A Neural Network Approach for Software Reliability Growth Modeling In the Presence of Code Churn

N. Karunanithi
Room: 2E-378, Bellcore
445 South Street, Morristown, NJ 07960
(201) 829-4466
Email: *karun@faline.bellcore.com*

Abstract

One of the key assumptions made in most of the time-domain based software reliability growth 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. Thus, the existing models may not be able to provide an accurate description of the failure process in the presence of code churn. Recently, Dalal and McIntosh [3] developed an extended stochastic model by incorporating continuous code churn into a standard Poisson process model and observed an improvement in the model's estimation accuracy. 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, a comparison is made between two neural network models, one with the code churn information and the other without the code churn information, for the accuracy of fit and the predictive quality using a data set from a large telecommunication system. The preliminary results suggest that the neural network model that incorporates the code churn information is capable of providing a more accurate prediction than the network without the code churn information.

Keywords: Software reliability growth modeling, Extended stochastic models, Neural network models, Continuous code churn, Complex models.

1 Introduction

There exists a large number of analytic software reliability growth models for estimating reliability growth of software systems. Existing analytic models describe the failure process as a function of execution time (or calendar time) and a set of unknown parameters. Some of the attractive features of the analytic models are that they are easy to analyze, interpret and make inferences. However, analytic models are based on many simplifying assumptions [9, 12]. For example, one of the key assumptions made in many of the existing analytic models (although this is often not stated explicitly) 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 (or evolves) during development. Programs can evolve due to either requirements changes or integration of parts during development. Requirement changes may occur due to enhancement of features, adaptation of the system to changing hardware and software, optimization of the system to realize performance improvement etc. Evolution of code may also occur during development for several reasons such as: parallel development of component systems by different groups, conducting tests in a step-by-step (or feature-by-feature) fashion, performing tests as and when subsystems are ready, and starting tests before the interfaces between subsystems are incomplete.

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, existing execution time based software reliability growth models may not be able to provide a

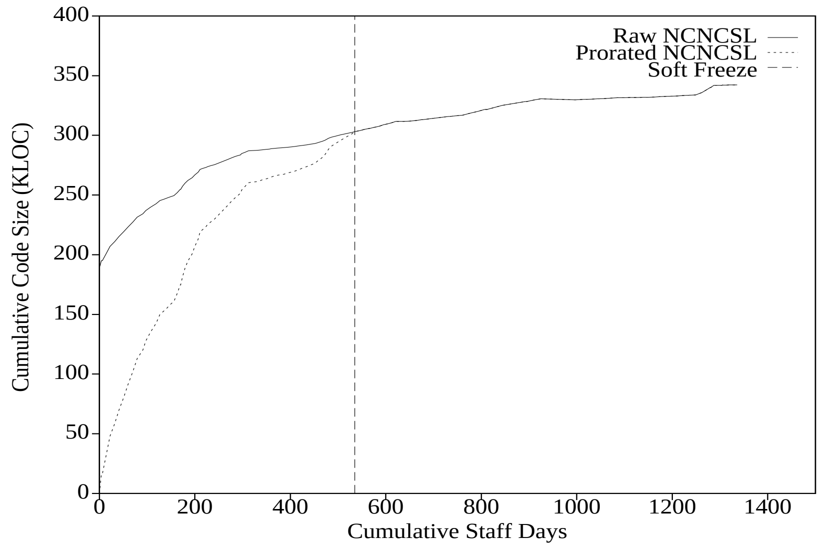

Figure 1: Code Churn Vs. Cumulative Staff Days.

trustworthy estimate without proper extensions.

To make a model more reliable one must incorporate relevant information about the evolutionary process of the code. Incorporating such additional information into a model can help the model to provide a more realistic picture of the failure process than a model that relies solely on the assumptions made by the model developer. Musa et al. [12] discuss three general approachs for tackling the issue of program evolution. However, all those approachs do not directly deal with continuous program evolution. Recently, Munson et al. [11] proposed an approach to incorporate the functional program complexity of the software into dynamic reliability growth models. The functional complexity of a system (under testing) represents the expected value of the relative complexity metrics of modules under a particular operational profile. However, this approach requires both a static analysis of the code as well as a precise definition of the operational profile of the software. In order to deal with a continuously evolving code (or "code churn") Dalal and McIntosh [3] proposed a simple extension to a Poisson process model proposed by Dalal and Mallows [1, 2]. (Their definition of "code churn" includes both new additions of code as well as changes in the existing code due to fault fixes.) In their approach, the size of the code churn was considered as one of the free variables of the model. They validated their extended model by applying it to a large telecommunication software project. Their empirical results clearly demonstrate 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. The idea is inspired by the work of Dalal and McIntosh [3]. In this preliminary study, a comparison is made between two neural network models, one with the code churn information and the other without the code churn information, for the accuracy of fit and the predictive quality using a data set from a large telecommunication system. The preliminary results suggest that the neural network model that incorporates the code churn information is capable of providing a more accurate prediction than the network without the code churn information. The rest of the paper is as follows. Section 2 describes the data set used in this study. Section 3 reviews the extended model developed by Dalal and McIntosh [3] and discusses the applicability of the neural network approach for developing extended software reliability growth models. Section 4 presents preliminary results from the neural network approach in terms of the fit of the models and their predictive quality. Section 5 concludes the paper with a summary of results and future extensions.

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 [3]. The particular release for which the reliability model

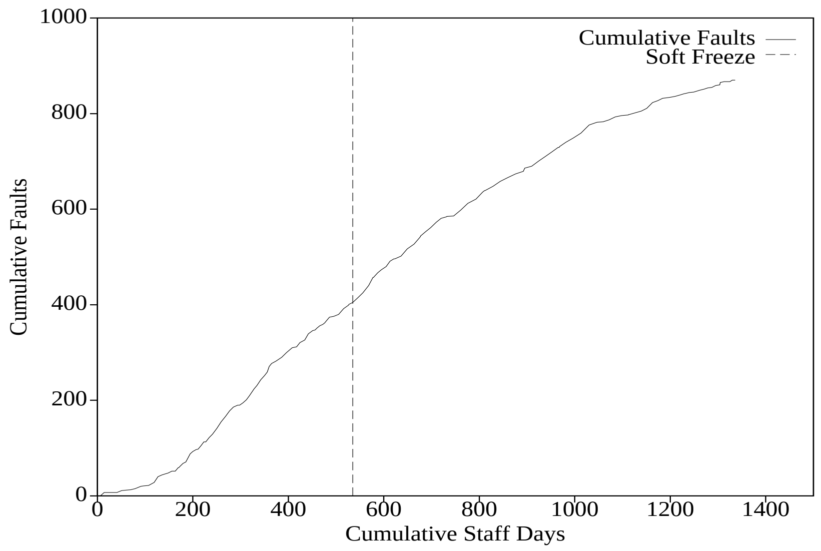

Figure 2: Cumulative Faults Vs. Cumulative Staff Days.

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. It should be noted that a staff day is not exactly the same as the calendar time. To collect the staff day data, the testing organization used a custom built login procedure to prompt the tester to enter the time information for the previous day when they logged on each morning. The code churn took place in two ways: first, the developers delivered code on daily basis; and second, testing in the early stages occurred by groups of subsystems. At the initial stages of testing the system contained only some of the modifications and additions of the code; not all of the subsystems were tested at once. As testing progressed, more additions and modifications were included. 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. To make the graph continuous, they used a simple linear interpolation for those days in which there was no code churn. Since they considered modifications in the code due to fault fixes also as part of the code churn, the system had a non-trivial code churn even after the soft freeze. Though several subsystems were delivered at the start of the testing, only a subset of them were actually tested. This necessitated a proper adjustment to the initial NCNCSL. So the initial NCNCSL was prorated across the NCNCSL received after the start of testing but before the soft freeze. The dotted line in Figure 1 shows the adjusted code size. Figure 2 illustrates the actual failure history of the system as a function of time.

In the model developed by Dalal and McIntosh [3] as well as the neural network approach examined in this paper, the code size is used as one of the free variables. Figure 3 illustrates the relationship between the cumulative faults and the cumulative code size. Note that nearly half of the faults were observed before the soft freeze and the amount of additional code added due to fault fixes constitutes about one sixth of the entire code.

3 Model Development

3.1 Dalal and McIntosh Model

If one assumes a continuous code churn and fits a basic model at each interval, the resulting cumulative fault vs. time curve will look like a waterfall (a cascade of exponentially decaying curves). This will also lead to an unmanageable number of parameters. In order to make the model computationally tractable, Dalal and McIntosh [3] make the following assumptions. Let the testing be divided into n intervals $(t_0, t_1), (t_1, t_2), \ldots, (t_{n-1}, t_n)$ and each code delivery corresponds to the start of a new interval. Let the number of faults N_i in each interval i be a Poisson random variable with mean α_i and that it satisfies

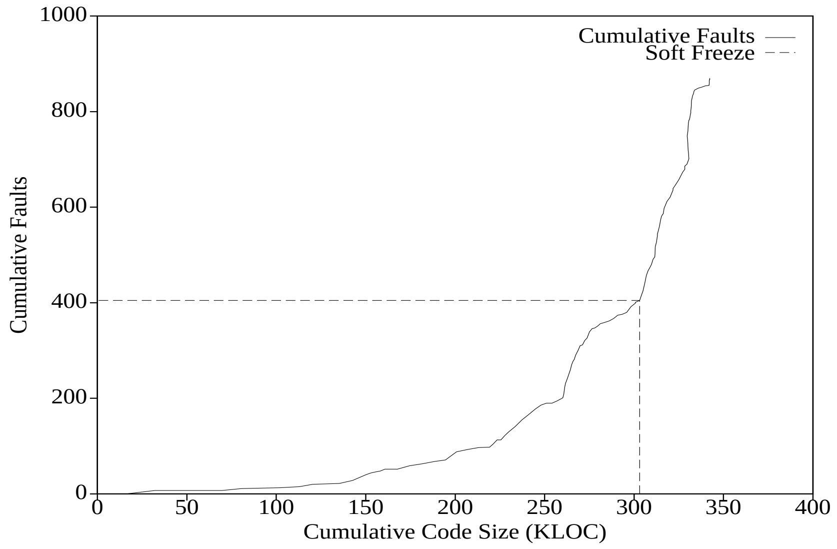

Figure 3: Cumulative Faults Vs. Cumulative Code Size.

the usual exponential assumptions. Let c_i be the size of the code added at the end of the i-th interval. If there was no code added in an interval then the corresponding $c_i = 0$. Since all code should undergo at least some testing, it is assumed that there were no further additions of code beyond interval $k \leq n-1$. The basic model of Dalal and Mallows [1, 2] on which the extended model was 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 c_i, is given by,

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

where $g(x, c, \theta)$ is a general (yet unspecified) function. 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, the structure of g_1 is assumed to be an identity function and c is equal to NCNCSL at the i-th delivery (i.e., $g_1(c, \theta) = \theta g_1(c)$). Finally, the model is simplified by assuming that the number of faults in the added code is proportional to the size of this 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. Their formulation also includes a maximum likelihood equation for estimating the parameters of the extended model and an economic heuristic to decide when one should stop testing. Observe that the estimate of the parameters α and β can very over time. 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 of their study 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

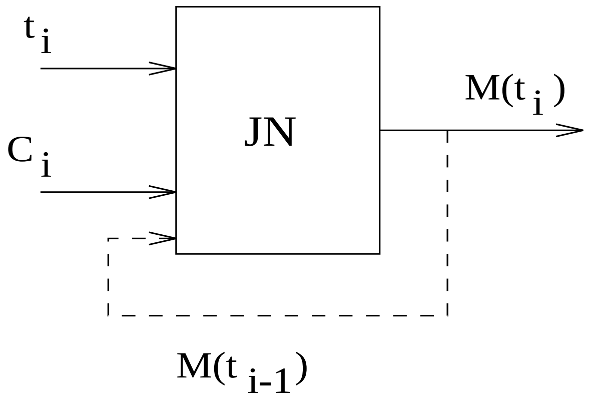

Figure 4: A Modified Jordan Style Neural Network.

by Karunanithi et al. [6, 7, 8]. (For more details on neural network models and their implementation refer to [6, 7, 8] and other references recommended therein.) Two important conclusions of the earlier 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 [5] networks with "Teacher Forced" training are capable of providing more accurate predictions than the feed-forward networks. Also, it was hypothesized that one can easily realize a complex software reliability growth model by incorporating additional information into the neural network framework. This paper verifies the later hypothesis by adding the code churn information to the existing neural networks framework.

The network models used in the previous research[6, 7, 8] 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 of view, 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 (C_i) for the cumulative code churn. Figure 4 represents an abstract model of the modified Jordan style network used in this study. The dotted line in Figure 4 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.

The function mapping of the network can be expressed as,

$$M(t_i) = f(M(t_{i-1}), t_i, C_i)$$

where f is an unknown mapping developed by the neural network. According to this model, the cumulative faults at time t_i is a function of the cumulative time, the cumulative code churn and the cumulative faults at the previous time step. This realization of the model is analogous to the extended model developed by Dalal and McIntosh [3].

4 Results

4.1 Fit of the Model

To evaluate the quality of the fit of the neural network approach, we constructed two different Jordan style networks. The first network had the staff day as the only external input. The second network, on the other hand, had both the staff day and the cumulative code churn as external inputs. Figure 4 shows the schematic representation of the second network model. The network had a clipped linear unit [8] in the output layer and sigmoidal units in the hidden layer. The network was constructed and trained using the Cascade-Correlation algorithm. The Cascade-Correlation algorithm is a constructive algorithm that can be used not only to train a network but also to construct a suitable network. During training, the Cascade-Correlation starts with a minimal network (i.e., a network with no hidden units) and progressively adds the required number of hidden units until the learning is successfully completed. However, the number of hidden units added to the network and the final weights of a trained network can vary depending on the random weight values used at the beginning of the training. This can introduce statistical variation in the fit as well as the predictions of the model. To reduce such variations and to get a better statistics we constructed and trained networks using 50 different initial weight vectors. The results reported in this

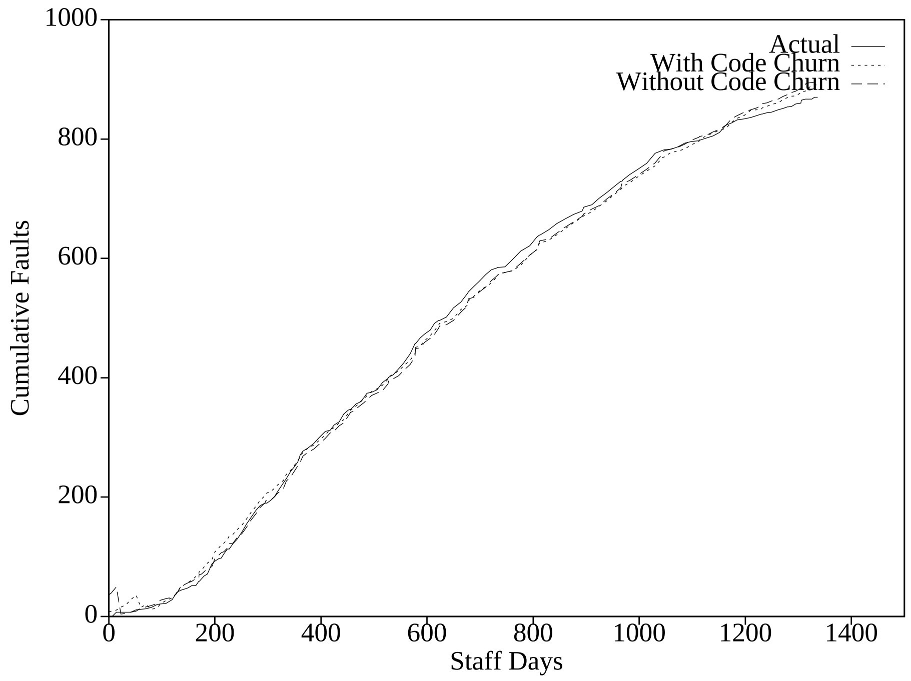

Figure 5: Fit of the Neural Network Models.

section are based on the averages obtained from 50 trials. The final fit of the networks with and without the code churn information are shown in Figure 5. The solid line in Figure 5 represents the actual cumulative faults observed. The dotted line is the fit of the network with the code churn input. The fit of the network without the code churn information is represented in a dash line. Note that both fits look almost similar. However, a closer look at these fits revealed that the network that used the code churn information is closer to the actual data than the network without the code churn information. This observation agrees with the result of Dalal and McIntosh [3].

4.2 A Diagnostic Plot

In order to check whether there are bias in the fit of the models, a simple diagnostic plot was constructed using the residuals from the final fit of the models. For the purpose of illustration, the diagnostic plot for the network with the code churn information is shown in Figure 6.

The residuals in Figure 6 represent the difference between the change in the observed number of cumulative faults and the change in the number of cumulative faults of the fit of the model. The residuals for the plot are obtained as follows:

$$Residuals = \{(M(t_i) - M(t_{i-1})) - (\hat{M}(t_i) - \hat{M}(t_{i-1}))\}$$

where $M(t)$ and $\hat{M}(t)$ represent the observed faults and the fit of the model respectively. The residuals are analogous to the derivative of $M(t)$ and reflect the sensitivity (or bias) of the local behavior of the fit. The diagnostic plot suggests that there is no overall major trend in the fit of the neural network model.

4.3 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

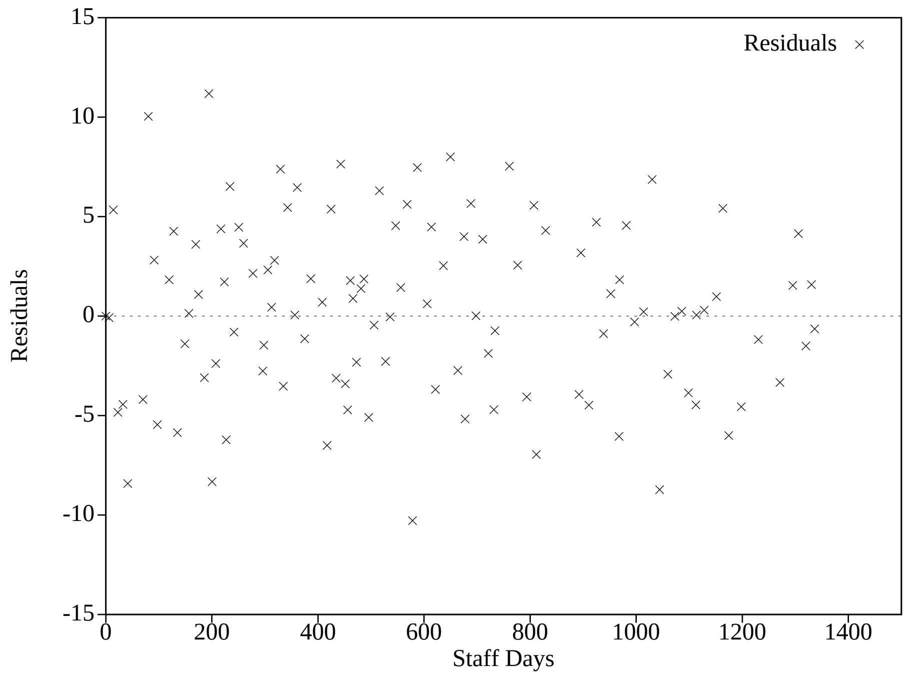

Figure 6: Diagnostics: Residuals vs. Staff Days.

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. [10]. 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.

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.

Dalal and Mallows [1, 2] proposed and integrated an economic model to decide when one should stop testing. The economic model is based on the tradeoff between the cost of testing software (i.e., not releasing in time) and the cost of releasing it. In the future, we plan to integrate their economic model with the neural network framework.

Often, one is interested in predicting not only 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

Neural Network Model	NSP		EPP	
	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*.

cumulative faults using the method outlined in [8].

There are several advantages with an extended software reliability growth model:

- From a modeling point of view, it demonstrates how one can extend a time-domain based model, whether it is based on the neural network framework or the standard stochastic approach, by incorporating continuous code churn information.
- From an information theoretic point view, the extended model is appealing because it is based on more information about the failure process than the traditional execution time-based model. As a first approximation, we directly included the code size in the extended model. However, this need not be the case; one can preprocess the code (i.e., perform a static analysis on the code to extract relevant static complexity measures) and then use the preprocessed outputs to build a more sophisticated software reliability growth model.
- One can also use the extended model in the early stages of the software development cycle because the model does not require the existence of code for the complete system.

References

[1] S. R. Dalal and C. L. Mallows, "When Should One Stop Testing Software?", *J. Am. Statist. Assoc.*, 83, pp. 872-879, 1988.

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

[3] S. R. Dalal and A. A. McIntosh, "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.

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

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

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

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

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

[9] B. Littlewood, "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.

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

[11] J. C. Munson and T. M. Khoshgoftaar, "The Functional Problem Complexity Metrics for Software Reliability Models", *Proc. 1991 Int. Symp. on Soft. Rel. Eng.*, pp. 2-11, May 1991.

[12] J. D. Musa, A. Iannino, and K. Okumoto, **Software Reliability - Measurement, Prediction, Applications**, McGraw-Hill, 1987.