Deep Learning-Based Software Fault Prediction Using Product and Process Metrics

Faryal Hayat[1,*], Aamer Nadeem[2]
[1]Department of Software Engineering, National University of Modern Languages, Islamabad, Pakistan
[2]Department of Software Engineering, Capital University of Science and Technology, Islamabad, Pakistan
*Correspondence: faryal.hayat@numl.edu.pk

Citation | Hayat. F, Nadeem. A, "Deep Learning-Based Software Fault Prediction Using Product and Process Metrics", IJIST, Vol. 7 Issue. 11 pp 24-43, November 2025
Received | October 9, 2025 **Revised |** November 10, 2025 **Accepted |** November 14, 2025 **Published |** November 19, 2025.

Software Fault Prediction aims to identify software faults in advance, enhancing performance. Machine learning techniques are used to predict software defects. While early approaches used product metrics, later advancements integrated process metrics to enhance predictive capabilities. Deep learning has emerged as a powerful technique for fault prediction. However, current research has primarily focused on utilizing product metrics. The effectiveness of deep learning models when applied to combined or process metrics has not yet been investigated. The novel contribution of this research is the integration of process metrics and product metrics, leveraging deep learning models to improve fault prediction. The study included two experiments: (i) using deep learning models such as Convolutional Neural Network, Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (BiLSTM) network to evaluate product-only metrics with combined metrics (product + process), and (ii) comparing the performance of deep learning models with conventional machine learning models (k-Nearest Neighbors (k-NN), Naive Bayes, and Logistic Regression) based on combined metrics. Experimental results show that fault prediction performance is enhanced by combining process metrics with product metrics. Deep learning models achieved an accuracy of up to 0.93 across five benchmark datasets, with precision, recall, and F1-scores of 0.93, 0.98, and 0.95, respectively. The combined metrics enhanced performance across all parameters when compared to product-only metrics. Additionally, deep learning models such as Convolutional Neural Networks and Bi-Directional Long Short-Term Memory performed better than machine learning techniques, demonstrating greater stability and efficiency in identifying complex patterns. Thus, this research demonstrates the effectiveness of integrating metrics within deep learning approaches for fault prediction.

Keywords: Software Fault Prediction; Software Metrics; Software Reliability; Deep Learning; Process Metrics

Introduction:

Software Fault Prediction (SFP) helps identify faulty modules during the development process. It helps to determine whether a software module is defective or not [1]. This serves as a safeguard against unexpected risks and ultimately increases software reliability. The key components of SFP models include data acquisition (product and process metrics), data preprocessing (cleaning, feature engineering, scaling), model selection (machine learning and deep learning algorithms), along with performance metrics such as accuracy, precision, recall, and F1 score [2]. Software metrics are essential for SFP because they are numerical measurements that offer objective information on code complexity, quality, performance, maintainability, and development efficiency. In order to find patterns and correlations associated with software errors, machine learning (ML) models are trained using these metrics as features [3]. Halstead metrics [4] McCabe metrics [5], Chidamber and Kemerer metrics [6], object-oriented metrics [7] and component-level metrics [8] are examples of product metrics, which quantify software attributes like size and complexity. Process metrics, on the other hand, offer insights into the software development process and how changes over time impact software quality. A developer's experience [9] and a software's change history [3] are two examples of sources from which process metrics can be obtained. Developer metrics, code churn, the number of revisions, lines added or removed, and the software's age are examples of common process metrics. At first, SFP made extensive use of product metrics such as McCabe metrics, which were created in the 1970s [5], because they could provide quantitative information on the features of the code. Process metrics, which represent the dynamics of software development and evolution, have also evolved to be an essential part of SFP over time. When combined, these metrics play a key role in creating prediction models that accurately identify faults.

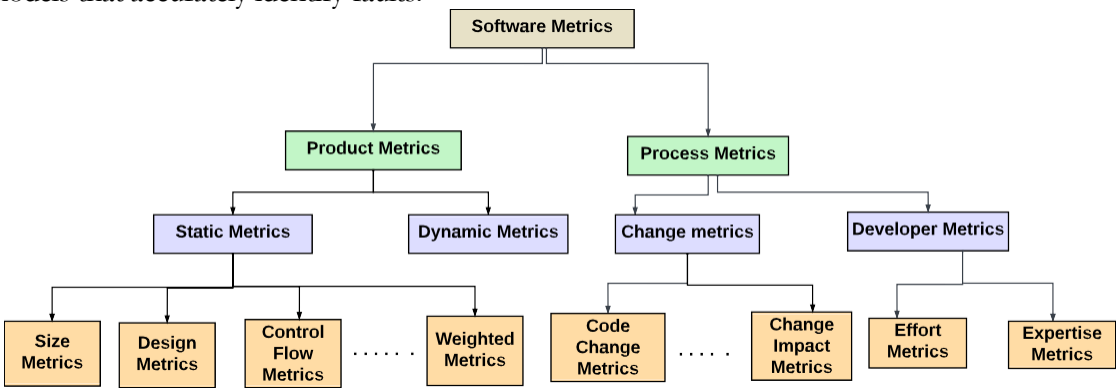

Figure 1. Metrics Categorization

A detailed classification of software metrics is shown in Figure 1. From Figure 1, we can observe that the software metrics are divided into product metrics, process metrics, and the related subcategories. This figure provides an overview of the structure and variation of these measures in the literature. Recognizing this classification helps researchers feel assured about the foundation of the analysis, which is essential for building trust in the study's approach. This classification is crucial to the study because it lays the groundwork for a comprehensive analysis of process and product metrics. The figure supports the proposed approach of integrating both metrics into deep learning (DL) models by highlighting the importance of considering them together rather than separately, which aligns with the study's objective.

Early detection of faulty modules of software development enables the testing team to use resources more effectively, ensuring the timely delivery of a high-quality product. In [10] demonstrated how predictive models may be used to find faults early in the software development lifecycle (SDLC), which lowers maintenance costs and increases dependability. It has been shown that using automated learning methods and historical software data can greatly increase the accuracy of problem identification and reduce the need for manual

inspection [11]. ML techniques hold great promise for early defect prediction in the SDLC by uncovering hidden patterns in historical data [12]. In recent years, ML and ensemble learning approaches have drawn a lot of interest for their ability to predict software faults using software metrics and historical fault data [13].

DL techniques have become a powerful way to predict software faults in recent years. But most of the research that has already been done has focused on using product metrics as input for these models. Product metrics offer only a partial view of the software development process, yet they are useful. The potential of integrated metrics, which incorporate both product and process, must be investigated in order to increase the precision and efficacy of DL models. In the field of software defect prediction, DL has become an effective method that greatly outperforms traditional ML methods. To the best of our knowledge, limited research has been done on integrating process and product metrics into DL models. By combining process and product data with DL models, our method aims to fill this gap and exploit their potential for more accurate and reliable defect predictions. In this study, we performed two different types of comparison. First, we compared product metrics with combined metrics (Product + Process) using DL to determine which metrics perform better. Second, we compared ML with DL methods using combined metrics to see which model gives better results. We utilized K-Nearest Neighbors (KNN), Naive Bayes (NB), and Logistic Regression (LR) as ML models, and Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM), and Bidirectional LSTM (BiLSTM) as DL models. The dataset used in this study was sourced from the AEEEM public repository [14], which contains software project data, metrics, and fault labels.

Research Gaps and Motivation:

DL and ML models are used to predict software faults and have achieved significant performance improvement, but there are still several issues that the current research has not addressed.

Earlier research has shown that, rather than utilizing both product and process metrics, they primarily focused on product metrics (such as complexity, size, and structural attributes). It has been unclear how using both sets of metrics will affect model performance, as previous research has consistently shown that process metrics have predictive significance beyond product metrics.

Secondly, previous studies rarely examine the effects of switching from product-only data to integrated product and process metrics on model performance. As a result, it is currently unclear how adding process-related data affects predictive efficacy.

Thirdly, even though both DL and ML models have been extensively used in this field, the majority of research assesses them separately rather than inside an integrated experimental approach.

Furthermore, there hasn't been a thorough comparison of DL and conventional ML techniques using combined metrics.

Research Objectives:

To enhance the quality of the software fault prediction process using DL algorithms.

To evaluate DL outcomes, it is important to compare them directly with ML results, using combined metrics (Product + Process).

To compare the results of the product metrics dataset with those of the combined dataset.

Novel Contribution:

The contribution of this paper is that we investigated the combined metrics (product + process) on both ML and DL models and carried out two different kinds of comparisons. First, we evaluated the effectiveness of DL models using the combined metrics dataset in comparison to the product metrics dataset. Second, we analyzed the results of DL and ML models using the combined metrics dataset.

This research paper is organized as follows: Section 2 conducts a detailed literature review of SFP using DL methods. Section 3 presents the proposed methodology, while Section 4 outlines the comparisons. Section 5 details the results; Section 6 discusses the significance of the results and practical implications. Lastly, Section 7 concludes and outlines future directions.

Literature Review:

Recent improvements in computing power and memory capacity in contemporary computer architectures have led to a notable evolution of conventional ML methods. DL algorithms have greatly improved advanced performance in a variety of software engineering activities, including code generation, defect prediction, software testing, and program recovery [15]. The idea of "deep learning" was presented by [16]. Researchers at academic institutions and industry practitioners alike are very interested in DL methods due to their success in image recognition and data mining. Consequently, they are currently investigating and utilizing DL algorithms for a variety of software engineering tasks, such as software implementation, requirements analysis, software testing, debugging, software design & modeling, and maintenance. As a subset of ML, DL utilizes both supervised and unsupervised methods to learn from neural networks. It excels at handling massive data volumes, discovering patterns from unlabeled data, and sharing learned representations across tasks. In [17] conducted a study on DL techniques, focusing on RBFN, BILSTM, and LSTM algorithms using two preprocessed datasets. Through extensive experiments, they found that LSTM and BILSTM achieved high accuracies of 93.53% and 93.75%.

In [18] proposed an enhanced CNN designed to identify defective instances from historical datasets sourced from the Tera-PROMISE data repository. The performance of the proposed enhanced CNN was compared with that of Li's CNN model. The results demonstrate that the enhanced CNN outperformed Li's model, achieving an average accuracy of 77%. The study concluded that the effectiveness of enhanced CNNs is significantly influenced by the quality of the input data and the architecture of the model. In [19] presented two DL models, Squeeze Net and Bottleneck, which were tested on seven NASA datasets. To address class imbalance and overfitting, SMOTE was employed in conjunction with the DL models. Squeeze Net achieved an F-measure of 0.93 ± 0.014, and Bottleneck reached 0.90 ± 0.013, showing that combining SMOTE with these models enhances predictive performance. In [20] presented a software defect prediction (SDP) model using CNN and BILSTM, which was tested on the GHPR dataset containing 6052 instances. In the CNN model, RELU activation functions were used for the input and hidden layers, and a Sigmoid function for the output layer, with Min-Max normalization employed for data preprocessing. CNN achieved an accuracy of 0.81%, while the BI-LSTM reached 0.80%. The study emphasizes the importance of conducting separate analyses of both models to select the most suitable one based on specific problem requirements.

In [21] introduced NB, DNN, and LSTM for predicting software defects using the PROMISE dataset. The proposed DNN model, with a softmax output layer, two hidden layers of fully connected LSTM units, and a 0.1% dropout rate, achieved the highest accuracy at 80.89%. In [22] they proposed a model, KPCA-ImbDNN, that uses Kernel Principal Component Analysis (KPCA) and a deep neural network (DNN) to capture feature insights and explore complex relationships. It addresses class imbalance with a weighted loss function and bootstrapping. Comparative analysis against SVM, RF, and Kernel Principal Component Weighted Ensemble (KPWE) demonstrated that KPCA-ImbDNN outperformed these existing approaches. In [23] proposed an enhanced CNN model that utilizes the Combined Defect Analysis (CDA) dataset and the PROMISE Source Code (SPSC) dataset for defect prediction. The ECNN model demonstrated superior performance with an F-measure of 0.660 on the SPSC dataset and an F-measure of 0.763 on the CDA dataset. Using LSTM and BiLSTM layers to examine software metrics, [24] presented a unique RNN-based deep learning

(RNNBDL) method for software fault prediction. The study made use of a novel, publicly accessible dataset called SFP XP-TDD, which was created from software projects that employed TDD and XP approaches. The DL model often produced more accurate results when compared to five conventional classifiers and related Rotation Forest (RF) ensembles. At the end, the study found that DL works better than ML for SFP in large datasets, with a 95% confidence level.

In order to address issues with software quality, reliability, and development costs, investigated SFP models. Two ML models, decision-tree regression and K-Nearest Neighbor were proposed. The implementation of a specialized metric suite that included Agile-based (Ma) and Requirement-based (Mr) metrics to be used as the input for fault prediction was a key contribution of this study. The researchers showed that the DTR-based model outperformed the KNN technique using the PROMISE repository. In the end, these ML frameworks help to optimize software standards and project budgets by helping with early detection of defects. A deep neural network (DNN) architecture for software defect prediction was presented by [25]. It uses 2D convolutional layers through a 3D representation of metrics. The investigation used random oversampling and dropout layers to address class imbalance and overfitting using the BugHunter dataset. With an average F1-score of 84.08%, the suggested DNN-17 model outperformed thirteen baseline models and represented a 20.01% improvement above conventional benchmarks. This work shows that the reliability of defect detection in complicated systems can be greatly improved by using specific DL architectures.

In [10] they investigated automating defect detection during the design and coding stages of the software development lifecycle (SDLC) by using ML. The study benchmarked Decision Tree and KNN models using a dataset of 10,885 modules with 22 static metrics. According to the experimental data, KNN performed better, recognizing a greater percentage of defective components with a recall of 0.906 and an AUC of 0.624. The authors come to the conclusion that switching to predictive, ML-driven quality assurance greatly lowers maintenance costs and boosts overall software dependability. In [11] utilized the JM1, CM1, PC1, and JM2 datasets to assess ML classifiers, such as SVM, NB, Random Forest, and KNN, for software fault prediction. The researchers obtained accuracy rates ranging from 98% to 99% by using 10-fold cross-validation and correlation-based feature selection. The best models for practical applications were found to be Random Forest and SVM. In the end, the research we conducted shows that ML-driven tools can greatly increase software reliability while lowering the requirement for manual inspection.

Using the PROMISE dataset, [26] assessed the effectiveness of Random Forest, XGBoost, and CNNs for early software problem identification. The research technique used SMOTE and Mutual Information-based feature selection to address important concerns such as class imbalance and feature redundancy. The use of Explainable AI (SHAP) to offer insights into model decision-making, emphasizing metrics such as cyclomatic complexity as significant defect indicators, was a key component of this study. With an F1-score of 0.82 and an AUC of 0.90, XGBoost outperformed both the CNN and Random Forest designs when validated using 5-fold cross-validation. In order to maintain maintenance costs and increase software reliability, the study ultimately improves the integration of these interpretable ML models into continuous development pipelines.

Materials and Methods:

This section presents the proposed methodology as shown in Figure 2. In this work, we used publicly available datasets that were downloaded from a public repository.

Subsequently, each dataset underwent a preprocessing phase, as discussed below. The preprocessed data was then applied to train ML and DL models; these models were chosen because they have been used in the literature. Moreover, the reliability of our study is demonstrated by its consistent performance in previous studies. Then, two types of

comparisons have been performed: (a) Comparison of product metrics only and combined metrics using DL models. (b) Comparing the performance of ML and DL models using combined metrics, predictions were made to categorize the data as faulty or non-faulty based on these models.

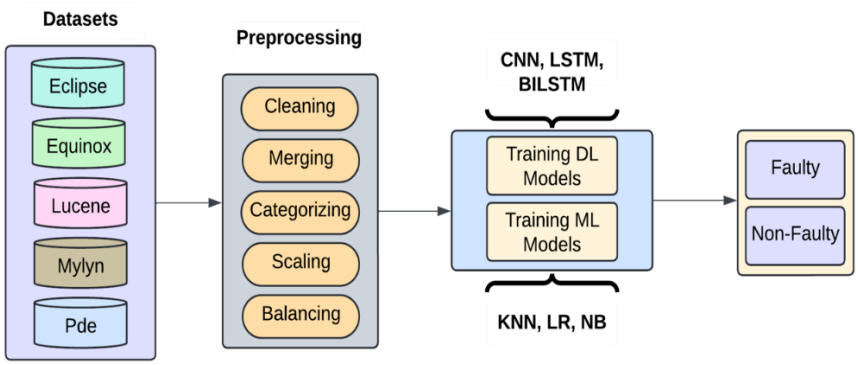

Figure 2. Proposed Methodology

Datasets:

The datasets are selected from the AEEEM public repository because of their exceptional multidimensionality as a key standard in the field of software defect prediction research. AEEEM uses a combination of both product metrics and process metrics, in contrast to traditional datasets like PROMISE or NASA that often use just product metrics. This is an essential part of our approach because it is based on DL methods, and it uses a variety of features to identify relationships between the evolution of code and its structural complexity. All five data sets, such as Lucene, JDT, Equinox, Mylyn, and PDE, provided by AEEEM are utilized in this study to optimize the generalizability of the experiment outcomes. These datasets include various application domains and software project architecture patterns. The dataset characteristics are presented in Table 1.

Table 1. Description of Dataset

Dataset	Description	Instances (Faulty, Non-Faulty)	Imbalanced Ratio
Eclipse JDT Core	Java Development Tools for the Eclipse IDE.	997 (206, 791)	3.84
Equinox Framework	Core runtime system for Eclipse	324 (129, 195)	1.51
Apache Lucene	High-performance, full-featured text search engine	691 (64, 627)	9.80
Mylyn	Task management tool for the Eclipse IDE	1862 (245, 1617)	6.60
Eclipse PDE UI	Plug-in Development Environment for Eclipse	1497 (209, 1288)	6.16

The datasets are named as Eclipse JDT core, Equinox framework, Apache Lucene, Mylyn, and Eclipse PDE UI. Each dataset contains both process and product metrics, with fifteen and seventeen metrics, respectively. From each dataset, we acquired these three files: Target File, Process Metrics, and Product Metrics File. All three files contain additional columns like unnamed, class name, Non-Trivial Bugs, Major Bugs, Critical Bugs, and High Priority Bugs columns, etc., representing the extensive details related to each project's bugs during the development process. We removed these additional columns because we only need one target label, which was the bug column. We utilized 16 columns from process metrics, 17 from product metrics, and 1 from the target file (Bug column). The details of the chosen metrics are mentioned in Tables 2 and Table 3.

Table 2. Product Metrics

Metric Type	Description
CBO (Coupling Between Objects)	Measures the number of classes a class is coupled to.
DIT (Depth of Inheritance Tree)	Depth of a class in the inheritance hierarchy.
Fan In	Number of classes referencing a given class.
Fan Out	Number of classes referenced by a given class.
LCOM (Lack of Cohesion of Methods)	Calculates unconnected method pairs in a class.
NOC (Number of Children)	Number of subclasses inheriting from a class.
Number of Attributes	Total attributes in a class.
Number of Attributes Inherited	Attributes inherited from parent classes.
Number of Lines of Code	Total lines of code in a class.
Number of Methods	Total methods within a class.
Number of Methods Inherited	Methods inherited from parent classes.
Number of Private Attributes	Count of private attributes in a class.
Number of Private Methods	Count of private methods in a class.
Number of Public Attributes	Count of public attributes in a class.
Number of Public Methods	Count of public methods in a class.
RFC (Response for Class)	Number of methods a class can invoke.
WMC (Weighted Methods per Class)	Aggregates method complexities in a class.

Table 3. Process Metrics

Metric Type	Description
Number of Versions until.	Total software versions up to a point.
Number of Fixes until.	Total bug fixes up to a point.
Number of Refactoring until	Code refactoring performed up to a point.
Number of Authors until	Total contributors to the codebase.
Lines Added Until	Cumulative lines added up to a point.
Max Lines Added Until	Max lines added in a single change.
Avg Lines Added Until	Average lines added per change.
Lines Removed Until	Cumulative lines removed up to a point.
Max Lines Removed Until	Max lines removed in a single change.
Avg Lines Removed Until	Average lines removed per change.
Code Churn Until	Cumulative code changes (additions/deletions).
Max Code Churn Until	Max code churn in a single change.
Avg Code Churn Until	Average code churn per change.
Age With Respect To	Age of code relative to a point.
Weighted Age with Respect To	Code age weighted by change frequency.

Data Preprocessing:

Figure 3 lists the preprocessing steps that were performed on the datasets.

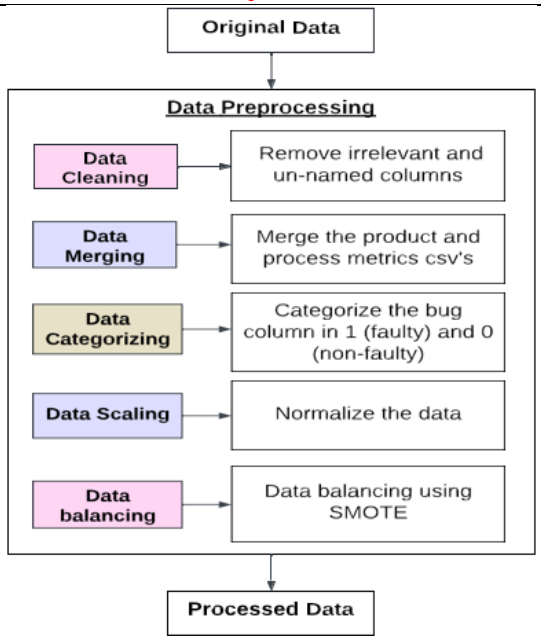

Figure 3. Processing Steps

Data Cleaning:

By eliminating columns that are not necessary for our research, we have simplified our dataset in order to improve model performance. Although the Non-Trivial Bugs, Major Bugs, Critical Bugs, and High Priority Bugs columns offer useful information, our main goal is to distinguish between faulty modules and those that are not. Therefore, we have just used the "Total Bugs" column since it gives us the information, we need to achieve our goals without requiring us to classify the various bug types. Furthermore, we continue to use the "class name" field for merging. By doing this, we can remove unnecessary complexity from the dataset and focus on the total number of bugs, which is essential for our research.

Data Merging:

For the product metrics dataset, we merged the product metrics file with the target file. For the combined metrics dataset, we merged the product, change-metrics, and target files. This merging was performed based on the class name column, because it was the unique identifier column among all the files. After merging, the class name column was removed from the merged file. To improve defect detection performance, we propose a method that combines product and process metrics into a single input feature set for DL models. This technique takes advantage of both the structural characteristics and the temporal dynamics of software development.

Data Labeling:

Our dataset was originally a regression dataset that represented the number of faults. So, we converted the dataset into a binary classification dataset by categorizing the bug column as faulty (1) or non-faulty (0). We chose binary classification because (as our objective was to identify faulty versus non-faulty modules) it has been commonly used in the literature, and we want to compare our results with those of other studies. This was done by converting all bug column instances with a value greater than 1 to label 1, and all values with 0 remained the same.

Data Scaling:

When working with numerical attributes, normalization is utilized to rescale data to a standard range based on an equation. We utilize the scikit-learn library's Standard Scaler in Python. The Standardization formula is represented by equation (i) below. After applying Z-score normalization, the data align with a standard normal distribution, typically within the range of

[-1, 1], making it easier to differentiate between faulty and non-faulty modules. For all the features except the bug column across the 5 datasets, normalization ensured that each attribute was transformed to have a mean of 0 and a standard deviation of 1. The original ranges, including minimum, maximum, and average values, varied across features and datasets but were effectively standardized, enhancing the comparability and interpretability of the data.

$$z = \frac{x-\mu}{\sigma} \qquad \text{(i)}$$

where:
z is the Z-score (normalized value),
x is the original value,
μ is the mean of the feature,
σ is the standard deviation of the feature.

Data Balancing:

Due to the uneven distribution of defective and non-defective instances in defect datasets, where defective modules are much less common, we performed data balancing [27]. Since the data was imbalanced, the minority class instances are less as compared to the majority class, so we balanced the data by using the over-sampling technique. We used the Synthetic Minority Over-sampling Technique (SMOTE) [28] before training the model to mitigate the class imbalance that existed in all of our datasets. By using this method as presented in equation (ii), we can balance the dataset and enhance the model's ability to learn from all classes by using this approach to generate synthetic examples for the minority class. To ensure an equal split of data, interpolation between nearest neighbors is used to create synthetic minority samples. A single feature set was then created by combining process and product metrics. The DL model is subsequently trained using this balanced and enhanced dataset. The capacity of this model to learn from minority class cases is improved by the addition of SMOTE, which raises overall prediction performance as follows.

$$x_{new} = x_i + rand(0,1) * (xnn - x_i) \qquad \text{(ii)}$$

where:
x_i= original minority sample
xnn = one of the k-nearest neighbors of xix_ixi
$rand(0,1)$= random value between 0 and 1
x_{new}= generated synthetic sample

Deep Learning Models:

The process of choosing a model comes after the data has been preprocessed and is ready. Based on the literature, we have selected three models: CNN, LSTM, and BILSTM. The selection of these models was based on their balanced abilities to handle sequential and structured data. CNNs can identify patterns [26] because they are good at extracting local and spatial characteristics. LSTMs are perfect for tasks involving temporal or contextual information because of their ability to capture long-term dependencies in sequential data. By processing input sequences [29] both forward and backward, BiLSTMs improve upon LSTMs and offer a more comprehensive contextual knowledge. When combined, these models offer a comprehensive approach: CNNs efficiently extract features, while LSTM and BiLSTM manage temporal interactions and sequence modeling, encompassing a broad range of DL capabilities. Because DL may discover intricate and non-linear relationships in the data that traditional ML models frequently find difficult to identify, DL models have significantly improved accuracy over traditional ML models. The subsections describe the details of DL models that we used for experimentation.

Convolutional Neural Networks:

The first DL model that we used is a 1D CNN for classification purposes. Figure 4 shows the proposed CNN architecture, and the hyperparameter settings are shown in Table 4.

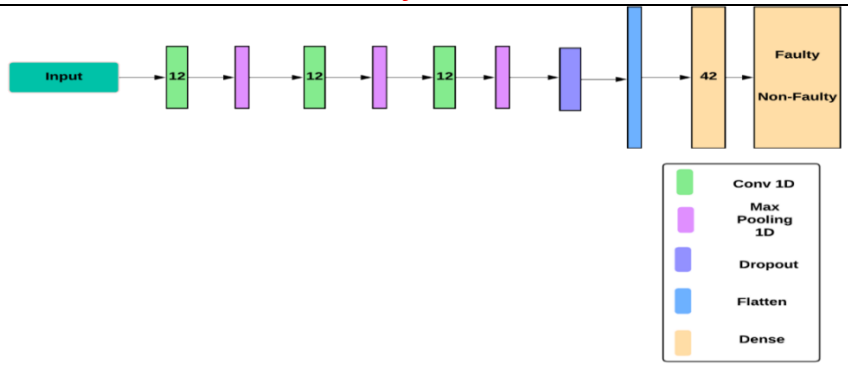

Figure 4. CNN Architecture for Software Fault Prediction

A 1D feature vector of length N is used in the suggested 1D CNN model, where N is the number of software metrics for each instance. To make the input data compatible with the Conv1D layer, it is transformed into a two-dimensional format of (N, 1). There are three consecutive convolutional blocks in the design. To gradually decrease the dimensionality of the feature maps, each block consists of a Conv1D layer with 12 filters, a kernel size of 4, and a ReLU activation function, followed by a MaxPooling1D layer with a pool size of 2. To reduce overfitting, a Dropout layer with a rate of 0.2 is placed after the convolutional layers. After that, a fully linked Dense layer with 42 neurons with ReLU activation receives the flattened feature maps. Lastly, binary classification (faulty vs. non-faulty) is carried out using a Dense output layer with a single neuron and sigmoid activation function. The Adam optimizer with binary cross-entropy as the loss function is used to train the model, which is built using the TensorFlow/Keras framework. The model is trained for 200 epochs using a batch size of 40, and the dataset is divided into training, validation, and testing sets 80:10:10.

Table 4. CNN Hyperparameters

Parameter	Value
Convolution Layer	3
Convolution Layer Filter	12, 12, 12
Convolution Layer Activation Function	ReLU
Optimizer	Adam
Loss Function	Binary Cross-Entropy
Batch Size	40
Epochs	200

Long Short-Term Memory:

LSTM networks are a subset of RNNs to identify patterns in data sequences using a memory cell, gates (forget gate, input gate, and output gate), and hidden states to control the flow of information. As new data is added by the input gate, the forget gate determines how much of the previous cell state should be retained. By merging new and retained data, the cell state is updated, and the output gate chooses which data is forwarded to the following hidden state. The presented LSTM architecture is shown in Figure 5, and the hyperparameter settings are shown in Table 5.

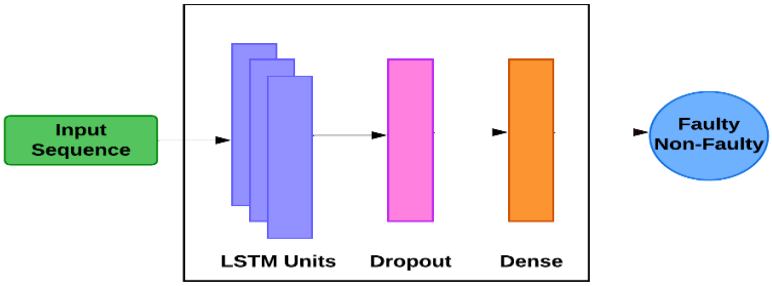

Figure 5. LSTM Architecture for Software Fault Prediction

Table 5. LSTM Hyperparameters

Parameter	Value
LSTM Layer 1	128 units
No. of dropout layers	2
Dropout ratio	0.2
LSTM Layer 2	64 units
Dense Layer	1
Activation Function	Sigmoid
Optimizer	Adam
Loss Function	Binary Cross-Entropy
Batch Size	128
Epochs	200

Bidirectional LSTM:

A BILSTM network is utilized in SFP to take advantage of its capability to consider both past and future contexts in sequential data. Figure 6 shows the proposed BILSTM architecture, and the hyperparameter settings are shown in Table 6. The BILSTM consists of a 128-unit Bidirectional LSTM returning sequences, followed by a 20% dropout layer. A second 64-unit Bidirectional LSTM with sigmoid activation and another 20% dropout layer is added. The final dense layer with 1 unit and sigmoid activation outputs the binary classification probability. Since the LSTM and BiLSTM models were chosen through rigorous tuning to achieve the best performance on the dataset, the same hyperparameters were used for both models. These parameters produced good results for the LSTM model, and a direct comparison focusing on the effect of bidirectionality was made possible by applying them consistently to the BiLSTM. We decided to use the same parameters for both LSTM and BiLSTM models to provide an objective and fair comparison between them. This method made sure that any variations in performance were mostly caused by the bidirectional structure of BiLSTM and not by variations in parameter settings. We were able to precisely evaluate the effect of bidirectionality by managing this variable. Additionally, since both models showed good performance under the same conditions, separate tuning was not necessary for this evaluation.

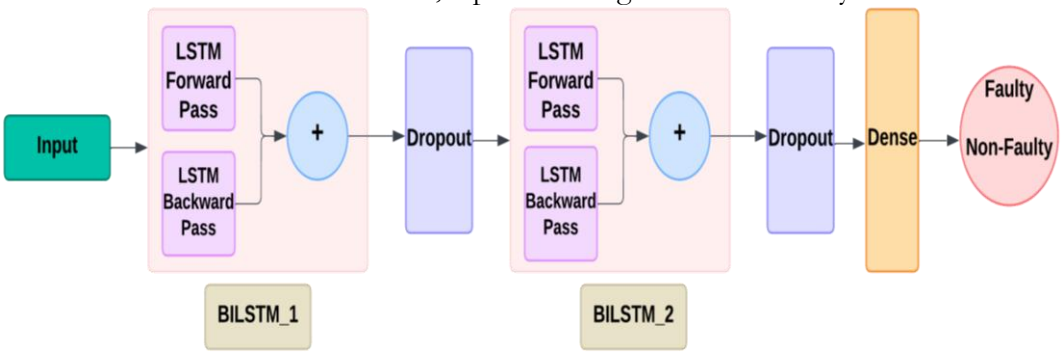

Figure 6. BiLSTM Architecture for Software Fault Prediction

Table 6. BiLSTM Hyperparameters

Parameter	Value
BiLSTM Layer 1	128 units
No. of dropout layers	2
Dropout ratio	0.2
BiLSTM Layer 2	64 units
Dense Layer	1
Activation Function	Sigmoid
Optimizer	Adam

Loss Function	Binary Cross-Entropy
Batch Size	128
Epochs	200

Comparison:

We performed two distinct types of comparison for the evaluation of DL models.

Comparison of product and combined metrics:

In metrics-based comparison, we used two types of metrics: product metrics and combined metrics (product + process). The analysis and comparison are performed using various DL models to evaluate their effectiveness in handling both types of metrics. Further, figure 7 shows the steps of metrics-based comparison. This process starts with gathering process metrics. After data preprocessing (cleaning, scaling, etc.), the dataset is split into 80%,10%, and 10% training, validation, and test sets. DL models, including CNN, LSTM, and BILSTM, were trained on the training set, and their performance was evaluated on the test set using metrics like accuracy, precision, recall, and F1-score. Their results are compared to analyze which dataset performs well.

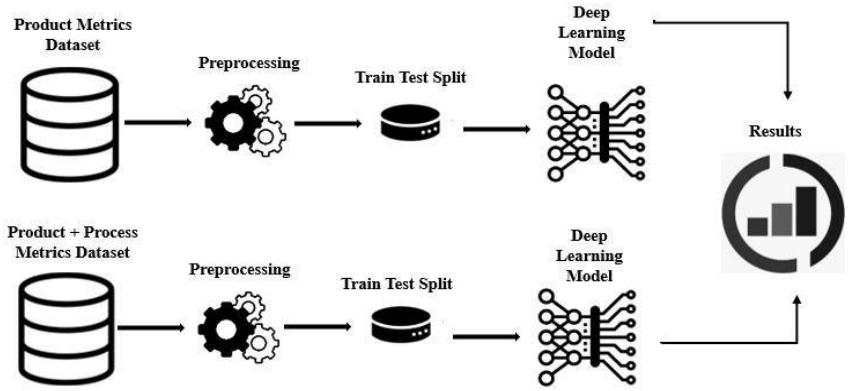

Figure 7. Methodology for the comparison of product and combined metrics

Comparison of ML and DL Models:

In model comparison, we employed both ML and DL models. The analysis is focused on the combined metrics dataset, where we compared the performance of DL models like CNN, LSTM, and BILSTM with ML models like KNN, NB, and LR to assess their accuracy and effectiveness across various evaluation criteria. Moreover, figure 8 shows model-based comparison steps. The process starts by creating a combined dataset. After cleaning and preparing the data, it's split into training, validation, and test sets. ML and DL models are trained and evaluated using accuracy, precision, recall, and F1-score. Their results are compared to analyze which model performs well using the proposed combined dataset.

K-Nearest Neighbors:

KNN is an easy-to-understand ML algorithm used for both classification and regression tasks. Essentially, it operates by identifying the 'nearest' known data points to a new data point using a selected distance metric, typically Euclidean distance [30]. The 'K' in KNN signifies the count of closest neighbors to consider. In classification, the class label of the new point is determined by the nearest neighbors.

Naive Bayes:

Naive Bayes is a probabilistic classification method that uses Bayes theorem, assuming that features are independent. The NB treats defect prediction as a binary classification problem. The Naive Bayes classifier is based on the Bayes theorem, which is stated as follows.

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}$$

Logistic Regression: To predict the likelihood that an observation will belong to a certain class (often represented as 1 or 0), logistic regression is employed in binary classification.

$$\hat{y} = b_1x_1 + b_2x_2 + \cdots + b_kx_k + a$$

The logistic model is based on the logistic function.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The result of applying the logistic function to the above regression equation is become:

$$\hat{y} = \sigma(b_1x_1 + b_2x_2 + \cdots + b_kx_k + a)$$
$$= \frac{1}{1 + e^{-(b_1x_1 + b_2x_2 + \cdots + b_kx_k + a)}}$$

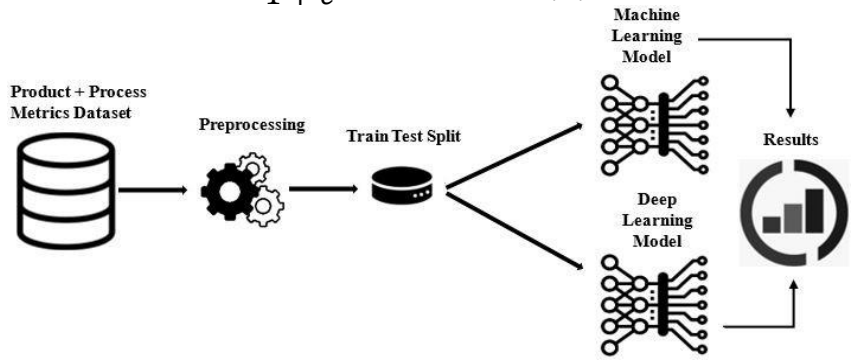

Figure 8. Methodology for the comparison of ML and DL models

Results and Discussion:

In this section, the results of metrics-based and model-based comparisons are displayed. These results are obtained using the experimental setup in which Adam optimizer, Binary cross entropy as loss function, 128 Batch size, with 200 Epochs was used. The evaluation metrics, F1 scores, accuracy, precision, and recall, were used to evaluate the models.

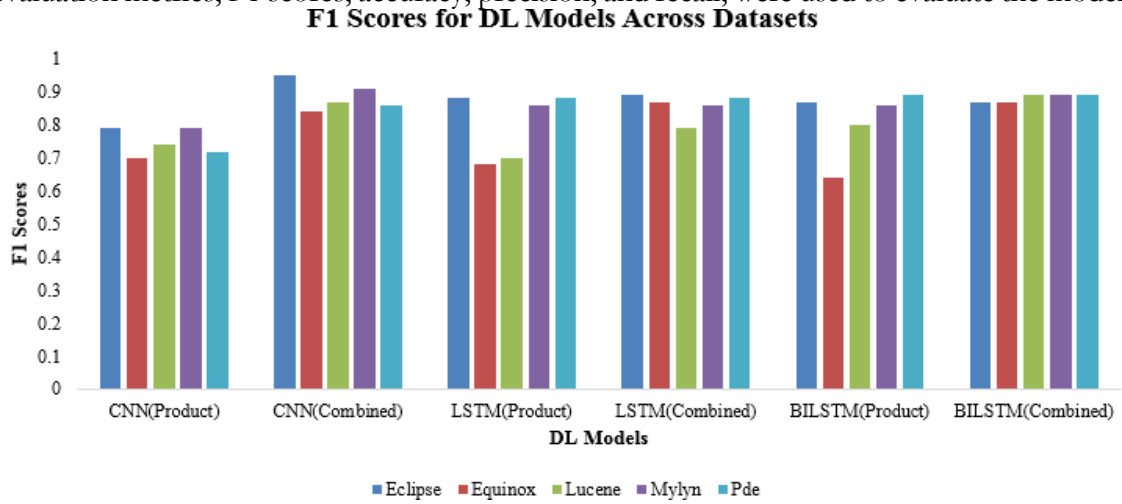

Figure 9. F1 Scores for DL Models across Datasets

Results for comparison of Product and Combined Metrics:

The results of the metrics-based comparison are shown in Table 7, referring to the F1 score results. The table indicates that when process and product data are combined, gradual improvement can be achieved. The CNN model increases on average from 0.75 to 0.88, which is a noticeable 0.13 gain in F1 score. In the same way, BILSTM rises from 0.81 to 0.88, and LSTM improves from 0.82 to 0.85.

The experimental results can be analyzed by visualizing the bar charts demonstrated in Figure 9. We started by evaluating the performance of the product metrics and combined the metrics datasets with DL models. Next, we evaluated both ML and DL models on the combined metrics datasets to conduct a comprehensive analysis. For the Eclipse, Equinox, Lucene, Mylyn, and PDE datasets, as shown in Figure 9. The CNN model demonstrates an

improved F1 Score with the combined dataset, suggesting that process metrics enhance the performance. The LSTM and BILSTM models show minimal changes with the combined dataset for Eclipse, where CNN models performance increases from 0.79 to 0.95, indicating that process metrics provide substantial additional information for fault prediction, while in the Equinox dataset, all models (CNN, LSTM, and BILSTM) show improved performance with the combined dataset, particularly BILSTM, which improves from 0.64 to 0.89, highlighting the strong impact of combined metrics in this case. For the Lucene dataset, the CNN and BILSTM models perform better with the combined data, while the LSTM shows stable performance across both datasets. In the Mylyn dataset, the CNN and BILSTM show marginal improvement with the combined data, but the LSTM performs consistently well on both datasets. Similarly, in the PDE dataset, the CNN and BILSTM models improve with the combined dataset, while the LSTM's performance remains the same across both datasets. Overall, the variations in the results are due to the performance of the dataset being influenced by its characteristics, and the number of instances. Each project dataset has unique characteristics, including its code base, commit history, bug reports, and development practices, which impact DL models differently. The analysis suggests that integrating process data into product generally leads to improved or stable performance across most projects, with models like LSTM and BILSTM compared to the CNN models. This underscores the importance of incorporating both product and process metrics in predictive modeling to enhance the robustness and accuracy of the results. Consequently, variations in results are natural and can be attributed to these dataset-specific differences, such as higher code complexity or varying commit frequency in different projects.

Table 7. F1 Scores of Products and Combined Metrics

	Product Metrics			**Combined Metrics**		
	CNN	LSTM	BILSTM	CNN	LSTM	BILSTM
ECLIPSE	0.79	0.88	0.87	**0.95**	0.85	0.84
EQUINOX	0.70	0.68	0.64	0.83	**0.86**	**0.89**
LUCENE	0.74	0.79	0.80	0.86	0.79	**0.88**
MYLYN	0.79	0.86	0.86	**0.92**	0.86	0.88
PDE	0.72	0.88	0.89	0.85	0.87	**0.90**
Average	0.75	0.82	0.81	**0.88**	**0.85**	**0.88**

Table 8. F1 Scores of ML and DL models

	ML Models			**DL Models**		
	KNN	NB	LR	CNN	LSTM	BILSTM
ECLIPSE	0.94	0.62	0.83	**0.95**	0.85	0.84
EQUINOX	0.75	0.86	0.87	0.83	0.86	**0.89**
LUCENE	0.84	0.56	0.71	0.86	0.79	**0.88**
MYLYN	0.91	0.64	0.75	**0.92**	0.86	0.88
PDE	0.87	0.68	0.73	0.85	0.87	**0.90**
Average	0.86	0.67	0.78	**0.88**	**0.85**	**0.88**

Results for Comparison for ML and DL models:

The results of the comparative analysis of ML and DL are shown in Table 8, referring to the F1 score results. Although DL models (CNN: 0.88, BILSTM: 0.88, LSTM: 0.85) perform better on average than the majority of conventional ML models, the difference is not consistently great across all datasets, suggesting that model efficacy is context-dependent. A more detailed review shows that KNN performs better, particularly in the Eclipse dataset (0.94), which is about equal to CNN (0.95). This suggests that for datasets with well-separated feature distributions, simpler models like KNN can perform better than DL methods.

However, KNN's performance declines in Equinox (0.75), highlighting the risk to data distribution and scaling limitations.

NB performs poorly on the majority of datasets, with Lucene showing not good results (0.56), suggesting that its strong independence assumption is inadequate for capturing the intricate relationships found in software metrics data. Although LR performs moderately (average 0.78), it is not as competitive as DL models due to its inability to accurately predict nonlinear interactions. With an accuracy of 0.89 on Equinox and 0.90 on PDE, BILSTM outperformed all other DL models in terms of consistency across the datasets. This implies that it is more effective to capture feature interdependence by framing the features as a bidirectional sequence. CNN outperforms LSTM, as displayed by its scores of 0.95 on Eclipse and 0.86 on Lucene. This suggests that CNN captures local patterns better than LSTM. Conversely, LSTM performs more consistently and somewhat less effectively, indicating that it provides no actual benefit for this task. The experimental results can be analyzed by visualizing the bar charts as shown in Figure 10.

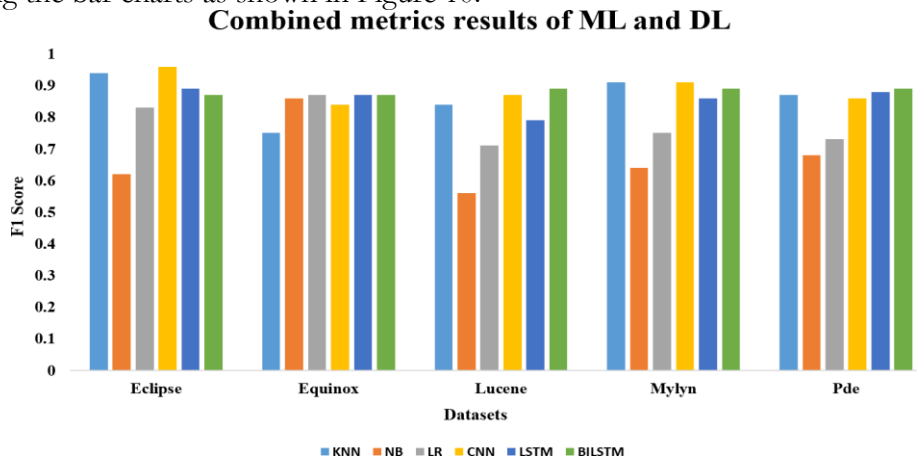

Figure 10. Analysis of ML and DL models

Overall, DL models like CNN and BILSTM are shown to be preferable due to their consistent high performance, making them suitable for tasks requiring high accuracy and reliability in SFP. This underscores the importance of selecting the appropriate model and tuning it to the specific dataset to achieve optimal results. Furthermore, in Figure 11 and Figure 12, the accuracy and loss graphs for training and validation give an accurate representation of the model's performance. The accuracy plot demonstrates a consistent improvement over the epochs, with both training and validation accuracy rising before reaching a high degree of stability. In the same way, the loss plot consistently declines, suggesting that the model was learning successfully. The close trend between training and validation metrics suggests that the model generalizes well. These charts provide a better understanding of the model's learning process and help in clearly proving the performance claims in the results section. They demonstrate that, whereas traditional ML models may be competitive in certain scenarios but are unable to generalize, DL models perform more consistently and accurately over a range of datasets. This highlights the need to select models based on dataset characteristics and the advantages of DL methods for handling complicated, explainable software metrics data.

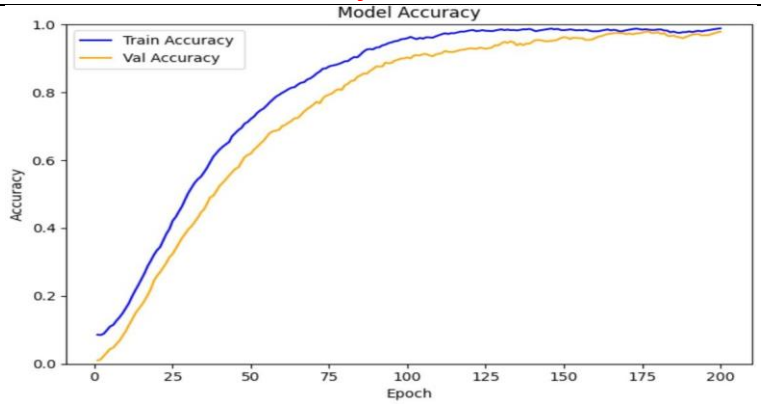

Figure 11. Validation accuracy curve for the CNN model trained on the Eclipse dataset using combined metrics.

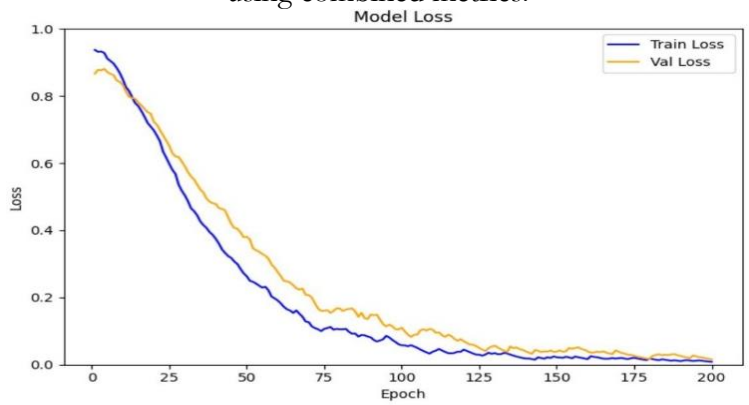

Figure 12. Validation loss curve for the CNN model trained on the Eclipse dataset using combined metrics.

Discussion:

Software metrics are essential to software defect prediction because both product and process metrics influence model performance. This study highlights the significance of using process metrics to improve prediction accuracy, whereas prior research has mainly focused on product metrics. The constant improvement across several datasets, as shown in Figure 13. Figure 13 indicates the integrating process, where product metrics yield related information rather than redundant features. This pattern implies that process metrics capture aspects of software development while product metrics alone do not, such as temporal and evolutionary aspects. The related nature of process and product data highlights the improvement achieved with combined metrics. While process metrics show the software's evolution, including code modifications along development activities, and product metrics capture the structural characteristics of the code. The model can learn deeper and more discriminative patterns linked to software errors because of this combination. Prior studies have mostly used ML and DL approaches with product metrics; however, there has not been much focus on integrating process metrics. Thus, our experimental results show that integrating process and product metrics consistently improves performance across several datasets, as shown in Figure 13, which emphasizes the significance of combined feature sets in fault prediction applications.

Furthermore, the trends in Figure 14, where we can observe that DL models constantly outperform conventional ML techniques across datasets, which provides a clearer understanding of the superior performance of DL models, especially CNN and BILSTM. This highlights that the combined metrics datasets, such as non-linear interactions are better captured by DL models.

DL models make better use of the larger merged dataset since they automatically extract structural features, in contrast to typical ML models that mostly rely on manual feature selection.

Moreover, DL models are theoretically justified by their capacity to learn hierarchical features, which makes them suitable for handling complex and large data sets, as summarized in Figure 14. Figure 14 supports the concept of representation learning, which shows that input features such as combination metrics contribute to the model's increased ability for generalization.

The above discussion concludes that the performance of fault prediction models is significantly enhanced by the use of DL techniques combined with metrics from the product and process domains. As a result, Figure 13 and Figure 14 demonstrate the importance of both features and models' capabilities for making reliable predictions.

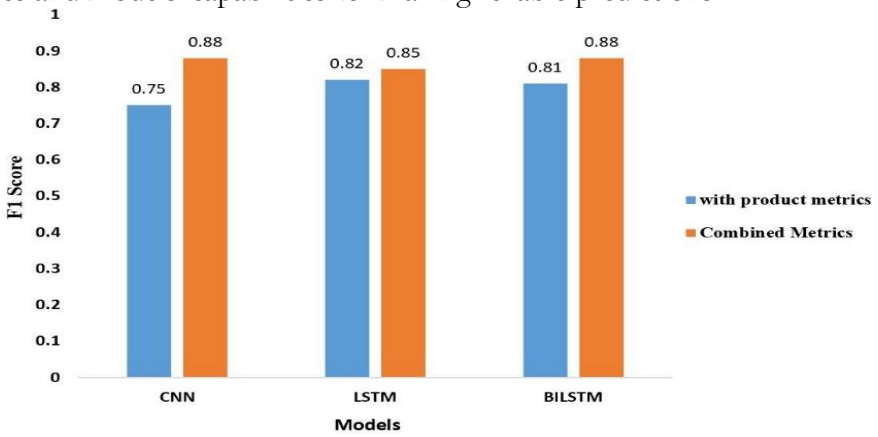

Figure 13. Comparison of product and combined metrics for DL models

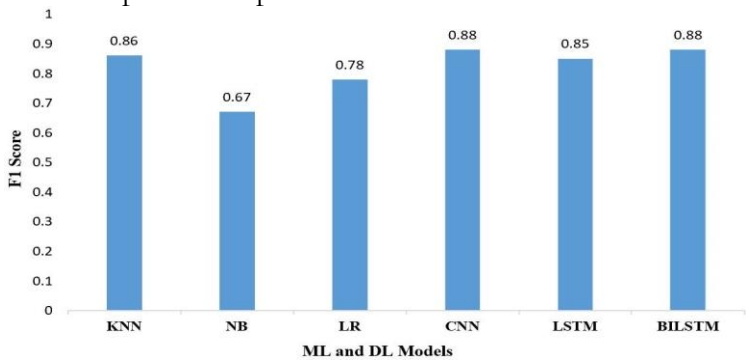

Figure 14. Comparison of ML and DL models

Practical Implication:

The suggested methodology is applicable in today's software development settings. Development teams can prioritize testing and debugging efforts, saving time, money, and post-release problems, by precisely identifying defect-prone modules early in the SDLC. The incorporation of process metrics, such as code churn, revisions, and developer activity, provides a deeper understanding of software development than conventional methods that rely solely on product metrics. As a result, the model is more suited for real-world projects where defect occurrence is influenced by both code structure and development dynamics.

Additionally, the proposed method can be easily incorporated into automated development workflows, such as enabling ongoing code quality monitoring and early identification of high-risk components. It enhances overall software reliability and promotes proactive quality assurance.

As a result, the model has both practical applications and theoretical significance for improving software quality, optimizing resource allocation, and making better decisions in real-world settings.

Conclusion:

This study introduces a novel approach that uses both process and product metrics to systematically assess DL-based software fault prediction, representing a major advancement in the area. Our research focused on analyzing both product and process metrics for fault prediction in software systems. We created a combined dataset from these metrics and used it for both training and testing. By integrating these diverse metrics, we aimed to enhance the accuracy of fault predictions. We conducted two types of experiments. First, we compared the results of ML models (KNN, NB Bayes, and LR) and DL models (CNN, LSTM, and BILSTM) using the combined dataset. This comparison enabled us to assess the performance of various modeling techniques in predicting software faults. Second, we specifically examined the performance of metrics using DL models to understand their impact on prediction accuracy. Additionally, our research revealed that adding process metrics to the product metrics significantly improved prediction results. This finding highlights the importance of considering both types of metrics in fault prediction models. In conclusion, the combined metrics identified in our study proved to be highly significant and effective for predicting software faults, demonstrating the value of a comprehensive approach to software fault prediction.

In the future, exploring more advanced DL models or ensembles could improve prediction accuracy by better capturing complex patterns in the data, leading to more reliable SFP. Validating the combined metrics approach on a variety of software datasets can test its robustness across different environments and fault types. Investigating dynamic metrics for real-time fault prediction would allow continuous monitoring of software changes, enhancing its timeliness and accuracy.

Acknowledgement: The author would like to thank Dr. Aamer Nadeem for his valuable guidance and support during this research.

Author's Contribution: The study was conceptualized and designed by Faryal Hayat in collaboration with Dr. Aamer Nadeem. Faryal Hayat carried out the data collection, analysis, and manuscript drafting with academic supervision and support from Dr. Aamer Nadeem. He reviewed the work and approved the final version of the manuscript.

Conflict of interest: The authors declare no conflict of interest.

References:

[1] Elena N. Akimova, Alexander Yu Bersenev, "A Survey on Software Defect Prediction Using Deep Learning," *Mathematics*, vol. 9, no. 11, p. 1180, 2021, doi: https://doi.org/10.3390/math9111180.

[2] S. Moudache and M. Badri, "Software fault prediction based on fault probability and impact," *Proc. - 18th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2019*, pp. 1178–1185, Dec. 2019, doi: 10.1109/ICMLA.2019.00195.

[3] Danijel Radjenović, Marjan Heričko, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013, doi: https://doi.org/10.1016/j.infsof.2013.02.009.

[4] "Elements of Software Science (Operating and programming systems series): | Guide books | ACM Digital Library." Accessed: Mar. 24, 2026. [Online]. Available: https://dl.acm.org/doi/10.5555/540137

[5] T. J. Mccabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976, doi: 10.1109/TSE.1976.233837.

[6] Shyam R. Chidamber, Chris F. Kemerer, "Towards a metrics suite for object oriented design," *OOPSLA '91 Conf. Proc. Object-oriented Program. Syst. Lang. Appl.*, 1991, [Online]. Available: https://dl.acm.org/doi/10.1145/117954.117970

[7] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002, doi:

10.1109/32.979986.
[8] P. Chandra and D. Sharma, "Towards recent developments in the methods, metrics and datasets of software fault prediction," *Int. J. Comput. Syst. Eng.*, vol. 6, no. 1, p. 14, 2020, doi: 10.1504/ijcsyse.2020.10031257.
[9] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014, doi: 10.1109/TSE.2014.2322358.
[10] V. Singh, A. Sahai, and P. Kumar, "Integrating ML Models Into The Software Development Lifecycle For Automated Fault Detection," *2025 5th Int. Conf. Adv. Electron. Commun. Eng. AECE 2025*, pp. 891–894, 2025, doi: 10.1109/AECE67531.2025.11386605.
[11] P. Bagla, R. Kumar, K. Kumar, K. P. Sharma, B. Punetha, and R. Malhotra, "Enhanced Software Fault Prediction Using Machine Learning," *2025 OITS Int. Conf. Inf. Technol.*, pp. 907–911, Dec. 2025, doi: 10.1109/OCIT66168.2025.11400152.
[12] A. Kundu, P. Dutta, K. Ranjit, S. Bidyadhar, M. K. Gourisaria, and H. Das, "Software Fault Prediction Using Machine Learning Models," *Proc. - 2022 OITS Int. Conf. Inf. Technol. OCIT 2022*, pp. 170–175, 2022, doi: 10.1109/OCIT56763.2022.00041.
[13] Nasraldeen Alnor Adam Khleel, Károly Nehéz, Montaser Fadulalla & Ahmed Hisaen, "Ensemble-Based Machine Learning Algorithms Combined with Near Miss Method for Software Bug Prediction," *Int. J. Networked Distrib. Comput.*, vol. 13, no. 11, 2025, [Online]. Available: https://link.springer.com/article/10.1007/s44227-024-00044-x
[14] "Find Open Datasets and Machine Learning Projects | Kaggle." Accessed: Mar. 31, 2026. [Online]. Available: https://www.kaggle.com/datasets
[15] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, "Deep learning-based software engineering: progress, challenges, and opportunities," *Sci. China Inf. Sci.*, vol. 68, 2025, [Online]. Available: https://link.springer.com/article/10.1007/s11432-023-4127-5
[16] G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006, doi: 10.1162/neco.2006.18.7.1527.
[17] I. Batool and T. A. Khan, "Software fault prediction using deep learning techniques," *Softw. Qual. J. 2023 314*, vol. 31, no. 4, pp. 1241–1280, Jun. 2023, doi: 10.1007/s11219-023-09642-4.
[18] Meetesh Nevendra and Pradeep Singh, "Software Defect Prediction using Deep Learning," *Acta Polytech. Hungarica*, vol. 18, no. 10, 2021, [Online]. Available: https://acta.uni-obuda.hu/Nevendra_Singh_117.pdf
[19] S. K. Pandey, A. Haldar, and A. K. Tripathi, "Is deep learning good enough for software defect prediction?," *Innov. Syst. Softw. Eng. 2023 212*, vol. 21, no. 2, pp. 501–516, Oct. 2023, doi: 10.1007/s11334-023-00542-1.
[20] Nasraldeen Alnor Adam Khleel, Károly Nehéz, "A new approach to software defect prediction based on convolutional neural network and bidirectional long short-term memory," *Prod. Syst. Inf. Eng.*, vol. 10, no. 3, pp. 1–15, 2022, doi: 10.32968/psaie.2022.3.1.
[21] Waleed Albattah, Musaad Alzahrani, "Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach," *AI*, vol. 5, no. 4, pp. 1743–1758, 2024, doi: https://doi.org/10.3390/ai5040086.
[22] Anh Ho, Nguyen Nhat Hai, "Combining Deep Learning and Kernel PCA for Software Defect Prediction," *ACM Int. Conf. Proceeding Ser.*, 2022, doi: https://doi.org/10.1145/3568562.3568587.
[23] K. Sekaran and L. S. P. Annabel, "A Deep Learning Based Model for Defect

Prediction in Intra-Project Software," *7th Int. Conf. Trends Electron. Informatics, ICOEI 2023 - Proc.*, pp. 1148–1155, 2023, doi: 10.1109/ICOEI56765.2023.10126014.

[24] "(PDF) Software Fault Prediction Using an RNN-Based Deep Learning Approach and Ensemble Machine Learning Techniques." Accessed: Apr. 20, 2026. [Online]. Available: https://www.researchgate.net/publication/367539625_Software_Fault_Prediction_Using_an_RNN-Based_Deep_Learning_Approach_and_Ensemble_Machine_Learning_Techniques

[25] Mehrasa Modanlou Jouybari, Alireza Tajary, "A novel deep neural network structure for software fault prediction," *PeerJ Comput. Sci.*, vol. 10, 2024, [Online]. Available: https://peerj.com/articles/cs-2270/

[26] D. Sharma, P. Kumar, and P. K. Rai, "Fault Prediction in Software Using Machine Learning," pp. 1411–1415, Dec. 2025, doi: 10.1109/CISES66934.2025.11265078.

[27] S. R. Goyal, "Current Trends in Class Imbalance Learning for Software Defect Prediction," *IEEE Access*, vol. 13, 2025, doi: 10.1109/ACCESS.2025.3532250.

[28] K. W. B. Nitesh V. Chawla, "SMOTE: Synthetic Minority Over-sampling Technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002, [Online]. Available: https://www.jair.org/index.php/jair/article/view/10302

[29] Alex Graves, Jürgen Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5–6, pp. 602–610, 2005, doi: https://doi.org/10.1016/j.neunet.2005.06.042.

[30] "Analysis of Software Project Reports for Defect Prediction Using KNN | Request PDF." Accessed: Mar. 24, 2026. [Online]. Available: https://www.researchgate.net/publication/275208478_Analysis_of_Software_Project_Reports_for_Defect_Prediction_Using_KNN