Using genetic programming to determine software quality

Matthew Evett, Taghi Khoshgoftaar, Pei-der Chien and **Ed Allen**
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida 33431
{matt, taghi, chienp, allene}@cse.fau.edu

Abstract

Software development managers use software quality prediction methods to determine to which modules expensive reliability techniques should be applied. In this paper we describe a genetic programming (GP) based system that classifies software modules as "faulty" or "not faulty", allowing the targetting of modules for reliability enhancement. The paper describes the GP system, and provides a case study using software quality data from a very large industrial project. The demonstrated quality of the system is such that plans are under way to integrate it into a commercial software quality management system.

Introduction

Buggy software is a fact of modern life. But while customers will tolerate some bugginess in consumer-market software, such as games and word processors, this is generally not the case in the enterprise market. In markets such as telecommunications, utility support, and factory floor production control, customers will not buy systems that do not have excellent reliability. Each software flaw in these environments can result in large financial loss to the owners. Consequently, software reliability is a strategic business weapon in today's competitive marketplace(Hudepohl 1990).

Correcting software faults late in the development life cycle (i.e., after deployment into the field) is often very expensive. Consequently, software developers apply various techniques to discover faults early in development(Hudepohl *et al.* 1996). These reliability improvement techniques include more rigorous design and code reviews, automatic test case generation to support more extensive testing, and strategic assignment of key personnel. While these techniques do not guarantee that all faults are discovered, they greatly decrease the probability of a fault going undiscovered before release. When a fault is discovered, it can be eliminated, and the repaired module possibly resubmitted for further reliability review.

Because reliability enhancement can be quite expensive, software development managers must attempt to apply reliability improvement techniques only where they seem most likely to pay off, that is, to those software modules that appear likely to suffer from flaws. In earlier work (Evett *et al.* 1998), we demonstrated a genetic programming-based system for targeting software modules for reliability enhancement. In this paper we extend this work in two significant ways: first, we demonstrate that the system's performance scales up by succesfully completing an industrial case study that is twenty-five times larger than that of our original study. Second, we provide a new mechanism (including a fitness evaluation process), *dynamic partitioning evaluation*, for optimizing the use of the system. The demonstrated quality of the resulting GP system is such that plans are under way to include its use in a commercial software quality management system.

Software Quality Modeling

Previous software quality modeling research has focused on classification models to identify *fault-prone* and *not fault-prone* modules(Khoshgoftaar *et al.* 1996). A software development manager can use such models to target those software modules that were classified as fault-prone for reliability improvement techniques.

However, such models require that "fault-prone" be defined before modeling, usually via a threshold on the number of faults predicted, and software development managers often do not know an appropriate threshold at the time of modeling. An overly high threshold could result in too few modules being classified as fault-prone, allowing some faulty modules to avoid the reliability testing that might have identified their flaws before being released. An overly low threshold, on the other hand, while it would likely result in the reliability testing of most of the faulty modules, would do so at excessive, probably prohibitive cost—more modules would be classified as fault-prone than resource limitations (manpower, deadlines) will permit.

To avoid this problem, we use *dynamic partitioning evaluation*, a technique that does not require an *a priori* threshold definition. Rather than providing a strict classification system, our goal is to develop models that predict the quality of each module relative to that of the other modules. These predictions are then used to rank the modules from the least to the most fault-prone.

We used GP to create models that predict the number of faults expected in each module, but we use these predictions only to rank the modules. Our evaluation of the quality of the generated models is based on ordinal criteria, rather than the amount of error in the predicted number of faults.

With a given rank-order of the modules, the software manager must decide which modules to submit to reliability testing. This is done by selecting a threshold (or cut-off), *c*, a percentage. In effect, the threshold delineates the set of modules that are considered "fault-prone" from those that are considered "not fault-prone". The top *c* percent of the modules according to the ranking (i.e., those designated "fault-prone") are chosen for testing. ***Dynamic partitioning evaluation*** is "dynamic" in the sense that the system presents the manager with an approximation of the cost of using different thresholds. The manager can use these approximations to optimize the selection of *c*.

These cost approximations are based on predicting the degree to which the module ranking is accurate. For the fault-prone detection, there are two distinct types of misclassifications, ***type I*** and ***type II error***. A ***type I error*** occurs when a low-risk (i.e., not fault-prone) module is classified as high risk (i.e., fault-prone). This could result in some wasted attention to low-risk modules. A ***type II error*** occurs when a high-risk module is classified as low risk. This could result in the release of a low quality product. From the perspective of a software manager, it is preferable to make a ***type I error*** than a ***type II error***, because the costs of ***type II error*** are higher (usually at least an order of magnitude or two) for correcting the faults later in the maintenance phase.

The software manager approximates the cost of each type of error (which varies greatly across domains), and then selects from a set of thresholds that one which appears most likely to minimize the overall cost, within the limitations of the available financial and man-power resources.

Predicting Software Quality

Most quality factors, including faultiness, are directly measurable only after software has been deployed. Fortunately, prior research has shown that software product and process metrics(Fenton & Pfleeger 1997) collected early in the software development life cycle can be the basis for reliability predictions.

Process metrics measure the attributes of the development process and environment. Some examples of process metrics are programmer attributes such as levels of programming experience, years of experience with a programming language, years of experience constructing similar software systems, use of object-oriented techniques, use of editors, and team programming practices.

Product metrics are measurements of the attributes of a software product. Product metrics are not concerned about how the program was created. Examples of product metrics include data structure complexity, program size, number of control statements, number of lines of code, number of calls to other modules, and number of calls to a certain module.

Our case studies of the models generated by our GP system are based on actual industrial software development projects. Our case study data consisted of the software metrics for each module in these projects, as well as the number of faults detected in the modules after deployment. The metrics were collected via the Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) system, a collection of decision support tools used to assess the risk of software faults. It was developed by Nortel (Northern Telecom) in partnership with Bell Canada and others(Hudepohl *et al.* 1996). Such industrial systems make practical use of software quality classification models.

The exact methodology used by our GP system to create models on the basis of this data, and our evaluation methodology is explained below.

System Description

We conducted a case study of a very large legacy telecommunications system, written in a high level language, and maintained by professional programmers in a large organization. The entire system had significantly more than ten million lines of code, and over 3500 software modules. This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment.

This case study focused on faults discovered by customers after release, *CUST_PR*, as the software quality metric. A module was considered *fault-prone* if any faults were discovered by customers.

$$Class = \begin{cases} \textit{not fault-prone} & \text{if } CUST\_PR = 0 \\ \textit{fault-prone} & \text{otherwise} \end{cases} \quad (1)$$

Fault data was collected at the module level by a problem reporting system. A module consisted of a set of related source code files. The proportion of modules with no faults among the updated modules was $\pi_1 = 0.9260$, and the proportion with at least one fault was $\pi_2 = 0.0740$.

Our goal was a model of the updated modules, where predictions could be made after beta testing. Software product metrics were collected from source code by the DATRIX software analysis tool (Mayrand & Coallier 1996), which is part of the EMERALD system, and were aggregated to the module level.

Development process data was largely drawn from a development metrics data base derived from configuration management and problem reporting data. EMERALD interfaces with this data base. The product and process metrics for a module consisted of those listed below:

FILINCUQ The number of distinct include files.

LGPATH The base 2 logarithm of the number of independent paths.

VARSPNMX The maximum span of variables.

USAGE A measure of the deployment percentage of the module. Higher values imply more widely deployed and used.

BETA_PR The number of problems found in this module during beta testing of the current release.

BETA_FIX Total number of different problems that were fixed for the current development cycle where the problems originated from issues found by beta testing of a prior release.

CUST_FIX Total number of different problems that were fixed for the current development cycle where the problems originated from issues found by customers in a prior release.

SRC_GRO Net increase in lines of code due to software changes.

UNQ_DES Number of different designers that updated this module.

UPD_CAR The total number of updates that designers had in their company careers when they updated this module.

VLO_UPD Number of updates by designers who had 10 or fewer total updates in entire company career.

Experimental Methodology

The basic unit for software quality modeling is an *observation*, which is a software module represented by a tuple of software measurements, $\mathbf{x}_j$, for observation j. The $\mathbf{x}_j$ values are vectors of the form $< x_{j:1}, x_{j:2}, \ldots, x_{j:n} >$, whose components are discrete or real-valued software metrics. The dependent variable of a model is the *quality factor*, $\mathbf{y}_j$, for each observation j. The *quality factor* usually is the number of faults in each software module for this study. The programs resulting from our GP system are the *models*. Let $\hat{\mathbf{y}}_{ij}$ be the estimate of $\mathbf{y}_j$ by model i. We develop software quality models based on training data where measurements and the quality factor are available for each module.

We impartially divided the available data on updated modules into approximately equal *fit* and *test* data sets, so that each data set had sufficient observations for statistical purposes. The *fit* data set was used to build the model, and the *test* data set was used to evaluate its accuracy.

The following steps outline the methodology we used to evaluate the effectiveness of our GP-based modeling system:

1. For each set of system parameters to be tested, make a number of runs sufficient to argue statistical validity (17 runs in this case).
   (a) Train a GP system using only the *training* data set to yield a best model for each run.
   (b) Use each best-of-run model to predict the quality factor in the *validation* modules, and order them accordingly.
   (c) Evaluate each best-of-run model using dynamic partitioning criteria (based on the dependent variable) detailed in Section .
2. Summarize the evaluation results across the best models of all runs.

Dynamic Partitioning Evaluation

Let C be management's preferred set of cut-off percentiles of modules ranked by predicted quality factor, and let n_c be the number of percentiles in C. In the case study, we chose the percentiles {90, 85, 80, 75, 70, 65, 60, 50} (corresponding to denoting the top {10, 15, 20, 25, 30, 35, 40, 50} percent of the ranking as fault-prone). Because the dataset was actually 7.4% fault-prone, the 92.6 percentile was added to the set as the highest reference point. Another project might choose different percentiles, but this set illustrates our methodology.

Let G_{tot} be the total number of modules, and G_{fp} be the total number of fault-prones in the validation data set's software modules. The following is our evaluation procedure used in Step 1c for each model. Given an individual, i, and a validation data set indexed by j:

1. Determine the perfect ranking of modules, $\mathbf{R}$, by ordering modules according to $\mathbf{y}_j$. Let $R(j)$ be the percentile rank of observation j.
2. Determine the predicted ranking, $\widehat{\mathbf{R}_i}$, by ordering modules according to $\hat{\mathbf{y}}_{ij}$. Let $\widehat{R}_i(j)$ be the percentile rank of observation j.
3. For each cutoff percentile value of interest, $c \in C$:
   (a) Account the number of modules, G_c, above the cutoff.
   (b) Account the number of actual fault-prones, $\widehat{G}_c(i)$, above the cutoff.
   (c) Calculate the *type I* error and *type II* error for various cutoff, c.

$$typeI_i(c) = \frac{G_c - \widehat{G}_c(i)}{G_{tot} - G_{fp}} \qquad (2)$$

$$typeII_i(c) = \frac{G_{fp} - \widehat{G}_c(i)}{G_{fp}} \qquad (3)$$

Because type II errors much more costly than type I errors, our primary measure of the accuracy of each quality model is the rate of *type II* errors.

Details of GP System

This section specifies the GP system used by this study.

Function and terminal sets The function set consists of

$$\mathcal{F} = \{+, -, \times, /, \sin, \cos, \exp', \log\} \qquad (4)$$

where $\exp'(x) = \exp(\sqrt{x})$, to lessen the risk of arithmetic overflow. Divide (/), exponentiation (exp'),

Table 1: The Tableau used in the case studies.

Terminal set:	Software product and process metrics available from each data set and $\Re$, varying over the range [0,1].
Function set:	$\{+, -, \times, /, \sin, \cos, \exp', \log\}$
Initialization:	Ramped half-and-half.
Fitness cases:	1825 (CCCS) module observations, each a tuple of numeric software metrics.
Raw fitness:	summation of correct prediction rate of fault-prones modules (see Equation 5).
Std. fitness:	same as raw fitness.
Wrapper:	Ranks fitness cases on basis of predicted number of faults.
Parameters:	$M = 2000$, $G = 50$.
Success Pred.:	Ranking of modules obtained by best-of-generation exactly equals that based on actual faults.
ADFs?:	No

and natural logarithm (log) are modified to protected against invalid inputs. The terminal set, $\mathcal{T}$, consists of the available software product and process metric variables (the independent variables of the data sets) and the ephemeral random constant generator function, $\Re$, taken from Koza(Koza 1992).

Fitness function. ***Raw fitness*** of individual i is defined as the *type II* error rate of individual i.

$$f_{adj}(i) = 1 - typeII_i(p) \qquad (5)$$

where p, the "training cut-off", is a cut-off percentage value. We made 17 runs for each of three different values of p: $\{ p_{f1} = 7.4\%, p_{f2} = 30\%, p_{f3} = 50\%\}$.

We intend to optimal the results at specific points of cutoff, in order to discover the impact to prediction results.

Run termination criterion. A run is terminated if an individual i with $f_{adj}(i) = 1$ is encountered, a "perfect" solution to the problem, or when the maximum number of generations is reached. The maximum number of generations is 30.

Table 1 is the Koza Tableau for the GP runs, listing the system parameters used.

Results of the Case Study

We completed 17 runs for each of the three different training cut-offs The runs were executed on a Sparc 20, using a modified version of Zongker *et al*'s Lil-gp system. Because of the size of the observation set and the number of variables involved, each run consumed over 6 hours of CPU time.

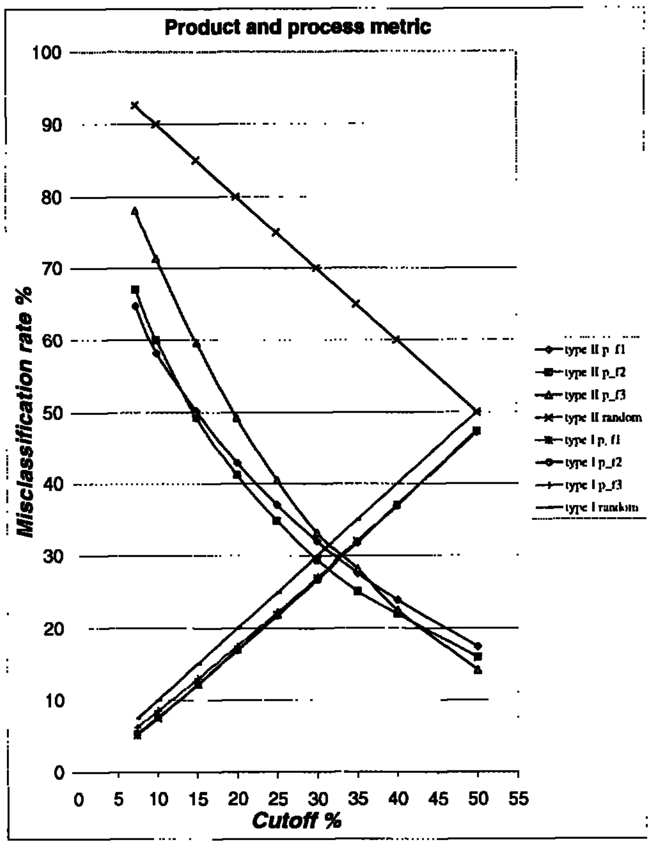

Figure 1: Misclassification rate for product and process metric.

Each best-of-run was applied to the validation set of observations to achieve a ranking of those modules. We then calculated the Type I and Type II error rates for each of the cut-offs in C. Figure 1 plots the results, the misclassification rates obtained at each percentile level, c, calculated over the validation data set. There are four pairs of curves in the graph, each pair showing the Type I and Type II error rates for each of the fitness functions (there are three, one for each training point), and one showing the error rates that would be expected if modules were chosen randomly for reliability enhancement (i.e., if the cut-offs were applied to random order rankings of the modules.) Each data point was the average of 17 values, and the standard deviation for the data points varied from 0.24% to 7.66%.

Comments

The results of our case study showed that the GP-generated software quality models far outperformed random reliability testing. This is not particularly surprising, but the degree of superiority is gratifying nonetheless.

The different values of the training cut-off, p, in the fitness function did not make any significant changes in type I error rate. However, the type II error rate did seem to be affected. The figures show that the predictive performance of the GP generated models were greatest at those cut-off percentages that were similar to the training cut-off. The p_{f1} (a 7.4% cut-off) trained GP models achieved the lowest Type II error rates at the 7.4% cut-off for the validation data. Likewise the p_{f2} and p_{f3} trained GP models achieved the lowest Type II error rates at the 30% and 50% cut-offs. To achieve optimum performance from the system, then, it should be trained at a pre-set training cut-off.

In using this system, a software manager would obtain a graph much like Figure 1 using different training cut-offs. The manager can then look at the error rates obtained by the various cut-off values, and use those rates to determine which cut-off value is obtains error rates that optimize the cost of reliability analysis, but which are within the financial and man-power resource constraints.

The manager can then execute an additional set of GP runs, using training cut-offs clumped closely around the desired cut-off value. Our experiments show that the resulting GP generated model should be quite accurate, enjoying the lowest type II error rate of all the models evaluated in the process on the validation or novel data.

Lastly, the addition of process metrics did yield models with greater accuracy. The type II errors rates of the models using both process and product metrics were generally superior to those models generated using only product metrics.

Conclusion

This paper, in combination with our related previous work (Evett *et al.* 1998) has demonstrated the effectiveness of GP to generate accurate software quality models in real-world, industrial domains. Because of the authors expertise in the field of software quality management, we are confident that our system's performance compares very well with other methods. Indeed, our current work involves the integration of this GP system into the existing EMERALD industrial software management system. Our future, however, work will concentrate on completing formal comparative studies between our GP-based system, and mature industry techniques already embraced by the software engineering community, such as discriminant analysis, and various regression tools. Such results will further bolster GP's claim to be a significant, real-world tool.

Acknowledgements

This work was supported in part by a grant from Nortel. The findings and opinions in this study belong solely to the authors, and are not necessarily those of the sponsor.

References

Evett, M.; Khoshgoftar, T.; der Chien, P.; and Allen, E. 1998. Gp-based software quality prediction. In Koza, J.; Banzhaf, W.; Chellapilla, K.; Kalyanmoym, D.; Dorigo, M.; Fogel, D. B.; Garzon, M. H.; Goldberg, D. E.; Iba, H.; and Riolo, R., eds., *Genetic Programming 1998: Proceedings of the Third Annual Conference.* San Francisco, CA: University of Wisconsin, Madison, Wisconsin.

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

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

Hudepohl, J. P. 1990. Measurement of software service quality for large telecommunications systems. *IEEE Journal of Selected Areas in Communications* 8(2):210–218.

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

Koza, J. 1992. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press.

Mayrand, J., and Coallier, F. 1996. System acquisition based on software product assessment. In *Proceedings of the Eighteenth International Conference on Software Engineering*, 210–219. Berlin: IEEE Computer Society.