Application of Recurrent Neural Networks to Covariate Software Defect Prediction

Fatemeh Salboukh[1], Priscila Silva[2], Vidhyashree Nagaraju[3], and Lance Fiondella[2]
[1] Engineering and Applied Science, University of Massachusetts Dartmouth, MA, USA
[2] Electrical and Computer Engineering, University of Massachusetts Dartmouth, MA, USA
[3]Department of Computer Science, Stonehill College, MA, USA
Email: {fsalboukh, psilva4, and lfiondella}@umassd.edu, vidhyashreenagaraju@gmail.com

***Abstract*—Traditional software reliability growth models (SRGM) assess defect discovery as a function of testing time or effort to quantify failure intensity based on the Non-Homogeneous Poisson Process (NHPP). More recently, covariate NHPP models have substantially improved defect prediction by characterizing software defect discovery in terms of underlying testing activities. Traditional NHPP models are based on parametric forms with specific distributional assumptions, which often fail to capture all details present on defect data. Therefore, this paper assesses the effectiveness of neural networks to predict software defects including covariates to encode factors driving defect discovery explicitly. Two neural networks are considered, including recurrent neural networks (RNNs) and long short-term memory (LSTM), which are then compared with the traditional covariate models to validate predictive capability. Our results suggest that LSTM achieves better overall goodness-of-fit measures, such as approximately 6 and 9 times smaller mean square errors and mean absolute percentage error, respectively, compared to traditional models when 75% of the data is used for training. These results suggest that neural networks are able to track and predict defect trends more accurately than traditional methods.**

***Index Terms*—Software reliability, non-homogeneous Poisson process, covariate models, defect discovery, neural networks**

I. INTRODUCTION

Non-homogeneous Poisson Process (NHPP) [1] software reliability growth models (SRGM)[2] are the most common class of models applied to track and predict software defect trends as a function of testing time or effort to quantify reliability. Covariate models [3] have been proposed in order to enhance software defect prediction by considering the relationship between defects and various testing activities. Machine learning algorithms have also been successfully applied to predict software defect discovery [4]. Given the effectiveness of machine learning on a wide array of prediction problems, it is therefore worthwhile to study the performance of machine learning in the context of covariate software reliability growth models.

For decades, researchers have applied the NHPP to model defect discovery during software testing [5], [6]. Covariate models were implemented to improve defect detection by explicitly considering software testing activities that lead to defect discovery. In this regard, Shibata et al. [7] presented NHPP metrics-based software reliability models to capture the effects of various metrics on software reliability and predict reliability growth. Okamura et al. [8] proposed a software reliability model based on Logistic regression, where the model captures the effects of multiple factors such as testing effort, system complexity, and personnel experience on the software reliability growth process. More recently, Nagaraju et al. [9] presented an optimal test activity allocation approach to maximize software defect discovery. Besides traditional NHPP models, machine learning [10] methods have also been explored for software discovery due to their ability to memorize trends. For example, Khoshgoftaar and Szabo [4] investigated the function of principal-components analysis to reduce the dimensional of multivariate data applied to neural networks to predict the number of faults of software products. Hochman et al. [11] applied Evolutionary Neural Networks (ENNs) to improve software reliability. Guo et al. [12] considered a random forest algorithm to predict software faults. Gupta et al. [13] introduced back-propagation for estimating software reliability. Li et al. [14] presented a Convolutional Neural Network approach to predict software defects. Rathore et al. [15] evaluated different techniques for predicting the number of failures including Artificial Neural Networks, Decision Trees, Naïve Bayes, and Support Vector Machines. However, machine learning methods considering covariates have never been explored for software defect detection.

To encourage the widespread adoption of covariates to software defect detection, this paper assesses the applicability of two neural network methods to predict software defects, including recurrent neural networks (RNNs) and long-short term memory (LSTM). Illustrations are performed to validate these model predictions in comparison with traditional NHPP SRGM, based on goodness-of-fit measures such as mean squared error, and mean absolute percentage error. Our results suggest that LSTMs are capable of predicting software defects with high precision, achieving at least 6 times lower mean square error when compared with NHPP SRGM.

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 highlights several measures to assess the goodness-of-fit of models. Section IV indicates numerical examples to compare existing and proposed methods. Section V offers conclusions and directions for future research.

II. Review of Models

This section reviews the traditional covariate NHPP SRGM based on the discrete Cox proportional hazard model, and the two neural network approaches considered in this paper, including RNN and LSTM.

A. Covariate Software Reliability Growth Model

The discrete Cox proportional hazards NHPP SRGM [9] correlates m covariates to the number of defects discovered in each of n intervals. The matrix $\mathbf{X}_{n\times m}$ quantifies the amount of effort dedicated to each of the m activities in n intervals. For example, $\mathbf{X}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ denotes the amount of each testing activity $(1 \leq j \leq m)$ performed in the i^{th} interval.

The mean value function (MVF) predicts the number of defects discovered up to and including the n^{th} interval given covariates $\mathbf{X}$ according to

$$m(\mathbf{X}) = \omega \sum_{i=1}^{n} p_{i,\mathbf{X}_i} \tag{1}$$

where $\omega > 0$ denotes the number of defects that would be discovered with infinite testing and

$$p_{i,\mathbf{X}_i} = \Big(1 - (1 - h(i))^{g(\mathbf{X}_i;\boldsymbol{\beta})}\Big) \prod_{k=1}^{i-1} (1 - h(k))^{g(\mathbf{X}_k;\boldsymbol{\beta})} \tag{2}$$

is the probability that a defect is discovered in the i^{th} interval, given that it was not discovered in the first $(i-1)$ intervals, $\boldsymbol{\beta}$ is the vector of m parameters contained within the Cox proportional hazards model

$$g(\mathbf{X}_i;\boldsymbol{\beta}) = \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_m x_{im}) \tag{3}$$

and $h(\cdot)$ is the baseline hazard function that can characterize a variety of real phenomena possessing different shapes.

Providing a wider variety of hazard functions increases model functionality, since some hazard functions can characterize certain defect detection processes better than others. Therefore, the Covariate Software Failure and Reliability Assessment Tool (C-SFRAT) [16] implements the covariate models and eight alternative hazard functions. Model results and predictions obtained from the tool suggest that three hazard functions performed well for the data set considered in this paper, including:

- Geometric (GM)
$$h(b) = b \tag{4}$$
where $b \in (0,1)$ is the probability of detecting a defect.
- Negative binomial of order two (NB2)
$$h(i;b) = \frac{ib^2}{1 + b(i-1)} \tag{5}$$
where $b \in (0,1)$ and 2 indicates the order.
- S distribution (S)
$$h(i;r,\rho) = r(1 - \rho^i) \tag{6}$$
where $r \in (0,1)$ is the probability of defect removal and $\rho \in (0,1)$ is the probability of a defect eluding detection in the first interval.

To estimate the parameters of the Covariate Software Defect Detection model, the log-likelihood function [9] is

$$\begin{aligned} LL(\mathbf{X}, \mathbf{k}; \boldsymbol{\gamma}, \boldsymbol{\beta}, \omega) &= -\omega \sum_{i=1}^{n} p_{i,\mathbf{x}_i} + \sum_{i=1}^{n} y_i \ln(\omega) \\ &\quad + \sum_{i=1}^{n} y_i \ln(p_{i,\mathbf{x}_i}) - \sum_{i=1}^{n} \ln(y_i!) \end{aligned} \tag{7}$$

where $\mathbf{X}$ represents a vector of covariates, $\mathbf{k}$ is the number of defects discovered in each interval, $\boldsymbol{\gamma}$ is the vector of model parameters contained in the hazard function, and y_i is the number of defects discovered in the i^{th} interval. Substituting one of the hazard functions specified in Equations (4), (5), and (6), into Equation (2) produces unique log-likelihood functions. Thus, given covariate data $\mathbf{X}$ and the vector of defects discovered in each of the n intervals ($\mathbf{y}_n$), the model fitting step identifies the numerical values of the total number of defects to be discovered (ω), vector of m covariate coefficients ($\boldsymbol{\beta}$), and hazard function parameters ($\boldsymbol{\gamma}$). The maximum likelihood estimates of the model parameters can be obtained by solving $\frac{\partial LL}{\partial \omega} = 0$, $\frac{\partial LL}{\partial \boldsymbol{\beta}} = 0$, $\frac{\partial LL}{\partial \boldsymbol{\gamma}} = 0$.

B. Neural Network

A neural network [17] is a type of machine learning model composed of interconnected artificial neurons that process and transmit information through weighted connections. A neural network consists of three primary elements: (i) the input layer, which embeds input variables; (ii) the hidden layer, which transforms the input features into processed features by applying non-linear functions, allowing the network to make more informed decisions about the data it is processing; and (iii) the output layer, which summarizes the processed data to produce the network's prediction. Neural network models can be considered for a range of tasks such as predicting the number of software defects.

Neural networks consist of three phases: training, validation, and testing. Training refers to the process of adjusting the network's weights and biases based on a portion of the data. Validation evaluates the performance of a neural network on another unseen portion of the data to refine the model parameters. Testing is the last stage, where the trained and validated neural network model is evaluated on the last portion of data not used in the previous steps.

1) Recurrent neural network (RNN) [18]: a type of artificial neural network that allows information from prior inputs, known as lookback period, to influence the subsequent input of the same node, and consequently the output of the neural network. An example of a simple recurrent neural network is shown in Fig. 1.

In a single hidden layer of the recurrent neural network, the hidden layer incorporates X_t with h_{t-1} as the input vector

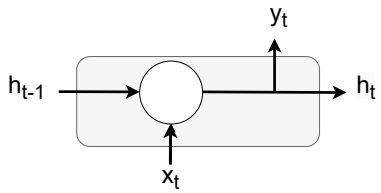

Fig. 1: A simple recurrent neural network

and the output of the hidden layer from the last time step as the weighted sum:

$$h_t = \sigma(W_h \cdot X_t + W_h \cdot h_{t-1} + b_h) \tag{8}$$

in which σ is the activation function, h_t the output of the hidden layer t, W_h the weights matrix, and b_h the bias. The terminal output of the network at each time step t is calculated as:

$$y_t = W_y \cdot h_t + b_y \tag{9}$$

whereas W_y indicates the weight and b_y the bias of the output layer. This equation contains the result of the output layer assuming a linear activation function.

2) Long-short Term Memory (LSTM) [19]: is a more advanced type of recurrent neural network architecture, which possesses both short and long term memories. While the RNN has difficulty to deal with long-term dependencies, the LSTM framework is able to store data in memory for extended periods of time and coordinate better the flow of information through the network. Three gated units in LSTM are used in the repeating modules to remember information for long periods of time, including an input gate, output gate, and forget gate.

Fig. 2 illustrates an LSTM framework, including the three gates present in the LSTM architecture.

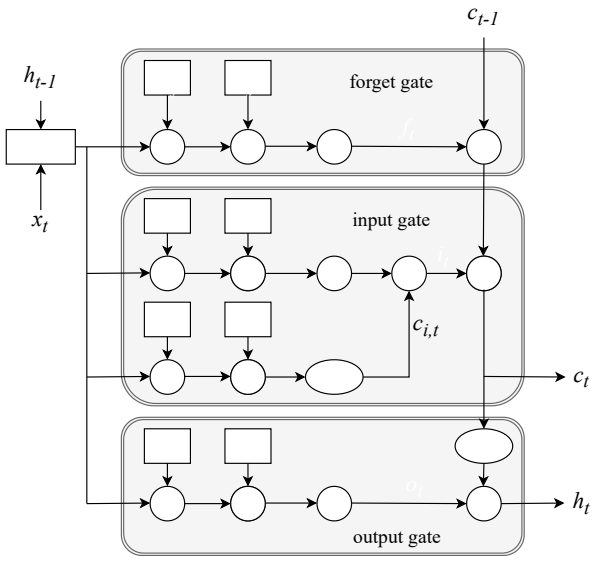

Fig. 2: Repeating modules of LSTM

The weight and bias gradients can be subsequently computed based on the LSTM structure.

- Input Gate: decides what new information to add to the memory cell by

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{10}$$

  where i_t is the input gate, σ the activation function, W_i the weight matrix, $[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input, and b_i the bias term.
- Forget Gate: decides what information from the previous memory cell state will be discarded as

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{11}$$

  where f_t is the forget gate.
- Output Gate: decides what information from the memory cell to output as the hidden state by

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{12}$$

  where o_t is the output.

The memory cell state c_t also shown in Fig 2 is updated according to

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{13}$$

where c_t is the memory cell state and $\tanh$ is the hyperbolic tangent activation function. The hidden state h_t is then computed as

$$h_t = o_t \cdot \tanh(c_t) \tag{14}$$

where h_t indicates the hidden state and corresponds to the number of failures in that specific interval. In each step of the LSTM network, the gates determine what information to store or discard in the memory cell and the hidden state, and the memory cell state is updated based on this information.

III. Goodness-of-Fit measures

This section summarizes goodness-of fit-measures [20] to assess how well the models described in this paper fit a set of data. In practice, there is no single model that performs best with respect to all measures. Hence, goodness-of-fit measures provide quantitative analysis to compare different models, where models with lower error are preferred.

Mean Squared Error (MSE) measures the average squared difference between predicted and actual values computed as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 \tag{15}$$

where n indicates the number of intervals, $f(x_i)$ and y_i are the predicted and actual number of the failures in the i^{th} interval respectively.

Predictive Mean Squared Error (PMSE) computes the sum of squares of the prediction residuals for the remaining $n-k$ observations not used for model fitting as

$$PMSE = \frac{1}{n-k}\sum_{i=k+1}^{n}(f(x_i) - y_i)^2 \tag{16}$$

where k indicates the number of intervals used during training.

Mean Absolute Percentage Error (MAPE) calculates the error as a percentage of how the predicted value deviates from the actual value as

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{f(x_i)-y_i}{y_i}\right| \times 100 \tag{17}$$

Predictive Ratio Risk (PRR) penalizes the difference between the predicted and actual number of defects as

$$PRR = \sum_{i=k+1}^{n}\left(\frac{f(x_i)-y_i}{f(x_i)}\right)^2 \tag{18}$$

where underestimation is penalized more than overestimation.

Predictive Power (PP) measures the distance between the estimated and actual number of defects as

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

where the term in the denominator penalizes overestimation.

IV. Illustrations

To illustrate the applicability of neural networks to covariate software defect discovery, a new data set was generated based on the DS1 data from Shibata et al. [21], with the goal of increasing the number of observations. The steps to generate data for training are:

- (S.1) Step 1 applied the covariate NHPP model described in Section II-A to fit the DS1 data, which contains 17 intervals and 3 covariates. By using the C-SFRAT tool to test eight different hazard functions, results suggested that this data set was best characterized by the model with geometric hazard function. The parameters estimated by maximum likelihood estimation for the geometric data (Eq. (7)) were $\widehat{\beta}_1 = 0.038$, $\widehat{\beta}_2 = 0.174$, $\widehat{\beta}_3 = 0.203$, and $\widehat{b} = 0.018$.
- (S.2) Step 2 generated 249 intervals of three hypothetical covariates as random variables following a uniform distribution with the same range of each actual covariate present in the DS1 data.
- (S.3) Step 3 generated 249 intervals of a Poisson distribution random variable using a mean described by the mean value function (Eq. (1)), where $\omega = 5000$ was chosen, assuming that a higher number of failures should be found in a long period of time, $\boldsymbol{\beta}$ vector assumes the estimators found in (S.1), and $\mathbf{x}_i$ the covariates generated in (S.2). These Poisson distributed random variable values are treated as the number of defects discovered in each interval.

The proposed models are illustrated using the data set generated with the steps above. For each neural network model, the number of hidden layers considered varied from 1 to 3, and the number of neurons in each hidden layer varied between 30 and 60. The models were trained with the ReLU activation function and a maximum of 200 epochs, where a dropout condition of 0.2 was assumed to avoid the overfitting problem. Different values of the learning rate were also considered, including 0.05, 0.01, and 0.001. The data set was split into three parts, Train-Validation-Test. A $75-5-20\%$ split was considered, since 75% is typically assumed for training in the machine learning community, and 5% was used for validation due to the need to have a small validation set to tune the model's hyperparameters and prevent overfitting during training. The hyper-parameters that achieved a good model fit are shown in Table I.

TABLE I: Models hyperparameters

	Hidden Layer	Neurons	Learning rate
RNN	2	$55-35$	0.001
LSTM	3	$55-45-30$	0.001

To assess how well neural network models predict software defects, these models are compared to the traditional NHPP models. Since the validation step is used to refine the training of neural networks, 80% of the data was considered training covariate models. Hence, to conduct an objective comparison, the remaining 20% of the data set was used to compare models based on machine learning and maximum likelihood estimation.

As noted in Section II-A, the C-SFRAT suggested that the geometric (GM), second order negative binomial (NB2), and S distribution (S) hazard functions performed best in the data generated. Table II compares the goodness of fit achieved by these model variants and the proposed neural network models.

TABLE II: Goodness of fit measures for model comparison

	MSE	PMSE	MAPE	PRR	PP
GM	10411.0	904.7	5.6	0.00193	0.00178
S	10921.8	678.0	6.9	0.00142	0.00141
NB2	19617.0	1420.2	5.6	0.00291	0.00294
RNN	4081.4	3066.4	1.2	0.00601	0.00612
LSTM	**1621.2**	**367.5**	**0.7**	**0.0007**	**0.0007**

Table II indicates that the S model performed best among all three hazard functions considered. However, the LSTM model achieved a better fit, decreasing MSE by a factor of $(\frac{10921.8}{1621.2}) = 6.73$, as well as achieving similar improvements in PMSE (3.86), MAPE (9.85), PRR (2.02), and PP (2.02).

For a graphical illustration, the model fits achieved by the S model and LSTM are compared in Fig. 3, and Fig. 4 is a close up view of last 20% of the data used for predictions. In Figure 3, it is important to notice that the first 10 intervals are not shown for the LSTM model fit since a $10-$lookback period is needed for the model predictions. In general, these models demonstrate the superior forecasting capabilities of neural networks compared to traditional models. Fig. 4 suggest the potential for the LSTM to capture small details in the defect discovery trend.

V. Conclusion

This study explored the applicability of neural network models, including recurrent neural network and long-short term memory, to covariate software defect prediction. Both

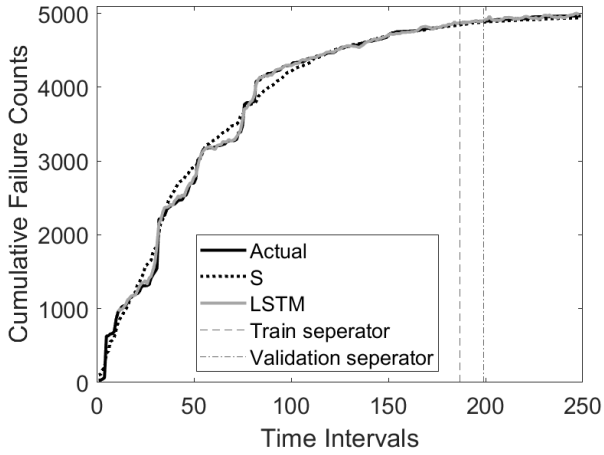

Fig. 3: The model fitted by best models of two approaches; S and LSTM

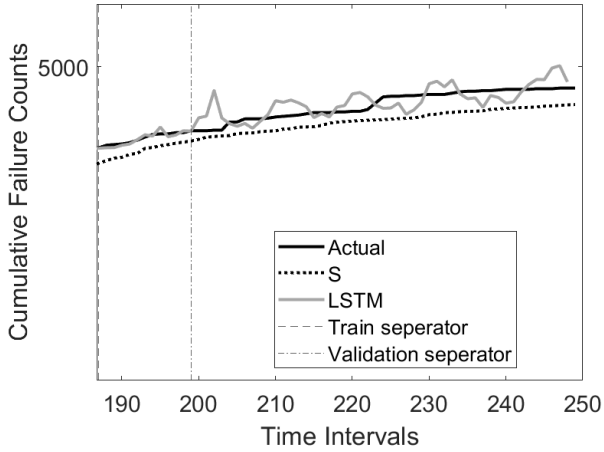

Fig. 4: Zoomed view of test part

neural network models considered were compared with the traditional covariate NHPP models possessing different hazard functions. Our results show that LSTM outperformed the traditional models in all goodness of fit measures considered, especially the mean square error and the mean absolute percentage error, where the measures achieved 6.73 and 9.85 times smaller values than the traditional models. This fact can be explained due to the LSTM's ability to store information in memory for extended periods, making them suitable for forecasting sequential software defect data.

Future studies will examine the effectiveness of feature selection methods to evaluate the stability of RNN and its variations on more complex data sets including more covariates.

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.

REFERENCES

[1] S. Ross, *Stochastic Processes*, ser. Probability and Statistics Series. Wiley, 1983, no. v. 1. [Online]. Available: https://books.google.com/books?id=Hj7bAAAAMAAJ

[2] W. H. Farr and O. D. Smith, "Statistical modeling and estimation of reliability functions for software (smerfs) user's guide. revision 3," 1993.

[3] K. Rinsaka, K. Shibata, and T. Dohi, "Proportional intensity-based software reliability modeling with time-dependent metrics," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, 2006, pp. 369–376.

[4] T. Khoshgoftaar and R. Szabo, "Using neural networks to predict software faults during testing," *IEEE Transactions on Reliability*, vol. 45, no. 3, pp. 456–462, 1996.

[5] S. Yamada and S. Osaki, "Software reliability growth modeling: Models and applications," *IEEE Transactions on software engineering*, no. 12, pp. 1431–1437, 1985.

[6] W. H. Farr, "Software reliability modeling survey," 1996.

[7] 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.

[8] H. Okamura, Y. Etani, and T. Dohi, "A multi-factor software reliability model based on logistic regression," pp. 31–40, 2010.

[9] V. Nagaraju, C. Jayasinghe, and L. Fiondella, "Optimal test activity allocation for covariate software reliability and security models," *Journal of Systems and Software*, vol. 168, p. 110643, 2020.

[10] T. M. Mitchell and T. M. Mitchell, *Machine learning*. McGraw-hill New York, 1997, vol. 1, no. 9.

[11] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl, "Evolutionary neural networks: a robust approach to software reliability problems," pp. 13–26, 1997.

[12] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," *15th International Symposium on Software Reliability Engineering*, pp. 417–428, 2004.

[13] 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.

[14] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 318–328.

[15] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Computing*, vol. 21, pp. 7417–7434, 2017.

[16] J. Aubertine, K. Chen, V. Nagaraju, and L. Fiondella, "A covariate software tool to guide test activity allocation," *SoftwareX*, vol. 17, p. 100909, 2022.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[18] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*, 1987, pp. 318–362.

[19] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[20] H. Pham and C. Deng, "Predictive-ratio risk criterion for selecting software reliability models," in *Proceedings of the 9th International Conference on Reliability and Quality in Design*, 2003, pp. 17–21.

[21] K. Shibata, K. Rinsaka, and T. Dohi, "Metrics-based software reliability models using non-homogeneous poisson processes," pp. 52–61, 2006.