A Connectionist Approach for Incorporating Continuous Code Churn into Software Reliability Growth Models

N. Karunanithi

Room 2E-378, Bellcore
445, South Street, Morristown, NJ 07960
karun@faline.bellcore.com

Abstract

A large number of execution-time based reliability growth models have been proposed for estimating reliability of software systems. One of the key assumptions made in almost all of the models is that the complete code for the system is available before testing starts and that the code remains frozen during testing. However, this assumption is often violated in large software projects because usually the code is developed in parts. This paper demonstrates the applicability of the neural network approach to the problem of developing an extended software reliability growth model in the face of continuous code churn. In this preliminary study, neural network reliability models with and without the code churn information are compared using a data set from a large telecommunication system. The results suggest that the neural network model with the code churn information is capable of providing a more accurate prediction of future faults than the model without the code churn information.

1 Introduction

A large number of dynamic reliability models for have been proposed for estimating reliability growth of software systems. These analytic models describe the failure process as a function of execution time (or calendar time) and are based on many simplifying assumptions [8]. One of the key assumptions made by many of the existing analytic models is that the code size remains unchanged during testing. This assumption may not be valid for most large software systems because the program undergoes change during development.

Evolving programs can introduce a variety of complications for both software engineers and the software reliability estimate. For software engineers, an evolving program becomes a moving target during testing. Furthermore, the testing team cannot have uniform confidance on all parts of an evolving program because the code which is included early on would be exercised more often than the parts that are included later. From software reliability estimation point of view, the program evolution introduces complications in efforts such as data collection, modeling, analysis and interpretation. Thus, in the face of program evolution the existing execution time based software reliability growth models may not be able to provide a trustworthy estimate without proper extensions.

In order to deal with a continuously evolving code (or "code churn") Dalal and McIntosh [2] proposed a simple extension to a Poisson process model and empirically demonstrated that adding the code churn information can improve the fit of the model. This paper demonstrates the applicability of the neural network approach to the problem of modeling software reliability growth in the face of continuous code churn. In this preliminary empirical study, neural network reliability models with and without the code churn information are compared using a data set from a large telecommunication system. The results suggest that the neural network model with the code churn information is capable of providing a more accurate prediction of future faults than the model without the code churn information.

2 The Data Set

The data set used for this study is from a large telecommunication system consisting of approximately 7 million non-commentary source lines [2]. The particular release for which the reliability model is applied had around 400,000 new or changed non-commentary source lines (NCNCSL). Since the testing was performed in a highly distributed environment the time was measured in "staff days". The "staff day" metric represents the amount of time a tester actually spends every day on testing the current release. Figure 1 illustrates the continuous code churn as a function of staff days. The vertical (dashed) line indicates the date on which the system was frozen. After the "soft freeze", no functionality was added to the system and subsequent changes reflect only fault fixes. Though several subsystems were delivered at the start of the testing, only a subset of them were actually tested. This

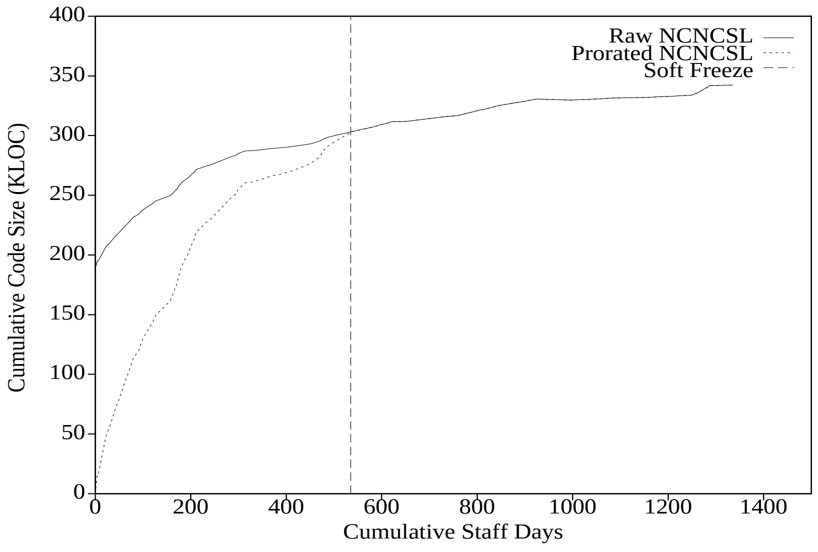

Figure 1: Code Churn Vs. Cumulative Staff Days.

necessitated a proper adjustment to the initial NCNCSL. The dotted line in Figure 1 shows the adjusted code size.

3 Model Development

3.1 Dalal and McIntosh Model

The basic model of Dalal and Mallows [1] on which the extended model is built is given by

$$\mu(t_i) = \alpha_i(1 - e^{-\beta(t_i - t_{i-1})})$$

where $\mu(t_i)$ is the mean number of faults found at time t_i, α_i is the number of faults at the beginning of the i-th interval and β is the rate parameter. The number of remaining faults at time t_i is equal to $\alpha_i e^{-\beta(t_i - t_{i-1})}$. The extended model which incorporates code churn is given by,

$$\alpha_{i+1} = g(\alpha_i e^{-\beta(t_i - t_{i-1})}, c_i, \theta) \qquad i = 1, \dots, n$$

where $g(x, c, \theta)$ is a general (yet unspecified) function and c_i is the size of the code added at the end of i-th interval. If there was no code added in an interval then the corresponding $c_i = 0$. Typically, the extended model can also be expressed as

$$g(x, c, \theta) = x + g_1(c, \theta) + g_2(x, c, \theta)$$

with $g_1 = g_2 = 0$ whenever $c = 0$. This formulation can be interpreted as follows.

The number of faults in the code after the code churn =
{ the number of faults immediately before the code churn } +
{ the number of faults in newly delivered code } +
{ the number of faults in the code because of the interactions between the two sets of faults}.

As a first step approximation, it is assumed that the function $g_2(x, c, \theta) = 0$. This is equivalent to saying that there is no additional faults due to interaction of the existing code and the newly added code. Next, they assume that the structure of g_1 is an identity function and that c is equal to NCNCSL at i-th delivery (i.e., $g_1(c, \theta) = \theta g_1(c)$). Finally, they simplify the model by assuming that the additional faults in the newly added code is proportional the size of the code. Thus, the resulting model is given by,

$$g(\alpha_i e^{-\beta(t_i - t_{i-1})}, c_i, \theta) = \alpha_{i+1} = \alpha_i e^{-\beta(t_i - t_{i-1})} + \theta c_i$$

where α_{i+1} is the number of faults in the system at the beginning of $(i+1)$st interval. The corresponding expression for the cumulative fault, $M(t_i)$, is given by,

$$M(t_i) = M(t_{i-1}) + \alpha_i(1 - e^{-\beta(t_i - t_{i-1})})$$

where $M(t_{i-1})$ is the cumulative faults found at the end of the $(i-1)$st interval. Note that the above expression has the structure of an autoregressive process. One of the important claims made by Dalal et. al., [1] is that the fit of the extended model is considerably superior than that of the basic model.

3.2 The Neural Network Approach

Applicability of neural network models to software reliability growth prediction has been demonstrated by Karunanithi et al. [5, 6, 7]. Two important conclusions of these studies are: i) the neural network approach is a "black box" approach (i.e., the neural networks are capable of developing an appropriate model for the failure process from the training data) and ii) the modified Jordan style [4] networks with "Teacher Forced" training are capable of providing a more accurate predictions than the feed-forward networks.

The network models used in the previous research[5, 6, 7] had the accumulated time (t_i) as the free variable and the cumulative faults ($M(t_i)$) as the dependent variable. From the modeling point, they are analogous to the traditional execution time based software reliability growth models. However, the networks used in the present study have an additional input

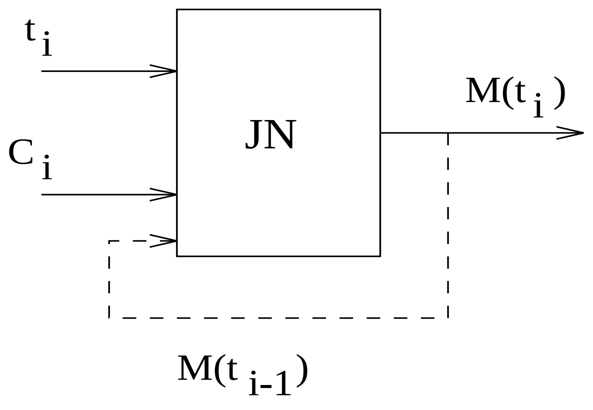

Figure 2: A Modified Jordan Style Neural Network.

(C_i) for the cumulative code churn. Figure 2 represents an abstract model of the modified Jordan style network used in this study. The dotted line in Figure 2 represents the feed-back from the output ($M(t_{i-1})$) required for realizing the Jordan style network. For simplicity, it is assumed that the network operates only in discrete time-steps. The "box" (JN) represents the neural network.

4 Prediction Results

Even if the fit of a model is good, that does not guarantee that its predictions will always be accurate. A good model also should provide accurate predictions of future faults. In order to assess the predictive capability of the model we used two extreme prediction horizons: the *next-step prediction* (NSP), for the cumulative faults at the end of the next time step; and the *end-point prediction* (EPP), for the cumulative faults at the end of the test phase. When an extended model like the one developed in this paper is used for predicting future events, one has to know not only the execution time corresponding to a future day but also a precise information about the code churn. We solved this issue by using the size of the code corresponding to the final value of the code churn in the training data to all future predictions (i.e., the value of the code size of the last point in the training data was considered as the size of the final code). Note that this issue does not arise if we do not use the code churn information.

The neural networks cannot predict well without sufficient training data. This is analogous to using insufficient data to estimate the parameters of a stochastic model. In our prediction experiment, the size of the training set was gradually increased from a minimal set consisting of all data points before the "soft freeze" up to a set with all but the last point in the failure history. In order to gauge the predictive quality of the neural network model we used the average relative error (ARE) used by Malaiya et al. [9]. The average relative error measure is defined as

$$ARE = \frac{1}{n-k-1} \sum_{i=k+1}^{n-1} \left| \frac{M(t_i) - \hat{M}(t_i)}{M(t_i)} \right|$$

where n is the number of points in data set, k is the number of points in the training set corresponding to the soft freeze and $n-k-1$ is the prediction window. ARE provides a summary of how well the model predicts across a window of the future failure history. The predictive performance of the neural network models for the two extreme prediction horizons are summarized in Table 1. The "Mean" and "SD" represent the mean and the standard deviation of the ARE measure over 50 trials. These results suggest that the network with the code churn information is capable of providing a more accurate prediction than the network without the code churn information.

Neural Network	NSP		EPP	
Model	Mean	SD	Mean	SD
Without Code Churn	3.834	1.883	26.342	8.417
With Code Churn	2.955	1.803	17.27	5.837

Table 1: A Summary of Prediction Results in Terms of ARE.

5 Conclusion

We demonstrated that one can easily extend the execution time based neural network framework to incorporate the code churn information. Our preliminary results suggest that incorporating the code churn information can help both the fit as well as the predictive accuracy of the neural network models. However, we do note that the results presented here are preliminary and they have yet to be compared with the results of the existing models. This will be explored in future.

Often, one is interested in not only predicting the cumulative faults but also other quantities such as the rate of occurrence of failure, mean time to failure etc. The extended neural network model developed in this study represent only the cumulative faults. However, if one is interested in the failure rate expression, it is straightforward to derive it from the expression for the cumulative faults using the method outlined in [7].

References

[1] Dalal, S. R. and Mallows, C. L., "Some Graphical Aids for Deciding When to Stop Testing Software", *IEEE J. Selected Areas in Communications.*, Vol. 8, No. 2, pp. 169-175, 1990.

[2] Dalal, S. R. and McIntosh, A. A., "Reliability Modeling and When to Stop Testing for Large Software Systems in the Presence of Code Churn: Analysis and Results for TIRKS Release 16.0", *Bellcore, TM-ARH-021705*, Aug. 1992.

[3] Fahlman, S. E. and Lebiere, C., "The Cascaded-Correlation Learning Architecture", School of Computer Science, Carnegie Mellone University, Tech. Rep. CMU-CS-90-100, Feb. 1990.

[4] Jordan, M. I., "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine", *Proc. of the 8th Annual Conf. of the Cog. Science*, 1986, pp. 531-546.

[5] Karunanithi, N., Whitley, D. and Malaiya, Y. K., "Prediction of Software Reliability Using Connectionist Models", *IEEE Trans. on Software Eng.*, Vol. 18, No. 7, pp. 563-574, July 1992.

[6] Karunanithi, N., Whitley, D. and Malaiya, Y. K., "Using Neural Networks in Reliability Prediction", *IEEE Software*, Vol. 9, No. 4, pp. 53-59, July 1992.

[7] Karunanithi, N. and Malaiya, Y. K., "The Scaling Problem in Neural Networks for Software Reliability Prediction", *Proc. 1992 Int. Symp. on Soft. Rel. Eng.*, pp. 76-82, Oct. 1992.

[8] Littlewood, B., "Theories of Software Reliability: How Good Are They and How Can They Be Improved?", *IEEE Trans. on Software Eng.*, Vol. SE-6, No. 5, pp. 489-500, Sep. 1980.

[9] Malaiya, Y. K., Karunanithi, N. and Verma, P., "Predictability of Software Reliability Models", *IEEE Trans. Reliability*, Vol. 41, No. 4, pp. 539-546, Dec. 1992.