REGRESSION TREE MODELING FOR THE PREDICTION OF SOFTWARE QUALITY

Swapna S. Gokhale*
Center for Advanced Computing and Communication
Dept. of Electrical and Computer Engineering
Box 90291, Duke University
Durham, NC 27708-0291
Email : ssg@ee.duke.edu

Michael R. Lyu
Room No. 2A-413, Lucent Technologies
Bell Laboratories
600, Mountain Avenue
Murray Hill, NJ 07974-0636
Email: lyu@research.bell-labs.com

Keywords: Software quality prediction, metrics, regression, modeling, risk analysis.

Abstract

This paper demonstrates the use of regression tree models to predict the number of faults in a software module based on the software complexity metrics, prior to the testing phase, which can help in channeling the validation and testing efforts in a productive direction. We also compare the regression tree modeling technique with the fault density technique which is a very commonly used approach to predict the number of faults.

1. Introduction

The production of modern computer software is one of the most complex and unpredictable activities in the industry. Software is a crucial part of many critical applications and certifying software integrity is absolutely essential. It is no longer acceptable to postpone the assurance of software quality until prior to a product's release. Delaying corrections until testing and operational phases can lead to higher costs, and it may be too late to improve the software product significantly.

A software failure is a manifestation of a fault, which is a defect in the executable software product. It is an old programming saw that 80% of the faults are found in 20% of the software code, i.e., a relatively small number of modules in a set of modules constituting a software product contain a disproportionate number of errors. A major portion of the effort expended in developing computer software is associated with program testing. The 80-20 rule described above suggests that applying equal testing and verification effort to all the modules of a software product is not very efficient. Scheduling and resource constraints often require testing to be conducted in such a manner that maximum number of faults are revealed in a minimum amount of time. Recent research in the field of computer program reliability has been directed towards the identification of software modules that are likely to be fault-prone, based on product and/or process-related metrics, prior to the testing phase. Software metrics represent quantitative description of program attributes and have been shown to be very closely related to the distribution of faults in the program modules[4, 5, 11]. Early identification of fault-prone modules in the life-cycle can help in channeling the program testing and verification efforts in the productive direction.

Several different techniques have sought to develop a predictive relationship between software metrics and the classification of the module into fault-prone and non fault-prone categories. These techniques include discriminant analysis[1, 10], classification trees[12], pattern recognition (Optimal Set Reduction (OSR))[2, 1] and neural networks[8]. Historical data from the past software development and maintenance scenarios are used to develop these predictive models, and then these quantitative models are used to provide insight and control into managing similar software projects, by identifying the troublesome modules earlier in the life-cycle of the software product. Most of these techniques however are classification models and they partition the modules into two categories viz. fault-prone and not fault-prone. Regression models have also been used in the context of software engineering[13, 14], though not very extensively. In this paper we explore *regression tree models* to predict the number of errors in a software module using the software metrics. The regression tree model

*This work was done while the first author was a summer intern at Bell Labs., Lucent Technologies provides for a finer classification than that provided by the classification models. We compare the predictions of the regression tree model with that of the fault density model which is one of the most commonly used techniques to predict the number of faults in a software product.

The rest of the paper is organized as follows: In Section 2 we describe the tree modeling procedure. Section 3 discusses the application of the algorithm described in Section 2, Section 4 compares the predictive performance of the tree modeling and the fault density procedures, and Section 5 presents conclusions based on the analysis of data.

1 2. Tree-based Modeling.

The use of tree-based models is an attractive way to encapsulate the knowledge of the experts and to aid in decision making. It is an exploratory technique used to uncover structure in the data, and it provides an alternative to linear and additive models for regression problems and to linear logistic and additive logistic models for classification problems. It originated from the search for alternative techniques to classical statistical models like linear regression which are highly unstable in the event of correlation between the variables, and hence were unsuitable to analyze the data in social sciences, where data are often characterized by complicated and unexplained irregularities[3].

2.1 Tree construction algorithm

The tree modeling approach is a goal oriented statistical technique which consists of recursive partitioning of the variable space using binary splits. The *dependent variable* or the *response variable* (usually denoted by y) in this context consists of the number of faults in a software module and the set of *classification*, *predictor* or *independent variables* (usually denoted by x) consists of the various software complexity metrics for the module. The algorithm attempts to partition the predictor variable space into homogeneous regions such that within each region the distribution of the response variable conditional to the predictor variables $f(y|x)$, is independent of the predictor variables (x)[15].

At each step, the tree-construction algorithm searches through all possible binary splits of all the predictor variables until the *overall deviance*, i.e., the sum of the deviances for each subset is minimized. The algorithm then begins the search again for the next binary split, reconsidering all the variables until the next binary split is made, and so on. Thus the tree-construction method uses a one-step lookahead, i.e., it chooses the next split by minimizing the deviance for that split, without making an effort to optimize the performance of the entire tree which is an NP-complete problem.

This can be viewed as estimating a step function $\tau(x)$, which is related to a primary parameter in the conditional distribution of $y|x$. The likelihood function provides the basis for choosing partitions. Deviance (likelihood ratio statistic) is used to determine which partition of a node is "most likely" given the data.

The model used for regression trees is based on the Gaussian distribution consisting of the following stochastic component

$$y_i \sim N(\mu_i, \sigma^2) \tag{1}$$

and the structural component

$$\mu_i = \tau(x_i) \tag{2}$$

The *deviance function* for an observation is defined as

$$D(\mu_i; y_i) = (y_i - \mu_i)^2 \tag{3}$$

which is minus twice the log-likelihood scaled by a factor of σ^2, where σ^2 is assumed to be a constant for all i.

The mean parameter μ is a constant for all the observations at a given node. The minimum-deviance estimate of μ, denoted by $\hat{\mu}$, is given by the average of all observations in that node.

The deviance of a node is defined as the sum of the deviances of all observations that belong to that node and is given by

$$D(\hat{\mu}, y) = \sum_j D(\hat{\mu}, y_j) \tag{4}$$

where j is the number of observations in the node. If all the y's belonging to a node are the same, (the node is known as a pure node in this case), then the deviance is identically equal to zero. The deviance increases from this minimum value as y's deviate from the ideal. In order to reduce the deviance of a given node, a binary split partitions the node into two children nodes. The deviance of a node is then compared to the deviance of the candidate children nodes, that allow for different means in the left and the right nodes. The deviance of the candidate nodes is given by

$$D(\hat{\mu_L}, \hat{\mu_R}; y) = \sum_L D(\hat{\mu_L}; y_j) + \sum_R D(\hat{\mu_R}; y_j) \tag{5}$$

where $\hat{\mu_L}$ is mean of the left node, and $\hat{\mu_R}$ is mean of the right node.

The change of deviance between the parent node and the children nodes is given by

$$\Delta D = D(\hat{\mu}, y) - D(\hat{\mu_L}, \hat{\mu_R}; y) \qquad (6)$$

The split that maximizes the change of deviance, called as the goodness-of-fit, is the one that is chosen at a given node.

Thus the nodes become more and more homogeneous as the splitting progresses. In the limiting case, the tree can have as many number of terminal nodes as the observations. In reality, this is far too many and the tree construction process is normally terminated based on some stopping rule. The stopping rule controls the granularity of the tree model and usually depends on the number of cases reaching each leaf. Tree construction can be stopped based on the cardinality threshold T_c, i.e., leaf node is smaller than some absolute minimum size, or by determining whether the leaf is homogeneous enough i.e., the deviance is less than the some small percentage (about 1%) of the deviance of the root node, also known as the homogeneity threshold T_h[13, 14]. The size of the tree is defined as the number of terminal nodes in the tree.

Intuitively, the algorithm uses a learning set of data to construct a regression tree which is used as a predicting device. Each terminal node in the tree represents a partition or a subset of the data that is homogeneous with respect to the dependent variable. The predicted value of the dependent variable is the average of all the observations in the node. In the present context, the tree-modeling procedure attempts to identify the modules with the same number of errors, and thus have the same degree of fault-proneness.

2.2 Pruning

The stopping rules described above tend to grow a tree that is too elaborate and has overfit the learning data set to a certain degree. *Pruning* provides the equivalent of variable selection in linear regression, and determines a nested sequence of subtrees of the given tree by recursively snipping off the least important splits, where the importance is given by the cost-complexity measure as follows:

$$R_\alpha(T') = R(T) + \alpha size(T') \qquad (7)$$

where $R_\alpha(T')$ is the deviance of the subtree T' for the cost-complexity parameter α, and $size(T')$ is the number of terminal nodes in the tree T'. Cost-complexity pruning determines the subtree T' that minimizes the deviance $R_\alpha(T')$ over all subtrees of the tree T for a given α.

The cost-complexity pruning generally prunes partitions with minimal gains in deviance reduction, since it is assumed that these partitions add little information or insight, so that the more significant partitions can be highlighted, which enables the understanding of the relationships between the response and predictor variables. However, in some cases, partitions which may not contribute heavily to the reduction in the overall deviance may have some practical significance and add insight based on semantic, contextual or environmental information. Pruning of partitions thus should be done indiscriminately.

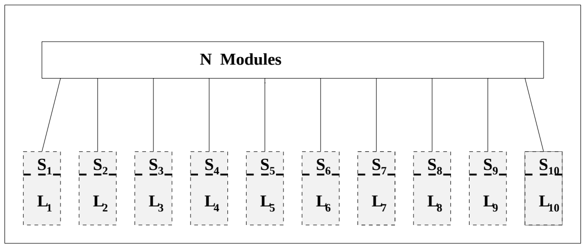

Figure 1: Schematic of 10-fold Cross Validation

2.3 Cross-Validation

Cross-validation is used to evaluate the predictive performance of the regression tree and also helps to determine the degree of pruning in the absence of a separate validation data set. The idea of cross validation is that the data are divided into two mutually exclusive sets, viz., learning sample and test sample. The learning sample is used to grow the tree and then the test sample is used to evaluate the tree sequence. Deviance is used as the measure to assess the performance of the prediction rule in predicting the number of errors for the test sample for different tree sizes. To reduce the sampling error in constructing the learning and the test samples 10-fold cross validation is used. In case of 10-fold cross validation, the training set is split into 10 equally sized parts, 9 of which are used to grow the tree and then the tree is evaluated using the 10th set. The learning sample which consists of 9 sets can be constructed in 10 different ways. Figure 1 shows the schematic of 10-fold cross validation.

In Figure 1 S_i denotes the test sample i and L_i denotes the corresponding learning sample. Let $N(S_i)$ denote the number of modules in the test sample S_i, and $N(L_i)$ denote the number of modules in the learning sample L_i. If N is the total number of modules in the data set, we have

$$N(S_i) = \lfloor \frac{N}{10} \rfloor \quad \& \quad N(L_i) = N - \lfloor \frac{N}{10} \rfloor \; i = 1, \ldots, 10 \qquad (8)$$

The deviance for 10 fold cross validation denoted by D_T is given by

$$D_T = \sum_{i=1}^{10} D_i \quad \& \quad D_i = \sum_{j=1}^{\lfloor \frac{N}{10} \rfloor} (y_{ij} - \hat{y_{ij}})^2 \qquad (9)$$

The deviance obtained from cross validation is plotted against the number of terminal nodes in the tree (tree size), and the number of nodes for which the deviance is minimum is determined. A tree of size suggested by cross-validation can then be grown and used as a predicting device without a significant loss in performance.

3. Data Analysis

The data used for an application of tree modeling represents the results of an investigation of software for a Medical Imaging System (MIS). The total system consisted of about 4500 modules amounting to about 400,000 lines of code written in Pascal, FORTRAN, assembler and PL/M. A random sample of about 390 modules, from the ones written in Pascal and FORTRAN was selected for analysis. These 390 modules consisted of approximately 40,000 lines of code. The software had been developed over a period of five years, and had been in commercial use at several hundred sites for a period of three years[8].

The number of changes made to the executable code of the module, documented by Change Reports (CRs) is an indicator of the software development effort. There is a one-to-one correspondence between the CRs and the number of faults found. In addition to the change data, the following 11 software complexity metrics were developed for each of the modules:

- Total lines of code (TC)
- Number of code lines (CL)
- Number of characters (Cr)
- Number of comments (Cm)
- Comment characters (CC)
- Code characters (Co)
- Halstead's Program Length (N), where $N = N_1 + N_2$ and N_1 represents a total operator count and N_2 represents a total operand count[6].
- Halstead's Estimate of Program Length Metric (Ne), where $Ne = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$, and η_1 and η_2 represent the unique operator and the operand count respectively[6].
- Jensen's Estimate of Program Length Metric (JE), where $JE = \log_2 \eta_1! + \log_2 \eta_2!$ [7]
- McCabe's Cyclomatic Complexity Metric (M), where $M = e - n + 2$, and e is the number of edges in a control flow graph representation of a program with n nodes[9]
- Belady's Bandwidth Metric (BW), where

$$BW = \frac{1}{n} \sum_i i L_i \qquad (10)$$

and L_i represents the number of nodes at level i in a nested control flowgraph of n nodes[7]. This metric indicates the average level of nesting or width of the control flowgraph representation of the program.

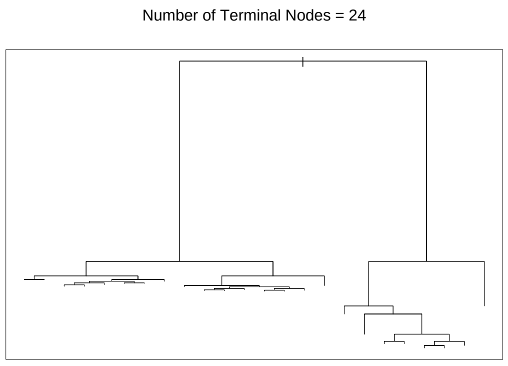

Figure 2: Dendrogram for 390 programs

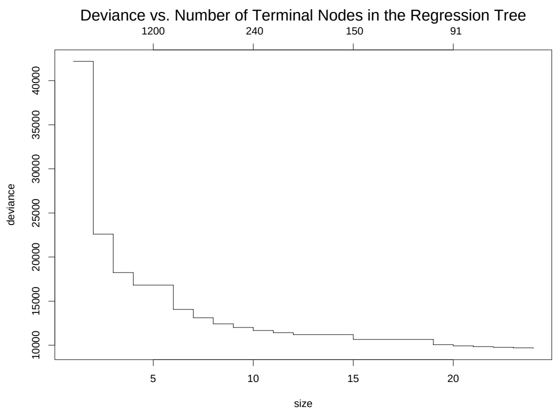

Figure 3: Reduction in deviance with number of terminal nodes

Figure 2 shows the dendrogram created when the tree construction algorithm was applied to the 390 programs. Binary splitting on the variables continues till the number of terminal nodes in the tree is 24. The lengths of the branches is proportional to the reduction in the overall deviance, thus the first binary split contributes to the greatest reduction in the overall deviance and the reduction in the overall deviance reduces for the subsequent splits.

Figure 3 shows the reduction in overall deviance with sequential splitting. The deviance for the root node (i.e., before any splits are made) is 42,200, and it decreases as the complexity of the tree increases.

Figure 4 shows the results of cross validation. A test sample of 39 modules was randomly selected from the 390 modules, and the remaining 351 modules were used to construct the tree. The deviance of 39 programs was then obtained for trees of different sizes. This process was repeated 10 times with a different set

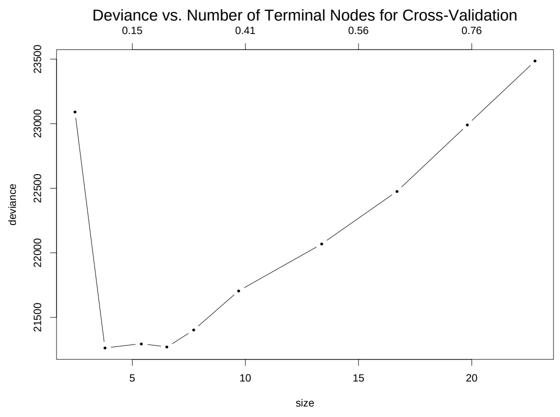

Figure 4: Results of Cross-validation

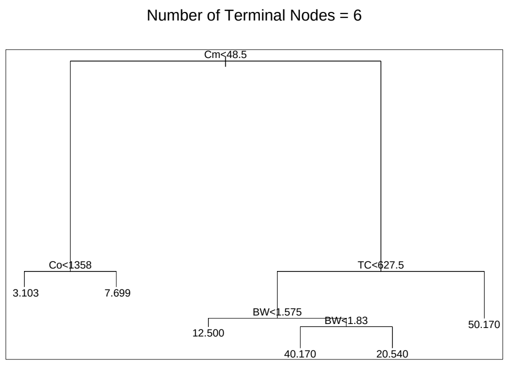

Figure 5: Pruned Tree with six terminal nodes

of 39 programs each time, and deviances corresponding to the same number of nodes is accumulated. The figure shows that the minimum deviance is obtained for a tree size of 6 nodes.

The original tree was snipped down to 6 nodes as suggested by cross validation and is shown in Figure 5. The first split occurs on the number of comments: those modules with the number of comments less than 48.5 and those with the number of comments greater than 48.5. For the left child of the root node, the next split occurs on the number of code characters: the modules with number of code characters less than 1358 and the ones with number of code characters more than 1358. The splitting process continues, visiting all variables each time a split is made until all the modules are divided into six partitions with predicted CRs (3.103, 7.699, 12.50, 20.540, 40.170, 50.170) based on only 4 variables, viz., Number of Comments, Number of Code Characters, Total Lines of Code and Belady's bandwidth metric, with a standard deviation of 3.77, 6.21, 7.52, 9.85, 2.59 and 21.8 respectively.

4. Comparative Performance of Tree Modeling and Fault Density.

A common practice today is to predict the number of CRs based on *fault density*, which is defined as follows:

$$FD = \frac{TCR}{TL} \tag{11}$$

where FD is the fault density of the software system, TCR is the total number of CRs in the software system and TL is the total number of lines in the software system.

The fault density for the Medical Imaging System was computed to be 0.0539464. The number of CRs predicted for a module i is given by

$$CR_i = NL_i * FD \tag{12}$$

where CR_i is the predicted number of Change Reports in module i, and NL_i is the number of lines in module i.

There are two types of errors that can be made in the prediction. A Type I error is when more faults are predicted than were actually observed. A Type II error is when fewer faults are predicted than were actually observed. Of the two types of errors, Type II error has more serious implications, since a product would seem better than it actually is, and testing effort would not be directed where it would be needed most.

The minimum of the observed CRs is 0 and the maximum is 98. Because of the large variation in the number of observed CRs, the modules are classified into 10 bins depending upon the number of CRs attributed to them. Thus modules with CRs in the range of 0-9 belong to bin 1, 10-19 belong to bin 2 and so on. Every module belongs to two bins, one corresponding to the observed number of CRs and the other corresponding to the predicted number of CRs. A Type I misclassification occurs when a module actually belongs to bin k, when it is predicted to belong to bin j, where $j > k$. Type II misclassification occurs when a module belongs to bin k and is predicted to belong to bin j, where $j < k$.

The overall deviance obtained by the tree modeling procedure is 9660.35, whereas the overall deviance obtained by the fault density approach is 19933.6. Type I misclassification rate for the tree modeling procedure is 6.16% and for the fault density procedure is about 7.5%. Type II misclassification rate for the tree modeling procedure is about 8.7% and for the fault density procedure is about 13.1% The overall misclassification rate is 14.86% and 20.6% for the tree modeling and fault density techniques respectively. These results indicate that the tree modeling approach gives considerable improvement over the fault density approach.

The pruned tree approach with six terminal nodes, as suggested by cross-validation was then used as a predicting device. The Type II misclassification rate in this case is 12.04% and the deviance is 14058.34. This result also shows an improvement over the fault density approach.

The regression tree approach can also be used to classify the modules into fault-prone and non fault-prone categories. A decision rule can be established which classifies the module as fault-prone if the predicted number of faults is greater than a certain number a. The choice of a determines the misclassification rate. Modules with very small number of changes are clearly non fault-prone, where as modules with relatively large number of changes are clearly fault-prone. The same data set has been analyzed using discriminant analysis[10] with a view to identifying fault-prone and non fault-prone programs. In this analysis, the modules with number of CRs in the range of 0-9 have been discarded. Note that in case of the regression tree modeling it is not necessary to discard the modules with an intermediate number of changes.

5. Conclusions

There is a lot of information to be extracted from software metrics, and the regression tree modeling is an effective way to analyze data, to understand the involved relationships among data attributes, to identify the troublesome modules, and thus take remedial actions before it is too late. The results presented in the tree-form are intuitive, aid decision making and are easy to use. Interesting subsets of modules can be identified along with their characteristics by following the path from the subset to the root of the tree. The technique is fairly robust to the presence of outliers, is stable with highly uncorrelated data, and can handle missing values. Thus it provides an effective way to predict software quality. This technique also enjoys lower misclassification rate and deviance as compared to the commonly used fault density procedure to predict the number of faults in software modules.

References

[1] L. C. Briand, V. R. Basili, and C. Hetmanski. "Developing Interpretable Models for Optimized Set Reduction for Identifying High-Risk Software Components". *IEEE Trans. on Software Engineering*, SE-19(11):1028–1034, November 1993.

[2] L. C. Briand, V. R. Basili, and W. M. Thomas. "A Pattern Recognition Approach for Software Engineering Data Analysis". *IEEE Trans. on Software Engineering*, SE-18(11):931–942, November 1992.

[3] L. A. Clark and D. Pergibon. *Statistical Models in S, J. M. Chambers and T. J. Hastie, Eds.*, chapter Tree-Based Models, pages 377–419. Chapman & Hall, New York, 1993.

[4] S. G. Crawford, A. A. McIntosh, and D. Pregibon. "An Analysis of Static Metrics and Faults in C Software". *J. Systems Software*, 15:37–48, 1985.

[5] J. Gaffney. "Estimating the Number of Faults in Code". *IEEE Trans. on Software Engineering*, SE-10(4):459–464, July 1984.

[6] M. Halstead. *"Elements of Software Science"*. New York Elsevier, North-Holland, 1977.

[7] H. Jensen and K. Vairavan. "An Experimental Study of Software Metrics for Real-Time Software". *IEEE Trans. on Software Engineering*, SE-11(2):231–234, February 1994.

[8] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. "A Comparitive Study of Pattern Recognition Techniques for Quality Evaluation of Telecommunications Software". *IEEE J. Selected Areas in Communication*, 12(2):279–291, February 1994.

[9] T. J. McCabe. "A Complexity Measure". *IEEE Trans. on Software Engineering*, SE-2:308–320, 1976.

[10] J. Munson and T. Khoshgoftaar. "The Detection of Fault-Prone Programs". *IEEE Trans. on Software Engineering*, SE-18(5), May 1992.

[11] A. J. Perlis, F. G. Sayward, and M. Shaw. *"Software Metrics: An Analysis and Evaluation"*. MIT Press, Cambridge, MA, 1981.

[12] A. A. Porter and R. W. Selby. "Empirically Guided Software Development Using Metric-Based Classification Trees". *IEEE Software*, pages 46–54, March 1990.

[13] J. Tian and J. Henshaw. "Tree-Based Defect Analysis in Testing". In *Proc. 4th Intl. Conf. on Software Quality*, McClean, Virginia, 1994.

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

[15] W. N. Venables and B. D. Ripley. *"Modern Applied Statistics with S-Plus"*. Springer-Verlag, New York, 1994.