A NEURAL NETWORK APPROACH FOR PREDICTING SOFTWARE DEVELOPMENT FAULTS

Taghi M. Khoshgoftaar, Abhijit S. Pandya and Hemant B. More

Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
taghi@cse.fau.edu

Abstract

Accurately predicting the number of faults in program modules is a major problem in quality control of a large scale software system. In this paper, the use of the neural networks as a tool for predicting the number of faults in programs is explored. Software complexity metrics have been shown to be closely related to the distribution of faults in program modules. The objective in the construction of models of software quality is to use measures that may be obtained relatively early in the software development life cycle to provide reasonable initial estimates of quality of an evolving software system. Measures of software quality and software complexity to be used in this modeling process exhibit systematic departures of normality assumptions of regression modeling. Neural networks have been earlier applied in dynamic software reliability growth prediction [7]. This paper introduces a new approach for static reliability modeling and compares its performance in the modeling of software reliability from software complexity in terms of the predictive quality and the quality of fit with more traditional regression modeling techniques. The neural networks did produce models with better quality of fit and predictive quality when applied to one data set obtained from a large commercial system.

1 Introduction

Recent research in the area of software reliability has been focused on the identification of program modules that are most likely to contain faults. It has been found that a relatively few number of modules, in a set of modules constituting a program, will contain a disproportionate number of faults. A major research effort is underway to try to determine, *a priori*, which modules might contain a significant number of faults so that the program test and validation process might be focused in the most productive direction.

The objective of this study is to apply neural networks which can contribute to the development of predictive models for use in software quality determination. Software complexity metrics have been shown to be related to the distribution of faults in program modules. That is, there is a direct relationship between some software complexity metrics and the number of changes attributed to faults found during the testing phase [11]. Many researchers have sought to develop a predictive relationship between complexity metrics and faults. In particular, a relationship has been found between the software measure of lines of code and faults; for example, studies [4,17] have developed estimates of fault rates ranging from 0.3 to 0.5 faults per hundred lines of code. Lipow [9] developed a model which predicts the number of faults per line of code based on Halstead's [6] software science metrics. Gaffney [5] proposed formulas relating the number of faults to the number of lines of code and also to the number of conditional jumps. Khoshgoftaar and Munson [8] developed a nonlinear regression model for predicting software development faults which generalizes Gaffney's model. Crawford, et. al. [3] found that no one software complexity measure provided significantly better results than the lines of code metric in predicting the number of faults in C programs; however, they recommended using multiple variable models. Akiyama [2] related program faults in an assembly language environment to decisions and calls (jumps) rather than to the number of program statements. Here the relationship between the number of faults in programs and the software metrics used to predict these faults is investigated.

During the last decade, neural networks have emerged as a promising technology in applications which require generalization, abstraction, adaptation and learning. The application of neural networks to

0-8186-2975-4/92 $03.00 © 1992 IEEE

diverse fields range from *autonomous vehicle control* [18] and *financial risk analysis* to *handwriting recognition* [1]. Therefore, it is only natural that the neural network technology is exploited to solve different software engineering problems. For example, neural networks have been earlier applied in dynamic software reliability modeling [7]. The purpose of this study is to compare neural networks and regression models in the modeling of software reliability from software complexity in terms of the predictive quality and the quality of fit.

2 The Neural Network Paradigm

Recent advances in learning algorithms for neural networks and parallel computation has led to renewed research in the area of statistical pattern classification. Neural networks have been applied both to time varying patterns as well as static patterns. In this section we shall briefly discuss applications to time invariant data for software reliability prediction. Neural nets are actually a set of neurons or processing elements which are interconnected through a connection strength (weight). The neurons in a multilayer perceptron model are arranged in layers. There is one layer for the input variables, and one for the output. There can be one or several layers between these which are referred to as hidden layers. The signal or input given to one neuron is passed to all the neurons to which it is connected in fractions equivalent to the weight between these neurons. Each neuron calculates its output based on a function which can be sigmoid, step or some such suitable function.

Figure 1 shows the two major phases of development of neural networks for predicting reliability. During the training phase, based on the *a priori* knowledge concerning the problem domain, a limited amount of training data is used to adjust neural net parameters. At the end of the training, the neural network develops an internal representation for the transformation that maps the input stimuli space onto the output response space. The process of training actually forms an internal representation which is the model for predicting the reliability characteristics of the data set. This can be compared to the curve fitting operation on the data for a selected model. This trained net can now be fed with the test data for which a prediction is needed. This is referred to as the prediction phase. If the net is trained well, and if the data set used for training is a good representation of the data to be encountered in actual prediction runs, it would be seen that the prediction has very good quality. During the prediction phase, the neural network designed from the training phase is evaluated on test data by providing an output for each input pattern. This approach can also be used to obtain outputs for a partial input pattern. Note, that test data is independent data that is only used to assess the generalization of the trained neural net defined as the error rate on the never-before-seen input patterns.

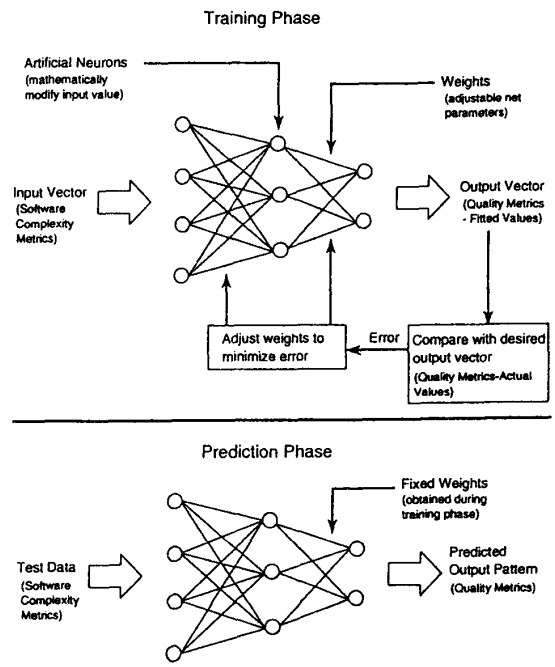

Figure 1: Two Phases of Neural Network Operation

Neural net learning algorithms allow a net to adapt and learn from its mistakes. Each artificial neuron is a simple processor with the ability to add all weighted inputs and then apply a mathematical transformation to generate an output. The resulting number (output) is typically fed into the next layer of neurons, which operates in a similar fashion and produces an output. The output of the neurons used in our simulations can be defined as :

$$Net_j = \sum_{i=1}^{n} W_{ij} Out_i - \theta_j$$
$$Out_i = f(Net_i) = 1/(1 + e^{-Net_i/T}) ,$$

where Out_i is the output of the ith neuron. Net_j is the weighted sum of inputs for neuron j with threshold θ_j, n is the number of input neurons, and W_{ij} is the connection strength (weight) between neurons i and j. T is the *temperature* for the sigmoid where small T will result in an approximation to a step function.

During the training phase, the network adapts by varying the connection strengths, so that its answers correspond with the reality. Initially all the weights are set randomly. The network generates an output vector for a given input pattern. Then it looks at the magnitude of error to see how much it needs to adjust to align its response with the desired one. Then it systematically increases or decreases the weights by a small amount in a direction that it estimates will reduce the error. Next a new input pattern is applied and the output is generated. This goes on until the complete data set is utilized. After one cycle is over, the set is fed again to the network. This is continued until the difference between the actual and desired output is less than the tolerance required. In our simulations, we have implemented a multilayer perceptron and back-propagation learning algorithm for the training phase [15]. The total error, E, between the desired output and the actual output can be expressed as:

$$\mathrm{E} = \tfrac{1}{2} \sum_p \sum_i (d_{pi} - O_{pi})^2 \ ,$$

where d_{pi} is the desired output of the output neuron i for the pth input pattern and O_{pi} is the actual output of the same unit. At the nth iteration the weights are adjusted using the following criterion :

$$\Delta W_{ij}(n+1) = \eta \delta_j Out_i$$
$$W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}(n+1) \ ,$$

where $W_{ij}(n+1)$ is the weight between the neuron i, which feeds the input, and neuron j in the next hidden or output layer at time epoch $n+1$. η is the learning rate. For neurons in the output layer the error signal δ_j is given by:

$$\delta_j = Out_j(1 - Out_j) * (d_{pj} - O_{pj}) \ .$$

In case of neurons (j) in the hidden layers there is no specified target. The error signal δ_j is determined recursively in terms of errors in neurons k which it (j) directly connects and weights of those connections, that is,

$$\delta_j = Out_j(1 - Out_j) * [\textstyle\sum_k \delta_k W_{kj}] \ .$$

Use of a *momentum* term in modification of weights is advised as it improves the training time and enhances the stability of the training process. The effect of past weight changes on the current direction of movement in the weight space is determined by α and the weights are modified according to the following equation :

$$W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}(n+1) + \alpha \Delta W_{ij}(n) \ .$$

The weight space is modified iteratively until the total error E is within the tolerance limit.

3 Experimental Methods

In order to use neural nets for any system it is essential that some thought be given to the encoding of the information available in such a manner that it would be amenable to be fed to the net as input. The net that was used for this experiment was fed with the input variables (software complexity metrics) which are integers. These software metrics represent quantitative descriptions of program attributes. Some of these metrics have been shown to somehow be related to quantitative measures of program quality. The output variable for our model is the total number of faults in program modules. This output varied between 0 and 42 for the software system being studied for our experiments. Moreover, the output variable was 0 for a substantial number of training sets. Software metrics and the number of faults for each program module form one training set. We scaled the output variable between 0 and 1. This formed the setup for training the net. After training was over, the trained net then was applied to the test data to give some assessment as to the validity of the prediction process, and the predictive values were transformed to the original scale. Predictive quality and quality of fit values for the data set were found.

The number of hidden layers and the number of neurons used in each hidden layer determine the architecture of the net apart from the input and output layers mentioned above. Having more number of hidden layers and more neurons in each layer will require more computations to be done for generating each output. The advantages are that a higher degree equation can be formed by greater number of hidden layers.

When the net is learning, the quantity *learning rate* determines the amount of correction to be made to the weights when the output generated is not equal to the desired output. The granularity of this change is determined by the learning rate. If the learning rate is high, a large change would be made in the weights, and situations which need a slight modification will experience oscillations. A small learning rate on the other hand will make it slower to train the network. Generally a combination of smaller and larger learning rates at different levels of training proves best.

4 Prediction Results

In our experiments, we used the data set with different network architectures. We used eight primitive metrics as input variables and one output variable that is the total number of faults in program modules. The architecture of the models was varied between one hidden layer to three hidden layers. The number of neurons in each hidden layer was also varied in different experiments.

The neural network predicts best when the training is done for a long time or until all the samples are learned by the net. It is very unlikely that all samples would be learned in a reasonably long training time, since the data set may contain significantly different output values for similar input patterns, i.e, we may have outliers in the data set. These outliers have a tendency to overbias neural net and regression models. The neural net can be used for prediction runs after more than a specific percentage of samples are learned. This specific percentage can be decided based on either the samples or heuristics. It was observed that the one hidden layer architecture performed the best in terms of the performance criterion (prediction error). It turned out that having a high learning rate at the initial stages, when less than 50 percent of the observations (program modules) are trained, and then progressively decreasing the learning rate as more and more observations get trained works best for training the net with our data set. The quality of fit and the predictive quality found for the data set give very optimistic results. The prediction results indicate that the net tries to track the behavior of the full data set and sometimes its predicted value is more than the actual and sometimes less. The data set itself is an instance of random behavior. But what the net predicts is how many faults should occur given the input variables based on what pattern it recognizes in the training data. The training data is the only data for the net to base its conclusions on since that is the only information it receives from the outer world.

5 Regression Analysis and Performance Criterion

In the use of software metrics to predict some measure of software quality, such as program faults, a linear regression model is customarily employed. In this regression model, the complexity metrics will appear as independent variables and the dependent variable will be program faults. Linear regression models are formed by choosing a subset from a set of independent variables in order to explain as much variation in the dependent variable as possible. Coefficients for the independent variables are produced by least squares estimation technique that attempt to fit these variables to sample data. Several methods of selecting an appropriate subset of independent variables that will not introduce additional variance (or noise) in the model have already been discussed by Munson and Khoshgoftaar [11].

It has been indicated [10] that minimization results of the average absolute error of a fit $y = f(x)$ given by:

$$AAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|, \qquad (1)$$

provide very little information about the fit's accuracy. y_i represents the number faults in the ith program module; x denotes an independent variable (complexity metrics such as lines of code, etc.); and $\hat{y}_i$ is the predictive value of the ith dependent variable by the model. Levitin [10] also adds, that "in order to get a better idea, one needs to relate the size of errors to the values being approximated". Furthermore, it is oftentimes useful in quality and reliability modeling to measure the performance of a model in terms of its relative error. The average of absolute relative errors is an appropriate "loss function" for determining the quality of the prediction equation [14]. The measure of average relative error is defined as:

$$ARE = \frac{1}{N}\sum_{i=1}^{N}|(y_i - \hat{y}_i)/y_i|. \qquad (2)$$

ARE has been established to be useful as a measure of model performance associated with complexity metrics in order to identify error-prone software [16]. Once a model has been fitted, its ARE value can be easily obtained by computing (2).

The performance of the model can be divided into the components ***quality of fit*** and ***predictive quality.*** Once a model is fitted, the quality of fit is determined from the magnitude of the corresponding ARE value calculated using only the values needed to fit the model. The predictive quality is assessed from the magnitude of the ARE using values other than those used to fit the models.

6 Example

6.1 The command and control communication system

An illustration is presented in this section comparing the performance of the regression and neural network models. The *quality of fit* as well as the *predictive quality* of each model is analyzed. The average relative error criterion defined in section 5 is used to evaluate the quality of performance of each of the models.

The data for the application of the regression analysis technique and neural network model were obtained from an Ada development environment for the command, and control of a military data link communication system (CCCS). All the data were collected from one large Ada project consisting of a total of 282 program modules. The criterion (dependent) variable, "number of faults per module", was determined from the Problem Change Reports (PCRs) recorded during the system integration and test phase and the first year of the program deployment. There was a one to one correspondence between the PCRs and changes made to programs, that is, one PCR per fault.

The metrics collected for the CCCS system consisted of the Halstead software science metrics together with other enumerative metrics listed below:

- Unique operator count (η_1)
- Unique operand count (η_2)
- Total operator count (N_1)
- Total operand count (N_2)
- Halstead's estimated program length ($\hat{N}$) where $\hat{N} = \eta_1 log_2 \eta_1 + \eta_2 log_2 \eta_2$
- Halstead's effort metric (E)
- Program volume ($V = N log_2(\eta_1 + \eta_2)$)
- McCabe's Cyclomatic complexity ($VG1$)
- Extended Cyclomatic complexity ($VG2$)
- Number of procedure calls ($PROCS$)
- Number of comment lines (COM)
- Number of blank lines ($BLNK$)
- Number of lines of code (LOC)
- Number of executable source lines of code ($ELOC$).

For the analysis of the data for these CCCS program modules, the initial set of 282 modules was subdivided into two groups. The first group of 188 program modules was randomly selected from the initial set of modules for the purposes of model building, and the remaining 94 modules were used for model validation.

Several techniques are available for selecting a subset of independent variables from the collection of random variables mentioned above in order to form a linear regression model. These include the R^2_p criterion, C_p criterion, forward selection procedure, stepwise regression, and backward elimination procedure. The authors used the latter two procedures and, in this case, each led to selection of the metrics $VG1$, $VG2$, η_1 and η_2 for the independent variables.

Stepwise regression is a procedure that incorporates independent variables into the linear model iteratively. An initial linear model is formed by selecting the independent variable having the largest squared simple correlation with the dependent variable. In subsequent iterations, new variables are selected for inclusion based on their partial correlation with variables already in the regression equation. Variables in the model may be removed when they no longer contribute significantly to the explained variance. In the backward elimination procedure, a linear model is formed using all the variables and then variables are eliminated, one at a time, that do not contribute significantly to the model. The reader interested in further details concerning model selection procedures is referred to Myers [13].

Neural and regression analysis techniques were employed, using the same random sample of 188 programs, to fit (train) a linear regression (neural network) model expressing program faults (dependent variable) in terms of the selected independent variables $VG1$, $VG2$, η_1 and η_2 for regression model and all primitive metrics for neural model. In our previous study [12], we have shown that the set of metric primitives measures everything that the larger set of primitive and non-primitive metrics does. The ARE values used to measure the quality of fit for each model were found for the 188 programs and are listed in Table I. Observe that the neural network model is superior and also had a small standard error (SD). Likewise, the predictive quality for each model was determined for the remaining 94 programs and is shown in Table II. Again, a neural network model is superior.

Modeling Technique	$\left\lvert\frac{y-\hat{y}}{y}\right\rvert$	
	Mean	SD
Regession	0.6249	0.8167
Neural Net	0.3333	0.2330

Table I: Model Quality of Fit for CCCS

Modeling Technique	$\left\|\frac{y-\hat{y}}{y}\right\|$	
	Mean	SD
Regression	0.5877	0.6248
Neural Net	0.3980	0.2786

Table II: Model Predictive Quality for CCCS

7 Conslusions

The software crisis focuses the attention of software engineers on the research of systematic techniques for software development in an attempt to make software systems more reliable. This calls for more research into building better models. The objective of this study was not to present a definitive model for the prediction of program quality measures, but rather to explore and evaluate neural and regression modeling techniques. Regression models are widely used by software reliability engineers to predict the number of faults in program modules. However, a neural network approach opens a broad new spectrum in our search to obtain models possessing superior quality of fit and prediction. Results from ARE values recorded in the tables suggest that neural network approach possess good properties from the standpoint of model quality of fit and predictive capability.

In order to build a linear regression model it is necessary to first select from a set of independent variables a subset of these variables that will explain the largest amount of variance in a dependent variable. The neural network approach presented here allows the user to directly use the collection of independent variables as the input to the neural net. During the training, the network itself will select the most significant software complexity metrics by computing the appropriate weight matrix. As a result the user is relieved from having to perform costly preprocessing, and the resultant model is likely to be superior in performance quality.

The regression analysis technique also involves assumptions regarding independence of response (dependent) variables, i.e, number of program faults, etc. The neural net approach does not require any assumptions of this nature and allows response variables which are linearly dependent.

The subject of predictive quality and appropriateness of the models has failed to receive the attention that it deserves in many reliability models we have studied. In general, there are many performance measures which aid in the determination of the predictive quality of models. We have examined some in this paper. Within the framework of this limited data set, our findings suggest an empirical basis for use of neural network model in order to identify fault-prone program modules.

Acknowledgment

The authors wish to express their sincere thanks to our colleagues in the aerospace industry for providing the Command and Control Communication System data. Dr. Khoshgoftaar was supported in part by a research grant from the Florida High Technology and Industry Council Applied Grants Program.

References

1. L.C. Agba, R. Shankar, A.S. Pandya, and C. Naylor, "A VLSI implementable handwritten digit recognition system", *Proceedings of International Joint Conference on Neural Networks*, Vol. II, Washington D. C., 1990, pp. 275-279.

2. F. Akiyama, "An example of software system debugging", *Inform. Processing, Proceedings of IFIP Congr.*, Aug. 1972, pp. 353-359.

3. S.G. Crawford, A.A. McIntosh, D. Pregibon "An analysis of static metrics and faults in C software", *The Journal of Systems and Software*, Vol. 5, 1985, pp. 37-48.

4. A. Endres, "An analysis of errors and their causes in system programs", *Proceedings of 1975 International Conf. Rel. Software*, 1975 April, pp. 327-336.

5. J.E. Gaffney, Jr., "Estimating the number of faults in code", *IEEE Transactions on Software Engineering*, Vol. SE-10, 1984 July, pp. 459-464.

6. M.H. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.

7. N. Karunanithi, Y.K. Malaiya, and D. Whitley, "Prediction of software reliability using neural networks", *Proceedings of the International Symposium on Software Reliability Engineering*, May 1991, pp. 124-130.

8. T.M. Khoshgoftaar, J.C. Munson, "Predicting software development errors using software complexity metrics", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 2, 1990 Feb., pp. 253-264.

9. M. Lipow, "Number of faults per line of code", *IEEE Transactions on Software Engineering*, Vol. SE-8, 1982 July, pp. 437-439.

10. A. Levitin, "The L_1 criteria in data analysis and the problem of software size estimation", *Proceedings of 21st Symposium on the Interface Computing Science and Statistics*, pp. 382-383, 1989.

11. J.C. Munson, T.M. Khoshgoftaar, "Regression modeling of software quality: an empirical investigation", *Journal of Information and Software Technology*, Vol. 32, No. 2, 1990 March, pp. 106-114.

12. J.C. Munson, T.M. Khoshgoftaar, "Some primitive control flow metrics", *Proceedings of the Annual Oregon Workshop on Software Metrics*, Silver Falls, Oregon, March 1991.

13. R.H. Myers, *Classical and Modern Regression with Applications*. Duxbury Press, Boston, MA, USA, 1990.

14. S.C. Narula, J.F. Wellington, "Prediction, linear regression and the minimum sum of relative errors", *Technometrics*, Vol. 19, 1977 May, pp. 185-190.

15. D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation", in D.E. Rumelhart and J.L. McClelland [Eds.], *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, MIT Press, Cambridge MA, 1986, pp. 318-362.

16. V. Shen, T. Yu, S. Thebaut and L. Paulsen, "Identifying error-prone software - an empirical study", *IEEE Trans. Soft. Eng.*, Vol. SE-11, no. 4, April 1985.

17. M.L. Shooman, and M.I. Bolsky, "Types, distribution, and test and correction times for programming errors", *Proceedings of 1975 Int. Conf. Rel. Software*, 1975 April, pp. 347-362.

18. K.P. Venugopal, A.S. Pandya, and R. Sudhakar, "Online control of autonomous underwater vehicles using feedforward neural networks", To appear in the *IEEE Journal of Ocean Engineering.*