Investigating the Predictability of Empirical Software Failure Data with Artificial Neural Networks and Hybrid Models

Andreas S. Andreou, Alexandros Koutsimpelas
Department of Computer Science, University of Cyprus,
75 Kallipoleos Str., P.O. Box 20537, CY1678, Nicosia, Cyprus
aandreou@ucy.ac.cy, cs99ak2@ucy.ac.cy

Abstract. Software failure and software reliability are strongly related concepts. Introducing a model that would perform successful failure prediction could provide the means for achieving higher software reliability and quality. In this context, we have employed artificial neural networks and genetic algorithms to investigate whether software failure can be accurately modeled and forecasted based on empirical data of real systems.

1 Introduction

One of the major demands in Software Engineering is reliability. Non-reliable systems lead to dissatisfied customers and system users, extra human hours devoted on testing and difficulties through the maintenance phase. High reliability ensures that there is a small failure probability during the execution of a program [8] [9]. The term failure defines the inability of the program to respond to user requests correctly, that is, as prescribed in its requirements [9], caused by a programming error during the implementation or an ill-defined user need during the analysis of the system. Measuring the occurrences of failures in a system provides a way to determine its reliability. These measurements reflect the system quality and can be used in decision-making and problem solving processes. Essentially, there are four ways to record failure occurrences [9]:

- Time of failure
- Time of interval between failures
- Cumulative failures expressed up to a give time t
- Failures experienced in a time interval Δt

Software failure data is a reliability measurement but suffers many disadvantages mainly because of its dependence to the simulation environment and user knowledge. A system being tested in a controlled environment produces different failure data if tested under real use and there is always an issue for the ability of the

Please use the following format when citing this chapter:
Andreou, Andreas, Koutsimpelas, Alexandros, 2006, in IFIP International Federation for Information Processing, Volume 204, Artificial Intelligence Applications and Innovations, eds. Maglogiannis, I., Karpouzis, K., Bramer, M., (Boston: Springer), pp. 524–532

tester. People tend to use software differently and failure data recorded from expert users differ dramatically from failure data of naive users. This partially explains the difficulties in modeling the behavior of software failures.

Having a model that predicts the occurrences of software failures may help software analysts and developers produce quality and reliable systems. Knowing a priori that a failure may occur in a specific time and monitoring thoroughly the problematic system at that moment increases the probability of successful error discovery. One way to achieve this is to study the behavior of empirical failure occurrence data and investigate its predictability, building upon a previous research that investigated the nature of software failures using a non-parametric analysis [2]. The findings of that study were quite interesting as they support a random explanation of the behavior of empirical software failures, which resembles that of pink noise. We will attempt to question or support those findings by utilizing several forms of neural networks, including hybrid models (i.e. combined with genetic algorithms) trained with the same data series to investigate their predictability level.

The rest of the paper is organized as follows: Section 2 provides an overview on neural networks and genetic algorithms, while section 3 describes the models proposed to produce the failure predictions. Finally, section 4 draws the concluding remarks.

2 Theoretical Background

2.1 Artificial Neural Networks

Artificial neural networks are based on the model of the human brain neuron cells and try to mimic their functions. The human brain contains billions of neuron cells connected to each other through synapses. The input signals of each cell are added in the body cell and the summation is provided to other cells through the axon. Based on this, an artificial neuron is a mathematical model, which tries to duplicate this function. The neuron input vector **x** is multiplied with the input weights vector **w** and an adder sums those products. The result is passed to an activation function φ and its output becomes the neuron's output. The mathematical representation of an artificial neuron is [6]:

$$u_k = \sum_{j=1}^{m} x_j w_{kj} \tag{1}$$

$$y_k = \phi(u_k) \tag{2}$$

where **x** is the input vector of neuron k, $\mathbf{w_k}$ the weighs vector and u_k the dot product of those two. The neuron output y_k is defined as the result of the activation function φ given the dot product u_k.

Combining several artificial neurons and organizing them into layers sets up an artificial neural network. By imitating the job of neuron cells the artificial neural networks inherit advantages such as generalization, input-output mapping fault

tolerance and adaptability. When the neuron outputs of a layer are provided as inputs to the neurons of the next layer, the neural networks are called fully connected. The outputs of the last layer are the network's outputs. The number of network inputs, the number of neurons per layer, their activation function and the number of layers define the network's architecture. The most popular network architecture is the Multi Layer Feed Forward Perceptrons [6]. Networks belonging in the category are composed of many layers, more than one and their neurons are fully connected with the neurons of the next layer.

Another category of neural network architecture is the Recurrent Networks. Observations of the human brain show that the output of a neuron cell is often redirected as an input to the same cell [6]. This kind of loop is implemented in the model of the recurrent neuron. Using this type of neurons produces networks that can solve more difficult problems. Typical examples of recurrent networks are the Elman networks, the Hopfield networks and the Kalman Filtering [5].

In order to be able to solve a problem, neural networks have to go through a training phase. In most cases, this training is supervised meaning that the networks try to learn by applying changes to there synaptic weights according to the errors computed on their outputs. The errors are the differences of the network output for a specific input pattern from the actual values provided. There are many training algorithms that implement this process, the most popular being the back propagation algorithm [6] and its variations. After the training phase the network generalizes its knowledge; it can provide satisfactory responses to unknown input patterns.

2.2 Genetic Algorithms

Genetic algorithms are stochastic, probabilistic algorithms that model natural phenomena such as inheritance and Darwinian strife for survival [7]. They are used to search a space of possible solutions through a process of evolving and evaluating individuals representing solutions and selecting the best one. The evolution works by encoding the possible solutions in a chromosome-like format, usually bit strings of specific length. A population of these chromosomes is randomly initialized and an evaluating loop begins. A fitness function selects the individuals with the highest fitness values. The selected atoms are undergone genetic operators like crossover (mating two individuals by exchanging their parts at a random position) and mutation (flipping bits from 0 to 1 and vice versa), which alter their values and hence their fitness [7].

3 Empirical Findings

This paper presents three attempts to find suitable neural networks for software failure occurrence predictions. The first one combines feed-forward neural networks trained with various algorithms and evolved by a genetic algorithm in terms of architecture. The second one utilizes recurrent networks and the third is a variation of the first one where the training data have been preprocessed.

3.1 Musa Dataset

In the 70s J. Musa created a dataset of software failures while working for the Bell Telephone Laboratories. The purpose of this work was to verify the ability of SRGM to simulate the occurrences of failures. Sixteen systems where monitored, such as operation system, word processors and real time systems, creating datasets that provided information about system code, day of failure and time intervals between failures in seconds. Three of these datasets were used in this work to train neural networks: Project 5 with 831 samples, SS1B with 375 samples and SS3 with 278 samples.

Previous research in this field [8] [9] have shown that the nature of the software failure data is non-deterministic and random. As seen in fig. 1, time intervals may vary as execution proceeds and it's very difficult to locate a pattern in this series especially with the presence of spikes adding more complexity. Furthermore, R/S analysis performed by Andreou and Leonidou [1] concluded that the characteristics of software failure data in Musa's datasets are similar to that of pink noise.

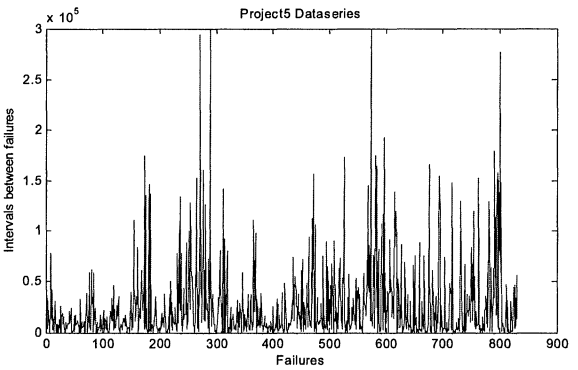

Fig. 1. Musa Dataset, Bell Laboratories, Project 5 interval between failures (seconds)

3.2 Feed Forward Neural Networks and Genetic Algorithms

The first attempt to locate an artificial neural network architecture that performs software failure data prediction was made using multi-layer feed-forward perceptrons that are trained with the Project 5, SS1B and SS1 data series. For the investigation part of the experiment, a dedicated genetic algorithm was designed and implemented according to the theory of evolutionary algorithms [7].

Each chromosome of the genetic algorithm represents a feed-forward neural network. The genes of the chromosome include information about the number of inputs, the number of nodes per layer and the activation functions per layer. Limits were set to the number of inputs, sixteen per networks, and the number of neurons,

thirty-two per layer. The output layer is consisted of one neuron with a linear activation function. Each network has two hidden layers by default.

Apart from information on the number of neurons and the activation function, an extra gene is used to specify the training algorithm used to train the network represented by the chromosome. The back propagation algorithm is used and its variations (back propagation with momentum, back propagation with adaptive learning rate) [5]. Furthermore, optimized algorithms that use Jacobian and Hessian matrices were utilized to perform weight corrections, such as the Levenberg-Marquardt back-propagation [4], the Quasi-Newton BFGS [3] and the Bayesian regularization back-propagation [4].

3.3 Recurrent Networks and Genetic Algorithms

In the case of recurrent neural networks the chromosomes were altered to accommodate the Multiple Extended Kalman filtering Algorithm (MEKA) [1], which is an alternation of the Extended Kalman Filtering Algorithm [5]. The synaptic weights wi of node i are computed by the following formulas [2]:

$$r_i(n) = \lambda^{-1} P_i(n-1) q_i(n) \tag{3}$$

$$k_i(n) = r_i(n)[1 + r_i^T(n) q_i(n)]^{-1} \tag{4}$$

$$w_i(n+1) = w_i(n) + e_i(n) k_i(n) \tag{5}$$

$$P_i(n+1) = \lambda^{-1} P_i(n) - k_i(n) r_i^T(n) \tag{6}$$

where, n = 1…N the iteration number and N the total number of input patterns. The vector $\boldsymbol{q_i(n)}$ is the estimation of the activation function made with Taylor series. $\boldsymbol{P_i(n)}$ is the current estimation of the inverse of the covariance matrix of $\boldsymbol{q_i(n)}$ and $\boldsymbol{k_i(n)}$ the kalman gain. The parameter λ is a forgetting factor with values in the range [0,1] and $\boldsymbol{e_i(n)}$ is the propagated error to node i. A good estimation of the initial values of $\boldsymbol{P}$ is $e^{-1}\mathbf{I}$, where e real in the range $[10^{-6}, 10^{-2}]$ and $\mathbf{I}$ the eye matrix [5].

The population of the genetic algorithm is consisted by recurrent networks where feedback loops go from each neuron to every neuron of its layer. The feedback time delay is one step. The chromosome encloses information about the number of inputs, the number of neurons per layer and the activation function of each layer (tansig and logsig).

3.4 SRGM Datasets

In order to compare the findings of real datasets with SRGMs, two artificial dataset were created based on the formulas of the Musa Basic and the logarithmic Musa-Okumoto models [2]. Those datasets, MB and MO respectively, were used in the same manner as real datasets searching for a network that would perform their prediction. The idea is that by succeeding to find a neural network that perform prediction on those artificial data and failing on real data one may argue their ability to model real software failures occurrences.

These data set were created by solving the mean value function $\mu(t)$ for t_i for each of the two models. The mean value function represents the number of failures expected to occur up to time moment t. Those functions are [8]:

$$MO: \mu(t) = [1 - e^{-fKBt}]N \quad (7)$$

$$MB: \mu(t) = N \ln(1 + \phi t) \quad (8)$$

where N is the expected number of failure in infinite time, φ is the failure rate per fault, K is the fault exposure ratio, B is the fault reduction factor and f a factor calculated as the average object instruction execution rate of the computer r divided by the number of source code instructions of the application under testing I_S times the average number of object instruction per source code instruction Q_s [8].

Assuming discreet time, replacing $\mu(t)$ with an integer positive variable i and setting [2]:

$$MO: i = N \ln(1 + \phi t) \quad (9)$$

$$MB: i = (1 - e^{-fKBt})N \quad (10)$$

we get:

$$MO: t_i = \frac{1}{\phi}\left(e^{i/N} - 1\right) \quad (11)$$

$$MB: t_i = -\frac{1}{fKB}\ln\left(1 - \frac{i}{N}\right) \quad (12)$$

Using the above formulas the times of failure occurrences were computed for each SRGM using the values suggested by [8]: (φ, f, K, B) = (7.8, 7.4 10^{-8}, 4.2 10^{-7}, 0.955).

3.5 Preprocessed Datasets

It is common practice to preprocess the data prior to training neural networks so as to remove possible biases and achieve better results. During the experiments of this study three different preprocessing methods were used; logarithms, difference of sequential logarithms (log (t_{n+1})-log (t_n), where t_n the n^{th} time interval of the software failure series) and logarithms with spikes threshold. The new data were used for training the different networks described in section 3.1. For this experiment only the Project 5 dataset was used because is the one with the most samples.

3.6 Results

The experiments were conducted with a population of 50 feed-forward neural networks and a limit of 20 epochs, and a population of 100 recurrent networks and a limit of 50 epochs for the genetic algorithm. During the evolution process the fitness evaluation was done with the Mean Relative Error (MRE) [2] and the fitness function:

$$fitness = \frac{1}{1 + MRE} \quad (13)$$

For each experiment two sets were created. The first one, the training set, consisted of the first 75% of the dataset under investigation. The second set, the remaining data of the dataset under investigation, were used to validate the neural network with data that did not participate in the training. The optimum networks, defined as I-H_1-H_2-1 (I: number of inputs, H_1: number of neurons in first hidden layer, H_2: number of neurons in second hidden layer) were simulated taking as inputs

the training and testing sets, and their responses were used for statistical analysis. The results of this analysis are presented in Table 1.

The Mean Squared Error (MSE) and the Mean Absolute Error (MAE), being data depended criteria, may provide a view on the success to predict the time series but for large input values they don't supply substantial information. On the other hand, the Mean Relative Error (MRE), by not being data depended, may offer a good criterion about the success to predict effectively future software failure intervals. The Normalized Root Mean Squared Error (NRMSE) examines the ability of a network to predict compared to a mean predictor. NRMSE values greater or equal to 1 imply that the network under verification doesn't curry out predictions but instead computes the mean value of its inputs. Finally, the Correlation Coefficient assess the ability of the network to follow the trend of the time series inspected.

Table 1. Statistical results for each dataset

Dataset	Algorithm	Network	Type	NRMSE	CC	MSE	MRE	MAE
Project5	traingdm	7-15-23-1	Train	1.2080	-0.0855	2.04×10^{9}	0.9834	2.54×10^{4}
			Test	1.1733	-0.2076	2.47×10^{9}	0.9624	2.62×10^{4}
	MEKA	16-32-32-1	Train	1.2061	-0.0196	2.06×10^{9}	0.9967	2.54×10^{4}
			Test	1.1746	-0.0067	2.45×10^{9}	0.9949	2.61×10^{4}
SS1B	trainbfg	16-17-24-1	Train	2.103	0.2395	6.23×10^{10}	0.9984	1.49×10^{5}
			Test	1.7328	0.1856	3.67×10^{10}	0.9992	1.08×10^{5}
	MEKA	15-16-24-1	Train	1.2425	-0.088	6.19×10^{10}	0.9323	1.48×10^{5}
			Test	1.1969	-0.0598	3.76×10^{10}	0.9285	1.08×10^{5}
SS3	trainbfg	16-24-7-1	Train	1.2286	0.1933	7.22×10^{10}	0.9949	1.57×10^{5}
			Test	1.2751	0.1257	2.4×10^{11}	0.9998	3.08×10^{5}
	MEKA	16-22-27-1	Train	1.2065	-0.074	7.19×10^{10}	9.2633	1.57×10^{5}
			Test	1.2709	-0.0751	3.14×10^{11}	1.2036	3.5×10^{5}
MB	trainbr	4-9-29-1	Train	9.67×10^{-9}	1	5.01×10^{-5}	2.03×10^{-11}	0.0069
			Test	2.91×10^{-8}	1	5.08×10^{-5}	2.03×10^{-11}	0.0069
MO	trainbr	15-20-30-1	Train	0.002	1	2.87×10^{7}	3.54×10^{-6}	4.56×10^{-4}
			Test	0.0854	1	5.69×10^{-5}	5.21×10^{-5}	0.0067
log (P5)	trainlm	6-28-28-1	Train	7.35×10^{-4}	1	2.13×10^{-6}	1.11×10^{-4}	8.45×10^{-4}
no spikes			Test	1.8568	0.017	19.7321	0.7552	3.4748
diff.	trainbr	16-4-12-1	Train	0.9992	Inf[1]	32.2626	Inf	3.3908
log (P5)			Test	0.974	Inf	77.6024	Inf	3.9268
log (P5)	trainlm	13-23-27-1	Train	0.0368	0.9993	0.0308	0.0085	0.077
			Test	1.6545	-0.0048	118.77	0.7896	7.2237

Even though there were cases where the statistical criteria showed that a network succeeded in predicting the training data set (preprocessed datasets), there was no success in predicting the testing data. This results in the failure of the networks to generalize and expand their knowledge. The NRMSE with values near and above 1 indicate that essentially only the mean values were actually computed. On the other hand, feed-forward neural networks managed to predict with high accuracy the artificial datasets created from equations (11) and (12).

[1] Division with zero

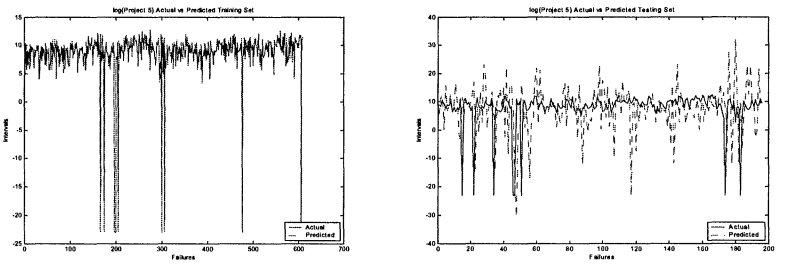

Fig. 2. Log values for Project 5: Training set (left) and testing set (right), with the response of the optimum network (actual samples are presented in solid line and predictions in dashed)

4 Conclusions

The predictability of software failure data was investigated using artificial neural networks and hybrid modeling in an attempt to understand the nature of software failure data with non linear (neural networks), probabilistic and stochastic (genetic algorithms) models.

Our research focused on utilizing neural networks (feed forward and recurrent networks) to predict empirical failure samples recorded by J. Musa for Bell Telephone Laboratories in the '70s for software reliability verification. The findings of our analysis come to support previous work [2] [9] on the subject stating that software failure data are non-deterministic in nature, resembling a random series and more specifically pink noise.

Examining the results in Table 1 and Fig. 3 it is clear that none of the networks investigated managed to produce accurate predictions. The statistical error between actual series values and network-simulated values exceeded the standards and the correlation coefficient showed that the predicted data series suffer from trend capturing and present the inability to reproduce accurate values. What the neural networks managed to do is provide a mean value computation of their inputs, as supported by the NRMSE, with results equal or greater than unity. On the other hand, artificially generated failure data created using known SRGM formulas were successfully predicted. This questions the ability of SRGMs to effectively capture and model the behavior of software failure and hence reliability.

Future work will focus on preprocessed data and advanced recurrent neural networks. The statistics gathered for preprocessed data indicated that the networks used presented better results, achieving higher prediction ability. Advanced recurrent neural networks having scaling time delays greater than one on their feedback loops will be examined to test whether prediction accuracy may be improved. Finally, new software failure data need to be recorded that would express failure tendency of today's software so as to examine possible changes in the reliability behavior of modern software.

References

1. Andreas, A., S., Georgopoulos, E., F., Likothanassis S., D.: Exchange-Rates Forecasting: A Hybrid Algorithm Based on Genetically Optimized Adaptive Neural Networks, Computational Economics (2002), 191-210
2. Andreas, A., S., Leonidou, C.: Nonparametric Analysis of Software Reliability. Revealing the Nature of Software Failure Dataseries. Department of Computer Science, University of Cyprus, Cyprus (2003)
3. Dennis, J., E., Schnabel, R., B.: Numerical Methods for Unconstrained Optimization and Nonlinear Equations, Englewood Cliffs, NJ Prentice-Hall (1983)
4. Hagan, M., T., Demuth, H., B., Beale, M., H.: Neural Network Design, Boston, MA: PWS Publishing (1996)
5. Haykin, S.: Kalman Filtering and Neural Networks, Wiley-Interscience (2001)
6. Haykin, S.: Neural Networks: A Comprehensive Foundation, Prentice-Hall, Second Edition (1999)
7. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, Third Edition, Springer-Verlag Berlin Heidelberg New York (1996)
8. Musa, J., D.: A Theory of Software Reliability and its Application, IEEE Trans. Software Eng. 1(3) (1975), 312-327
9. Musa, J., D.: Software Reliability Engineering, McGraw-Hill (1999)
10. Patra, S.: A neural network approach for long-term software MTTF prediction, Fast abstracts of 14th IEEE International Symposium on Software Reliability Engineering (ISSRE2003), Chillarege Press (2003).