Application of Recurrent Neural Network to Software Reliability and Defect Prediction

Shadow Pritchard[1], Timothy Flavin[2], Vidhyashree Nagaraju[3], and Lance Fiondella[4]
[1]Department of Computer Science, Stonehill College, MA, USA
[2]Tandy School of Computer Science, University of Tulsa, OK, USA
[3]Bastion Technologies, TX, USA
[4]Electrical and Computer Engineering, University of Massachusetts Dartmouth, MA, USA
Email: vidhyashreenagaraju@gmail.com, lfiondella@umassd.edu

***Abstract*—Non-homogeneous Poisson process software reliability growth models assume the defect detection process can be characterized by curves based on parametric forms. However, such parametric forms may not capture all changes or details present in the data, which could limit a model's ability to predict defects accurately beyond a certain level of precision. Therefore, this paper assesses the application of simple neural network methods including artificial and recurrent neural networks to predict defects based on data collected during software testing. Two variations of recurrent networks, long short-term memory and gated recurrent unit, are considered and compared with traditional growth models to assess the predictive capability. Results suggests that RNN and its variations achieve approximately $1.6-37$ and $1.8-68$ times better overall goodness-of-fit measures compared to ANN and traditional NHPP models when 80% and 90% of the data is used for model fitting.**

***Index Terms*—Software reliability, recurrent neural network, software failure, software reliability growth models, non-homogeneous Poisson process, machine learning**

I. Introduction

Non-homogeneous Poisson process (NHPP) [1] software reliability growth models (SRGM) [2] are a common methodology that enables quantitative assessment of software systems by characterizing defect data collected during testing. Traditional NHPP models are developed based on the assumption that the defect detection process can be accurately characterized by a continuous smooth curve. This assumption restricts the model's ability to characterize and predict changes in the failure rate, thus reducing the accuracy of defect prediction. Changepoint models [3] were developed to characterize the changes in the data. However, such models are limited to characterization of the number of changepoints incorporated in the parametric form. Therefore, the mean value function of traditional NHPP SRGM can only exhibit curves based on parametric forms and therefore cannot predict defects accurately beyond a certain level of precision.

Previous studies have proposed several models and methods to improve the predictive accuracy of NHPP models. Dohi et al. developed new models such as the mixed-type [4], [5], Markov modulated process [6], Hyper-Erlang [7], logistic regression [8], Poisson regression [9], Cumulative Binomial Trial Process [10] as well as additive [11] and covariate [12] models. Other studies focused on development of software reliability models with changepoints to characterize changes and stages in a software testing process. Single changepoint models have been developed for hazard rate [3], [13] and failure count data [14], [15]. The majority of studies present changepoint models for various classes of parametric models such as discrete [16], imperfect debugging [17], fault correction [18], environmental factors [19], [20], test effort [21], and execution time [22] models.

Successive generations of software reliability models possess an increased number of parameters, which requires stable and efficient algorithms for model fitting. Statistical algorithms include the expectation-maximization (EM) [23], expectation conditional maximization (ECM) [24], and hybrid algorithms [25], which combine the EM and ECM algorithms with the Newton-Raphson [26] method. Several studies have also explored application of soft computing techniques [27], including the genetic algorithm [28] and particle swarm optimization (PSO) [29], [30]. More recent studies utilize machine learning algorithms [31], [32], specifically variations of neural networks. Promising results based on the artificial neural network (ANN) led to further studies, including evolutionary neural network [33] and ANN with changepoints [34]. Most recently, focus has shifted towards recurrent neural networks (RNN) [35], [36] for software failure prediction [37] along with several variations, including the combined recurrent ANN model [38] for fault detection and correction, dynamic weighted model [39], and Bayesian regularization [40].

This paper presents an application of simple ANN and RNN to characterize and predict software defects. Two variations of the RNN model are explored, including long-short term memory (LSTM) [41] and gated recurrent unit (GRU) [42]. Traditional NHPP models, including the Goel-Okumoto (GO) [43] and Weibull [44] models are considered in order to compare the predictive accuracy of the neural network models. All models considered are applied in the context of historical software failure data [45]. Results suggests that RNN and its variations exhibit approximately $1.6-37$ and $1.8-68$ times better overall goodness-of-fit measures than ANN and traditional NHPP models when 80% and 90% of the CSR1 data is used for model fitting. Specifically, RNN exhibit $1.6-15$ and $1.8-21$ times better information theoretic measure and $1.7-36$ and $2.5-68$ times better predictive measure with 80% and 90% of data used for model fitting.

The remainder of the paper is organized as follows: Section II reviews models considered in this paper including NHPP SRGM and neural networks. Section III lists several measures to assess the goodness-of-fit of models. Section IV presents numerical examples to compare existing and proposed approaches. Section V offers conclusions and directions for future research.

II. Review of Models

This section provides an overview of software reliability growth models and presents the Goel-Okumoto (GO) [43] and Weibull [44] model. A review of neural network methods, including artificial neural networks (ANN) [46] and recurrent neural networks (RNN) [35] is also provided.

A. Non-homogeneous Poisson Process Software Reliability Growth Model

In the context of software reliability, the non-homogeneous Poisson process is a stochastic process that counts the number of defects discovered as a function of testing time t. The expected value of a NHPP is characterized by a mean value function, $m(t)$, which can be written as

$$m(t) = a \times F(t) \tag{1}$$

where a denotes the number of defects that would be discovered if software was tested indefinitely and $F(t)$ is the cumulative distribution function.

1) Goel-Okumoto (GO) SRGM: The MVF of the GO [43] model is

$$m(t) = a\left(1 - e^{-bt}\right) \tag{2}$$

where parameter b is interpreted as the defect detection rate.

2) Weibull SRGM: The MVF of the Weibull [44] model is

$$m(t) = a\left(1 - e^{-bt^c}\right) \tag{3}$$

where b and c are the scale and shape parameters respectively. Setting $c = 1.0$ simplifies to the GO model of Equation (2).

Parameters of the GO and Weibull models are estimated using maximum likelihood estimation (MLE) [47]. Specific solution techniques include the expectation conditional maximization (ECM) [25] algorithm, which utilizes CM steps to iteratively update the parameters until convergence such that the numerical values of the parameters maximizes the likelihood function.

B. Neural Networks

An artificial neural network (ANN) [46], [36] is a machine learning approach composed of three components: (i) nodes, (ii) a learning algorithm, and (iii) size, shape, and network connections. Nodes, also referred to as neurons, take input values and process them using an activation function and weights to output a new value. Two categories of neural network are discussed, including the ANN and three variations of the RNN.

1) Artificial Neural Network (ANN): An artificial neural network consists of nodes and layers that process information and pass it on to the next layer or set of nodes. Each neuron in the network sums results from the previous layer and inputs it into the activation function

$$f_i(X) = a_i\left(W_{i,j} \cdot X + b_i\right) \tag{4}$$

where $a_i(\cdot)$ is the activation function, $W_{i,j}$ is the weight for input from the jth node in the ith layer, X is the vector of inputs, and b_i is the bias of the ith layer. Figure 1 illustrates a single neuron with pre-activation and activation.

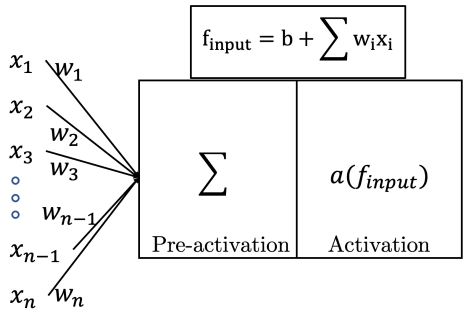


Fig. 1: Example of a neuron in neural network

The result of the activation function is then passed to all the nodes in the succeeding layer.

Figure 2 shows the network considered in this paper, which is made up of an input layer with a single input, a hidden layer with the n nodes, and an output layer with one output value. This model serves as a baseline and facilitates comparison with other models considered in this paper.

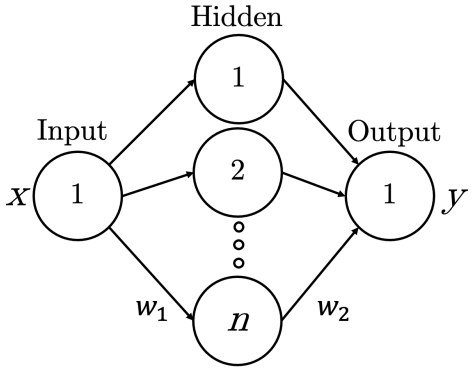


Fig. 2: Example of a simple artificial neural network

The activation in the input and output layers are linear while the hidden layer uses the ELU activation function

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha\ (e^x - 1) & x \leq 0 \end{cases} \tag{5}$$

where $\alpha > 0$ is a positive constant.

The weights of the network are updated iteratively by training on data and using gradient descent like methods. In this paper, Adam[48], a modification of gradient descent, is used to update the weights of the network.

2) Recurrent Neural Network (RNN): The recurrent neural network [35], [36] (RNN) model uses both the input value and the prediction of the previous time step to make predictions. In contrast to the ANN, which takes one input, processes it, and produces an output, the RNN uses a weighted sum of the input and the previous result of the hidden layer as the input to the activation function.

Figure 5 shows an example of a simple recurrent neural network.

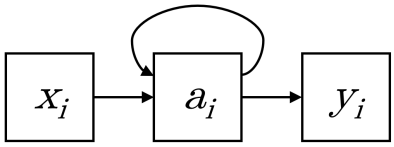


Fig. 3: An example of a simple recurrent neural network

where x_i is the i^{th} input, a_i is the network at the i^{th} time instance, and y_i is the output of the network. The loop pointing back to the network is the recurrent portion of the network. In a single hidden layer recurrent network, the hidden layer combines the input vector with the output of the hidden layer from the previous time step as the weighted sum

$$h_t = ELU\left(W_x \cdot X_t + W_h \cdot h_{t-1} + b_h\right) \tag{6}$$

where h_t is the output of the hidden layer t, ELU is the activation function, W_x and W_h are the weights for the input and the previous result, X_t is the vector of inputs, h_{t-1} is the output from the previous layer, and b_h is the bias.

The final output of the network at each time step is

$$y_t = W_y \cdot h_t + b_y \tag{7}$$

where W_y is the weight and b_y is the bias of the output layer. The simple equation is the result of the output layer possessing a linear activation function.

Long Short-Term Memory (LSTM): Long Short Term Memory (LSTM) networks [41] are a modification of RNN. While RNN accept input and previous state values, the LSTM also takes in a "cell" value, c. The previous cell value, c_{t-1}, previous network output, h_{t-1}, and current input, X_t, are processed using three gates to decide what information is remembered and what is forgotten.

Figure 4 shows an example of a LSTM cell.

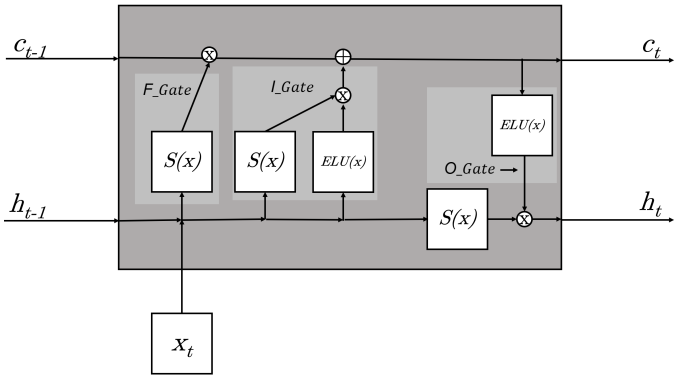


Fig. 4: Example LSTM cell

In Figure 4, $\bigotimes$ denotes vector multiplication and $\bigoplus$ denotes element wise addition. Going from left to right the paths from bottom to the top are the forget gate (F_Gate), input gate (I_Gate), which is made up of two activation functions, and the path at the end going from top to the bottom is the output gate (O_Gate). The forget gate is used to determine what information from previous states should be remembered. The input gate is used to update the cell state. The output gate calculates the prediction of the network and the hidden state.

In addition to the ELU activation function in Equation (5), the LSTM model uses a sigmoid activation function

$$S(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

In Figure 4, the forget gate takes the weighted sum of the input ($W_x \cdot X_t$) and the previous network's output ($W_h \cdot h_{t-1}$) and processes the result using a sigmoid activation function (S). The output of the activation function is then multiplied by the input cell value (c_{t-1}). Thus the forget gate is

$$f_t = c_{t-1} \times S\left(W_x \cdot X_t + W_h \cdot h_{t-1} + b\right) \tag{9}$$

The input gate is made up of two activation functions with the output of the two multiplied together. The first activation function takes the weighted sum of the input and the previous network's output and processes the result using sigmoid activation. The second activation is an ELU activation that takes in the weighted sum of the input and the previous network's output. Thus, the result of the input gate is

$$i_t = S\left(W_x \cdot X_t + W_h \cdot h_{t-1} + b\right) \times ELU\left(W_x \cdot X_t + W_h \cdot h_{t-1} + b\right) \tag{10}$$

The result of the input gate is then summed with the results of the forget gate to obtain the new cell value

$$c_t = f_t + i_t \tag{11}$$

The final gate in the LSTM is the output gate. This gate also uses two activations. The first is a sigmoid activation that takes in the weighted sum of the input and the previous network's output and the second is an ELU activation that takes in the newly updated cell value c_t. The results of each activation are multiplied together to get the final output of the current state.

$$S\left(W_x \cdot X_t + W_h \cdot h_{t-1} + b\right) \times ELU\left(W_c \times c_t + b\right) \quad (12)$$

c_t and h_t are the final outputs of the network with c_t as the cell state that is used in the next time step and h_t is the hidden state and the prediction of the network.

Gated Recurrent Unit (GRU): Gated recurrent unit (GRU) [42] is a recurrent unit similar to the LSTM. The GRU is a simpler architecture which operates using three components, the update gate, z, the reset gate, r, and a candidate hidden state $\tilde{h}$. At a given time step, t, the inputs to the GRU are the hidden layer activation's from the previous time step, h_{t-1} and the input from this time step, X. The output is the hidden layer activation at this time step, h_t. Unlike the LSTM, the GRU lacks the c_t parameter.

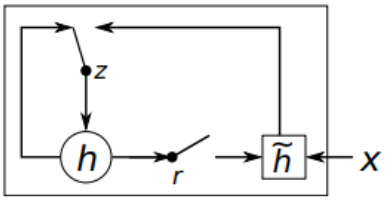


Fig. 5: An illustration taken from [42]

In the network flow illustration above, h represents both the previous activation h_{t-1} and the current activation being calculated h_t. Two loops can be seen, one on the left from h at time $t-1$ through z and back to h at time t, and another from h at time $t-1$ through r to contribute to $\tilde{h}$ before returning back to h at time t through z. The update (z) and reset (r) gates are both vectors with the same dimensions as h and $\tilde{h}$ filled with values between 0 and 1 due to the Sigmoid activation function, S. They can be calculated using equations (14, 13).

$$r_t = S\left(W_r X + U_r h_{t-1}\right) \quad (13)$$

$$z_t = S\left(W_z X + U_z h_{t-1}\right) \quad (14)$$

In the equations above, W denotes a learned weight matrix being applied to the input, X, and U denotes another learned matrix being applied to the previous hidden state.

The reset gate at time t, r_t, is used to determine how much of the previous hidden layer's information will be used in the candidate hidden layer $\tilde{h}_t$. The candidate hidden layer can be calculated using equation (15).

$$\tilde{h}_t = tanh\left(WX + U(r_t \odot h_{t-1})\right) \quad (15)$$

In the equation above, $\odot$ refers to the element-wise, Hadamard, product between two vectors or matrices so that r_t will decide how much influence each activation in h_{t-1} has on calculating $\tilde{h}_t$.

The update gate at time t, z_t, will then be used to determine how much the hidden state, h_t, should be updated from h_{t-1} using the candidate state $\tilde{h_t}$. The calculation of h_t can be seen in equation (16).

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \quad (16)$$

In the above equation, it can be seen that each activation in h_t will be some combination of h_{t-1} and $\tilde{h}_t$ that is decided on an element-wise basis by z_t.

III. Goodness-of-fit measures

This section summarizes goodness of fit measures [49] for model comparison based on information theoretic and predictive accuracy.

A. Mean Squared Error (MSE)

Mean squared error is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \quad (17)$$

where n is the number of input values, $f(x_i)$ is the predicted value and y_i is the actual value.

B. Mean Absolute Percentage Error (MAPE)

Mean absolute percentage error calculates the error as a percentage of how the predicted value deviates from the actual value

$$MAPE = \frac{1}{n} \times \sum_{i=1}^{n}\left|\frac{f(x_i) - y_i}{y_i}\right| \times 100 \quad (18)$$

C. Predictive Mean Squared Error (PMSE)

Predictive mean squared error computes average error between prediction of the model with data not used to perform model fitting.

$$PMSE = \frac{1}{n-k}\sum_{i=k+1}^{n}\left(f(x_i) - y_i\right)^2 \quad (19)$$

where k is the subset of data used for model fitting.

D. Predictive Ratio Risk (PRR)

The predictive ratio risk of a model is

$$PRR = \sum_{i=k+1}^{n}\left(\frac{f(x_i) - y_i}{f(x_i)}\right)^2 \quad (20)$$

where the term in the denominator penalizes underestimation of the number of defects more heavily than an overestimate.

E. Predictive Power (PP)

The predictive power of a model is

$$PP = \sum_{i=k+1}^{n}\left(\frac{f(x_i) - y_i}{y_i}\right)^2 \quad (21)$$

where the term in the denominator penalizes overestimation of the number of defects.

	MSE		MAPE		PMSE		PRR		PP	
Model	80 : 20	90 : 10	80 : 20	90 : 10	80 : 20	90 : 10	80 : 20	90 : 10	80 : 20	90 : 10
ANN	5.40	4.19	5.71	6.11	4.29	3.88	0.60	0.77	0.96	1.70
RNN	2.41	1.95	3.71	3.53	2.54	2.06	0.40	0.32	0.34	0.46
LSTM	**2.04**	2.05	**3.40**	3.36	**2.15**	1.82	0.43	0.38	**0.32**	**0.30**
GRU	4.03	**1.82**	3.79	**3.25**	3.19	**1.54**	**0.34**	**0.30**	0.51	0.40
GO	30.71	38.60	14.11	17.07	28.01	41.16	12.55	20.55	3.66	5.50
Weibull	11.86	12.21	7.00	7.30	12.32	12.77	1.08	1.20	0.89	1.04

TABLE I: Goodness-of-fit comparison on CSR3 data

IV. Illustrations

This section presents examples to illustrate the application of machine learning models specifically ANN and RNN with LSTM and GRU as well as NHPP models, including the GO and Weibull to characterize failure data taken from the Handbook of Software Reliability Engineering [45]. The GO and Weibull models were selected based on results from the SFRAT [50] as the models that best characterized the CSR3 data set used here. These models are further assessed based on several goodness-of-fit measures and runtime.

The ANN uses one hidden layer with ELU activation and 50 nodes. A learning rate of 0.001 was selected to fit the shape of the curve well while avoiding overfitting. The RNN, LSTM, and GRU use eight nodes and ELU and sigmoid activations as defined in Section II. All the models were trained with an upper limit of 2000 epochs with an early stopping condition of a change in loss of less than 0.001 for 10 epochs.

A. Assessment of Model Fit

Figure 6 shows the fit of the NHPP and neural network models presented in Section II when 80% of the CSR3 data was used for training or model fitting and the remainder was used for prediction. Other than the observed data, plots to the right of the vertical line indicate predictions, which is zoomed in the second part of Figure 6 for the sake of more detailed comparison.
The GO and Weibull models shown in Figures 6 exhibit under and over prediction in the training phase compared to both ANN and RNN models. Specifically, the RNN and LSTM characterize the data well and also predict future defects most closely to observed value, as shown in the right part of Figure 6.

Similarly, Figure 7 shows the fit of the NHPP and neural network models as well as a close up view of the predictions when 90% of the CSR3 data is used for training or model fitting and the remainder is used for prediction. RNN in Figure 7 appear to not only characterize the data well but also predict the remaining 10% of the defects with low error value.

B. Performance Assessment

This section compares different models based on the goodness-of-fit measures given in Section III. The models were also applied to the CSR3 data found in the Handbook of Software Reliability Engineering [45]. Table I reports goodness-of-fit measures for each of the six models discussed in Section II, when applied to 80 and 90% percent of the CSR3 data for model fitting and the remaining data used to test predictive accuracy. Models achieving the best goodness-of-fit values are shown in bold. Table I suggests that variations of the RNN, including the LSTM and GRU, achieved significantly lower error values compared to other models.

V. Conclusions and Future Research

This paper assessed the ability of neural network models, including ANN and RNN along with two variations of RNN, including the LSTM and GRU, to predict defects based on data collected during software testing. Neural network models and traditional NHPP models, including the GO and Weibull models were applied to the CSR3 dataset with 80 and 90% of the data for training and the remainder used for testing. The results indicated that RNN and its variations exhibit approximately $1.6 - 37$ and $1.8 - 68$ times better overall goodness-of-fit measures than ANN and traditional NHPP models when 80% and 90% of CSR1 data is used for model fitting respectively. Specifically, RNN exhibit $1.6 - 15$ and $1.8 - 21$ times better information theoretic measures and $1.7 - 36$ and $2.5 - 68$ times better predictive measures when 80% and 90% of the data was used for model fitting.

Future research will explore speed and stability assessment of RNN on more challenging data sets.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant Number 1749635. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Student work was supported by the University of Tulsa & Team8 Cyber Fellows program.

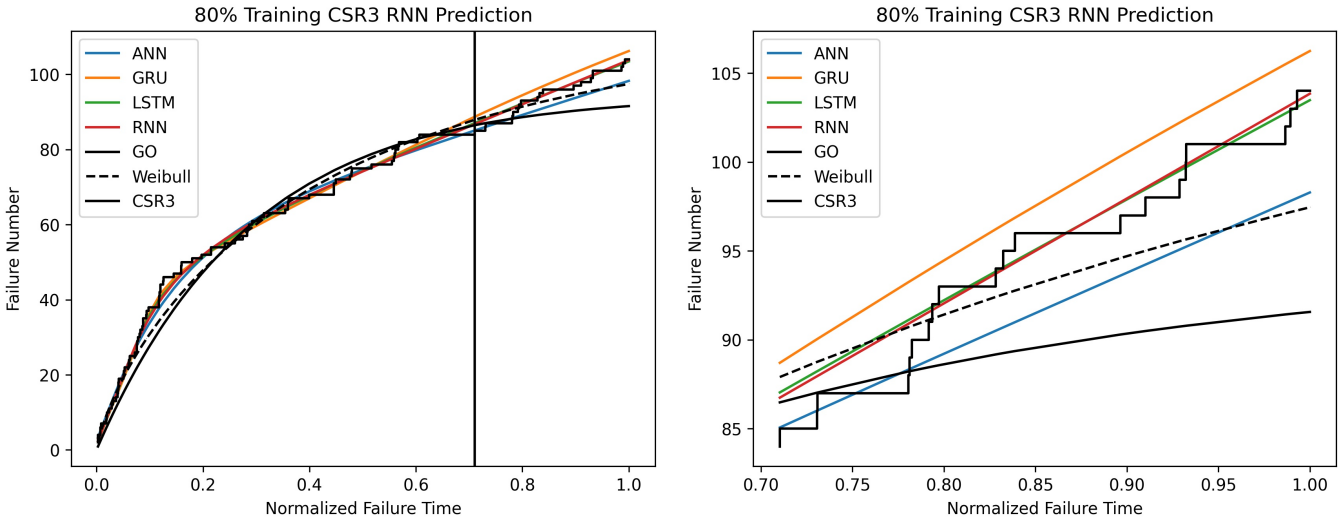
References

[1] S. Ross, *Introduction to Probability Models*, 8th ed. New York, NY: Academic Press, 2003.

[2] W. Farr and O. Smith, "Statistical modeling and estimation of reliability functions for software (SMERFS) users guide," Naval Surface Warfare Center, Dahlgren, VA, Tech. Rep. NAVSWC TR-84-373, Rev. 2, 1984.

[3] M. Zhao, "Change-point problems in software and hardware reliability," *Communications in Statistics-Theory and Methods*, vol. 22, no. 3, pp. 757–768, 1993.

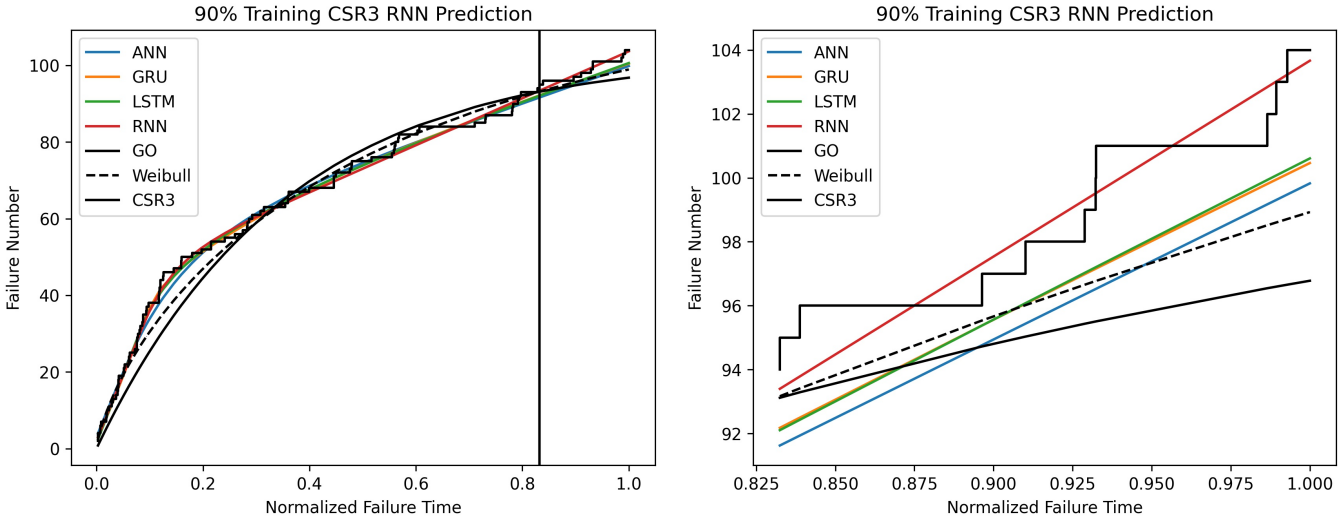
[4] H. Okamura, Y. Watanabe, and T. Dohi, "Estimating mixed software reliability models based on the em algorithm," in *IEEE Proceedings International Symposium on Empirical Software Engineering*, 2002, pp. 69–78.



(a) Model fit

(b) Zoomed view of prediction

Fig. 6: Model fit with train/test split of 80:20



(a) Model fit

(b) Zoomed view of prediction

Fig. 7: Model fit with train/test split of 90:10

[5] H. Okamura, K. Tateishi, and T. Dohi, "Statistical inference of computer virus propagation using non-homogeneous Poisson processes," in *IEEE International Symposium on Software Reliability*, 2007, pp. 149–158.

[6] T. Ando, H. Okamura, and T. Dohi, "Estimating markov modulated software reliability models via em algorithm," in *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, 2006, pp. 111–118.

[7] H. Okamura and T. Dohi, "Hyper-erlang software reliability model," in *IEEE Pacific Rim International Symposium on Dependable Computing*, 2008, pp. 232–239.

[8] H. Okamura, Y. Etani, and T. Dohi, "A multi-factor software reliability model based on logistic regression," in *IEEE International Symposium on Software Reliability Engineering*, 2010, pp. 31–40.

[9] H. Okamura and T. Dohi, "A novel framework of software reliability evaluation with software reliability growth models and software metrics," in *IEEE International Symposium on High-Assurance Systems Engineering*, 2014, pp. 97–104.

[10] H. Okamura, A. Murayama, and T. Dohi, "EM algorithm for discrete software reliability models: a unified parameter estimation method," in *IEEE International Symposium on High Assurance Systems Engineering*, 2004, pp. 219–228.

[11] H. Okamura, Y. Watanabe, and T. Dohi, "An iterative scheme for maximum likelihood estimation in software reliability modeling," in *IEEE International Symposium on Software Reliability Engineering*, 2003, pp. 246–256.

[12] K. Shibata, K. Rinsaka, and T. Dohi, "Metrics-based software reliability models using non-homogeneous poisson processes," in *2006 17th International Symposium on Software Reliability Engineering*, 2006, pp. 52–61.

[13] S. Inoue and S. Yamada, "Software hazard rate modeling with multiple change-point occurrences," in *IEEE International Conference on Industrial Engineering and Engineering Management*, 2014, pp. 1151–1155.

[14] Y.-P. Chang, "Estimation of parameters for nonhomogeneous poisson process: Software reliability with change-point model," *Communications in Statistics-Simulation and Computation*, vol. 30, no. 3, pp. 623–635, 2001.

[15] S. Inoue and S. Yamada, "Software reliability assessment with multiple changes of testing-environment," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 98, no. 10, pp. 2031–2041, 2015.

[16] C.-Y. Huang, C.-T. Lin, and C.-C. Sue, "Software reliability prediction and analysis during operational use," in *IEEE International Conference on Information Technology: Research and Education*, 2005, pp. 317–321.

[17] F.-Z. Zou, "A change-point perspective on the software failure process," *Software testing, verification and reliability*, vol. 13, no. 2, pp. 85–93, 2003.

[18] C.-Y. Huang and T.-Y. Hung, "Software reliability analysis and assessment using queueing models with multiple change-points," *Computers & Mathematics with Applications*, vol. 60, no. 7, pp. 2015–2030, 2010.

[19] J. Zhao, H.-W. Liu, G. Cui, and X.-Z. Yang, "Software reliability growth model with change-point and environmental function," *Journal of Systems and Software*, vol. 79, no. 11, pp. 1578–1587, 2006.

[20] S. Inoue and S. Yamada, "Software reliability growth modeling frameworks with change of testing-environment," *International Journal of Reliability, Quality and Safety Engineering*, vol. 18, no. 04, pp. 365–376, 2011.

[21] C.-T. Lin, C.-Y. Huang, and J.-R. Chang, "Integrating generalized weibull-type testing-effort function and multiple change-points into software reliability growth models," in *IEEE Asia-Pacific Software Engineering Conference (APSEC'05)*, 2005, pp. 8–pp.

[22] V. Singh, P. Kapur, and M. Basirzadeh, "Open source software reliability growth model by considering change-point," *International Journal of Information Technology*, vol. 4, no. 1, p. 405, 2012.

[23] V. Nagaraju, L. Fiondella, P. Zeephongsekul, and T. Wandji, "An adaptive em algorithm for the maximum likelihood estimation of non-homogeneous poisson process software reliability growth models," *International Journal of Reliability, Quality and Safety Engineering*, vol. 24, no. 04, p. 1750020, 2017.

[24] P. Zeephongsekul, C. Jayasinghe, L. Fiondella, and V. Nagaraju, "Maximum-likelihood estimation of parameters of NHPP software reliability models using expectation conditional maximization algorithm," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1571–1583, 2016.

[25] V. Nagaraju, L. Fiondella, P. Zeephongsekul, C. Jayasinghe, and T. Wandji, "Performance optimized expectation conditional maximization algorithms for nonhomogeneous Poisson process software reliability models," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 722–734, 2017.

[26] R. Burden and J. Faires, *Numerical Analysis*, 8th ed. Belmont, CA: Brooks/Cole, 2004.

[27] J. Steakelum, J. Aubertine, K. Chen, V. Nagaraju, and L. Fiondella, "Multi-phase algorithm design for accurate and efficient model fitting," *Annals of Operations Research*, pp. 1–23, 2021.

[28] L. Tian and A. Noore, "Evolutionary neural network modeling for software cumulative failure time prediction," *Reliability Engineering & system safety*, vol. 87, no. 1, pp. 45–51, 2005.

[29] V. Nagaraju and L. Fiondella, "A hybrid model fitting approach incorporating particle swarm optimization and statistical algorithms," *Reliability and Maintenance Engineering Summit*, 2021.

[30] A. Sheta, "Reliability growth modeling for software fault detection using particle swarm optimization," in *Proc. IEEE Congress on Evolutionary Computation*, 2006, pp. 3071–3078.

[31] N. Karunanithi, Y. K. Malaiya, and L. D. Whitley, "Prediction of software reliability using neural networks," in *ISSRE*, 1991, pp. 124–130.

[32] N. Karunanithi, D. Whitley, and Y. K. Malaiya, "Using neural networks in reliability prediction," *IEEE Software*, vol. 9, no. 4, pp. 53–59, 1992.

[33] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl, "Evolutionary neural networks: a robust approach to software reliability problems," in *Proceedings The Eighth International Symposium on Software Reliability Engineering*. IEEE, 1997, pp. 13–26.

[34] N. Gupta and M. P. Singh, "Estimation of software reliability with execution time model using the pattern mapping technique of artificial neural network," *Computers & operations research*, vol. 32, no. 1, pp. 187–199, 2005.

[35] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[37] M. Bhuyan, D. Mohapatra, and S. Sethi, "Prediction strategy for software reliability based on recurrent neural network," in *Computational Intelligence in Data Mining—Volume 2*. Springer, 2016, pp. 295–303.

[38] Q. Hu, M. Xie, S. Ng, and G. Levitin, "Robust recurrent neural network modeling for software fault detection and correction prediction," *Reliability Engineering & System Safety*, vol. 92, no. 3, pp. 332–340, 2007.

[39] P. Roy, G. Mahapatra, P. Rani, S. Pandey, and K. Dey, "Robust feedforward and recurrent neural network based dynamic weighted combination models for software reliability prediction," *Applied Soft Computing*, vol. 22, pp. 629–637, 2014.

[40] L. Tian and A. Noore, "Software reliability prediction using recurrent neural network with bayesian regularization," *International Journal of Neural Systems*, vol. 14, no. 3, pp. 165–174, 2004.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[43] A. Goel and K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Transactions on Reliability,*, vol. 28, no. 3, pp. 206–211, 1979.

[44] S. Yamada and S. Osaki, "Reliability growth models for hardware and software systems based on nonhomogeneous poisson processes: a survey," *Microelectronics Reliability*, vol. 23, no. 1, pp. 91–112, 1983.

[45] M. Lyu, Ed., *Handbook of Software Reliability Engineering*. New York, NY: McGraw-Hill, 1996.

[46] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[47] S. Hossain and R. Dahiya, "Estimating the parameters of a non-homogeneous poisson-process model for software reliability," *IEEE Transactions on Reliability*, vol. 42, no. 4, pp. 604–612, 1993.

[48] Z. Zhang, "Improved adam optimizer for deep neural networks," in *IEEE/ACM 26th International Symposium on Quality of Service*, 2018, pp. 1–2.

[49] D. Kleinbaum, L. Kupper, A. Nizam, and K. Muller, *Applied Regression Analysis and Other Multivariable Methods*, 4th ed., ser. Applied Series. Belmont, CA: Duxbury Press, 2008.

[50] V. Nagaraju, V. Shekar, J. Steakelum, M. Luperon, Y. Shi, and L. Fiondella, "Practical software reliability engineering with the software failure and reliability assessment tool (sfrat)," *Elsevier SoftwareX*, vol. 10, p. 100357, 2019.