International Journal of Mathematical, Engineering and Management Sciences
Vol. 10, No. 5, 1192-1217, 2025
https://doi.org/10.33889/IJMEMS.2025.10.5.057

Ram Arti Publishers

DNN-based Software Reliability Model for Fault Prediction and Optimal Release Time Determination

Shikha Dwivedi
Subir Chowdhary School of Quality and Reliability,
Indian Institute of Technology, Kharagpur, West Bengal, India.
Corresponding author: shikhadwivedi1969@iitkgp.ac.in

Neeraj Kumar Goyal
Subir Chowdhary School of Quality and Reliability,
Indian Institute of Technology, Kharagpur, West Bengal, India.
E-mail: ngoyal@hijli.iitkgp.ac.in

Hariom Chaudhari
Subir Chowdhary School of Quality and Reliability,
Indian Institute of Technology, Kharagpur, West Bengal, India.
E-mail: chaudhariharіomvijay@kgpian.iitkgp.ac.in

(Received on January 8, 2025; Revised on April 8, 2025; Accepted on April 22, 2025)

Abstract
The accurate prediction of both detected and corrected faults is crucial for enhancing software reliability and determining optimal release times. Traditional Software Reliability Growth Models (SRGMs) often focus on either fault detection or correction, potentially overlooking the comprehensive view needed for effective software maintenance. This paper introduces a Dense Neural Network (DNN)-based model that predicts both detected and corrected faults using data from the initial testing phase. The proposed model adopted a simpler architecture to reduce computational overhead and minimize time complexity, making it suitable for real-world applications. By incorporating logarithmic encoding, the model effectively manages missing data and performs well with smaller datasets, which are common in early testing stages. The proposed model is compared with existing approaches, demonstrating superior results across multiple datasets. This comparative analysis highlights the model's enhanced predictive accuracy, computational efficiency, and less time complexity. Additionally, the predicted faults are used to determine the optimal release time, based on the customer's reliability requirements and the minimum cost necessary to achieve that reliability. By offering a more comprehensive and accurate prediction of software reliability, this model provides a practical solution for software development teams, facilitating better decision-making in testing, maintenance, and release planning.

Keywords- Software reliability, Faults prediction, Artificial neural networks, Logarithmic encoding, Detected faults, Corrected faults.

Abbreviations

SRGM	Software Reliability Growth Model
NHPP	Non-Homogeneous Poisson Process
FRF	Fault Reduction Factor
ANN	Artificial Neural Network
DNN	Dense Neural Network
RNN	Recurrent Neural Network
CNN	Convolutional Neural Networks
EDRNN	Encoder-Decoder RNN
LSTM	Long Short-Term Memory
BSO-LAHC	Brainstorm Optimisation and Late Acceptance Hill-Climbing
ReLU	Rectified Linear Unit
MSE	Mean Squared Error
MAPE	Mean Absolute Percentage Error
MAE	Mean Absolute Error

1. Introduction

The increasing complexity of modern software systems has made it crucial to ensure their reliability and minimise faults. Software faults can lead to system failures, data corruption, security vulnerabilities, and other serious issues, particularly in mission-critical applications. Software reliability is a critical aspect of software quality, impacting safety, performance, and development costs. Thus, there is a pressing need for efficient approaches to predict software faults for ensuring reliable software. To address software reliability, researchers have categorized models in Parametric and Non-Parametric models. Parametric models, such as Goel-Okumoto, Jelinski Moranda, Delayed S-shaped etc, are based on predefined mathematical functions that describe the software failure process over time. These models assume that fault detection follows a known probability distribution, such as the Binomial or Poisson type. While parametric models provide analytical insights into software failure behavior, they often rely on restrictive assumptions, such as immediate and perfect fault correction, which may not reflect real-world testing and debugging conditions (Dwivedi and Goyal, 2024). On the other hand, non-parametric models leverage data-driven techniques, such as machine learning and neural networks, to model software reliability without assuming an explicit mathematical structure. These models can learn complex patterns from historical data and adapt to varying fault detection and correction trends. While both approaches have contributed to reliability prediction, they suffer from key limitations that hinder their practical effectiveness.

Over the years, researchers have proposed enhanced parametric models to address the shortcomings of traditional NHPP-based SRGMs (Huang et al., 2022; Li and Pham, 2017; Li et al., 2022). A key improvement has been the integration of fault reduction factors (FRF) to better represent the dynamics of fault detection and correction. For example, Xie et al. (2007) introduced an exponential FRF, while Hsu et al. (2011) considered time-variable fault reduction factor to reflect varying debugging efficiencies. Another critical area of improvement has been the incorporation of imperfect debugging and fault dependency. Pachauri et al. (2015) proposed three software reliability models, one considering perfect debugging and two considering imperfect debugging. Chatterjee and Shukla (2016) introduced a Weibull curve to capture the behaviour of the Fault Reduction Factor (FRF) in software reliability growth models to relieve the limitation of remaining faults of the software. To enhance the practical applicability of SRGMs, researchers have also explored multi-release modeling. Kumar et al. (2016) introduced a two-dimensional multi-release software reliability model that focuses on fault detection and correction processes using a Cobb-Douglas production function. Peng and Zhai (2017) introduced a modelling framework for software fault detection and correction processes, considering fault dependency. They proposed models considering various debugging lags and determined the optimal software release policy within this framework. Pradhan et al. (2022) assumed an s-shaped fault reduction factor and extended the proposed model to predict the optimal release policy. Dhaka and Nijhawan (2024) considered the environmental effect on the debugging process while modelling fault detection and correction by integrating a change point-based fault reduction factor. However, Bibyan et al. (2023) used stochastic differential equation to handle the random effect of testing coverage while modelling multi-release software model. In Wang et al. (2024) proposed an open-source software reliability model considering three-parameter lifetime distribution-based fault detection. Dwivedi and Goyal (2025) proposed a multi-version software reliability model considering changing failure patterns across multiple versions of software and used different distributions for modelling the failure rates of each version depending upon its behaviour.

Despite these advancements, parametric SRGMs still face fundamental limitations. Over time, researchers have introduced additional parameters into these models to better represent fault detection and correction processes. While this has improved their accuracy, it has also made them more complex, increasing the risk of overfitting and reducing their ability to perform well for all software projects. Additionally, these models address fault correction delays, they generally assume that all faults have equal severity, failing to account

for the prioritization required for debugging faults with varying impacts. Moreover, parametric SRGMs often require extensive historical failure data for accurate parameter estimation, limiting their effectiveness for new or evolving projects. Their inability to handle large-scale Agile projects with frequent updates and changing failure pattern further constrains their practical applicability.

These challenges have motivated researchers to explore non-parametric machine learning approaches for reliability assessment. Non-parametric models do not rely on predefined assumptions about fault detection and correction rates, allowing them to learn complex, non-linear patterns directly from historical failure data. Early research in this domain applied feedforward backpropagation networks and other feedforward architectures to model software reliability (Amin et al., 2013; Ho et al., 2003; Karunanithi et al., 1991), but these models lacked the ability to capture temporal dependencies in fault trends. Although, Bisi and Goyal (2015) improved prediction accuracy using particle swarm optimization-based ANN models, they did not incorporate fault correction, limiting their applicability in real-world scenarios. To address this limitation, Hu et al. (2007) have explored fault correction processes using recurrent neural network (RNN) architectures like Elman neural networks to predict detected and corrected faults**.** Wang and Zhang (2018) introduced a deep encoder-decoder model to predict the number of software faults by leveraging their ability to capture temporal patterns and handle sequences of unequal length.

Other studies have integrated testing effort considerations into fault prediction models. Xiao et al. (2020) used dilated casual convolution, a temporal convolutional network method for stepwise fault prediction. Li et al. (2022) introduce an attention-based encoder-decoder RNN (EDRNN) for software fault prediction. Long short-term memory has been used as the encoder-decoder layer with detection time as time series input to predict a cumulative number of faults. Raamesh et al. (2022) proposed a stepwise fault prediction model using brainstorm optimization and a late acceptance hill-climbing algorithm (BSO-LAHC). Their LSTM-based model demonstrated a 27% improvement in fault correction prediction and a 32% enhancement in fault detection prediction compared to existing approaches but it is dependent on hyperparameter tuning and extensive training data. Samal and Kumar (2024) proposed deep neural network model considering six different hidden layers with varying neutrons ranging between 10000, 5000, 2500, 1000, 500, 250 respectively for each hidden layer. Their approach demonstrated the improvements in fault detection and correction processes compared to existing models.

However, these models are computationally intensive and required large datasets for effective learning. In cases where data is sparse, imbalanced, or noisy, these models may fail to generalize well, leading to unreliable predictions. Computational complexity is another drawback. Training deep learning models, such as LSTM or encoder-decoder architectures, requires significant computational resources. Hyperparameter tuning, architecture selection, and optimization processes further increase the complexity. Furthermore, most existing non-parametric approaches primarily focus on fault detection, with limited attention given to fault correction modeling. While some studies have attempted to address this gap, the challenge of integrating detection and correction in a unified framework remains. Additionally, ANN-based models may struggle to adapt to evolving software development practices and changing operational environments, as they typically require retraining with updated datasets to maintain accuracy.

Both parametric and non-parametric models have contributed significantly to software reliability modeling, but they face key limitations that need to be addressed for better fault prediction and reliability assessment as summarized below:

- Traditional SRGMs, which rely on predefined distributions such as exponential, Weibull, or logistic, often struggle to accurately model the evolving fault patterns of dynamic software development approaches like Agile.

- SRGMs incorporate multiple parameters (e.g. fault reduction factors, environmental effects, imperfect debugging and multi-release dependencies) which increases their complexity and prone to overfitting.
- Existing non-parametric approaches, especially deep learning models, require large datasets for accurate prediction, but fault data is limited and spans only for two to four weeks, restricting effective training.
- Advanced deep learning models (e.g. RNNs, LSTMs, attention-based networks) involve extensive hyperparameter tuning, high time complexity, and longer processing times.
- Many ANN-based models focus on predicting the number of detected faults but overlook the fault correction process, which is crucial for assessing software reliability comprehensively.
- Existing models trained on specific datasets may not perform well when applied to different software projects, requiring extensive retraining and parameter optimization.

Given the challenges associated with both parametric and non-parametric models, there is a critical need for a more efficient and simpler software reliability model that can effectively integrate fault detection and correction processes without relying on unrealistic assumptions and complex deep neural architectures. The proposed Dense Neural Network (DNN) based non-parametric model that addresses the limitations of existing models by simplifying the learning process, reducing time complexity and handling missing data using logarithmic encoding. This model integrates fault detection and correction processes while avoiding the complexities of deep neural architectures, making it more suitable for real-world software development. Specifically, the model offers the following benefits:

- The proposed model explicitly incorporates both fault detection and fault correction without using the complex architecture and excessive layers, reducing the training costs, and processing time.
- The proposed DNN-based approach achieves better prediction accuracy while reducing time complexity, as validate using four existing parametric and non-parametric models.
- The proposed model reduces the risk of overfitting by employing early stopping, which is common concern in deep learning-based reliability models.
- The proposed model is designed to perform well with both large and smaller datasets, as validated using five different datasets ensuring adaptability with varying length of failure data.
- In addition to fault prediction, the model also assists in optimal release time determination, helping project managers make informed release time-to-market by balancing reliability with cost constraints.

By addressing these limitations, the proposed model seeks to provide a more practical and accessible solution for software reliability prediction. This will enable developers to leverage the benefits of data-driven software reliability modeling without encountering the complexities often associated with deep learning approaches. The subsequent sections of the paper cover the Proposed Model in Section 2, followed by model implementation and validation criteria in Section 3. Next, the results and discussion of the model implementation are discussed in Section 4. Furthermore, in Section 5, observation and comparison of the proposed model with existing work are discussed in detail to provide a better understanding. Then, in Section 6, the optimal release time determination is done based on cost and reliability. Additionally, the Limitations and the possibility of future work are presented in Section 7. Finally, the conclusion of the work is discussed by summarising the findings and implications in Section 8.

2. Proposed Model

This section outlines the methodologies employed in the development of the proposed machine learning-based model for software fault prediction. It provides a comprehensive overview of the techniques used, including the architectural design of the Dense neural network (DNN) model, data preprocessing steps, model training procedures, and validation strategies.

2.1 Proposed Model Architecture

The proposed software reliability model uses a dense neural network architecture to predict both detected and corrected software faults, addressing key limitations of existing models. The architecture is designed to enhance prediction accuracy while effectively handling missing data and variations in input parameters through logarithmic encoding as shown in **Figure 1**. The architecture of the model consists of a sequential neural network with the following layers:

- ***Input layer*:** The model takes input parameters such as time, cumulative detected faults, and corrected faults.

- ***Hidden layers*:** The model architecture includes two dense (fully connected) hidden layers, with 64 and 32 neurons respectively and ReLU (Rectified Linear Unit) activation functions:
  - *Dense Layer* 1: This layer captures complex, non-linear relationships between the input features. The ReLU activation function introduces non-linearity in the model, allowing it to learn intricate patterns within the data.
  - *Dense Layer* 2: This layer further refines the feature representations learned by the first dense layer. The additional depth allows the model to capture interactions between input features.

- The output layer of the proposed DNN model is designed to provide predictions for the cumulative number of detected and corrected software faults. This layer consists of two neurons, one dedicated to detected faults and the other to corrected faults. By using a linear activation function the model can accurately forecast the cumulative faults.

2.2 Data Preprocessing

Before training the DNN model, several data preprocessing steps are performed:

2.2.1 Feature Scaling Using Logarithmic Encoding

The input layer applies logarithmic encoding to the raw input parameters. This is essential for managing wide variations in input data and mitigating the impact of missing values. By converting input features into a logarithmic scale, the model can better capture the underlying patterns in the data, leading to more accurate predictions. To make sure that a Dense Neural Network (DNN) learns effectively, it's crucial to scale its inputs within a specific range, usually between 0 and 1. This scaling ensures that the network can process the data optimally. One way to achieve this is by using an encoding function. This function converts the actual value, such as testing time, into a value within the range of 0 to 1, allowing the DNN to work efficiently with the data.

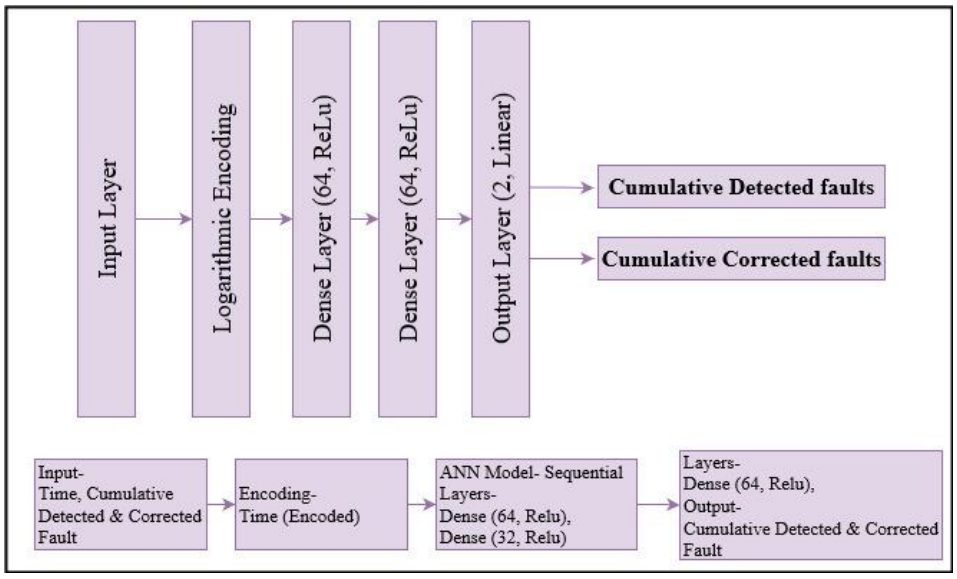

Figure 1. Model architecture.

The input variable is encoded using a logarithmic transformation to account for the non-linear nature of the data using following function:

$$x^* = ln(1 + \beta x) \quad (1)$$

Here, β is the encoding parameter, x is input (in proper units) and $x*$ is encoded value. However, the value of x^* need to be confined within range [0, 1] for input parameters by selecting a proper value of β. Let, x_{max} is the maximum value of input variable (in proper units) and x^*_{max} is the maximum value of the encoded input. Taking these values, value of β is determined as following:

$$\beta = \frac{(exp(x^*_{max}) - 1)}{x_{max}} \quad (2)$$

The experiments done by (Bisi and Goyal, 2016) shows that choosing x^*_{max} within the range of 0.85 to 0.96 for the logarithmic encoding leads to consistently lower Mean Absolute Percentage Error (MAPE) across diverse datasets. This indicates that scaling the data within this particular range allows the neural network to learn more effectively. By utilizing logarithmic encoding with a carefully chosen β and x^*_{max}, we ensure that the normalized data falls within the desired range while preserving valuable information about the original data distribution.

This encoding technique transforms the input values using the logarithm function, which helps in several ways:

- ***Handling missing data*:** Logarithmic encoding reduces the skewness of input data distributions, making the model more robust to missing data. When data points are missing or incomplete, the transformed values provide a more stable input to the model, reducing the impact of these gaps on overall prediction performance.
- ***Managing variation in input*:** Software testing data often exhibit high variability, with testing done in days, weeks, CPU hours etc. Logarithmic encoding compresses the range of input values, ensuring that large variations do not disproportionately influence the model and helps the network learn more effectively from the data.

2.2.2 Dataset Splitting

To evaluate the performance and robustness of the proposed software reliability model, we conducted experiments using three different data splitting ratios: 60-40, 70-30, and 80-20. These ratios represent the proportions of the dataset allocated to the training and testing sets, respectively. Through extensive experimentation, we found that the 80-20 split ratio yielded the best performance across all datasets. This ratio provided a substantial amount of data for training, enhancing the model's learning process, while still maintaining a sufficient testing set to validate its performance.

2.3 Model Training

The DNN model is trained using a systematic and efficient process designed to maximize predictive accuracy while minimizing computational overhead. Here are the key steps involved in the training process:

2.3.1 Data Preparation

The training dataset is divided into input features (X_train) and target variables (Y_train). The input features consist of time, detected faults and corrected faults, while the target variables are the cumulative detected and corrected faults.

2.3.2 Model Compilation

The model is compiled using the Adam optimizer, a popular choice for training neural networks due to its adaptive learning rate and efficient handling of sparse gradients. The mean squared error (MSE) is selected as the loss function to measure the difference between the predicted and actual values of cumulative detected and corrected faults. MSE is particularly suitable for regression tasks, providing a clear objective for the model to minimize prediction errors.

2.3.3 Early Stopping

To prevent overfitting, early stopping is employed during the training process. This technique monitors the loss function on the validation dataset and halts training if the loss does not improve for a specified number of epochs. By early stopping, the model avoids overfitting to the training data, ensuring better generalization to unseen data.

2.3.4 Training Process

The model is trained for a fixed number of epochs or until the early stopping criteria are met. This iterative process involves feeding batches of training data through the model, adjusting the weights and biases to minimize the loss function. The use of two dense layers with Rectified Linear Unit (ReLU) activation functions promotes faster training times and lower computational requirements compared to more complex deep learning architectures like RNNs or CNNs. This simplicity allows for efficient training while still capturing the essential patterns in the data.

3. Model Implementation and Validation Criteria

In this section, the proposed model has been implemented on actual datasets to predict future faults. Following the methodology discussed in Section 2.1, 80 percent of the data has been utilized for training the model. The results obtained from this implementation are evaluated using four error metrics: Mean Squared Error (MSE), Bias, Variance, and Adjusted R-Squared. Additionally, the performance of the proposed model is compared with existing models. The existing models are also implemented on these datasets, and the predicted faults are compared to assess their performance relative to the proposed model. The detailed description of the datasets used, evaluation criteria, and comparative models is provided in the subsequent subsections. This comprehensive evaluation highlights the effectiveness of the proposed model

in predicting both detected and corrected faults, showcasing its advantages in terms of prediction accuracy, time complexity, and computational requirements.

3.1 Datasets

For the training and evaluation of our proposed software reliability model, we utilized five project datasets containing historical software testing data. These datasets provide a rich source of information essential for building an accurate and reliable prediction model. The key attributes in each dataset include:

- *Testing time* (t): The time elapsed during the software testing process.
- *Cumulative number of faults detected* (D_t): The total number of faults detected up to a given testing time.
- *Cumulative number of faults corrected* (C_t): The total number of faults corrected up to a given testing time.

The study utilized five project datasets containing historical software testing data for model training and evaluation naming:

i. Jira API V2.1
ii. Jira API V2.2
iii. Jira API V2.3
iv. Firefox 3
v. Firefox 3.5

Each dataset offers a detailed account of testing times, cumulative detected faults, and cumulative corrected faults, which are crucial for training the proposed DNN-based model. Detailed descriptions of first three datasets are provided in the Appendix (**Tables 15** to **17**) and last two datasets are obtained from the existing research (Xiao et al., 2020).

3.2 Validation Criteria

The proposed DNN-based SRGM is evaluated against existing parametric and non-parametric models. This section presents a criteria used (**Tables 1** and **2**) for comparative analysis of the proposed model on above dataset, highlighting the advantages of the proposed model in terms of prediction accuracy, adaptability, and its ability to predict fault corrected.

3.2.1 Evaluation Criteria

To assess the effectiveness of the proposed model compared to existing approaches, we employ six different error criteria

- ***Mean squared error* (*MSE*):** Measures the average squared difference between predicted and actual values. Lower MSE indicates better prediction accuracy.
- ***Bias*:** Evaluates the systematic tendency of the model to under- or overestimate the actual values. Ideally, a model should exhibit minimal bias.
- ***Variance*:** Assesses the spread of the predictions around the average. Lower variance signifies higher consistency in the model's predictions.
- ***Adjusted R-squared*:** An adjusted version of the R-squared statistic that accounts for the number of model parameters. Values closer to 1 indicate a better fit between the model and the data.

3.2.2 Comparative Models

We compare the proposed DNN-based SRGM with four existing models as summarized in **Table 2**:

- ***Xiao model*** (Xiao et al., 2020) ***and Hu model*** (Hu et al., 2007)**:** These are non-parametric models utilizing neural networks for predicting detected and corrected faults.
- ***Dhaka model*** (Dhaka and Nijhawan, 2024) ***and Li model*** (Li and Pham, 2017)**:** These are statistical software reliability models designed for fault correction.

Therefore, the comparison with these models (**Table 2**) is carried out for predicting detected and corrected faults to ensure a fair comparison.

Table 1. Error metrics.

Error metrics
Mean squared error
Bias
Variance
Adjusted R squared

Table 2. Comparison model.

Comparison model
Xiao model (Xiao et al., 2020)
Hu model (Hu et al., 2007)
Dhaka model (Dhaka and Nijhawan, 2024)
Li model (Li and Pham, 2017)

4. Results and Discussion

This section presents the comprehensive results obtained from the evaluation of the proposed software fault prediction model. It provides a detailed analysis of the Dense neural network (DNN) model's performance in predicting both detected and corrected software faults across various datasets. The results are presented through a combination of comparison tables and prediction graphs, allowing for a clear understanding of the model's predictive capabilities.

4.1 Dataset 1 (Jira API V2.1)

In the first experiment with the Jira API V2.1 dataset, the proposed model, as well as the comparison models (Xiao Model (Xiao et al., 2020), Hu Model (Hu et al., 2007), Dhaka Model (Dhaka and Nijhawan, 2024), and Li Model (Li and Pham, 2017)), are implemented to predict detected and corrected faults. The proposed model required 100 epochs and a batch size of 8 to reach the optimal solution. In contrast, the Xiao model required 250 epochs with a batch size of 28, and the Hu model required 300 epochs with a batch size of 20. The parametric models (Dhaka Model, Li Model), which rely heavily on initial parameter settings, struggled to provide optimal solutions despite multiple attempts to adjust these parameters. The prediction error remained high for parametric models as shown in **Tables 3** and **4.** The proposed model demonstrated superior performance across all four-error metrics (Mean Squared Error, Bias, Variance, and Adjusted R-Squared), highlighting the effectiveness of the non-parametric approach. The proposed model outperformed the existing comparison models consistently, as the predicted faults obtained from the implementations of the proposed model is much closer to the actual faults in the testing sets.

Table 3. Prediction error of detected faults (Dataset 1).

Metric	Proposed mode	Xiao model	Hu model	Dhaka model	Li model
MSE	7.771	528.727	319.602	1832.527	1596.262
Adjusted R2	0.999	0.994	0.997	0.975	0.978
Bias	1.464	-1.705	-1.966	-11.177	-7.400
Variation	5.092	20.812	17.871	41.560	39.487

Table 4. Prediction error of corrected faults (Dataset 1).

Metric	Proposed mode	Xiao model	Hu model	Dhaka model	Li model
MSE	25.274	415.227	430.682	3360.877	30118.699
Adjusted R^2	0.999	0.997	0.997	0.937	0.630
Bias	1.906	-7.671	-6.818	-18.118	97.763
Variation	8.805	14.192	15.747	55.385	144.213

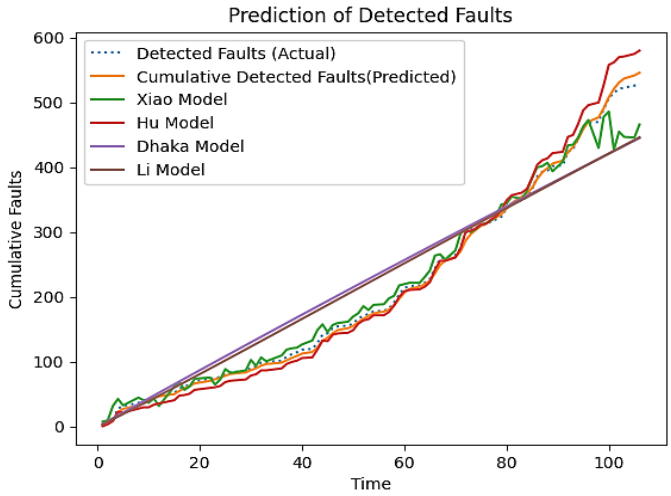

Figure 2. Actual vs predicted detected faults (Dataset 1).

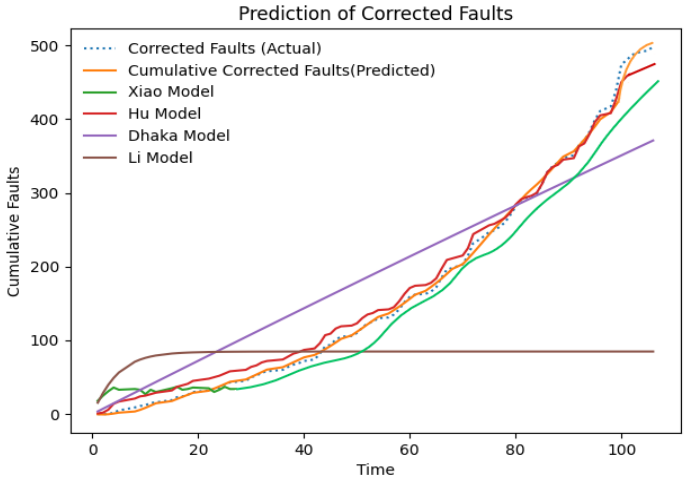

Figure 3. Actual vs predicted corrected faults (Dataset 1).

The graphical representation of the predicted fault and actual fault with respect to time has been presented in **Figures 2** and **3**. In these graphs, the detected faults obtained from the proposed model as well as comparison models has been plotted to highlight the prediction accuracy.

4.2 Dataset 2 (Jira API V2.2)

For the Jira API V2.2 dataset, the proposed model also showed a significant improvement in predicting detected and corrected faults. Similar to the first experiment, the proposed model reached optimal performance with 80 epochs and a batch size of 8, while the Xiao and Hu models required more epochs and larger batch sizes. The results indicated in **Tables 5** and **6** shows that the proposed model had lower Mean Squared Error, Bias, and Variance, as well as better Adjusted R-Squared values compared to the existing models. This suggests that the proposed model is more accurate and reliable in predicting software faults.

Table 5. Prediction error of detected faults (Dataset 2).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	9.38	13.93	491.22	6908.89	6908.89
Adjusted R^2	0.9999	0.9998	0.9996	0.0006	0.0006
Bias	-0.13	-3.33	1.89	0.23	0.23
Variation	1.09	3.60	20.95	84.06	84.06

Table 6. Prediction error of corrected faults (Dataset 2).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	10.30	34.51	105.04	5191.93	7161.24
Adjusted R^2	0.9999	0.9985	0.9971	0.0095	0.008
Bias	1.12	-4.96	-9.13	-21.20	-18.00
Variation	3.04	3.19	7.27	72.40	85.58

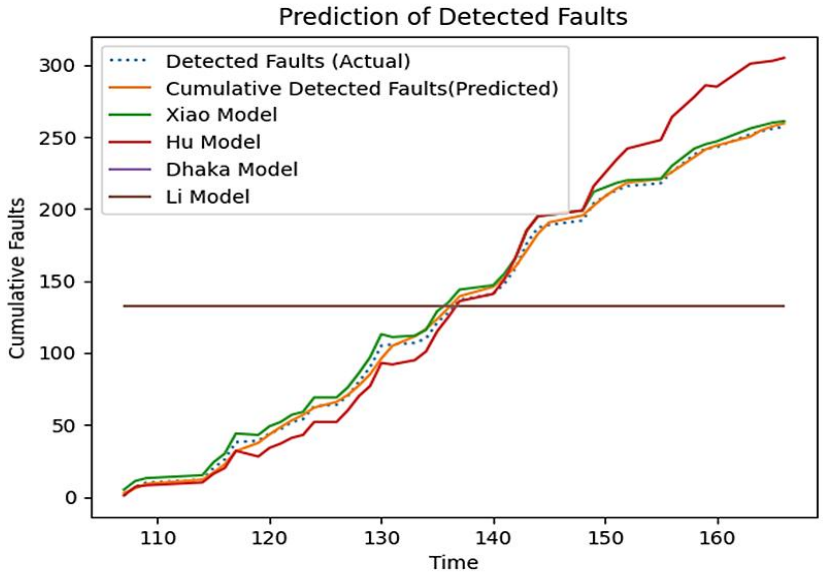

Figure 4. Actual vs predicted detected faults (Dataset 2).

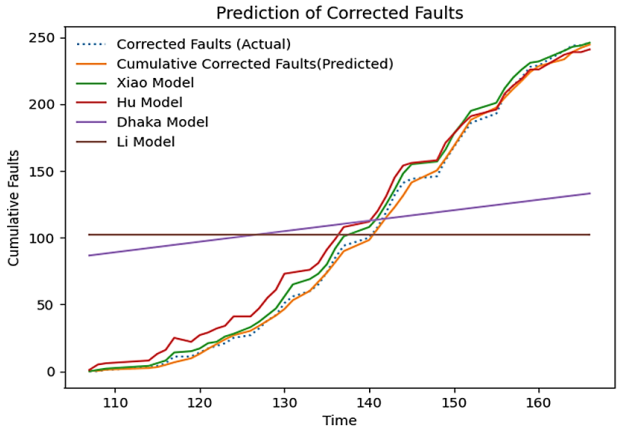

Figure 5. Actual vs predicted corrected faults (Dataset 2).

Similarly, **Figures 4** and **5** highlights the graphical representation of actual faults vs predicted fault for the 20 percent of testing data. The **Figure 4** shows the cumulative number of detected faults over time, comparing actual data (dotted line) with predictions from proposed model and various existing models (Xiao Model, Hu Model, Dhaka Model, and Li Model). The proposed model aligns closely with the actual detected faults as compared to Xiao and Dhaka models, whereas the Hu model underestimates significantly and the Li model shows an almost constant prediction, failing to capture the trend. The second **Figure 5** illustrates the cumulative number of corrected faults over time with these models. Similar to the **Figure 4**, the proposed model gives the better accuracy rather than the Xiao and Dhaka models, while the Hu model overestimates and the Li model underestimates the faults.

4.3 Dataset 3 (Jira API V2.3)

In the third experiment with the Jira API V2.3 dataset, the proposed model continued to demonstrate superior performance. With the training parameters (50 epochs and a batch size of 8), it outperformed the Xiao, Hu, Dhaka, and Li models across all error metrics as shown in **Tables 7** and **8.**

Table 7. Prediction error of detected faults (Dataset 3).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	0.53	2.35	28.92	518.05	518.05
Adjusted R^2	0.99	0.94	0.93	0.76	0.76
Bias	-0.13	-1.45	5.18	9.10	3.46
Variation	0.72	0.98	1.47	22.95	22.95

The parametric models again failed to achieve optimal results due to their dependence on initial parameter settings and assumptions. The proposed model's flexibility and ability to learn from data without restrictive assumptions made it more effective in fault prediction as depicted in **Figures 6** and **7.**

Table 8. Prediction error of corrected faults (Dataset 3).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	0.70	2.95	5.37	215.35	513.97
Adjusted R^2	0.99	0.96	0.92	0.88	0.85
Bias	0.30	-0.85	-0.92	-2.99	0.00
Variation	0.79	0.47	2.14	14.48	22.86

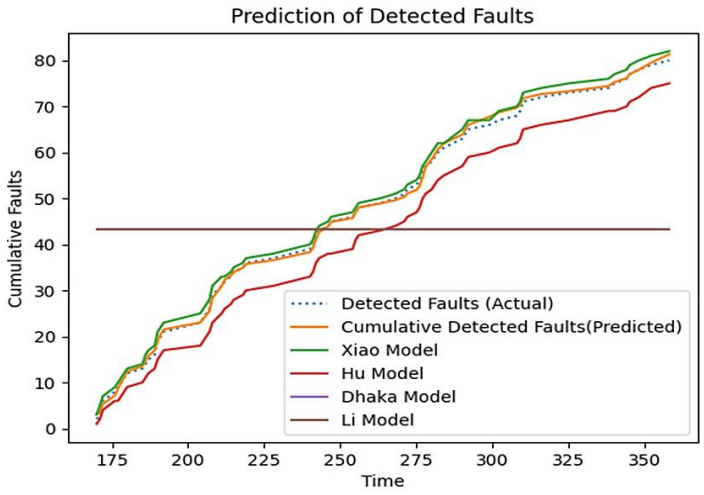

Figure 6. Actual vs predicted detected faults (Dataset 3).

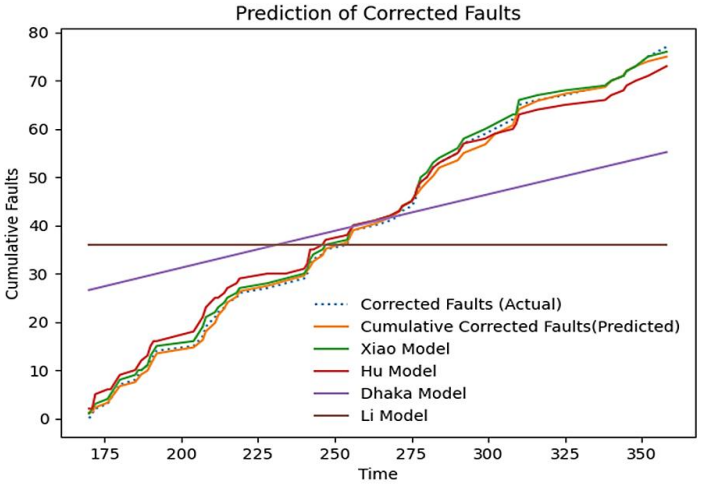

Figure 7. Actual vs predicted corrected faults (Dataset 3).

4.4 Dataset 4 (Firefox 3)

The Firefox 3 dataset provided further validation of the proposed model's effectiveness. The proposed model achieved optimal performance with 100 epochs and a batch size of 8, whereas the comparison models required more epochs and larger batch sizes to reach similar levels of performance.

Table 9. Prediction error of detected faults (Dataset 4).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	2.436	35.264	5.490	3.844	15.489
Adjusted R^2	0.999	0.989	0.997	0.996	0.9930
Bias	-0.0417	-0.8566	-0.215	-0.459	0.58
Variation	1.5747	15.994	2.9557	2.9277	12.93

Table 10. Prediction error of corrected faults (Dataset 4).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	2.21	51.74	33.25	6.78	8.46
Adjusted R^2	0.999	0.95	0.98	0.99	0.900
Bias	0.33	0.45	-5.13	0.21	0.27
Variation	1.46	7.25	6.65	2.42	2.11

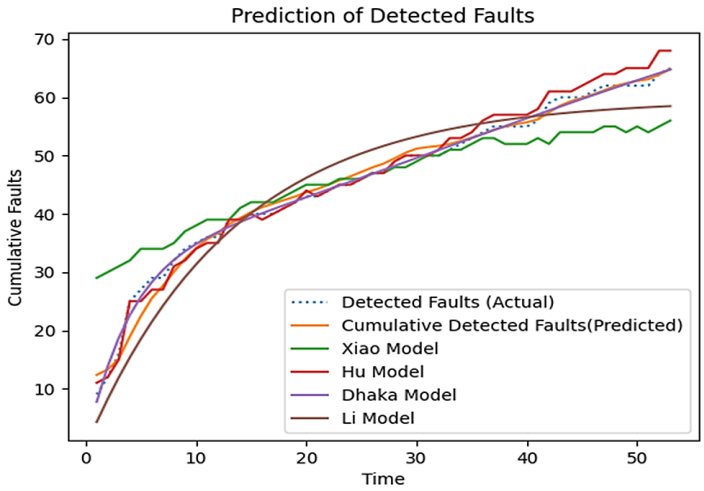

Figure 8. Actual vs predicted detected faults (Dataset 4).

The proposed model's predictions are more accurate, with lower Mean Squared Error and Bias, and better Variance, and Adjusted R-Squared values as shown in **Tables 9** and **10**. These results show the advantages of using proposed model for fault prediction in complex software systems. The graphical representations of predicted faults over time in **Figures 8** and **9** showed that the proposed model's predictions aligned more closely with the actual fault data across different projects.

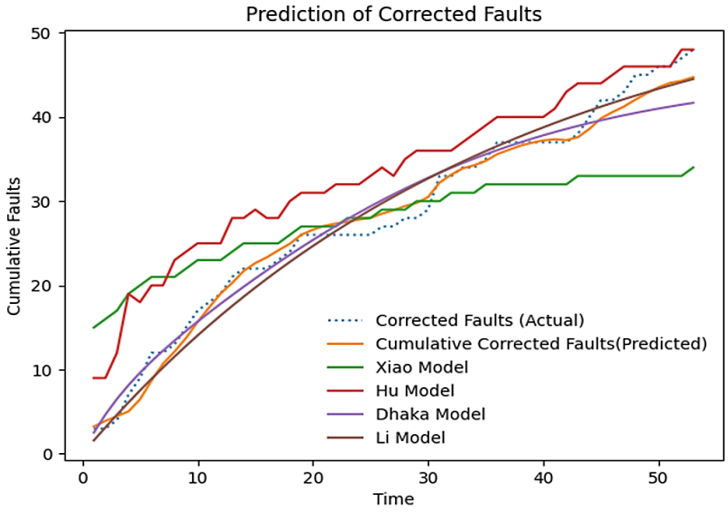

Figure 9. Actual vs predicted corrected faults (Dataset 4).

4.5 Dataset 5 (Firefox 3.5)

In the final experiment with the Firefox 3.5 dataset, the proposed model once again demonstrated its superiority. It required the 50 epochs and 8 batch size to outperformed the existing models in all four-error metrics.

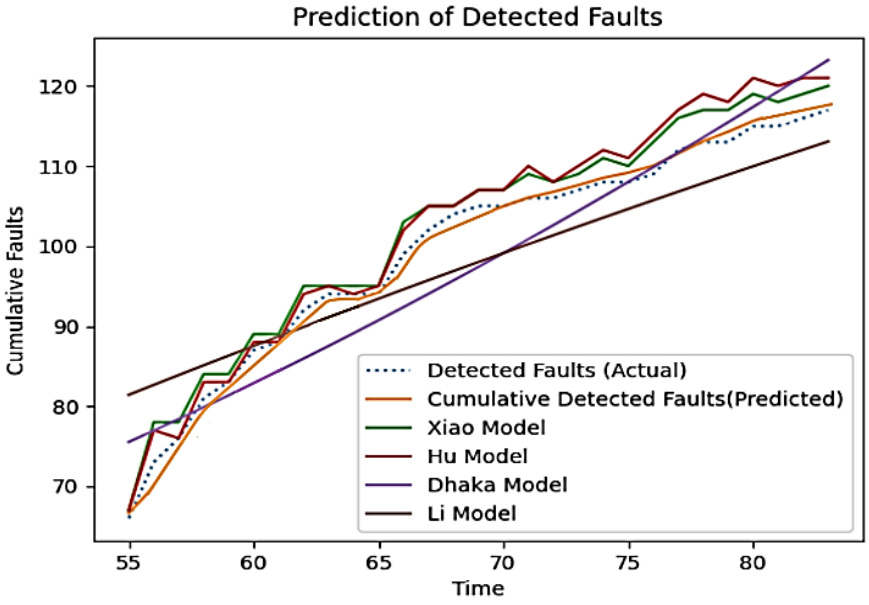

Figure 10. Actual vs predicted detected faults (Dataset 5).

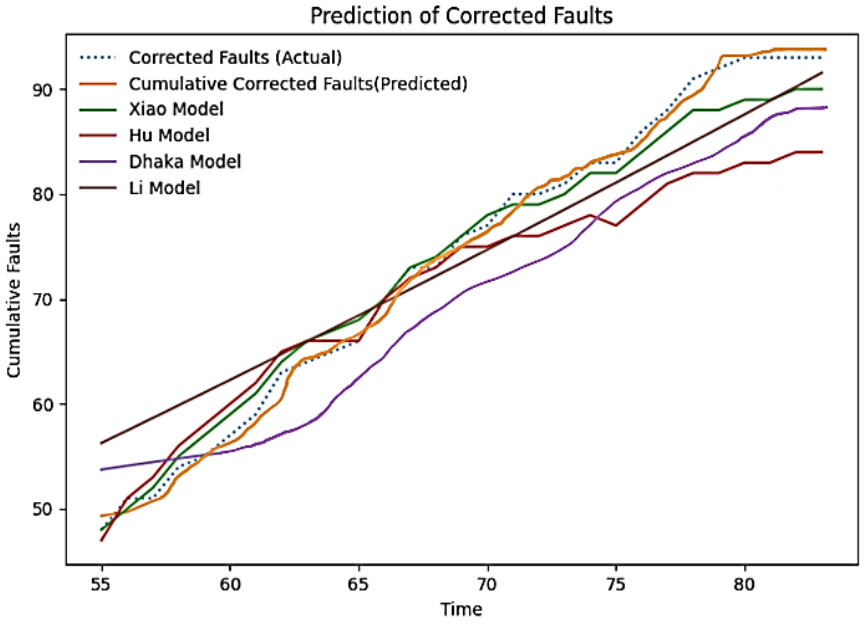

Figure 11. Actual vs predicted detected faults (Dataset 5).

Table 11. Prediction error of detected faults (Dataset 5).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	3.11004	13.8658	15.0122	34.9230	59.5101
Adjusted R^2	0.9998	0.9989	0.9984	0.9971	0.9941
Bias	0.1107	-1.3293	-0.8902	-0.0184	2.3789
Variation	0.7709	1.4577	2.0668	3.8868	6.2134

Table 12. Prediction error of corrected faults (Dataset 5).

Metric	Proposed model	Xiao model	Hu model	Dhaka model	Li model
MSE	5.2830	11.9390	11.8780	181.6377	23.2319
Adjusted R^2	0.9976	0.9954	0.9957	0.9409	0.9910
Bias	1.0137	-1.0366	-0.4878	4.522	1.0181
Variation	2.7102	3.3164	3.4328	12.7742	4.7402

Across all five datasets, the proposed DNN-based software reliability model consistently outperformed the existing models (Dhaka and Nijhawan, 2024; Hu et al., 2007; Li and Pham, 2017; Xiao et al., 2020) in terms of Mean Squared Error, Bias, Variance, and Adjusted R-Squared as depicted in **Tables 11** and **12**. The non-parametric approach of the proposed model, combined with its ability to handle missing data and variation in input through logarithmic encoding, proved to be a more effective method for software fault prediction. The detailed results and graphical representations (**Figures 10** and **11**) of predicted faults underscore the advantages of the proposed model in delivering accurate and reliable predictions for software.

5. Observations and Comparison

From the above experiments, it can be illustrated that the proposed model consistently outperformed the existing models in all four-error metrics. This section discusses about the results obtained from the proposed model and existing models to show the effectiveness of proposed model. The comparative analysis is conducted based on three critical parameters: prediction accuracy, computational requirements of hyperparameters, and time complexity. These parameters are essential for evaluating the overall performance and feasibility of the models in practical applications as detailed in further subsections.

5.1 Prediction Accuracy

Accurate prediction of detected and corrected faults can provide a more comprehensive view of software reliability and better guide testing and maintenance efforts. The proposed model demonstrated superior prediction accuracy compared to existing models such as Xiao Model (Xiao et al., 2020), Hu Model (Hu et al., 2007), Dhaka Model (Dhaka and Nijhawan, 2024) and Li Model (Li and Pham, 2017). The error prediction is evaluated using four error metrics, as discussed in **Tables 3** to **12**. After training the model, predictions for future faults are performed on the testing set of each dataset using the proposed and comparative models, as detailed in **Tables 13** and **14.** These tables present the average cumulative faults at the conclusion of the testing period for each dataset, offering insight into the models' predictive performance over the entire testing duration. The Prediction accuracy is evaluated using the mean absolute error (MAE) between the actual and predicted faults for the 20% testing sets of each dataset. The accuracy is calculated using the following formula,

$$\boldsymbol{Accuracy} = \mathbf{100\%} - \left(\frac{\boldsymbol{Mean\ Absolute\ Error\ (MAE)}}{\boldsymbol{Average\ Actual\ Value}} \times \mathbf{100\%}\right) \quad (3)$$

The proposed model achieved the highest prediction accuracy with an average of 98% for detected faults

and 96% for corrected faults across all five datasets as shown using graphs in **Figure 12**. Additionally, the overall prediction accuracy for each model is depicted in **Figures 13** and **14** showcasing that the proposed model consistently outperformed other models in terms of prediction accuracy across all datasets.

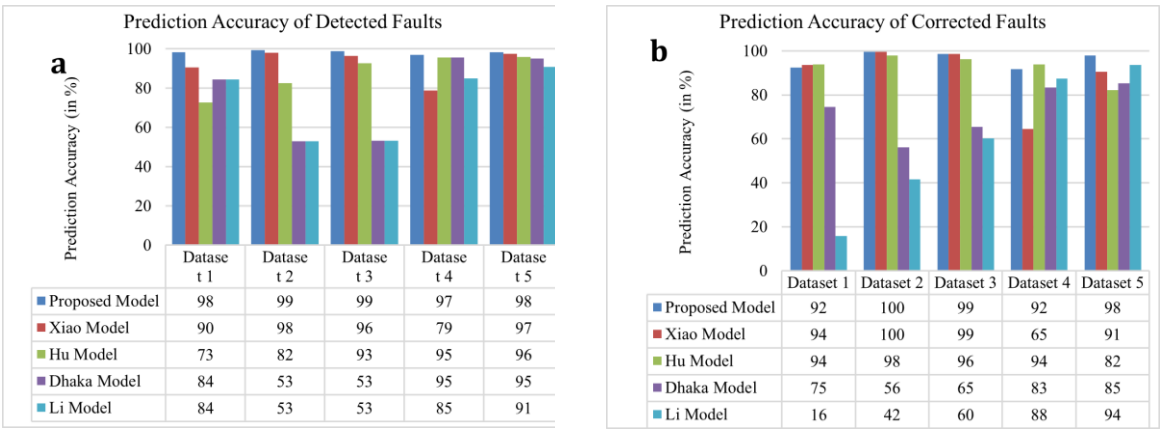

Figure 12. Prediction accuracy of detected and corrected faults (a-b).

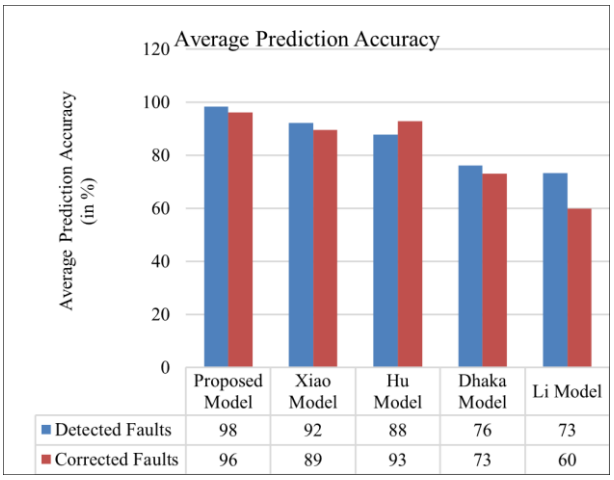

Figure 13. Average prediction accuracy of the models.

Table 13. Prediction of cumulative detected faults at the end of testing.

	Actual	Proposed mod	Xiao model	Hu model	Dhaka model	Li model
Dataset 1	534	544	483	680	450	450
Dataset 2	261	263	266	307	138	138
Dataset 3	81	82	84	75	43	43
Dataset 4	66	64	52	69	63	56
Dataset 5	119	117	122	124	125	108

Table 14. Prediction of Cumulative corrected faults at the end of testing

	Actual	Proposed mod	Xiao model	Hu model	Dhaka model	Li model
Dataset 1	504	512	472	473	376	80
Dataset 2	248	247	249	243	139	103
Dataset 3	78	77	79	75	51	47
Dataset 4	48	44	31	51	40	42
Dataset 5	95	93	86	78	81	89

5.2 Computational Performance

The computational requirements for hyperparameters, such as the number of epochs, batch size, and learning rate, significantly affect the feasibility and efficiency of a model. Models with lower computational requirements are more practical for real-world applications, especially when computational resources are limited. The details of the hyperparameters required by models for training the dataset has been shown in **Figure 14.** It can be clearly observed that the proposed model required fewer epochs and a smaller batch size to reach optimal performance compared to the existing models. This efficiency in training indicates lower computational requirements and faster convergence, making the proposed model more practical for real-world applications.

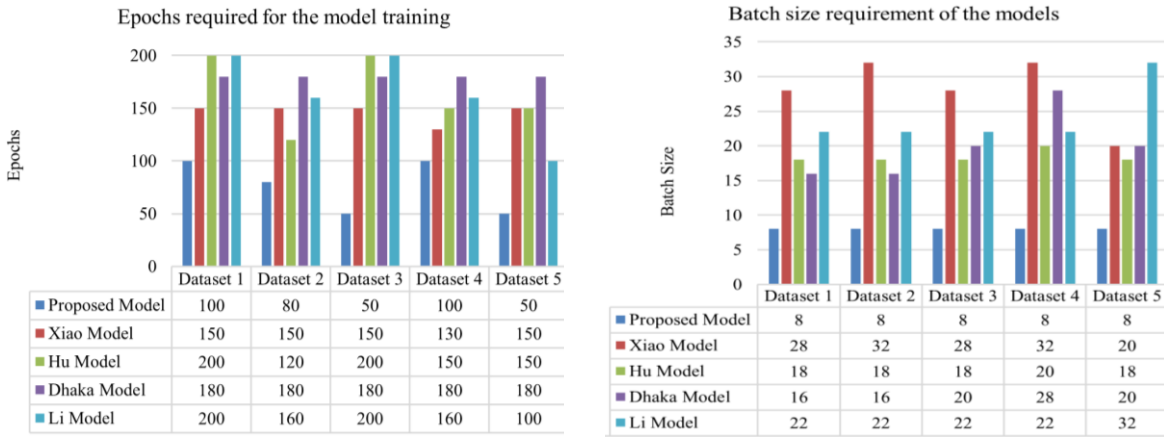

Figure 14. Hyperparameters used for the model validation.

5.3 Time Complexity

Time complexity is a critical factor in evaluating the scalability and real-time applicability of a model. Lower time complexity indicates that the model can be trained and deployed faster, making it more suitable for dynamic and large-scale environments. The proposed model employs a relatively simple architecture with two dense layers, each consisting of 64 neurons with ReLU activation functions. The time complexity for a single forward pass through this model is $O(n \cdot d)$ where n is the number of input features and d is the number of neurons in a layer. Considering the model's simplicity and the use of dense layers, the backward pass has a similar time complexity, leading to an overall training time complexity of $O(E \cdot B \cdot n \cdot d)$, where E is the number of epochs and B is the batch size. This ensures faster training times and lower computational requirements compared to more complex architectures.

In contrast, the LSTM model and simple RNN model, with its ability to capture temporal dependencies, involves a more intricate architecture. Each LSTM cell requires the computation of multiple gates (input, forget, and output gates) and the cell state update, which significantly increases the computational load. The time complexity for a single forward pass through an LSTM layer is $O(T \cdot n \cdot h^2)$, where T is the sequence length, n is the number of input features, and h is the number of hidden units. The backward pass similarly involves complex gradient calculations, resulting in a training time complexity of $O(E \cdot B \cdot T \cdot n \cdot h^2)$. This higher complexity often translates to longer training times and greater computational demands unlike proposed model. This makes the proposed model a robust and practical solution for software fault prediction, suitable for various real-world applications where resource efficiency and quick adaptability are crucial.

6. Optimal Release Time Determination Considering Cost and Reliability

The process of fault prediction helps in analysing the reliability growth of the software product. Although, determining the number of detected and corrected faults in the software is not the sole concern for a software developer. Deciding when to release software is critical for market competition. Releasing too early can lead to numerous remaining defects and increased maintenance costs. While, delaying the release to reach high reliability incurs significant development and testing costs. The optimal release time ensures the software meets acceptable reliability standards at minimal cost, maximizes market opportunities by hitting crucial market windows, and efficiently manages resources. It enhances customer satisfaction and retention by delivering reliable products. Therefore, in this study the optimal release time has been determined using the proposed model by ensuring that the software meets a reliability requirement while minimizing the associated costs. The cost required to develop the software is divided into phases as discussed below:

Setup and development cost **(C_s):** This includes all costs associated with setting up the development environment, initial design, coding, integrating tools and 3rd party APIs.

Market opportunity cost **(C_m):** Market opportunity cost refers to the potential revenue and competitive advantages lost when a software product is not released at the most opportune time. Releasing a software product late can result in missed market windows, allowing competitors to capture market share and diminishing the product's potential impact. The Market Opportunity Cost is quadratic function of software release time established by Jiang et al. (2012) as formulated as

$$\boldsymbol{C_m = c_0 t^2} \tag{4}$$

Testing cost **(C_t):** Cost incurred to perform the testing of various components of the project until the release time. It can be calculated as the product of cost of testing per unit time and the time to release the software as given below.

$$\boldsymbol{C_t = c_1 t} \tag{5}$$

Debugging cost in-house testing phase **(C_h):** Costs for fixing bugs discovered during the testing phase. After or during testing, the debugging phase starts, where the detected faults are corrected. This phase often runs concurrently with testing.

$$C_h = c_2 m_c(t) \tag{6}$$

where, c_2 is the cost required to correct a fault, and $m_c(t)$ is the number of faults corrected.

Debugging cost in field-testing phase **(*Beta testing*):** This phase occurs after in-house testing and involves releasing the software to a controlled group of users. It's typically shorter but crucial for catching

environment-specific or real-world faults.

$$C_f = c_3[m_d(t) - m_c(t)] \quad (7)$$

Debugging in warranty period **(C_w):** Costs associated with fixing defects discovered during the warranty period offered to customers. If w is the warranty period, then the cost for removing faults in warranty period can be given as

$$C_w = C_4[m_d(t_w) - m_c(t)] \quad (8)$$

Therefore, the total cost can be calculated as the summation of all the above costs and it can be represented as

$$\boldsymbol{C(t) = C_s + C_m + C_t + C_h + C_f + C_w} \quad (9)$$

$$\boldsymbol{C(t) = c_s + c_0 t^2 + C_1 t + C_2 m_c(t) + C_3[m_d(t) - m_c(t)] + C_4[m_d(t_w) - m_c(t)} \quad (10)$$

Let T_{lc} is the lifecycle of software and $m_C(t)$ denotes the total number of corrected faults up to period *t*. $a = m_d(T_{lc})$ denotes the number of total faults of software in the lifecycle. The reliability target set by the client and the development team through mutual agreement is denoted as *R** The software reliability at time *t*, represented as *R(t)*, can be formulated in terms of the cumulative number of corrected faults as

$$\boldsymbol{R(t) = \frac{m_c(t)}{a}} \quad (11)$$

The objective is to minimize the expected cost of software *C*(*t*) i.e.

$$\boldsymbol{min\, C\,(t)} = \min\{\boldsymbol{c_s + c_0 t^2 + C_1 t + C_2 m_c(t) + C_3[m_d(t) - m_c(t)] + C_4[m_d(t_w) - m_c(t)]}\} \quad (12)$$

Subject to

$$R(t) \geq R^* \quad (13)$$

The cost associated with each phase of software development can vary significantly depending on factors such as the size of the project, specific requirements, and the functionalities being implemented. To model these variations, we assume the costs related to fault correction in different phases as $c_s = \$5, c_0 = \$2, c_1 = \$2, c_2 = \$3, c_3 = \$4, c_4 = \$5$. This cost distribution has been determined based on insights from the literature (Aggarwal et al., 2015; Anand et al., 2022; Franch and Ruhe, 2016; Kapur et al., 2012), where the costs are carefully constrained according to the relationship $c_1 < c_2 < c_3 < c_4$ and the value of c_s and c_0 can vary depending upon the software. This ordering reflects the understanding that as faults are detected and corrected in later stages of the software lifecycle, the cost associated with their correction generally increases.

We are implementing the cost model on Dataset 1 (Jira API V2.1), focusing on determining the optimal release time based on reliability constraints. The lifecycle of the software is set at 150 days, with a warranty period *w* of 30 days. The reliability requirement R^* is specified as 0.96. The time to reach this reliability target with minimal cost can be obtained using Equation (12). From the analysis, as illustrated in the **Figure 15**, the software's reliability begins to increase significantly above 0.95 from day 108, eventually reaching the target reliability of 0.96 at day 110. At this point, the associated cost is $1924. Therefore, the optimal release time for the software, considering both the reliability target and cost minimization, is determined to be 110 days.

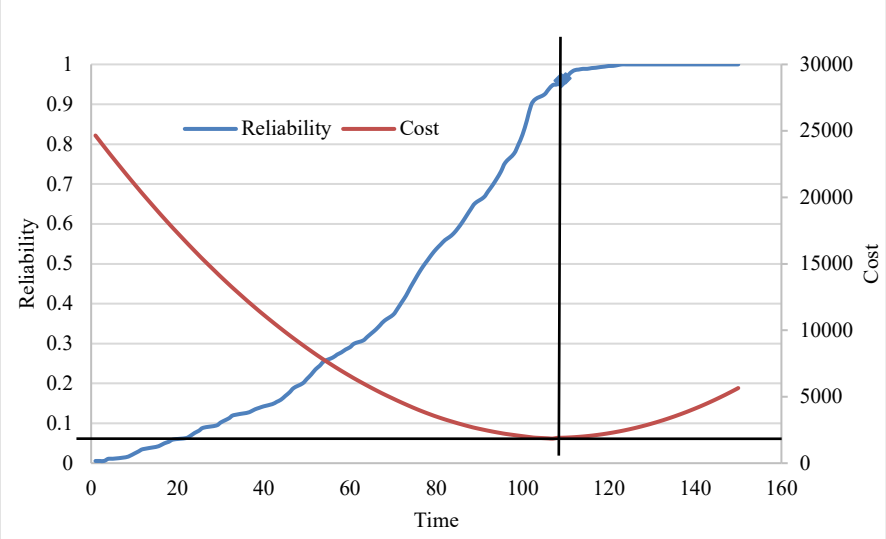

Figure 15. Optimal cost determination.

This method can also be applied to other datasets as well to determine the optimal release time, enabling better decision-making in software project management. By calculating the time at which the reliability target is met with minimal cost, software developers can make informed decisions about the best time to release the software, ensuring a balance between product reliability and development costs.

7. Limitations and Future Work

The proposed system demonstrates promising results, it is important to acknowledge its limitations and outline potential areas for future research and development. Some limitations of the proposed work include:

- While the proposed model has been validated using five datasets with varying characteristics, its generalizability across industrial-scale datasets from different domains (e.g., embedded systems, safety-critical software) can be further explored.
- The current model does not incorporate fault severity levels. It can provide a better reliability assessment, enabling better prioritization of fault correction efforts.
- Due to the unavailability of actual cost structures and their variability across projects and organizations, cost calculations in this study are based on assumed values.

By acknowledging these limitations, future research efforts can focus on expanding the model by incorporating diverse software fault datasets and integrating fault severity levels to enhance reliability assessment. Additionally, addressing fault dependencies across multiple sprints will improve long-term fault prediction in Agile development. Further advancements can include refining data preprocessing techniques and optimizing model performance for real-world applications.

8. Conclusion

This paper proposes an DNN-based non-parametric model to predict both detected and corrected faults, addressing the gap of simpler model in existing research. The proposed model demonstrated promising results in accurately predicting both detected and corrected faults across various software failure datasets. It utilizes a dense neural network architecture combined with logarithmic encoding of input parameters. Logarithmic encoding helps manage the wide range of input values, reducing skewness and enhancing the

training process's efficiency and accuracy. By capturing intricate patterns and interactions within the data that might be overlooked by linear or non-logarithmic methods, the model demonstrates superior performance in predicting faults across various software failure datasets. Accurate prediction of both detected and corrected faults is essential for software maintenance. It significantly reduces the need for post-deployment debugging and enhances overall software quality, contributing to the broader field of software reliability. Furthermore, the model's ability to determine the optimal release time based on the predicted faults ensures that software is released when it meets the reliability requirements of the customer, thereby minimizing costs associated with over-testing or premature release. This dual focus on fault prediction and optimal release time determination makes the proposed model a valuable tool for supporting the development and maintenance of high-quality software systems.

Conflicts of Interest

The authors confirm that there is no conflict of interest to declare for this publication.

Acknowledgments

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. The author would like to thank the editor and anonymous reviewers for their comments that help improve the quality of this work.

AI Disclosure

During the preparation of this work the author(s) used generative AI in order to improve the language of the article. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Appendix: Datasets 1 to 3

Dataset 1: Jira V2.1

Table 15. Jira V2.1 dataset.

Tin	Detected f	Cumulative dete faults	Correcte faults	Cumulative corrected faul	Tin	Detected f	Cumulative detec faults	Correcte faults	Cumulative corre faults
1	1	1	0	0	59	12	204	10	152
2	3	4	0	0	60	10	214	7	159
3	7	11	0	0	61	3	217	3	162
4	17	28	2	2	63	1	218	1	163
5	4	32	3	5	64	4	222	3	166
8	5	37	4	9	65	6	228	6	172
9	4	41	2	11	66	16	244	13	185
10	1	42	1	12	67	15	259	11	196
11	3	45	3	15	68	2	261	2	198
12	2	47	1	16	70	4	265	5	203
13	5	52	3	19	71	10	275	12	215
16	7	59	4	23	72	24	299	17	232
17	3	62	1	24	73	5	304	4	236
18	3	65	2	26	74	3	307	5	241
19	5	70	4	30	75	4	311	6	247
22	4	74	2	32	76	1	312	2	249
23	1	75	2	34	77	5	317	5	254
24	2	77	2	36	78	3	320	6	260
25	5	82	4	40	79	4	324	10	270

Table 15 continued…

2	2	84	3	43	8	13	337	11	281
2	2	86	2	45	8	7	344	9	290
3	5	91	5	50	8	3	347	3	293
3	2	93	2	52	8	3	350	1	294
3	6	99	4	56	8	4	354	5	299
3	1	100	2	58	8	13	367	13	312
3	2	102	2	60	8	19	386	16	328
3	6	108	4	64	8	7	393	8	336
3	4	112	3	67	8	2	395	5	341
3	3	115	2	69	8	7	402	6	347
4	4	119	3	72	9	1	403	4	351
4	1	120	2	74	9	21	424	14	365
4	9	129	8	82	9	4	428	6	371
4	13	142	10	92	9	16	444	12	383
4	2	144	2	94	9	19	463	16	399
4	8	152	7	101	9	6	469	12	411
4	3	155	4	105	9	1	470	6	417
4	1	156	1	106	9	20	490	22	439
5	3	159	4	110	10	17	507	35	474
5	8	167	7	117	10	8	515	6	480
5	4	171	6	123	10	6	521	7	487
5	3	174	2	125	10	2	523	3	490
5	4	178	5	130	10	3	526	3	493
5	1	179	1	131	10	3	529	5	498
5	3	182	4	135	10	5	534	6	504
5	10	192	7	142					

Dataset 2: Jira V2.2

Table 16. Jira V2.2 dataset.

Tin	Detected fa	Cumulativ detected fau	Correct faults	Cumulative corrected fau	Tin	Detected f	Cumulative detec faults	Correct faults	Cumulative corre faults
10	1	1	0	0	13	9	137	9	94
10	6	7	0	0	14	4	141	6	100
10	3	10	1	1	14	7	148	8	108
11	2	12	2	3	14	11	159	11	119
11	8	20	1	4	14	17	176	13	132
11	6	26	2	6	14	11	187	9	141
11	12	38	5	11	14	2	189	3	144
11	1	39	0	11	14	3	192	2	146
12	5	44	3	14	14	12	204	12	158
12	3	47	3	17	15	5	209	10	168
12	5	52	2	19	15	4	213	9	177
12	2	54	2	21	15	3	216	9	186
12	9	63	4	25	15	2	218	7	193
12	1	64	2	27	15	8	226	14	207
12	6	70	5	32	15	6	232	7	214
12	10	80	6	38	15	6	238	6	220
12	10	90	4	42	15	4	242	8	228
13	15	105	9	51	16	1	243	1	229
13	1	106	5	56	16	9	252	11	240
13	1	107	4	60	16	2	254	4	244
13	3	110	5	65	16	2	256	0	244
13	11	121	9	74	16	1	257	2	246
13	7	128	11	85	16	4	261	2	248

Dataset 3: Jira V2.3

Table 17. Jira V2.3 dataset.

Tim	Detected f	Cumulative dete faults	Correcte faults	Cumulative corre faults	Tir	Detected f	Cumulative dete faults	Correcte faults	Cumulative corre faults
170	2	2	0	0	25	1	46	1	36
171	1	3	1	1	25	1	47	2	38
172	3	6	1	2	25	1	48	1	39
176	2	8	1	3	26	1	49	1	40
177	1	9	1	4	26	1	50	1	41
178	1	10	1	5	27	1	51	1	42
179	1	11	1	6	27	1	52	1	43
180	1	12	1	7	27	1	53	1	44
185	1	13	1	8	27	1	54	1	45
186	1	14	1	9	27	2	56	2	47
187	1	15	1	10	27	1	57	2	49
189	1	16	1	11	28	1	58	1	50
190	3	19	1	12	28	2	60	2	52
191	1	20	1	13	28	1	61	1	53
192	1	21	1	14	29	2	63	2	55
204	2	23	1	15	29	2	65	2	57
207	3	26	2	17	29	1	66	2	59
208	3	29	2	19	30	1	67	1	60
211	2	31	2	21	30	1	68	2	62
212	1	32	1	22	30	1	69	1	63
214	1	33	1	23	31	2	71	2	65
215	1	34	1	24	31	1	72	1	66
218	1	35	1	25	32	1	73	1	67
219	1	36	1	26	33	1	74	2	69
228	1	37	1	27	34	1	75	1	70
234	1	38	1	28	34	1	76	1	71
240	1	39	1	29	34	1	77	1	72
241	1	40	1	30	34	1	78	1	73
242	2	42	2	32	35	1	79	2	75
243	1	43	1	33	35	1	80	2	77
246	1	44	1	34	35	1	81	1	78
247	1	45	1	35					

References

Aggarwal, A.G., Nijhawan, N., & Kapur, P.K. (2015). A discrete SRGM for multi-release software system with imperfect debugging and related optimal release policy. In *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management* (pp. 186-192). IEEE. Greater Noida, India. https://doi.org/10.1109/ablaze.2015.7154990.

Amin, A., Grunske, L., & Colman, A. (2013). An approach to software reliability prediction based on time series modeling. *Journal of Systems and Software*, *86*(7), 1923-1932. https://doi.org/10.1016/j.jss.2013.03.045.

Anand, A., Das, S., Agarwal, M., & Inoue, S. (2022). An optimal scheduling policy for upgraded software with updates. *International Journal of Quality & Reliability Management*, *39*(3), 704-715. https://doi.org/10.1108/ijqrm-04-2021-0092/full/pdf.

Bibyan, R., Anand, S., Aggarwal, A.G., & Kaur, G. (2023). Multi-release software model based on testing coverage incorporating random effect (SDE). *MethodsX*, *10*, 102076. https://doi.org/10.1016/j.mex.2023.102076.

Bisi, M., & Goyal, N.K. (2015). Prediction of software inter-failure times using artificial neural network and particle swarm optimisation models. *International Journal of Software Engineering, Technology and Applications*, *1*(2-4), 222-244. https://doi.org/10.1504/ijseta.2015.075629.

Bisi, M., & Goyal, N.K. (2016). Software development efforts prediction using artificial neural network. *IET Software*, *10*(3), 63-71. https://doi.org/10.1049/iet-sen.2015.0061.

Chatterjee, S., & Shukla, A. (2016). Modeling and analysis of software fault detection and correction process through Weibull-type fault reduction factor, change point and imperfect debugging. *Arabian Journal for Science and Engineering*, *41*(12), 5009-5025. https://doi.org/10.1007/s13369-016-2189-0.

Dhaka, V., & Nijhawan, N. (2024). Effect of change in environment on reliability growth modeling integrating fault reduction factor and change point: a general approach. *Annals of Operations Research*, *340*(1), 181-215. https://doi.org/10.1007/s10479-022-05084-6.

Dwivedi, S., & Goyal N.K. (2025). Fault prediction of multi-version software considering imperfect debugging and severity. *International Journal of System Assurance Engineering and Management*. https://doi.org/10.1007/s13198-024-02671-7.

Dwivedi, S., & Goyal, N.K. (2024). Effect of fault correction delay on software reliability modelling in agile software development. In: Varde, P.V., Vinod, G., Joshi, N.S. (eds) *International Conference on Reliability, Safety, and Hazard*. Springer Nature, Singapore, pp. 795-802. ISBN: 978-981-97-3087-2. https://doi.org/10.1007/978-981-97-3087-2_70.

Franch, X., & Ruhe, G. (2016). Software release planning. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp 894-895). ACM. New York, USA. https://doi.org/10.1145/2889160.2891051.

Ho, S.L., Xie, M., & Goh, T.N. (2003). A study of the connectionist models for software reliability prediction. *Computers & Mathematics with Applications, 46*(7), 1037-1045. https://doi.org/10.1016/S0898-1221(03)90117-9.

Hsu, C.J., Huang, C.Y., & Chang, J.R. (2011). Enhancing software reliability modeling and prediction through the introduction of time-variable fault reduction factor. *Applied Mathematical Modelling, 35*(1), 506-521. https://doi.org/10.1016/j.apm.2010.07.017.

Hu, Q.P., Xie, M., Ng, S.H., & Levitin, G. (2007). Robust recurrent neural network modeling for software fault detection and correction prediction. *Reliability Engineering & System Safety, 92*(3), 332-340.

Huang, Y.S., Chiu, K.C., & Chen, W.M. (2022). A software reliability growth model for imperfect debugging. *Journal of Systems and Software, 188*, 111267. https://doi.org/10.1016/j.jss.2022.111267.

Jiang, Z., Sarkar, S., & Jacob, V.S. (2012). Postrelease testing and software release policy for enterprise-level systems. *Information Systems Research, 23*(3-part-1), 599-848. https://doi.org/10.1287/isre.1110.0379.

Kapur, P.K., Pham, H., Aggarwal, A.G., & Kaur, G. (2012). Two dimensional multi-release software reliability modeling and optimal release planning. *IEEE Transactions on Reliability, 61*(3), 758-768. https://doi.org/10.1109/tr.2012.2207531.

Karunanithi, N., Malaiya, Y.K., & Whitley, D. (1991). Prediction of software reliability using neural networks. In *Proceedings 1991 International Symposium on Software Reliability Engineering* (pp. 124-130). IEEE, Austin, USA. https://doi.org/10.1109/issre.1991.145366.

Kumar, V., Mathur, P., Sahni, R., & Anand, M. (2016). Two-dimensional multi-release software reliability modeling for fault detection and fault correction processes. *International Journal of Reliability, Quality and Safety Engineering, 23*(3), 1640002. https://doi.org/10.1142/s0218539316400027.

Li, C., Zheng, J., Okamura, H., & Dohi, T. (2022). Software reliability prediction through encoder-decoder recurrent neural networks. *International Journal of Mathematical, Engineering and Management Sciences, 7*(3), 325-340. https://doi.org/10.33889/ijmems.2022.7.3.022.

Li, Q., & Pham, H. (2017). NHPP software reliability model considering the uncertainty of operating environments with imperfect debugging and testing coverage. *Applied Mathematical Modelling, 51*, 68-85. https://doi.org/10.1016/j.apm.2017.06.034.

Pachauri, B., Dhar, J., & Kumar, A. (2015). Incorporating inflection S-shaped fault reduction factor to enhance software reliability growth. *Applied Mathematical Modelling*, *39*(5-6), 1463-1469. https://doi.org/10.1016/j.apm.2014.08.006.

Peng, R., & Zhai, Q. (2017). Modeling of software fault detection and correction processes with fault dependency. *Eksploatacja i Niezawodnosc - Maintenance and Reliability, 19*(3), 467-475. https://doi.org/10.17531/ein.2017.3.18.

Pradhan, V., Kumar, A., & Dhar, J. (2022). Modelling software reliability growth through generalized inflection S-shaped fault reduction factor and optimal release time. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, *236*(1), 18-36. https://doi.org/10.1177/1748006x211033713.

Raamesh, L., Jothi, S., & Radhika, S. (2022). Enhancing software reliability and fault detection using hybrid brainstorm optimization-based LSTM model. *IETE Journal of Research*, *69*(12), 8789-8803. https://doi.org/10.1080/03772063.2022.2069603.

Samal, U., & Kumar, A. (2024). A neural network approach for software reliability prediction. *International Journal of Reliability, Quality and Safety Engineering, 31*(3), 2450009. https://doi.org/10.1142/s0218539324500098.

Wang, J., & Zhang, C. (2018). Software reliability prediction using a deep learning model based on the RNN encoder-decoder. *Reliability Engineering & System Safety, 170*, 73-82. https://doi.org/10.1016/j.ress.2017.10.019.

Wang, J., Li, P., Hu, J., & Zhang, C. (2024). A multi-release reliability model of open source software with fault detection obeying three-parameter lifetime distribution. *Scientific Reports*, *14*(1), 1-17. https://doi.org/10.1038/s41598-024-70536-x.

Xiao, H., Cao, M., & Peng, R. (2020). Artificial neural network based software fault detection and correction prediction models considering testing effort. *Applied Soft Computing, 94*, 106491. https://doi.org/10.1016/j.asoc.2020.106491.

Xie, M., Hu, Q.P., Wu, Y.P., & Ng, S.H. (2007). A study of the modeling and analysis of software fault-detection and fault-correction processes. *Quality and Reliability Engineering International*, *23*(4), 459-470. https://doi.org/10.1002/qre.827.