Data Mining of Software Development Databases

Taghi M. Khoshgoftaar (taghi@cse.fau.edu)
Florida Atlantic University, Boca Raton, Florida USA

Edward B. Allen (edward.allen@computer.org)*
Mississippi State University, Mississippi USA

Wendell D. Jones (wendellj@nortelnetworks.com)
Nortel Networks, Research Triangle Park, North Carolina USA

John P. Hudepohl (hudepohl@nortelnetworks.com)
Nortel Networks, Research Triangle Park, North Carolina USA

Abstract. Software quality models can predict which modules will have high risk, enabling developers to target enhancement activities to the most problematic modules. However, many find collection of the underlying software product and process metrics a daunting task.

Many software development organizations routinely use very large databases for project management, configuration management, and problem reporting which record data on events during development. These large databases can be an unintrusive source of data for software quality modeling. However, multiplied by many releases of a legacy system or a broad product line, the amount of data can overwhelm manual analysis. The field of data mining is developing ways to find valuable bits of information in very large databases. This aptly describes our software quality modeling situation.

This paper presents a case study that applied data mining techniques to software quality modeling of a very large legacy telecommunications software system's configuration management and problem reporting databases. The case study illustrates how useful models can be built and applied without interfering with development.

Keywords: knowledge discovery, data mining, software quality modeling, software metrics, classification trees

1. Introduction

High software quality is essential for mission-critical systems. However, assuring high quality often entails time-consuming, costly, development processes, such as more rigorous design and code reviews, automatic test-case generation, more extensive testing, and reengineering of high-risk portions of a system. One cost-effective strategy is to target enhancement activities to those software modules that are most likely to have problems (Hudepohl et al., 1996). A *software quality model* can

* This work was performed while Edward B. Allen was at Florida Atlantic University.

predict which modules will probably have customer-discovered faults. Enhancement efforts can then be effectively targeted.

A typical software quality model consists of an algorithm in which a single *response* variable is a function of a set of *predictor* variables.[1] Predictors can be measured earlier in the development life cycle than the response variable, whose value is predicted. Prior research results (Arthur and Henry, 1995; Fenton and Pfleeger, 1997; Oman and Pfleeger, 1997) have shown that software product and process metrics can be quality predictors. *Software product metrics* are measures of product attributes, and *software process metrics* are measures of attributes of software development processes (Fenton and Pfleeger, 1997).

Decision-support systems that provide software quality models during development are a reality today. For example, Nortel's Enhanced Measurement for Early Risk Assessment of Latent Defects system, EMERALD, routinely assesses the risk of faults in software under development (Hudepohl et al., 1996). Such systems deliver predictions by software quality models to developers, so that risks can be dealt with before there is a quality problem for the end user. Accurate, robust, timely quality models are the keys to success for these systems.

A recent status report (Pfleeger et al., 1997) on the field of software measurement highlights gaps between current research and practice. For example, practitioners want accurate, timely predictions of which modules have high risk, but researchers have yet to find adequate, widely applicable measures and models. Faults are a result of mistakes or omissions by developers, and relevant human behavior in the workplace is notoriously difficult to measure directly.

We take a more pragmatic approach. We capture relevant variation among modules with practical metrics, even though the underlying human behavior is not well understood. Instead of expensive, specialized data collection, we leverage existing databases collected for other purposes, so that the marginal cost of data collection is modest. Rather than waiting for researchers to formulate a general theory, we achieve useful accuracy by empirically calibrating models to each local development environment.

Fayyad (1996) defines *knowledge discovery in databases* as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data." Fayyad restricts the term *data mining* to one step in the knowledge-discovery process, namely, extracting patterns or fitting models from data. Others use the term more broadly. "Primary data analysis" in statistics is motivated by

[1] We follow terminology in the classification tree literature, calling dependent and independent variables "response" and "predictor" variables, respectively.

a particular set of questions that are formulated before acquiring the data. In contrast, data mining analyzes data that has been collected for some other reason. Hand (1998) defines data mining as "the process of secondary analysis of large databases aimed at finding unsuspected relationships which are of interest or value to the database owners." Data mining is most appropriate when one seeks valuable bits of knowledge in large amounts of data collected for some other purpose, and when the amount of data is so large that manual analysis is not possible.

This aptly describes software quality modeling, especially when faults discovered by customers are rare. Many software development organizations have very large databases for project management, configuration management, and problem reporting which capture data on individual events during development. For large legacy systems or product lines, the amount of available data can be overwhelming. Manual analysis is certainly not possible. However, we have found that these databases do contain indicators of which modules will likely have customer-discovered faults.

2. Knowledge-Discovery Process

Given a set of large databases or a data warehouse, Fayyad et al. (1996b) give a framework of major steps in the knowledge-discovery process: (1) selection and sampling of data; (2) preprocessing and cleaning of data; (3) data reduction and transformation; (4) data mining; and (5) evaluation of knowledge. This paper presents a case study in which we applied Fayyad's framework to predicting software quality from software development databases. We extracted knowledge from a very large legacy telecommunication systems's configuration management and problem reporting databases. Our framework is shown in Figure 1.

Figure 1 has two similar tracks of processing steps. The upper track processes data on past releases where fault data is known. The results of this track are an empirical model, an assessment of its accuracy, and an interpretation of its structure. The lower track processes data on a current release that is still under development, predicting which modules will be fault-prone through the empirical model. The human figure in the corner represents a developer who will make use of the predictions, the expected accuracy, and the knowledge derived from the model's structure.

In Figure 1, the *Data Warehouse* represents software development databases, such as configuration management systems and problem reporting systems, irrespective of the storage system implementation.

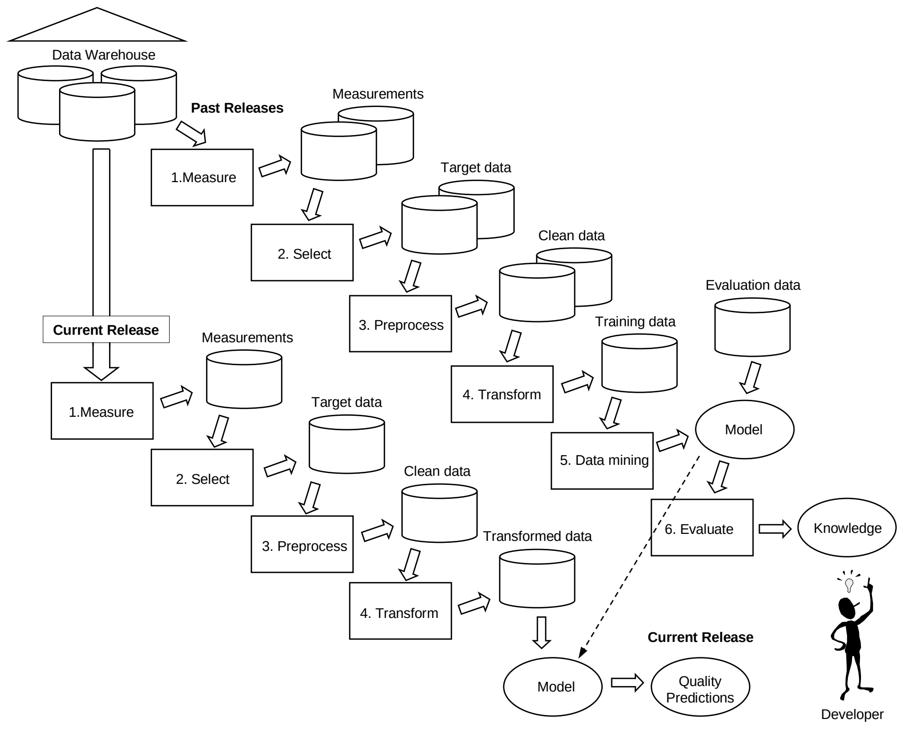

Figure 1. Knowledge Discovery from Software Development Databases

A *configuration management system* is an information system for managing multiple versions of artifacts produced by software development processes. For example, most configuration management systems support storage and retrieval of versions of source code. Other features may regulate changes to source code, so that team members do not interfere with each other, and record the history of changes for later review. A *problem reporting system* is an information system for managing software faults from initial discovery through distribution of fixes. In other words, it records events in the debugging process. Most developers of large software products use such systems.

The first step measures available software development databases to derive variables from source code, configuration management transactions, and problem reporting transactions for one or more past releases.

Pragmatic considerations usually determine the set of available predictors. We do not advocate a particular set of software metrics for software quality models to the exclusion of others recommended in the literature. Because marginal data collection costs are modest, we prefer to apply data mining to a large well-rounded set of metrics rather than limit the collection of software metrics according to predetermined research questions.

Step 2, Select, chooses data for study, resulting in target data. Step 3, Preprocess, accounts for missing data and outliers in the target data, resulting in clean data. Step 4, Transform, may extract features from the clean data, and may transform data for improved modeling. The result is separate transformed data sets for training and for evaluation. Step 5, Data mining, builds a model based on the training data. Step 6, Evaluate, assesses the model's accuracy using the evaluation data, and analyzes the model's structure.

In the case study, Step 5, Data mining, resulted in a classification tree model that predicts whether a module is likely to have faults discovered by customers. Classification trees are a well-known modeling technique in the field of data mining (Fayyad et al., 1996a; Weir et al., 1995). The case study confirmed prior empirical work (Gokhale and Lyu, 1997; Khoshgoftaar et al., 1998; Porter and Selby, 1990; Takahashi et al., 1997; Troster and Tian, 1995) that demonstrated the potential usefulness of classification trees for identifying fault-prone modules based on patterns of software metrics. Alternative classification techniques have been applied to this problem, including discriminant analysis, the discriminative power technique, logistic regression, pattern recognition, artificial neural networks, and fuzzy classification. Future work may explore other classification techniques, as well.

3. Case Study

We conducted a case study of a very large legacy telecommunications system (Naik, 1998). The results are summarized in Table I. This embedded-computer application included numerous finite-state machines and interfaces to other kinds of equipment. Such systems require very high software reliability. A *module* consisted of a set of related source-code files. The software was written in a high level language (Protel) using the procedural development paradigm, and was maintained by professional programmers in a large organization. The entire system had significantly more than ten million lines of code (LOC). Updated modules had more than five million lines of code in more than three thousand modules.

Table I. Case Study Results

Goal	Predict fault-prone modules
fault-prone	Customer-discovered faults > 0
Constraint	About 20% of modules predicted to be fault-prone
Selection	
Training data set	More than 3 thousand updated modules
Evaluation data set	Updated modules in next release (also > 3K modules)
Data Mining	
Modeling technique	Classification tree (CART)
Candidate predictors	42 product, process, and deployment metrics
Knowledge	
Training data set	7.4% were fault-prone
Significant predictors	

Metric	Definition	Concept
Product metrics:		
FILINCUQ	Number of distinct files included	Interfaces
VARSPNMX	Maximum span of variables	Locality
CNDSPNSM	Total span of vars in control structures	Locality
NDSENT	Number of procedures	Size
NDSINT	Number of internal nodes in CFG	Size
STMCTL	Number of control statements	Size
Process metrics:		
TOT_UPD	Number of updates	Code churn
SRC_MOD	Net new and changed *LOC*	Code churn
UNQ_DES	Number of designers making updates	Size of team
UPD_CAR	Number of updates in company career	Experience
Deployment metric:		
USAGE	Fraction of sites installed	Extent of use

Accuracy

Fault-prone modules in evaluation data set	4.7%
Modules predicted to be fault-prone	21.0%
Fault-prone modules correctly predicted	63.0%
Not fault-prone modules correctly predicted	81.1%

A preliminary empirical study (Khoshgoftaar et al., 1998) examined one release of this system. Here, we took advantage of additional data that became available on the subsequent release, as well. The first release had more than thirteen thousand configuration management transactions which were due to both normal development and resolution of more than six thousand problem reports (reported both before and after release). The scale of the second release was similar. In other words, the study was based on a large amount of data.

A module was considered fault-prone if any faults were discovered by customers, and not fault-prone otherwise. Faults discovered in deployed telecommunications systems are typically extremely expensive, because, in addition to down-time due to failures, visits to customer sites are usually required to repair them. Fault data was collected at the module-level by the problem reporting system. The following sections describe each step of the knowledge-discovery process for this case study.

3.1. Measure Software Product and Process

The framework of Fayyad et al. (1996b) puts feature extraction in Step 4, Transform. In our context, we performed feature extraction on two levels: (1) in Step 1, Measure, we measured software source code to obtain software product metrics and tabulated other data to obtain process metrics and deployment metrics, and (2) in Step 4, Transform, we transformed some measurements to be more suitable for modeling.

The EMERALD system retrieved the source code from the configuration management system and measured static software product metrics for each module, similar to those collected by commercially available metric analyzers (Mayrand and Coallier, 1996). Such metric analyzers measure a wide variety of source-code attributes, which we have found to have adequate variation for software quality modeling (Khoshgoftaar et al., 2000). Attributes of a call graph, control flow graphs, and source code statements were measured. A call graph depicts invocation relationships among procedures. A control flow graph (CFG) shows the flow of control where each arc represents a series of in-line statements and each node represents a decision statement, such as an IF, FOR, or WHILE statement, or a branch destination.

We extracted the software process metrics from configuration management data and problem reporting data (Khoshgoftaar et al., 1999a). For each change to a source file, the configuration management system recorded the identity of the developer, and a counter of the number of times that person had modified any source file since the inception of the configuration management system. In other words, at the time of a change, the number of changes made by that person in their company career was recorded. Software process metrics quantified attributes of the updates and of the developers, aggregated to the module level.

For each module, we forecast the proportion of systems expected to have that module installed ($USAGE$) based on past deployment records and installation plans. This is an approximation for the extent of usage of a module (Jones et al., 1999).

3.2. Select Modules for Study

Step 2, Select, produced a target set of modules. Preliminary data analysis was part of this step. Configuration management data included the date of the most recent update to each module. Compared to the beginning date of the current development cycle, this yielded an update indicator, identifying modules that were unchanged from the prior release. Prior empirical research has shown that such an indicator can be a significant predictor in software quality models (Khoshgoftaar et al., 1996a; Khoshgoftaar et al., 1996b). Approximately 99% of the unchanged modules had no faults. Consequently, modeling of this subset was not appropriate, because there were too few fault-prone modules.

This case study considered only updated modules, i.e., those that were new or had at least one update to source code since the prior release. The proportion of modules with no faults among the updated modules was $\pi_{nfp} = 0.926$, and the proportion with at least one fault was $\pi_{fp} = 0.074$. Such a small set of modules is often difficult to predict accurately early in development.

We selected two consecutive past releases for study where the actual numbers of customer-discovered faults were known for a substantial period of operations. The first release was the basis for the *training* data set, and the subsequent release was the basis for the *evaluation* data set. A preliminary study (Khoshgoftaar et al., 1998) split data from only the first release into training and evaluation data sets. We prefer to use subsequent releases for evaluation when the data is available, because this simulates use of a model on a current project better than the data-splitting approach.

3.3. Preprocess Data Sets

Preprocessing and cleaning resolves issues such as outliers and missing data. This case study did not need to perform extensive preprocessing. We omitted from the cleaned data sets the few modules that were missing relevant data.

3.4. Transform Data

Data reductions and transformations include techniques to reduce the number of predictors or to project data onto spaces for easier problem solution. In the case study, we transformed selected variables for improved modeling.

The measurement data included the total number of nodes (NDS) in the control flow graph, as well as the number of entry nodes ($NDSENT$), exit nodes ($NDSEXT$), and inaccessible nodes (dead code) ($NDSPND$). The latter three types of nodes are somewhat correlated with the total, because they are components. Correlation among variables risks a less robust model. Therefore, we transformed the variables and used internal nodes, $NDSINT = NDS - NDSENT - NDSEXT - NDSPND$, as a candidate variable in the modeling, rather than the total (NDS).

The measurement data included the total number of independent paths ($PTHIND$) in the control flow graph. The range was greater than 10^{32} which was too large for preliminary statistical analysis. We made a monotonic transformation to a more practical range; the base 2 logarithm of independent paths was a candidate variable, $LGPATH = \log_2 PTHIND$. It turned out that this variable was not significant in the model.

The measurement data included several averages, such as the average number of updates in developers' company careers ($UPDAV$), and the average span of variables per conditional structure ($CNDSPNAV$). The span of a variable is the number of lines of code between its first and last use in a procedure. Because we measure quality by customer-discovered faults over the entire module and not as an average, we transformed averages to the totals, for example, $UPD\_CAR = (UPDAV)(TOT\_UPD)$ and $CNDSPNSM = (CNDSPNAV)(CND)$, where CND is the number of conditional structures.

3.5. Perform Data Mining

Step 5, Data mining, extracts patterns or models from clean, transformed data, for example, fitting a model or finding a pattern. This step discovered significant and important relationships between customer-discovered faults and certain predictors. The result was a classification tree model that was suitable for predicting whether customers were likely to discover faults in a module.

In our application, a classification tree represents an algorithm as an abstract tree of decision rules to classify a module as a member of the not fault-prone group or the fault-prone group. Each module has a set of predictors, namely, software product and process metrics,

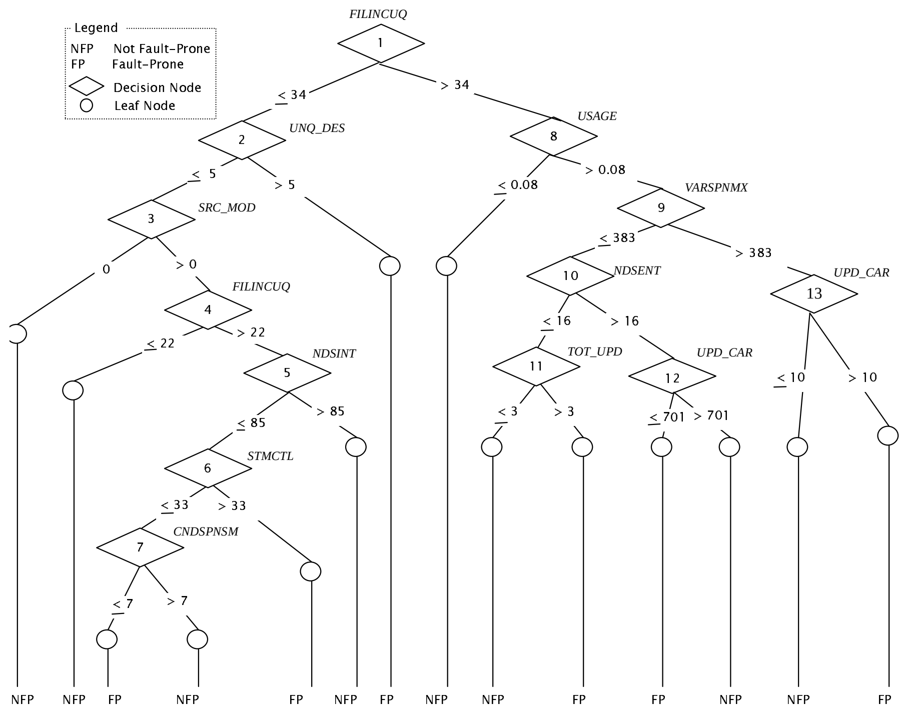

Figure 2. Classification Tree

and a response variable with two possible values, *not fault-prone* or *fault-prone*. Figure 2 depicts the preferred classification tree in our case study. Each diamond node represents a decision, and each edge represents a possible result of that decision. Each circular node is a leaf that classifies a module into the group noted at the bottom. The root of the tree is the node at the top.

In Figure 2, a module is represented by its measurements. Beginning at the root node, the algorithm traverses a downward path in the tree, one node after another, until it reaches a leaf node. The current decision node is applied to one measurement. For example, in Figure 2, Node 1 examines the module's measurement of the number of unique file-includes, *FILINCUQ*. If $FILINCUQ \leq 34$ then the algorithm chooses the left edge to Node 2. Otherwise the algorithm proceeds to Node 8 which is another decision. At Node 8, if $USAGE \leq 0.08$ then the algorithm proceeds along the left edge where the module is classified as not fault-prone. In other words, many interfaces but low usage implies customers are not likely to discover mistakes. Otherwise, the algorithm proceeds to the right to Node 9.

The process is repeated for each node along the path. When a decision node is reached, it is applied to the module. When a leaf node is reached, the module is classified as not fault-prone or fault-prone, and the path is complete. Each module in a data set can be classified using such an algorithm. Each path from the root to a leaf specifies a pattern of predictors. The tree as a whole is a classification model, which can predict the class membership of each module from a similar system or subsequent release. Moreover, each path from root to leaf can be interpreted as a combination of software development attributes that is associated with the class of the leaf. This can yield insights into development processes.

The Classification And Regression Trees (CART) algorithm (Breiman et al., 1984) builds a classification tree. It is implemented as a supplementary module for the SYSTAT package (Steinberg and Colla, 1995). We model each module with a set of ordinal-scaled predictors, namely, software product and process metrics, and a nominal-scaled response variable with two categories, *not fault-prone* or *fault-prone*. Classification tree algorithms, such as CART, automatically choose the predictor and threshold for each decision node.

Beginning with all modules in the root node, the algorithm recursively partitions ("splits") the set into two leaves until a stopping criterion applies to every leaf node. A goodness-of-split criterion is used to minimize the heterogeneity ("node impurity") of each leaf at each stage of the algorithm. CART's default goodness-of-split criterion is the "Gini index of diversity" which is based on probabilities of class membership (Breiman et al., 1984). Other classification tree algorithms that have been applied to software engineering use other goodness-of-split criteria, such as entropy (Porter and Selby, 1990; Takahashi et al., 1997), deviance (Troster and Tian, 1995), and statistical significance (Khoshgoftaar et al., 1999b). CART stops splitting if a node has too few modules (e.g., less than ten modules), or if all modules have exactly the same measurements. The result of this process is typically a maximal tree which overfits the data set, and consequently, is not a robust model. CART then generates a series of trees by progressively pruning branches from the maximal tree. The accuracy of each size of tree in the series is estimated by ν-fold cross-validation and the most accurate tree in the series is selected as the final classification tree.

ν-fold cross-validation is a method for estimating model accuracy (Efron, 1983; Gokhale and Lyu, 1997; Lachenbruch and Mickey, 1968). The algorithm has these steps: Randomly divide the training data set into ν approximately equal subsets. Our case study used the CART default, $\nu = 10$ (Breiman et al., 1984). Set aside one subset as an evaluation sample, and build a tree with the modules of the remaining $\nu - 1$ subsets. Classify the modules in the evaluation subset and note the accuracy of each prediction. Repeat this process, setting aside each

subset in turn. Calculate the overall accuracy. This is an estimate of the accuracy of the tree built using all the modules.

CART allows one to specify prior probabilities, and costs of misclassifications. Let π_{fp} and π_{nfp} be prior probabilities of membership in the fault-prone and not fault-prone classes, respectively. We define a Type I misclassification to be when the 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. Let C_I and C_{II} be the costs of Type I and Type II misclassifications, respectively. These parameters are used to evaluate goodness-of-split of a node as a tree is recursively generated.

We applied the CART algorithm to the training data set to build a classification tree model that predicts whether a module will be fault-prone or not. The case study illustrates a hypothetical application of our approach. Suppose we had only enough resources to enhance about 20% of the modules. We built numerous trees with various parameter values that choose a balance between Type I and Type II misclassification rates, which in turn, determine the proportion of modules predicted to be fault-prone. Figure 2 depicts the preferred classification tree generated by CART.

Given a candidate value of $\zeta = (\pi_{fp}/\pi_{nfp})(C_{II}/C_I)$. We have observed a tradeoff between the Type I and the Type II misclassification rates as functions of ζ, irrespective of the component values of π_{fp}, π_{nfp}, C_I, and C_{II}. We build a tree and estimate the Type I and Type II rates using ν-fold cross-validation. We repeat for various values of ζ, until we arrive at the preferred ζ for the project. In the case study, we preferred $\zeta = 0.65$, so that the proportion predicted to be fault-prone would be 19.4% when calculated by cross-validation of the training data set. Other projects should choose their own preferred value of ζ based on local criteria. A parameter like ζ is useful with other classification techniques, as well (Khoshgoftaar and Allen, 2000).

3.6. Evaluate Knowledge

Step 6, Evaluation, assesses the accuracy of the model and draws lessons from the structure of the model. We applied each model to the evaluation data set to predict the class membership of each module. Table I above summarizes the results of our case study. The Type I misclassification rate was 18.9%, the Type II rate was 37.0%, and the overall misclassification rate was 19.8%. In other words, 81.1% of the not fault-prone modules were correctly predicted, and 63% of the the fault-prone modules were in the set of modules that the model predicted to be fault-prone. This level of accuracy is very valuable to a telecommunications software project, because fixing fault-prone modules after release is very expensive, and they are difficult to identify prior to release. This balance between misclassification rates implied that 21% of the modules in the evaluation data set were predicted to be fault-prone. This achieved our goal of "about 20%", when we chose a preferred value of ζ.

Table I lists the significant predictors that appear in Figure 2. In general, one predictor does not indicate whether a module will be fault-prone, or not. One must consider the combination of predictors that lie along a path from the root to a leaf. The root node was a decision node on the number of distinct file-includes (*FILINCUQ*, Nodes 1 and 4). Include-file directives are commonly used to insert header files into the source code at compile-time. Header files are used to define shared data abstractions and function prototypes. Hence, the number of distinct includes indicated the amount of interfaces a module has. Two predictors were related to the span of variables (*VARSPNMX*, Node 9 and *CNDSPNSM*, Node 7). The span of a given programming variable is related to how close together are its first and last references. A large span implies that use of the variable is not localized. The number of entry nodes (*NDSENT*, Node 10) to control flow graphs was equivalent to the number of procedures, because in this system, all procedures have only one entry point. Preliminary data analysis showed that the number of procedures (*NDSENT*, Node 10), the number of internal nodes (*NDSINT*, Node 5) in the control flow graph, and the number of control statements (*STMCTL*, Node 6) were all highly correlated with other measures of module size, such as lines of code (*LOC*).

The amount of update activity, also called "code churn", was measured both by the number of updates (*TOT_UPD*, Node 11) and by the number of new and changed lines of code (*SRC_MOD*, Node 3). The number of designers (*UNQ_DES*, Node 2) indicated how many people modified the same module. Too many people implied the modules was more likely to be fault-prone. The overall amount of experience the designers had was indicated by the number of updates made in their company-careers (*UPD_CAR*, Nodes 12 and 13). The usage (*USAGE*, Node 8) of a module indicated the relative opportunity customers had to find faults. Very low usage implied a module will not be considered fault-prone.

Our case study concluded with this step.

3.7. Application to a Current Release

The payoff for building a model based on past releases is to make predictions of the quality of a current release that is still under development.

The model built in Step 5, Data mining, can be used on data from a current release, as indicated by the dashed arrow in Figure 1. Processing of the current release follows similar steps as past releases, until one uses the model to predict software quality on a module-by-module basis.

1. Measure software products and processes. Products and process history of the current release are measured at the module level in the same way as the past releases.

2. Select modules for predictions. Modules from the current release are selected for prediction using the same criteria that were used in the knowledge-discovery process. For example, the case study considered updated modules only.

3. Preprocess data. Relevant preprocessing and cleaning steps are applied to data from the current release. For example, one might omit modules that are missing data for significant variables in the model. The status of data for variables not in the model is irrelevant.

4. Transform data. The same data reductions and transformations are applied to data from the current release as were applied to the past releases. For example, in the case study, we transformed several variables.

5. Apply the model. The above steps have prepared the data from the current release to be compatible with the model. This step applies the model to the data from the current release, making a prediction for each module.

The EMERALD team has observed progressive improvement in fault-levels as this methodology has been adopted by software developers.

3.8. Infrastructure

Routine knowledge discovery and quality prediction as depicted in Figure 1 requires a technology infrastructure. The data warehouse must enable convenient access to configuration management and problem reporting data without time-consuming manual processing. Adequate processing power must be applied to the knowledge-discovery steps. Application to a current release must be early enough in the life cycle that developers have time to improve the software's quality. Finally, results must be delivered to developers in a timely, convenient, attractive manner. The payoff for the system comes when developers use the predictions to target those modules that need enhancement the most.

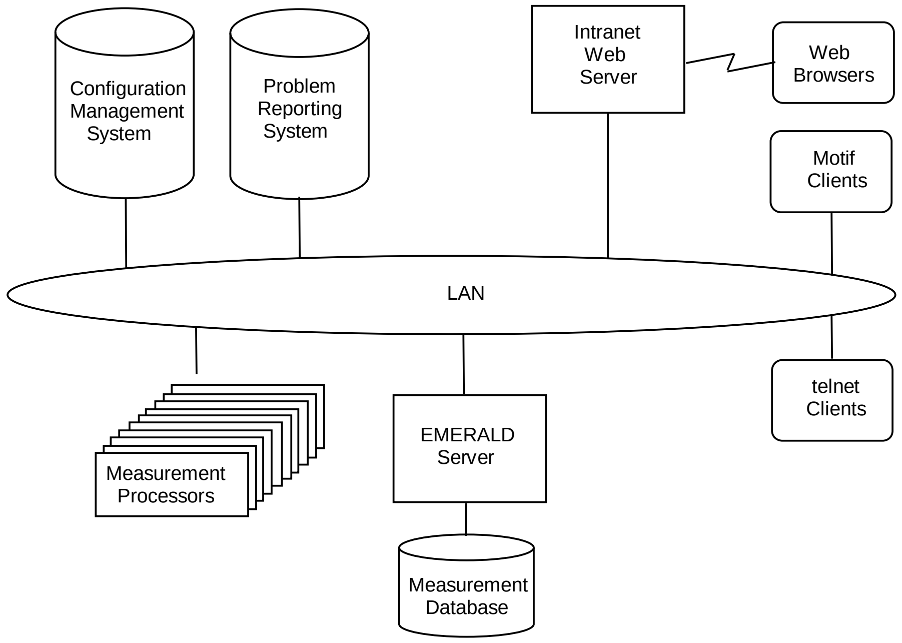

Figure 3. EMERALD System Architecture

For example, Figure 3 depicts EMERALD's client-server architecture (Hudepohl et al., 1996). A network of engineering workstations periodically extracts the latest source code from the library, and calculates software metrics. Distributed computation was required to handle the large number of modules in our case study's very large databases. The EMERALD server stores measurement history and calculates predictions. The EMERALD graphical user interface (GUI) is available as a company-private World Wide Web page, using a Web browser, and alternatively, a Motif-compatible interface via a local area network. There is also a batch command interface and a command line interface using telnet.

Interfaces to the configuration management and problem reporting systems make the process of collecting data unintrusive. Experiments with the various risk assessment models give an empirical basis for relying on model predictions. An inviting user interface facilitates acceptance of the system by the developers, so that software metrics and models are being integrated into day-to-day development activities.

4. Conclusions

The knowledge-discovery steps of Fayyad et al. (1996b) are readily adapted to the process of building and applying a software quality model. This paper shows that one can extract useful information from large configuration management and problem reporting databases. The case study developed a classification tree using the CART algorithm to predict whether a module would likely have customer-discovered faults. A software developer would benefit from predictions of which modules are fault-prone, the expected accuracy of those predictions, and knowledge derived from the model's structure.

Future research will extract new, useful software product and process metrics from software development databases whose collection need not intrude into development activities. New metrics, in turn, will stimulate the need for new modeling techniques, as well. Future research will also explore infrastructures for doing data mining of software development databases as a routine part of software engineering.

Acknowledgements

We thank Archana Naik for performing the experiments. We thank the anonymous reviewers for their thoughtful comments. This work was supported in part by a grant from Nortel Networks through the Software Reliability Engineering Department. 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

Arthur, J. D. and S. M. Henry (eds.): 1995, *Software Process and Product Measurement*, Vol. 1 of *Annals of Software Engineering*. J. C. Baltzer.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone: 1984, *Classification and Regression Trees*. London: Chapman & Hall.

Efron, B.: 1983, 'Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation'. *Journal of the American Statistical Association* **78**(382), 316–331.

Fayyad, U. M.: 1996, 'Data Mining and Knowledge Discovery: Making Sense Out of Data'. *IEEE Expert* **11**(4), 20–25.

Fayyad, U. M., D. Haussler, and P. Stolorz: 1996a, 'Mining Scientific Data'. *Communications of the ACM* **39**(11), 51–57.

Fayyad, U. M., G. Piatetsky-Shapiro, and P. Smyth: 1996b, 'The KDD Process for Extracting Useful Knowledge from Volumes of Data'. *Communications of the ACM* **39**(11), 27–34.

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

Gokhale, S. S. and M. R. Lyu: 1997, 'Regression Tree Modeling for the Prediction of Software Quality'. In: H. Pham (ed.): *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*. Anaheim, CA, pp. 31–36.

Hand, D. J.: 1998, 'Data Mining: Statistics and More?'. *The American Statistician* **52**(2), 112–118.

Hudepohl, J. P., S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand: 1996, 'EMERALD: Software Metrics and Models on the Desktop'. *IEEE Software* **13**(5), 56–60.

Jones, W. D., J. P. Hudepohl, T. M. Khoshgoftaar, and E. B. Allen: 1999, 'Application of a Usage Profile in Software Quality Models'. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. Amsterdam, Netherlands, pp. 148–157.

Khoshgoftaar, T. M. and E. B. Allen: 2000, 'A Practical Classification Rule for Software Quality Models'. *IEEE Transactions on Reliability* **49**(2), 209–216.

Khoshgoftaar, T. M., E. B. Allen, W. D. Jones, and J. P. Hudepohl: 1999a, 'Data Mining for Predictiors of Software Quality'. *International Journal of Software Engineering and Knowledge Engineering* **9**(5), 547–563.

Khoshgoftaar, T. M., E. B. Allen, W. D. Jones, and J. P. Hudepohl: 2000, 'Classification-Tree Models of Software-Quality Over Multiple Releases'. *IEEE Transactions on Reliability* **49**(1), 4–11.

Khoshgoftaar, T. M., E. B. Allen, K. S. Kalaichelvan, and N. Goel: 1996a, 'Early Quality Prediction: A Case Study in Telecommunications'. *IEEE Software* **13**(1), 65–71.

Khoshgoftaar, T. M., E. B. Allen, K. S. Kalaichelvan, and N. Goel: 1996b, 'The Impact of Software Evolution and Reuse on Software Quality'. *Empirical Software Engineering: An International Journal* **1**(1), 31–44.

Khoshgoftaar, T. M., E. B. Allen, A. Naik, W. D. Jones, and J. P. Hudepohl: 1998, 'Using Classification Trees for Software Quality Models: Lessons Learned'. In: *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*. Bethesda, Maryland USA, pp. 82–89.

Khoshgoftaar, T. M., E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl: 1999b, 'Assessing Uncertain Predictions of Software Quality'. In: *Proceedings of the Sixth International Software Metrics Symposium*. Boca Raton, Florida USA, pp. 159–168.

Lachenbruch, P. A. and M. R. Mickey: 1968, 'Estimation of Error Rates in Discriminant Analysis'. *Technometrics* **10**(1), 1–11.

Mayrand, J. and F. Coallier: 1996, 'System Acquisition Based on Software Product Assessment'. In: *Proceedings of the Eighteenth International Conference on Software Engineering*. Berlin, pp. 210–219.

Naik, A.: 1998, 'Prediction of Software Quality Using Classification Tree Modeling'. Master's thesis, Florida Atlantic University, Boca Raton, FL USA. Advised by Taghi M. Khoshgoftaar.

Oman, P. and S. L. Pfleeger (eds.): 1997, *Applying Software Metrics*. Los Alamitos, CA: IEEE Computer Society Press.

Pfleeger, S. L., R. Jeffery, B. Curtis, and B. A. Kitchenham: 1997, 'Status Report on Software Measurement'. *IEEE Software* **14**(2), 33–43.

Porter, A. A. and R. W. Selby: 1990, 'Empirically Guided Software Development Using Metric-based Classification Trees'. *IEEE Software* **7**(2), 46–54.

Steinberg, D. and P. Colla: 1995, 'CART: A supplementary modules for SYSTAT'. Salford Systems, San Diego, CA.

Takahashi, R., Y. Muraoka, and Y. Nakamura: 1997, 'Building Software Quality Classification Trees: Approach, Experimentation, Evaluation'. In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering.* Albuquerque, NM USA, pp. 222–233.

Troster, J. and J. Tian: 1995, 'Measurement and Defect Modeling for a Legacy Software System'. *Annals of Software Engineering* **1**, 95–118.

Weir, N., U. M. Fayyad, and S. G. Djorgovski: 1995, 'Automated Star/Galaxy Classification for Digitized POSS-II'. *Astronomical Journal* **109**(6), 2401–2412.

Authors' Vitae

Taghi M. Khoshgoftaar
is a professor of the Department of Computer Science and Engineering, Florida Atlantic University and the Director of the Empirical Software Engineering Laboratory. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, computational intelligence, computer performance evaluation, multimedia systems, and statistical modeling. He has published more than 150 refereed papers in these areas. He 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, the IEEE Computer Society, and IEEE Reliability Society. He served as the general chair of the 1999 International Symposium on Software Reliability Engineering (ISSRE'99), and the general chair of the 2001 International Conference on Engineering of Computer Based Systems. 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, Providence, Rhode Island USA, in 1971, the M.S. degree in systems engineering from the University of Pennsylvania, Philadelphia, Pennsylvania USA, in 1973, and the Ph.D. degree in computer science from Florida Atlantic University, Boca Raton, Florida USA, in 1995. He is currently an assistant professor in the Department of Computer Science at Mississippi State 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, software quality, and computer performance modeling. He has more than 60 refereed publications in these areas. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Wendell D. Jones
received his B.S. degree in Mathematics from Furman University, Greenville, South Carolina in 1983 and his M.S. and Ph.D. degrees in Math Science (Statistics emphasis) from Clemson University, Clemson, South Carolina in 1985 and 1987 respectively. He has held various positions in his eleven years with Nortel Networks, most recently as Advisor and Chief Researcher of the Dept. of Software Reliability Engineering and Tool Development. He is an active member of American Statistical Association and has served on several technical committees related to software reliability. His research interests include applying quantitative methods to software development and management, and applications of data analysis to large datasets.

John P. Hudepohl
received a B.S. in both electronic engineering and communication arts from the University of Dayton, and an M.S. in system management from Florida Institute of Technology. He is currently manager of Software Reliability Engineering at Nortel Networks, Research Triangle Park, North Carolina, USA. He has more than 20 years experience in the reliability, maintainability and quality fields for both hardware and software, with Cincinnati Electronics, with ITT, and since 1986, with Nortel Networks. He has published and presented numerous papers at IEEE conferences and workshops. He is a member of the IEEE, and past chair of the IEEE Communications Society's Quality Assurance Management Committee.

Address for Offprints: 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/.