Developing Accurate Software Quality Models Using a Faster, Easier, and Cheaper Method

Taghi M. Khoshgoftaar*
Linda Lim
Erik Geleyn
Florida Atlantic University
Boca Raton, Florida USA

Keywords: software reliability, faults, fault-prone modules, software metrics, classification, prediction,return on investment, case-based reasoning.

Abstract

This paper demonstrates how software quality models like case-based reasoning can be effective in facilitating the decision-making processes for managers so they can achieve the all-important goals of releasing a product on time and within budget, all the while adhering to established quality standards.

Case-based reasoning (CBR) is an alternative modeling method based on automated reasoning processes. It has proven useful in a wide variety of domains. CBR is especially useful when there is a limited knowledge and an optimal process solution is not known.

This paper primarily focuses on our empirical work. Through a case study, we attempt to build useful software quality models in order to predict the quality of the software prior to system test and to determine, by analyzing the fault removal process, where time, money, and energy could be saved.

1. Introduction

Recent publications have stated that models and measurements were the means for understanding, controlling, and improving development processes [14]. Various classification and prediction modeling techniques can use those metrics as input to compute either a class (fault-prone, not fault-prone) or an estimation of a quality factor (number of faults in the module under development). Previous research at the Empirical Software Engineering Lab at FAU has used discriminant analysis [2], logistic regression [4], decision trees [5][15], artificial neural networks [6][7], and fuzzy logic [17].

An effective prediction and classification technique is case based reasoning. Based on the assumption that a future module will have the same number of faults as a similar previously developed module, a case based reasoning model is able to retrieve a prediction from past instances [1]. Those past instances are the *cases* of our library, they are well-known data from past projects.

In our case study we wanted to predict the class (fault-prone, not fault-prone) of a currently developed program module to be able to take corrective actions early in the life cycle.

Our research is primarily focused on an empirical case study. Through this case study, we attempt to build useful software quality models in order to predict the quality of the software prior to system test and to determine, by analyzing the fault removal process, where time, money, and energy can be saved. We used few primitive software metrics and yielded good results. The case study sought to determine the quality of the software just prior to system test. In this case study, we built classification models. The classification models classified source files as either fault-prone or not fault-prone based upon a threshold determined by the user. Experiments were conducted by varying the user-defined thresholds and model validation tech-

*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/.

niques.

From this research, we have proven that through the use of a simple yet very effective methodology, CBR, we can build useful software quality models using very few primitive metrics and yield good results. The Type I (classifying a not fault-prone module as fault-prone) and Type II (classifying a fault-prone module as not fault-prone) misclassification rates were used to evaluate the classification accuracy of the software quality classification models built.

2. Case-based Reasoning

Based on the assumption that a future module will have the same number of faults than a similar previously developed module, a case based reasoning model is able to retrieve a prediction from past instances [1]. Those past instances are the *cases* of our library, they are well-known data from past projects. Those *cases* contain all the available information about the described program module.

1. The independent variables to evaluate the similarities between the modules.
2. The dependent variable to make a prediction for future program modules.

In our case study we wanted to estimate the fault-proneness of a currently developed program module to be able to take corrective action early in the life cycle. Case based reasoning models use *similarity functions* to determine the most similar items from the case library to the module under development. Various *similarity functions* are available and they all compute a distance from the target module to the *cases* in the library. In our experiment we used the Mahalonobis distance. The following is the equation for Mahalonobis Distance:

$$d_{ij} = (c_j - x_i)' S^{-1} (c_j - x_i) \quad (1)$$

Where c_j is the vector of independent variables of the j^{th} case of the library and x_i is the vector of independent variables of the target module. Prime (') means transpose, and S is the variance-covariance matrix of the independent variables over the entire case library. S^{-1} is its inverse. The closest modules determined by the *similarity function* are called *nearest neighbors.* We can then select the number of *nearest neighbors* (n_N) we want to consider to build the prediction for the dependent variable using a *solution algorithm. Solution algorithms* compute a prediction of the dependent variable using the dependent variables of the *nearest neighbors* from the case library. Some algorithms give the same importance to all the *nearest neighbors* while others give more importance to the closest *cases.* We selected one of the two available classification methods: data clustering and majority voting. In our case study we used data clustering. With the Data Clustering method, the case library is partitioned into clusters according to the actual class of each case. The distances to the clusters are then computed for the current case. The current case's classification is then determined by comparing the ratio of the average of these distances to the cost ratio. The classification terminology for Data Clustering is as follows [8]: for an unclassified case, x_i, let $d_{nfp}(x_i)$ be the average distance to the not fault-prone nearest neighbor cases, and let $d_{fp}(x_i)$ be the average distance to the fault-prone nearest neighbor cases. The following is the generalized classification rule for Data Clustering:

$$Class(x_i) = \begin{cases} NFP, & \text{if } \frac{d_{fp}(x_i)}{d_{nfp}(x_i)} > \frac{C_I}{C_{II}} \\ FP, & \text{Otherwise} \end{cases} \quad (2)$$

Our experiment were conducted using the Software Measurement Analysis and Reliability Toolkit (SMART) [8]. SMART is a research tool for software quality modeling. It was developed at the Empirical Software Engineering Lab (ESEL) at Florida Atlantic University.

3. Empirical Case Study

Our case study was performed on a real life system. The data sets were collected from two large Windows-based applications used for customizing wireless products. The applications are written in C++ and provide similar functionality.

During our classification experiments we used three different thresholds for classification:

- *Threshold 1*: a file is fault-prone if it contains one or more faults- $y \geq 1$
- *Threshold 2*: a file is fault-prone if it contains two or more faults- $y \geq 2$
- *Threshold 3*: a file is fault-prone if it contains three or more faults- $y \geq 3$

During the case study the cost ratios ($\frac{C_I}{C_{II}}$) were varied, and the one yielding the most balanced misclassification rates was selected for each number of *nearest neighbors* (n_N).

Once the cost ratio is determined we could compute the return on investment (ROI) using C_I as the cost for a Type I misclassification and C_{II} for the type II misclassification cost. The notation used *x:y* means that an investment of y units in extra reviews saved x units of effort later in the life cycle.

The case study analyzed a data set, which contains data on source files. After the data set was preprocessed and cleaned [3], 1211 observations remained. In this particular case, an observation is a source file. We used the following product metrics for our experiments:

- BASE_LOC, number of lines of code for the source file version just prior to the coding phase.
- SYST_LOC, number of lines of code for the source file version delivered to system test.
- BASE_COM, number of lines of commented code for the source file version just prior to the coding phase.
- SYST_COM, number of lines of commented code for the source file version delivered to system test.

We also added the process metric INSP (number of times the source file was inspected prior to system test) for our experiment.

Classification experiments: We conducted six classification experiments. The first three experiments involved the entire data set with one experiment conducted for each fault-proneness threshold chosen. The target data (test data) set for these experiments was the same as the fit data set, and the models built were validated using cross-validation [12]. The second set of three experiments involved the use of a fit data set and a separate target data set for building and validating the models.

Considering our experiments for *Threshold 1*, the optimum model shows a Type I misclassification rate of 14.34% and a Type II misclassification rate of 13.68%. Following the same analysis posed in [11], suppose we propose to increase our reliability by conducting extra reviews for those source files that are suspected of being fault-prone (FP) at a cost of one unit. If any of the files were later determined to be not fault-prone (NFP), then the extra reviews would simply be a waste of time. In our experiment, the best cost ratio was determined to be 0.50. Therefore, if the cost of a Type I misclassification is one unit, then the cost of a Type II misclassification is two (1/0.50) units. However, the two units actually represent the net cost of forfeited benefit. Because the cost of extra reviews is one unit, the actual cost of a Type II misclassification is three units. The three units consist of the cost of the forfeited benefit plus the cost of reviews. A perfect model for Threshold 1 would cost 402 units for review and yield

$$402 * 3 = 1206 \tag{3}$$

units in avoided debugging costs. When this model is compared to the optimum model, the expected cost of misclassifications would be

$$(809 * 0.1434 * 1) + (402 * 0.1368 * (3 - 1)) = 226 \tag{4}$$

units. The cost of reviews would be

$$(809 * 0.1434 * 1) + (402 * 0.8632 * 1) = 463 \tag{5}$$

units. The benefit would be reliability improvement for

$$(402 * 0.8632) = 347 \tag{6}$$

fault-prone files. This might avoid

$$(347 * 3) = 1041 \tag{7}$$

units in debugging costs later. This would be considered a good return on investment (ROI), 1041:463. The model identifies the FP files that need extra reviews. In this experiment, the cost of the extra reviews for the FP files would be 463 units. By focusing efforts on the source files that are deemed to be FP early, managers could potentially save a net of

$$(347 * 2) = 694 \tag{8}$$

units of debugging costs in the future.

Table 1 displays the results for the three classification experiments on the entire data set. For each of the thresholds, we obtained similar Type I and Type II misclassification rates. In addition, the ROI for each of the thresholds were also quite similar. If managers were to implement any of the three classification models, the percentage of FP source files could be reduced to less than 5%. In addition, they could expect to save anywhere from 47% ((602-321)/602) to 56% ((1041-460)/1041) in debugging costs. However, these ROI figures are only based on our modeling analysis. In reality, the actual ROI may be much greater. This is attributed to the fact that a higher quality product could be delivered to market. Customer satisfaction would be greater which could help a company's market share and image, and the cost of fixing faults late in the project would be minimized. In addition, the actual cost of latent faults discovered by customers may be higher than the costs obtained by our CBR modeling. These are just some of the benefits that would allow the ROI for building software quality models to be much greater.

Table 2 summarizes the results from the six classification experiments conducted on the entire and on the Fit/Test data sets. Splitting the data set into a fit data set and a target data set allowed us to validate

Table 1. Classification Model Results-Entire Data Set Experiments

T	Type I	Type 2	ROI Debug cost	Prior (%) Fault-prone	Reduction (%) Fault-prone
1	14.34%	13.68%	1041:463	33.20%	4.54%
2	13.91%	14.50%	721:356	21.64%	3.13%
3	14.74%	14.00%	602:321	16.52%	2.13%

Table 2. Classification Model Results-All Experiments

T	Cross-Validation Entire Data Set		Cross-Validation Fit/Target		Data Splitting Fit/Target		Average	
	Type I	Type II	Type I	Type II	Type I	Type II	n_N	CI/CII
1	14.34%	13.68%	15.91%	15.55%	15.75%	16.37%	2.28	0.57
2	13.91%	14.50%	15.09%	14.71%	15.08%	16.19%	2.10	0.40
3	14.74%	14.00%	15.65%	15.28%	15.08%	16.55%	3.16	0.37

the models built in our study. Data splitting offers the ability to simulate the use of the model in practice because the target data set can represent data from subsequent releases. 50 models were built to ensure the results obtained were due to accurate modeling instead of a fortunate data split. The data splitting results were quite similar to the cross-validation results which provides additional confidence that successful classification models were built. The Type II errors were a bit higher for the models validated using data splitting, but the difference was not significant. The cross-validation results from the entire data set was also quite similar to the data splitting results which used 2/3 of the data set as the fit data set. In addition, the results were similar for the three fault-proneness thresholds. This indicates that our classification models were not sensitive to the particular fault-proneness thresholds chosen in our research.

4. Conclusions

With the number of software driven devices on the rise, and more people depending on them, it is imperative for organizations to ensure the reliability of its software. However, the costs associated with developing software are also ever increasing. It then becomes difficult for project managers to remain within budget and on schedule. As a result, quality and reliability are often reduced in order to meet the promised delivery dates.

Through the use of software quality models, managers have the opportunity to discover areas for improvement in their development processes. The models can provide managers with the insight that will allow them to improve their reliability enhancement and resource allocation strategies.

We showed that through the use of software quality models, managers have the opportunity to discover areas for improvement in their development processes. The models can provide managers with the insight that will allow them to improve their reliability enhancement and resource allocation strategies.

From our research, we have proven that using a simple modeling methodology (CBR) and few metrics we can develop accurate and useful models faster, easier, and cheaper. In both of our case studies, only five independent variables were needed. This makes CBR very cheap to implement.

The case study proved that accurate software quality models can be built to predict the quality of the software prior to system test. The classification models indicated a ROI for determining the FP files early at approximately 50%. This reveals that by using classification models, managers have the opportunity to save time and money by identifying FP files early. By having this information early, test managers will know exactly which files are more likely to contain faults. As a result, resource allocation during system test could be greatly enhanced.

Software quality models can be of great use to managers. They have the ability to provide management with insight to better control their development and test processes. By making their processes more efficient and effective, managers have the ability to provide a reliable, high quality product in less time.

Acknowledgments

This work was supported in part by Cooperative Agreement NCC 2-1141 from NASA Ames Research Center, Software Technology Division (Independent Verification and Validation Facility). The findings and opinions in this paper belong solely to the authors and are not necessarily those of the sponsors, or collaborators.

References

[1] K. Ganesan, T. M. Khoshgoftaar, and E. B. Allen. Case-based software quality prediction. *International Journal of Software Engineering and Knowlegde Engineering*, 10(2):139–152, 2000.

[2] 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.

[3] T. M. Khoshgoftaar and E. B. Allen and W. D. Jones and J. P. Hudepohl. Data Mining for Predictiors of Software Quality *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547-563, 1999.

[4] 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.

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

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

[7] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the International Conference on Software Maintenance*, pages 58–67, Victoria, BC Canada, Sept. 1994. IEEE Computer Society.

[8] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. Software quality modeling: The software measurement analysis and reliability toolkit. *IEEE International Conference on Tools with Artificial Intelligence*, pages 54–61, Nov 2000.

[9] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi and J. McMullan Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371, Oct. 1996.

[10] 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, New Mexico USA, Nov. 1997. IEEE Computer Society.

[11] T. M. Khoshgoftaar and E. B. Allen Classification of Fault-Prone Software Modules: Prior Probabilities, Costs, and Model Evaluation *Empirical Software Engineering: An International Journal*, Volume3:275–298, Sep. 1998.

[12] L. Lim Developing Accurate Software Quality Models Using a Faster, Easier and Cheaper Method. Master's thesis, FAU, May 2001. Advised by T. M. Khoshgoftaar.

[13] G. J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold, New York, 1978.

[14] S. L. Pfleeger. Assessing measurement. *IEEE Software*, 14(2):25–26, Mar. 1997. Editor's introduction to special issue.

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

[16] N. Sundaresh. An Empirical Study of Analogy Based Software Fault Prediction. Master's thesis, FAU, May 2001. Advised by T. M. Khoshgoftaar.

[17] Z. Xu Fuzzy Logic Techniques for Software Reliability Engineering. Ph.D. dissertation, FAU, May 2001. Advised by T. M. Khoshgoftaar.