CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD

Deep Learning Based Software Fault Prediction using Process and Product Metrics

by

Faryal Hayat

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the

Faculty of Computing
Department of Computer Science

2024

I would like to dedicate my dissertation to my family and teachers. I am extremely grateful to my loving parents and brothers, who have been a constant source of encouragement and inspiration throughout my life. I also want to express sincere appreciation to my supervisor, whose unwavering confidence and guidance have helped me reach this milestone.

CERTIFICATE OF APPROVAL

Deep Learning Based Software Fault Prediction using Process and Product Metrics

by

Faryal Hayat

(MCS223002)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Majid Iqbal	COMSATS, Islamabad
(b)	Internal Examiner	Dr. Nayyer Masood	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

Dr. Aamer Nadeem
Thesis Supervisor
September, 2024

Dr. Abdul Basit Siddiqui
Head
Dept. of Computer Science
September, 2024

Dr. M. Abdul Qadir
Dean
Faculty of Computing
September, 2024

Author's Declaration

I, **Faryal Hayat** hereby state that my MS thesis titled "**Deep Learning Based Software Fault Prediction using Process and Product Metrics**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Faryal Hayat)

Registration No: MCS223002

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**Deep Learning Based Software Fault Prediction using Process and Product Metrics**" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Faryal Hayat)
Registration No: MCS223002

Acknowledgement

First and foremost, I would like to thank Allah (S.W.T) for blessing me with knowledge, strength, courage, and patience throughout the course of this research. I am deeply appreciative of my esteemed supervisor **Dr. Aamer Nadeem** for his invaluable guidance, meticulous oversight, and insightful feedback, which have been pivotal in shaping this thesis.

My heartfelt thanks extend to my beloved parents and supportive and encouraging brothers for their unwavering support, encouragement, and belief in my abilities during both triumphs and challenges.

Lastly, I would like to appreciate myself a little. Now, as I look back at my academic journey for completing this thesis, I thank myself for staying positive and having self belief in hard times, managing social life and home chores efficiently, staying focused on my research, staying steadfast and learning from every challenge I faced during this research . All of these efforts have proven to be very crucial in overcoming obstacles and achieving this academic milestone.

(Faryal Hayat)

Abstract

Software Fault Prediction (SFP) helps to identify potential faults and defects before they occur, enhancing overall software per formance. SFP research focuses on creating prediction models that use software metrics and fault data from previous releases to identify fault-proneness in new modules. While traditional defect prediction methods use product metrics like McCabe and Halstead etc. and process metrics like NumberOfVersions, Refactorings etc, recent advancements have combined process and product metrics to improve fault prediction accuracy. Data mining, machine learning, and deep learning have all emerged as popular techniques in this field.

Our proposed technique investigates the application of deep learning algorithms for analyzing process metrics. While machine learning algorithms have long been used for fault prediction, the rise of deep learning has produced promising results due to its ability to handle large datasets, to capture complex patterns and dependencies in the software fault data. Despite their potential, process metrics have not been widely used with deep learning models. The use of deep learning in SFP, especially with process metrics, could provide new insights and significantly improve fault prediction accuracy compared to current techniques. So, we chose CNN, LSTM and BILSTM as deep learning models.

In this thesis, we performed two type of comparisons: first, we compared product metrics with combined metrics (Product + Process) using deep learning, and second, we compared machine learning (ML) with deep learning (DL) methods using combined metrics. The combined metrics results are better with all deep learning models compared to the product metrics, and deep learning results are better than machine learning results when using combined metrics. Based on the first experiment, the results demonstrated that the CNN achieved 93% accuracy with upto 16% improvement over KNN using combined metrics. Furthermore, in second comparison DL model BILSTM outperformed ML model LR by achieving accuracy of 91% with upto 5% improvement.

Contents

List of Figures

List of Tables

Abbreviations

BILSTM	Bidirectional long short-term memory
CNN	Convolutional Neural Networks
CK	Chidamber and Kemerer
CBO	Coupling Between Objects
CDA	Combined defect dataset
DNN	Deep Neural Network
DM	Deep Mining
DBN	Deep belief network
DL	Deep Learning
HSBA	Hybrid Search Based Algorithms
KNN	K-Nearest Neighbour
LR	Logistic Regression
LOC	Lines Of Code
LSTM	Long Short-Term Memory
ML	Machine Learning
MLT	Machine Learning Technique
NB	Naive Bayes
NOC	Number Of Children
SDP	Software Defect Prediction
SFP	Software Fault Prediction
SDLC	Software Development Life Cycle
SMOTE	Synthetic Minority Over-sampling Technique
SLR	Systematic Literature Review

Chapter 1

Introduction

1.1 Software Fault Prediction

Software fault prediction (SFP), aids in identification of faulty modules during the development process and helps to quickly and efficiently enhance the final product's software quality. Fault prediction helps to determine whether a software module is defective or not [1], which serves as a shield that might avoid any later unexpected risk and ultimately increases the efficiency and effectiveness of software. This helps to reduce software failures and improve planning, controlling, and execution of software development activities. The development team will have more opportunities due to timely fault prediction to perform tests multiple times on modules or files that have a high probability of errors. The purpose of SFP models is to detect modules within a software system (like files, classes and functions) that are likely to experience faults. Moreover, the key components of software fault prediction models include [2] data acquisition (product and process metrics), data preprocessing (cleaning, feature engineering, scaling), model selection (machine learning and deep learning algorithms), along with performance metrics like accuracy, precision, recall, and F1-score. Software quality declines when there are flaws in it. It can be difficult to guarantee that a system is error-free, and it can take time to find and fix problems. A software fault prediction model forecasts a

module's or software release's fault proneness based on prior software metrics—a measure of the quality of the software. Software metrics are numerical values or specific characteristics derived from predetermined rules that are used to characterize real-world entities [3]. There are two types of software metrics utilized in SFP [4]: (i) Product metrics which includes metrics of size and complexity, such as Halstead, McCabe, and LoC metrics etc and (ii) Process metrics, such as developer metrics, history metrics, and code churn. Process metrics are a relatively new addition to SFP, whereas the first category have been used extensively. Various methodologies are employed in SFP using product metrics and process metrics, with data mining (DM), machine learning (ML) and deep learning (DL) emerging as prominent techniques. While traditional ML algorithms such as Neural Networks, Decision Trees, and Naive Bayes have demonstrated improved prediction accuracy, they often require substantial expert knowledge, human intervention, and extensive training data, posing challenges in dynamic environments [5]. The limitations of conventional methods are addressed by deep learning, a subset of machine learning, presents an exciting potential solution by using multi-layered neural networks to adapt changing trends in data [6], a feature that greatly favors their use for datasets used in software fault prediction.

SFP trains machine learning models using a variety of software metrics as features, which then predict faults. This process is divided based on the models' results. The most common method is binary classification, in which the model returns a label indicating whether a module is faulty (1) or not (0). Faulty modules are then identified for revision and improvement by applying appropriate machine learning classification techniques.

1.2 Software Metrics

Software metrics are numerical measurements that evaluate various aspects of software and the process of developing it. These metrics provide objective information about code complexity, quality, performance, maintainability, and development efficiency. These metrics can be used to train predictive models to identify patterns

and correlations associated with software flaws [4]. This allows for early detection and proactive management of defects, resulting in improved software quality and lower maintenance costs.

Software metrics classify modules as defective or non-defective. Software metrics can be classified according to the software's change history, composition, coding details, structure and developer involvement. Figure 1.1 shows how the metrics have been grouped into different categories, showing how each category is structured and linked to the others.product metrics are widely used in Software Fault Prediction (SFP) because they provide quantitative data about the software's characteristics, which aids in the identification of potential flaws.The product metrics are further categorize to static and dynamic metrics. **Static metrics** [7] are derived from analyzing the software code without execution, examining attributes like lines of code, cyclomatic complexity, coupling, cohesion and number of methods. On the other hand, **dynamic metrics** [7] are obtained by analyzing the software during execution, offering insights into execution time, memory usage, response time, frequency of function calls, and error rates. Furthermore, process/change metrics, whereas less commonly used, have received attention from researchers due to their superior effectiveness in detecting post-release flaws when compared to static code metrics.

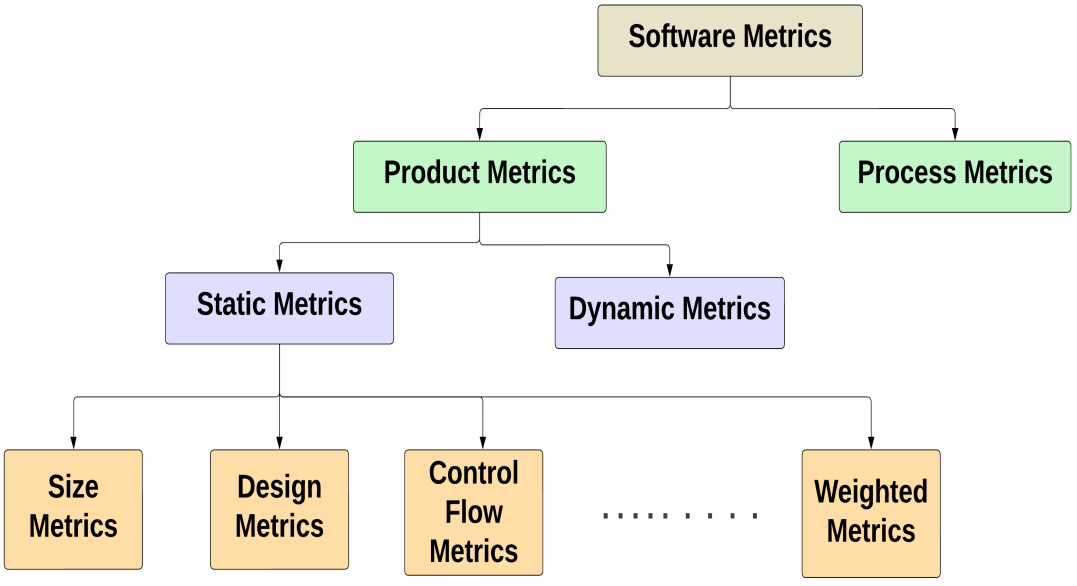

FIGURE 1.1: Metrics Categorization

1.2.1 Product Metrics

Product metrics are used to predict software faults because they provide quantitative data on various software characteristics. These metrics include measurements of code complexity, size, and design quality. By analyzing these metrics, potential flaws can be identified because they highlight areas of the code that are prone to errors or difficult to maintain. For example, high code complexity can indicate a greater likelihood of bugs, whereas frequent code changes may indicate instability. As a result, product metrics are critical for detecting faults early on and improving overall software quality. Primary product metrics, such as McCabe metrics, were developed in the 1970s [8]. Some common product metrics are Chidamber and Kemerer [9], Halstead metrics [10], McCabe metrics [11], object oriented metrics [12] and Component-level metrics [13] CK metrics includes various measures for object oriented, such as number of children (NOC), Lack of Cohesion in Methods (LCOM), coupling Between Object classes (CBO), depth of inheritance tree (DIT), and response for class (RFC) etc. McCabe's suite include Cylomatic complexity, Modular Design Complexity and Essential Complexity. Halstead metrics are based on the idea that the number of operators and operands in a program can indicate its complexity.

1.2.2 Process Metrics

Process metrics are used in software fault prediction because they provide information about the development process and how changes over time affect software quality. The software development process' effectiveness is demonstrated by these metrics. They track issues like the amount of time needed to complete a task, the quantity of defects discovered during testing, and how often the requirements have changed. It is challenging to collect process metrics because they are mainly collected in industrial context and are often unavailable to researchers.

Process metrics are available from a number of sources, including developer's experience [14] and from software's change history [4]. Additionally, how a particular piece of code—or the entire system—was written is determined by the developer's

experience. Common change metrics include number of revisions, lines added or deleted and age of software etc.

1.3 Datasets Used in Software Fault prediction

A dataset is a structured collection of data that is commonly arranged in rows and columns, with each row denoting an instance and each column denoting a feature or variable. Multiple datasets are used in software fault prediction. Datasets can be divided into three categories based on their availability [15]: Public, Partially public and Private datasets. Public datasets are easily available on the internet. They typically provide source code, metric values, and fault data for various modules.Table 1.1 shows publically available datasets.

TABLE 1.1: Public Datasets used in SFP

Datasets	Description
NASA [16]	datasets are widely used in software fault prediction comprised of 14 software projects in the metrics data program (MDP) version and 13 software projects in the PROMISE version.
Jureczko and Madeyski [17]	dataset includes a diverse range of software metrics extracted from a variety of software projects, covering open-source, proprietary, and academic domains.Examples of projects included are Tomcat, Ant, Camel, CKJM, Ivy, JEdit, Log4j, Lucene, POI, Prop, Synapse, Velocity, Xalan, Xerces, among others.
AEEEM [18]	This dataset comprises class-level data from five software systems: Apache Lucene 2.4, Eclipse Equinox Framework 3.4, Eclipse JDT Core 3.4, Eclipse PDE UI 3.4.1, and Mylyn.
ECLIPSE [19]	includes file and package-level data from three versions of Eclipse: 2.0, 2.1, and 3.0. It encompasses product metrics at the file level and 40 metrics at the package level.
SoftLab [20]	dataset comprises five projects known as Autoregressive-1 (AR1), AR2,3,4 and 5. Each project is characterized by its unique set of data and parameters.

In Partially public datasets [21] the project's source code and fault data are generally accessible, but the metrics values must be taken out of the code and compared to the defect data in the repository. Such as open-source projects like Eclipse [19] and Mozilla. Private datasets are less commonly used by researchers compared to other types. These datasets typically lack publicly available data including datasets, source code or fault data. As a result, it is challenging for other researchers to reproduce the study or perform comparative analysis. Many companies have developed fault prediction models using proprietary data, but verifying their accuracy is difficult. [21]

1.4 Methods

1.4.1 Machine Learning Methods

Early detection of faulty modules of software development enables the team to use resources more effectively, ensuring the timely delivery of a high-quality product. Machine learning techniques hold great promise for early defect prediction in the software development life cycle (SDLC) by uncovering hidden patterns in historical data. [22]

Software fault prediction techniques utilizing machine learning techniques, most commonly used ML models include Decision Trees [23], Random Forest, SVM [24], Naive Bayes, neural networks [25], Logistic Regression and KNN. Large labeled datasets are used in these methods to precisely detect and forecast software defects. These techniques have significantly enhanced the prediction and detection of software flaws by identifying complex relationships and patterns within the data, making them essential in real-world applications.

It has been observed that machine learning models are highly effective in classifying software faults. Therefore, it is important to explore their potential by leveraging combined data from both process metrics and product metrics. This approach could enhance the effectiveness and reliability of fault prediction in software development.

Figure 1.2 demonstrates how various machine learning techniques are employed to assess the likelihood of faults in modules or classes. It highlights that among these techniques, DT, SVM, NB, and LR are the most frequently used. The chart provides a visual representation of the prevalence of these methods in the literature, showcasing their relative usage in fault prediction and estimation tasks.

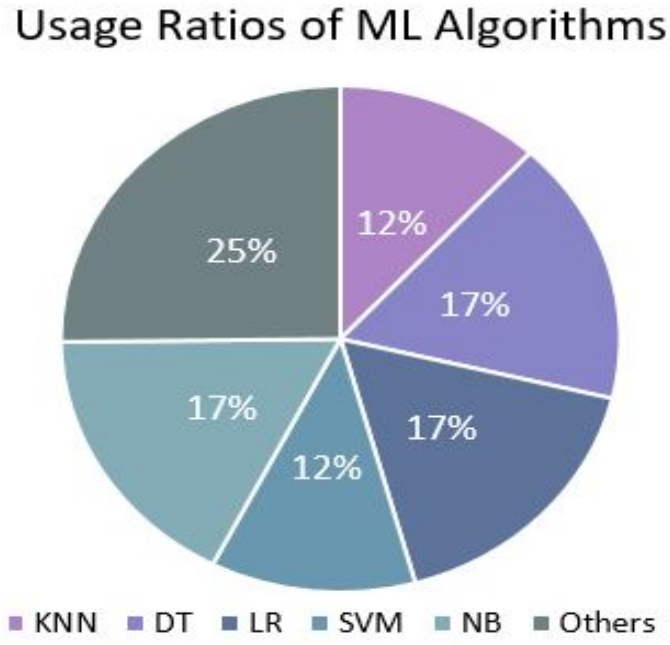

FIGURE 1.2: Commonly used ML Algorithms

1.4.2 Deep Learning Methods

In the domain of predicting software faults, deep learning has proven to be an effective method, showing notable improvements over conventional techniques. These models make use of advanced neural networks to examine and identify trends in large datasets that include different software development process and product metrics. Deep learning algorithms can use this data to identify subtle patterns and dependencies, which are frequently evidence of possible flaws or defects in software systems. Usually, the procedure involves instructing the model on past data so that it can identify patterns related to known defects. After being trained, the model can identify new occurrences or forecast the possibility of errors in software components that are not visible. This predictive ability improves the overall

robustness and reliability of software systems in addition to assisting in the early identification of possible issues throughout the development lifecycle.

Commonly used deep learning models in SFP are CNN [26], LSTM [27] and RNN[28]. Deep learning models have shown remarkable effectiveness in classifying software fault predictions. Therefore, there is a need to explore their capabilities further by leveraging integrated process and product metrics data, and analyzing their respective performances is critical.

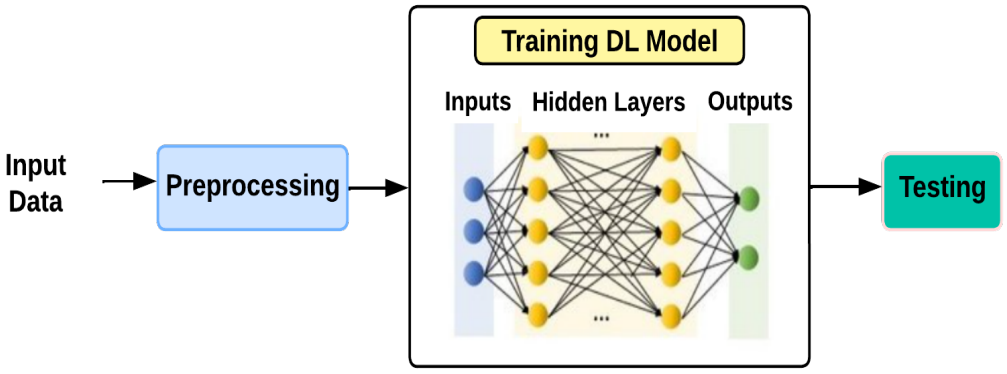

FIGURE 1.3: Basic Structure of DL Models

1.5 Problem Statement

Researchers are currently exploring various metrics to find the most effective combinations for predicting software faults, aiming to improve the accuracy and reliability of these prediction methods. Based on our review of the literature, we analyzed two aspects; that there has been relatively less investigation into the application of process metrics in Software Fault Prediction compared to product metrics and researchers used various machine learning algorithms on diverse datasets, leveraging product and process metrics to accurately predict software fault proneness.

However, deep learning has been used for fault prediction and compared with machine learning, but there has been no investigation into deep learning techniques specifically applied to process metrics in the context of software fault prediction.

1.6 Research Objectives

1. To enhance the quality of software fault prediction process using deep learning algorithms.

2. To evaluate the outcomes of deep learning, it is important to compare them directly with the results of machine learning, using combined metrics (Product + Process).

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

1.7 Research Questions

- **RQ1: Do combined metrics (Product + Process) provide better results than solely using product metrics when employing deep learning algorithms?**

  To provide an answer to this question, we evaluated the product dataset and the combined dataset using deep learning models, and the findings are documented in Chapter 4.

- **RQ2: Does deep learning produce better results on combined metrics compared to machine learning algorithms?**

  To provide an answer to this question, we evaluated the combined metrics (Product + Process) datasets using machine learning and deep learning methods, with detailed results including comparative analyses and performance metrics, are discussed in Chapter 4.

1.8 Research Contribution

1. Initially, Literature review identified that solely relying on traditional machine learning methods may not achieve sufficient accuracy, prompting a shift to deep learning approaches.

2. Then we investigated the metrics used in SFP and found that there has been comparatively less study conducted on process metrics compared to product metrics.

3. Next, in order to conduct experiments to compare performance of models and metrics, we created two datasets: a product dataset and a Product + Process dataset by merging Process and Product metrics.

4. Lastly, our findings, which are presented in Chapter 4, showed that deep learning models outperform machine learning models and that the combined dataset performs better than the product metrics dataset.

1.9 Thesis Organization

In this thesis chapters are structured as follows:

Chapter 1 introduces the Software fault prediction and its essential components like metrics,datasets and approaches.

Chapter 2 conducts an extensive literature review based on two distinct categories; DL based SFP and process metrics used in SFP. In the end,the comperative analysis is performed.

Chapter 3 outlines the methodology, including comparisons of the performance of the proposed models and metrics. We will also provide details of the datasets used for the proposed methodology and the preprocessing steps.

Chapter 4 presents the results of the methodology. It includes a detailed discussion of the results and addresses potential threats to validity.

Chapter 5 captures the conclusions and future directions derived from the study.

Chapter 2

Literature Review

Machine learning techniques have long been used in software fault prediction, traditionally utilizing methods like Logistic regression and Naive Bayes. But deep learning has been used recently, significantly enhancing predictive accuracy through its ability to model complex patterns and relationships in datasets. In this literature review, approaches using deep learning are mentioned.

we reviewed the most relevant researches that were SFP-focused and discussed about the prior state-of-the-art outcomes using machine learning and deep learning. These studies highlight the effectiveness of various techniques and methodologies in achieving these outcomes. Consequently, thoroughly investigating these studies is crucial for a comprehensive understanding of the different aspects of SFP and to ensure that effective strategies are implemented for optimal software development and maintenance.

Machine learning and deep learning are the two main approaches that are highlighted in this literature review. While there are other approaches utilized in software fault prediction as well. Additionally, two categories of metrics are employed in software fault prediction: product metrics and process metrics.
To highlight the key research areas of our study, we have analyzed deep learning in SFP and the process metrics used in SFP. Figure 2.1 illustrates the categorization of these research areas according to the literature.

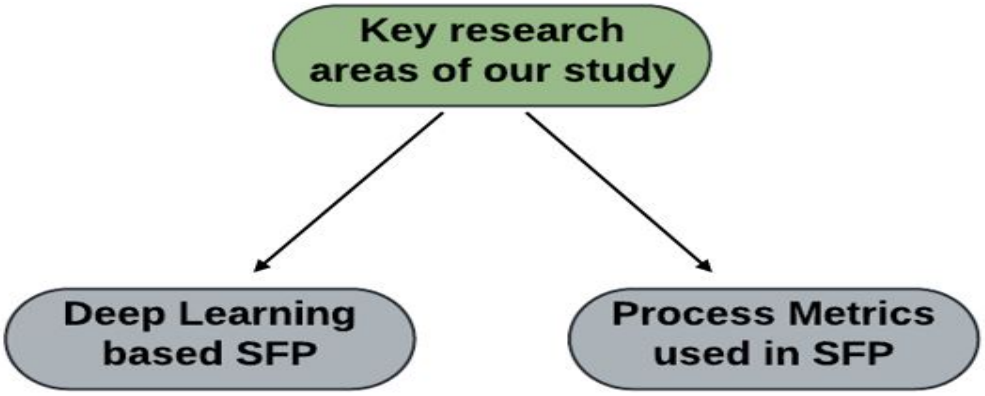

FIGURE 2.1: Categorization of Our Research

2.1 Existing Literature Surveys on SFP

The literature based on Surveys and SLR of our research study is given below:

Catal and Diri [29] conducts a comprehensive review of journal articles and conference papers on SFP to assess advancements and direct future research. It classifies studies by metrics, methods, and datasets, focusing on those before and after the 2005 establishment of the PROMISE repository, which provides public datasets. The review follows a systematic approach inspired by previous reviews and addresses eight research questions. It includes papers based on relevance to fault prediction, without regard to publication year or methods. Key findings include challenges with proprietary datasets, the predominance of method-level metrics, and a shift from statistical methods to machine learning techniques post-2005. The review highlights the need for more research on class-level metrics, public datasets, and repeatable models to advance the field.

Catal [30] provides an extensive overview of the literature on SFP in software engineering, where fault data is recorded from error reports during system testing, and software metrics are used as independent variables. It emphasizes the need for version control and change management systems for effective fault data collection and analysis. Historical context is provided, noting that fault prediction models, developed since the 1990s, can outperform traditional reviews, with

a detection probability of 71% versus 60%. The thesis will review both machine learning and statistical approaches, covering studies from 2000-2003 that explore correlations between static metrics and fault-proneness, 2003-2005 that examine module complexity and machine learning techniques, 2005-2007 focusing on algorithm performance and the importance of class-level metrics, and 2007-2009 highlighting advancements and challenges with limited fault data. Practical issues include clustering fault-prone modules and the limitations of current methods in sparse fault data contexts. The conclusion calls for further research into additional prediction areas such as software reusability and security.

Hall et al. [31] underscores the importance of fault prediction modeling in software engineering for targeting fault-prone code and improving quality. It highlights the need for a comprehensive review of diverse fault prediction models and distinguishes this systematic literature review (SLR) by its rigorous methodology, which includes examining 36 relevant studies in depth. Studies were selected based on specific inclusion criteria and evaluated using standardized assessment criteria, with data extracted on performance metrics like precision, recall, and f-measure. Findings reveal gaps in contextual information and variable model performance, with Naive Bayes and Logistic Regression generally performing well, while SVMs showed inconsistent results. The review emphasizes the need for more rigorous methods and detailed reporting to enhance model reliability and applicability. The appendix provides methodologies for calculating performance metrics.

Radjenovi et al. [32] examined the significance of software metrics in fault prediction, noting that the choice of metrics influences classification accuracy more than modeling techniques. I identified key metrics suites, such as MOOD, QMOOD, and CK, through a systematic review of 106 studies. This review focused on the effectiveness of these metrics rather than modeling techniques, highlighting gaps in study quality and contextual reporting. I used a rigorous search strategy and inclusion criteria to assess metrics, data set availability, and their application across different software development phases. The findings emphasize the importance of clear metrics definitions and data set accessibility, and the conclusion stresses the review's value for future research in diverse contexts.

Malhotra [33] underscores the importance of early fault detection in software development for timely corrections and enhanced quality. It explores how software metrics and fault data can be used to model fault-prone modules, optimizing testing resources. The paper focuses on soft computing and machine learning (ML) techniques in SFP, noting the lack of systematic reviews on these methods. A comprehensive review of studies from 1991 to 2013 identified seven ML techniques, comparing their performance with statistical methods. Results show ML techniques, especially Random Forest, perform well with AUC values of 0.7 to 0.83 and accuracies of 75% to 85%. Limitations include potential biases and study variability. The conclusion highlights the need for more comparisons between ML and traditional methods and recommends further exploration of ML techniques to improve software quality.

Rathore and Kumar [34] addresses the limitations of current software fault prediction techniques, which achieve 70% to 85% accuracy but often suffer from high misclassification rates and biased learning due to uneven fault distribution. It highlights the role of publicly available datasets and reviews past studies, emphasizing the need for improved performance evaluation and context consideration. The paper notes that object-oriented metrics generally perform better than traditional ones and calls for a taxonomy of fault prediction components. It also discusses the importance of accurate fault data, the impact of software metrics and contextual variables, and challenges in predicting fault severity and density. The conclusion advocates for detailed methodologies, new metrics, and adaptation to agile development environments.

Caulo and Scanniello [35] presents a comprehensive taxonomy of 512 software metrics used in Software Fault Prediction (SFP) to standardize terminology and improve communication among researchers and practitioners. By extending two previous Systematic Literature Reviews and including studies from 2012 to 2017, the authors classify these metrics into process and product categories, each with various subcategories. Their findings highlight that product metrics are more frequently used than process metrics, with many metrics having a short lifespan of around 3.1 years. The taxonomy aims to aid in selecting appropriate metrics for

developing accurate SFP models and is designed to be continually updated through community contributions on GitHub. This work underscores the importance of standardization in SFP research and the need for ongoing updates to maintain its relevance.

Pandey et al. [36] conducts a SLR to evaluate the efficacy of various ML techniques in SFP from 1990 to June 2019. They analyzed 154 studies, assessing their relevance based on predefined research queries and quality criteria. The study highlights the significance of ML models in enhancing SFP by comparing them against traditional statistical methods. Key findings indicate that ML models generally outperform statistical models, with specific ML techniques such as RF and SVM showing high performance across diverse datasets. The paper also identifies common challenges in SFP, including class imbalance and overfitting, and underscores the importance of proper dataset selection and hyperparameter tuning. The review provides guidelines for software practitioners to select optimal ML techniques for SFP and suggests directions for future research to address existing challenges and improve predictive modeling in software engineering.

Khan et al. [37] examines Artificial Neural Networks (ANNs) for SDP from 2015 to 2018. The study reviews 8 key papers, highlighting ANNs' effectiveness in improving software quality and reducing costs by predicting defects early. It emphasizes the advantage of hybrid approaches, combining ANNs with techniques like feature selection and preprocessing to enhance prediction accuracy. The authors conclude that ANNs, especially when integrated with other methods, show great promise for SDP and recommend exploring additional machine learning techniques for future research.

Pandey and Kumar [38] examines recent advancements in software fault prediction (SFP) techniques that address the issue of imbalanced datasets. Imbalanced data, where one class significantly outnumbers another, poses a challenge for predictive models, leading to biased and inaccurate results. The authors review various ML and DL approaches that have been developed to mitigate this issue, such as Synthetic Minority Over-sampling Technique (SMOTE) and other data sampling methods. They highlight that the most popular method for resolving problems

with data quality is SMOTE. The paper categorizes and compares different SFP algorithms, focusing on their performance metrics like Acc, precision, and recall. It concludes that balancing the dataset is crucial for accurate fault prediction and gives recommendations for deciding which methods, depending on various datasets and algorithms, are the best.

Son et al. [39] systematically maps software defect prediction models, reviewing 98 studies from 1995 to 2018. It addresses key areas including data preprocessing, model building techniques, performance evaluation, and the detection of security vulnerabilities. Findings show that while many studies focus on dataset size and machine learning methods, few address multicollinearity analysis, feature selection, cross-project predictions, and security-related defects. The authors recommend thorough data preprocessing, using industrial datasets, comparing various techniques, and focusing on security-related defect prediction for future research.

Abaei and Selamat [40] reviews ML techniques for software fault prediction, using NASA datasets. It evaluates methods like decision trees, random forest, neural networks, and artificial immune systems (AIS). Results show that random forest is best for large datasets, and Naïve Bayes is effective for small ones. Immunos99 is the top AIS classifier with feature selection. Performance is measured using AUC, probability of detection, and false alarm rates. The study suggests that while feature selection reduces execution time, the type of algorithm plays a more crucial role in prediction performance, suggesting a focus on algorithm improvement for better fault prediction.

Aziz et al. [41] examines the impact of the inheritance component on SFP, investigating different methods and approaches for software fault prediction. It examines the effects of applying inheritance metrics to SFP machine learning models, evaluating their performance and efficacy on various datasets. The study emphasizes that although inheritance metrics can enhance prediction models, their impact varies. It discusses the significance of feature selection, the function of various classifiers, and the need to combine different metrics in order to improve prediction accuracy. The results indicate that improved fault prediction models may result from combining inheritance features with other software metrics.

2.2 Deep Learning Based Fault Prediction

Recent improvements in computing power and memory capacity in contemporary computer architectures have led to a notable evolution of conventional machine learning methods, which is referred to as deep learning (DL). The idea of "deep learning" is presented by Hinton et al. in 2006 [42]. Researchers at academic institutions and industry practitioners alike are very interested in deep learning (DL) methods due to their success in image recognition and data mining. Consequently, they are currently investigating and utilizing deep learning (DL) algorithms for a variety of software engineering (SE) tasks, such as software implementation,requirements analysis, software testing and debugging, software design & modeling, and maintenance.

Deep learning enables computational models with several processing layers to understand data at various levels of abstraction. By leveraging the backpropagation algorithm, DL uncovers intricate patterns within large datasets. This algorithm guides the machine in adjusting its internal parameters, refining the representation in each layer based on the previous layer's output. DL is a subset of ML that employs supervised and/or unsupervised strategies. It is a collection of methods for learning from multiple layers in neural networks.

Additionally, deep learning can handle massive volumes of data, offers multiple techniques that enable the use of unlabeled data to discover essential patterns, and allows representations acquired by deep neural networks to be shared across various tasks.

Below is a list of the deep learning-based literature that supports our research study:

Batool et al. [43] conducted a comprehensive study on deep learning techniques, RBFN, BILSTM, and LSTM algorithms, utilizing two datasets that underwent normalization and label encoding during preprocessing. They performed extensive experiments, including hyperparameter tuning, which significantly impacted the algorithms' performance. By employing a combined open source dataset from the

various sources, they ensured diverse and comprehensive training and evaluation. Their results showed that LSTM and BILSTM achieved superior accuracies of 93.53% and 93.75%, respectively, highlighting their effectiveness. Despite this, RBFN outperformed the other two algorithms in computational efficiency, demonstrating significantly faster processing speeds, making it ideal for scenarios where speed is crucial. This study underscores the importance of hyperparameter tuning and provides insights into the trade-offs between accuracy and computational efficiency in deep learning models.

Nevendra and Singh [44] proposed an Enhanced Convolutional Neural Network (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 demonstrated that the Enhanced CNN outperformed Li's model, achieving an average accuracy of 0.775. 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. These findings highlight the importance of both data preprocessing and model design in improving the accuracy of defect prediction models.

Pandey et al. [45] presented two deep learning models, Squeeze Net and Bottleneck, were evaluated using seven datasets from the NASA repository. The research focused on addressing class imbalance and overfitting issues commonly encountered in predictive modeling. To mitigate these issues, the Synthetic Minority Over-sampling Technique (SMOTE) was employed in conjunction with the deep learning models. The results demonstrated that both Squeeze Net and Bottleneck models achieved significant improvements in performance metrics. Specifically, Squeeze Net achieved an F-measure of 0.93 ± 0.014, while the Bottleneck model reached an F-measure of 0.90 ± 0.013. These results indicate that the integration of SMOTE with these deep learning models can effectively improve the accuracy and precision of SFP, outperforming traditional methods.

Nehéz and Khleel [46] presented software defect prediction (SDP) model utilizes high-performance deep learning algorithms, specifically CNN and BI-LSTM, applied to a public benchmark dataset GHPR 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. Results indicated that the 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. Furthermore, the study compares these models with existing approaches like DNN, LSTM, and Random Forest.

Tadapaneni et al. [47] introduced the Naïve Bayes machine learning algorithm along with advanced DL approaches, including Deep Neural Networks (DNN) and LSTM, for predicting software defects. The proposed DNN model featured a softmax output layer and two hidden layers, each comprising fully connected LSTM layers with a dropout rate of 0.1% to enhance model robustness. Utilizing the PROMISE dataset, the researchers conducted a comparative analysis among the Naïve Bayes, LSTM, and DNN models. The results demonstrated that the DNN achieved the highest accuracy at 80.89%, significantly outperforming both Naïve Bayes and LSTM models. This highlighted the DNN's superior effectiveness for binary classification tasks in software defect prediction, emphasizing the importance of leveraging advanced deep learning techniques and optimal network configurations to enhance predictive performance.

Ho et al. [48] proposed model, KPCA-ImbDNN, leverages Kernel Principal Component Analysis (KPCA) to capture crucial insights from feature design and introduces a deep neural network (DNN) to explore the complex relationships among these features. To address class imbalance, the model incorporates a weighted loss function and a bootstrapping method. The KPCA employs the Gaussian Radial Basis Function kernel, a variation of PCA suited for nonlinear data. Experiments were conducted using datasets from the NASA, ECLIPSE, and PROMISE open-source repositories. Comparative analysis against SVM, RF, and Kernel Principal Component Weighted Ensemble (KPWE) demonstrated that KPCA-ImbDNN outperformed these existing approaches.

Sekaran and Annabel [49] proposed Enhanced Convolutional Neural Network (ECNN) model utilizes the Combined Defect Analysis (CDA) dataset and the PROMISE

Source Code (SPSC) dataset for defect prediction. The initial step in this framework involves collecting source files from repositories and converting them into abstract syntax trees (ASTs). These ASTs are then mapped to integer input vectors for the CNN model, which are subsequently transformed into string token vectors. To address class imbalance, an oversampling technique is employed. The ECNN model was evaluated using both datasets, demonstrating superior performance with an F-measure of 0.660 on the SPSC dataset and an F-measure of 0.763 on the CDA dataset.

The following Table 2.1 provides a detailed overview of the DL studies discussed in this chapter. The table summarizes various techniques, including CNN, LSTMs, Bidirectional LSTMs (BILSTMs), and Radial Basis Function Networks (RBFNs), among others. These techniques have been applied to different datasets, and the results of these applications are presented in the table below.

TABLE 2.1: Studies in which Deep Learning has been used for Fault Prediction

Ref	**Year**	**Approach**	**Dataset**	**Best Results**
[44]	2021	Enhanced Convolution Neural Networks(CNNs)	CM1 KC1 KC2 PC1	**Accuracy** PC1 0.93
[46]	2022	CNN BI-LSTM	GHPR Dataset	**Accuracy** CNN: 0.80 BI-LSTM: 0.80
[47]	2022	LSTM DNN	PROMISE	**Accuracy** DNN 80.89
[48]	2022	KernalPCA-ImbDNN	PROMISE NASA ECLIPSE	**AUC** PROMISE 0.78

TABLE 2.1: (Continued from Previous Page)

Ref	Year	Approach	Dataset	Best Results
[43]	2023	LSTM BILSTM RBFN	PROMISE GHPR	**Accuracy** BILSTM 0.93
[49]	2023	Enhanced CNN	PROMISE NASA	**F-measure** E-CNN 0.83
[45]	2023	Squeeze Net Bottle Neck	CM1 KC1,KC2 MC1 JM1,MW1 PC4	**F-measure** Squeeze Net: 0.93 ± 0.014

2.3 Metrics Used in Software Fault Prediction

Typically, the following metrics are employed as features in software fault prediction:

1. Product metrics
2. Process metrics

Product metrics assess the current state of a software product, identify potential issues, and track risks, with examples including lines of code, number of methods and cyclomatic complexity. In contrast, process metrics aim to enhance the long-term workflow of a team or organization, such as measuring the time required for software development tasks, number of defects found during testing. Table 2.2 shows commonly used product and process metrics in SFP literature. Chapter 3 provides the detailed discussion of product and process metrics.

Product Metrics	**Process Metrics**
Halstead Complexity Measures	Number of Commits
Lines of Code (LOC)	Number of Changes
Lack of Cohesion in Methods (LCOM)	Number of Bug Fixes
Number of Methods/Functions	Time to Fix Bugs
Cyclomatic Complexity	Code Review Effort
Coupling between Objects (CBO)	Developer Experience
Depth of Inheritance Tree (DIT)	Development Lead Time
Number of Attributes/Variables	Code Review Comments
Class Size	Number of Fixes
Code Coverage	Number of Authors

TABLE 2.2: Product and Process Metrics

2.4 SFP Using Process Metrics

The DL models for SFP have primarily relied on product metrics like code complexity and size, often neglecting process metrics. This section explores the integration of process metrics, which are crucial as they offer insights into the software development lifecycle and track code modifications that can lead to defects. Process metrics include data on the frequency and nature of code changes, developer activity, code churn, and historical defect data.

Recent research highlights their importance, as they provide a dynamic and comprehensive view of the software's evolution and associated risks. By incorporating these metrics, models can better account for the temporal and contextual aspects of code changes, improving the accuracy and robustness of fault prediction. This shift towards including process metrics represents a significant advancement, offering a more holistic approach to identifying potential defects early in the development process.

Choudhary et al. [50] examines the combination of process metrics with product metrics to enhance the effectiveness of fault prediction models. Various versions of Eclipse projects are utilized in the experimental investigations, with process metrics extracted from GIT repositories. Additional process metrics are identified and collected from the ECLIPSE repository to provide a more comprehensive dataset. Machine learning algorithms are then applied to this combined dataset of process and product metrics to build and evaluate SFP models. By integrating these metrics, the study aims to capture a broader range of factors influencing software faults, thereby improving the accuracy and reliability of the prediction models.

Sikic et al. [51] proposed new metrics known as aggregated change metrics (ACM) to enhance software defect prediction. ACMs are derived by consolidating data from all software changes between two versions, taking into account the chronological sequence of these changes. Tests conducted on open-source Java projects show that adding aggregated change metrics improves the performance and stability of widely used models for classification. Additionally, the proposed metrics are applicable for detecting defect-prone software modules across various programming languages. Statistical tests are utilized to demonstrate the significant improvement in results when employing the proposed metrics over process metrics.

Alshehri et al. [52] undertakes a comparison between different machine learning algorithms for predicting software defect proneness by leveraging process metrics, product metrics, and a hybrid of both. Naive Bayes, J48 and Logistic regression, are applied across three releases of Eclipse (2.0, 2.1, and 3.0) to assess their efficacy. Findings reveal that employing a smaller set of metrics for SFP yields a bit improved accuracy and reduced false positive rates compared to utilizing all process metrics.

However, the comprehensive utilization of all process metrics surpasses the reduced number in terms of recall and G score. Moreover, from their study it is concluded that integrating process metrics and product metrics generally yields superior performance compared to relying solely on process metrics or product metrics. Compared to Naive Bayes and logistic regression, J48 outperforms.

Yu et al. [53] proposed two novel change metrics, which depends on historical package defect rates and class change degree. Nine open-source projects are tested on thirty-three versions,the efficacy of these proposed process metrics was compared with process metrics and other product metrics. This study revealed that integrating these PPMs notably enhanced defect prediction performance, particularly when defect rates remained stable between adjacent versions. When compared to traditional software defect prediction methods, the proposed change metrics exhibited superior performance in evolution-based defect prediction.

Jiang et al. [54] conducts a comprehensive empirical investigation into process metrics crucial for detecting changes in defect states within evolving projects. 5 change metrics were gathered from 37 different versions across twelve software projects, and their correlation values and classification performance were analyzed.

The findings highlight the significant role of the Number of Distinct Committers in defect state changes, particularly in defect elimination. Additionally, the Number of Revisions emerges as the second most influential metric, while the Degree of Code Modification is identified as the least influential. Moreover, the Average Number of Modified Lines (ANML) is deemed more effective than the Number of Modified Lines(NML). The paper also offers insights into software development and SDP strategies based on the experimental findings.

Lomio et al. [55] examines the relationship between fault-proneness and code quality, focusing on product metrics and code smells. It aims to enhance fault-inducing commit prediction by utilizing variables such as SonarQube rules, software code, change metrics, and a range of ML and DL techniques. An empirical investigation was conducted on 29 Java projects, encompassing 58,125 commits and identifying 33,865 faults.

The analysis revealed a set of features that achieved highly accurate fault prediction, with an Area Under the Curve (AUC) exceeding 95%. Additionally, deep learning models demonstrated higher accuracy compared to traditional machine learning models, highlighting their superior capability in handling complex patterns and relationships within the data. This study underscores the importance

of combining diverse metrics and advanced modeling techniques to improve the predictive performance of fault-inducing commits.

Rahman and Devanbu [56] evaluates the significance and validity of product and process metrics across 85 releases of 12 major open-source projects in the creation of prediction models. It finds that process metrics are more effective than product metrics for prediction, with product metrics exhibiting high stasis and limited usefulness.

The study employs Spearman correlation to measure metric stasis between releases and uses AUC, AUCEC at 10 and 20, and F-Measure for model comparison. The results indicate that process metrics outperform product metrics in building prediction models. The paper seeks to understand why process metrics are more effective, noting that product metrics are less stable, portable, and tend to stagnate in prediction models by focusing on less defect-dense files.(investigates the applicability and effectiveness of product and process metrics in developing prediction models for 85 releases of 12 large open-source projects. It discovers that process metrics outperform product metrics for prediction, while product metrics exhibit high stasis and limited usefulness.

In this paper they uses Spearman correlation to measure metric stasis between releases and AUC, AUCEC at 10 and 20, and F-Measure to compare the models. The findings show that change (process) metrics outperform product metrics when developing prediction models. The paper attempts to understand why process metrics are more effective, pointing out that product metrics are less stable, portable, and tend to stagnate in prediction models when focusing on less defect-dense files.

Enhancing fault prediction with deep learning uses neural networks to analyze historical data, improving accuracy and preventing software errors. The following Table 2.3 offers a comprehensive overview of studies utilizing process metrics, as discussed in this chapter. It highlights various machine learning models, such as KNN, NB, LR and MLP, along with other models and process metrics used in these studies, providing insights into their applications and effectiveness.

TABLE 2.3: Studies in which Process Metrics have been used for Fault Prediction

Ref	Year	Technique	Models	Dataset	Process-Metrics used	Best Results
[50]	2018	Change metrics with new ones.	J48 KNN RF	Eclipse	LOC-WORKED-ON, COMMITS, LOC-ADDED, LOC-DELETED, CODECHURN, SINGLE-COMMITS	**Precision** **RF** 0.77
[52]	2018	Code-metrics, Change-metrics, Code+Change	LR NB J48	Eclipse	LOC-Added LOC-Deleted Refactoring Bug fixes Authors Code churn	**Accuracy** **J48** 80.9
[53]	2020	PCDC PCCM	NB KNN LR	PROMISE	Package-Defect Change Degree Package-Change Count Metric	**AUC** **KNN** 0.90
[54]	2020	Empirical analysis of process metrics	NB KNN LR MLP SVM	PROMISE	NR NDC NML DCM=(NML/LOC), ANML=(NML/NR)	**F1** **MLP** 0.77

TABLE 2.3: (Continued from Previous Page)

Ref	Year	Technique	Models	Dataset	Process-Metrics used	Best Results
[51]	2021	aggregated change metrics.	RF SVMC DTC MLPC	PROMISE	Lines Added Lines Deleted No. of Developers No. of Commits Developer's-experience.	**F1 DTC** 0.55±0.03
[55]	2022	SonarQube and SZZ algorithm	XG-Boost RF Gradient-Boost	Technical Debt dataset	No of classes Physical lines Effective lines No of Java classes Java interface No of package statements No of functions No of comments Density of comment lines	**AUC XG-Boost** 81.73

2.5 Research Gap

Deep learning models excel in adapting to changing data trends, making them ideal for software fault prediction. Unlike traditional machine learning models that need manual feature engineering and frequent updates, deep learning models automatically learn features and adjust to new data patterns. This capability is crucial for software fault prediction, where faults and their patterns can quickly

evolve due to software updates, changing user behaviors, and evolving codebases. By continuously learning from the latest data, deep learning models maintain high prediction accuracy and provide timely insights, ensuring potential faults are quickly identified and addressed.

From the literature [43–49, 57], the results of deep learning-based fault prediction with product metrics are superior than traditional machine learning methods. According to these studies, deep learning consistently achieves promising results in terms of accuracy and effectiveness in predicting faults. This is mostly due to deep learning's ability to automatically learn intricate representations and patterns from large datasets, which enhances its predictive capabilities compared to traditional machine learning techniques.

Software metrics refer to quantitative measures used to assess various aspects of software development and maintenance. From the literature [50–56], it has been observed that both product and process metrics are employed solely with traditional machine learning algorithms, and integrating process metrics with product metrics using conventional machine learning leads to improved results.

Therefore, it is observed that deep learning approach has not been used with process metrics. From literature it has been found that machine learning models like KNN, NB and LR have performed well with process metrics as discussed earlier, also deep learning model like CNN, LSTM and BILSTM have performed well with product metrics. Moreover, we have observed that results from deep learning are better than those from conventional machine learning using product metrics, and incorporating process metrics improves results. This suggests a potential area that has not been explored yet—whether using deep learning with combined metrics(Product + process) leads to better results.

Chapter 3

Methodology

ML models have traditionally been employed to assess metrics for SFP; however, recent advancements have seen the adoption of deep learning models, which offer improved accuracy. The proposed methodology aims to significantly enhance software fault prediction by leveraging process metrics. In this chapter,we present a detailed overview of the proposed approach, detailing its methodology and how it contributes to more accurate fault prediction.

3.1 Explanation of Methodology

To investigate the role of change metrics in deep learning models for predicting software faults and the importance of process metrics in predicting the number of faults that integrates both metrics into a comprehensive predictive framework.

Our approach involves these steps as depicted in Figure 3.1:

1. For our experiments, the deep learning models CNN, LSTM, and BILSTM are used. We used publically available dataset that is downloaded from public repository. These models were chosen because they have been used in the literature. Moreover, the reliability for our study is demonstrated by their consistent performance in previous studies.

2. After selecting the deep learning models to be implemented, first step is to acquire a dataset comprising product metrics and evaluate the outcomes by applying and analyzing results from a deep learning model.

3. Subsequently, we obtained merged dataset that includes both product and process metrics. Using this dataset, we employed a deep learning model to measure and assess the accuracy of the predictions.

4. Two types of comparisons have been performed:

   (a) Comparison of product metrics only and combined metrics using deep learning models.

   (b) Comparing the performance of ML and DL models using combined metrics.

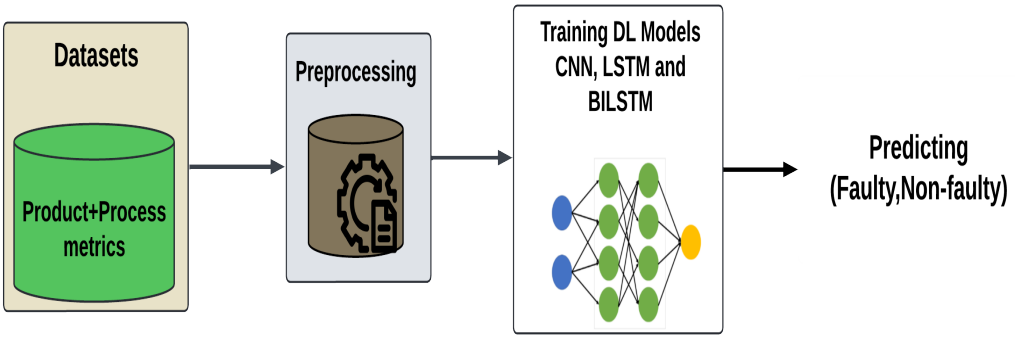

FIGURE 3.1: Proposed Methodology Steps

3.2 Datasets

In our research, we utilized five datasets with fault/bug counts as the target class. Table 3.1 lists the publicly available datasets from open-source Eclipse projects. The datasets are named Eclipse, Equinox, Lucene, Mylyn, and Pde. Each dataset contains both process and code metrics, with fifteen and seventeen metrics respectively. The data acquisition section contains detailed information about these metrics. In these datasets, there are between 324 and 1,862 instances, as shown in table below. The outcomes were then evaluated through the application of

machine and deep learning models like CNN, LSTM and BILSTM to assess its predictive accuracy across these diverse datasets.

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

TABLE 3.1: Description of Datasets

3.2.1 Data Acquisition

In this section, we described how we obtained and prepared the data for our study. We used publicly available datasets from open-source Eclipse projects, which contain a wide range of metrics and fault counts [58]. The subsections that follow describe the steps involved in preprocessing the data to ensure its quality and relevance to our research objectives. The dataset includes different projects, namely Eclipse, Equinox, Lucene, Mylyn, and PDE, each containing various files in csv format such as defect logs, code metrics, and change history, all of which were crucial for our analysis. The details are mentioned below:

3.2.1.1 Target File

The target file of this dataset contains 11 columns; some of them are displayed in figures 3.2, 3.3 and 3.4. This file have some extra columns like class name, unnamed

columns and Irrelevant columns like nonTrivialBugs, majorBugs, criticalBugs and highpriorityBugs.

	classname	numberOfBugsFoundUntil:	numberOfNonTrivialBugsFou
0	org::eclipse::jdt::internal::core::search::ind...	3	2
1	org::eclipse::jdt::internal::compiler::codegen...	0	0
2	org::eclipse::jdt::internal::compiler::ast::AS...	55	48
3	org::eclipse::jdt::internal::compiler::lookup:...	3	3
4	org::eclipse::jdt::internal::eval::CodeSnippet...	15	13
...	...	...	...

FIGURE 3.2: Target File Columns

numberOfMajorBugsFoundUntil:	numberOfCriticalBugsFoundUntil:	numberOfHighPriorityBugsFoundUntil:
0	0	0
0	0	0
6	4	2
0	0	0
1	1	0
...	...	...

FIGURE 3.3: Target File Columns

bugs	nonTrivialBugs	majorBugs	criticalBugs	highPriorityBugs	Unnamed: 11
0	0	0	0	0	NaN
0	0	0	0	0	NaN
1	0	0	0	0	NaN
0	0	0	0	0	NaN
0	0	0	0	0	NaN
...	...	...	...	...	...

FIGURE 3.4: Target File Columns

3.2.1.2 Process Metrics File

The process metrics file of this dataset contains 20 columns; some are displayed in figures 3.5 and 3.6. This file also have some extra columns like class name, unnamed columns, etc.

Process metrics are called change metrics used in software fault prediction that focus on how the software is developed and maintained rather than its inherent characteristics. These metrics include factors such as the number of times a piece of code has been revised, the number of developers involved in its modification, the duration since it was last changed, and the frequency of bug fixes applied to it.

Understanding process metrics helps assess the stability and reliability of software. In this study, we utilized different process metrics in our experiments. The following is a list of all seventeen metrics, with their names and brief descriptions:

1. **NumberOfVersionsUntil:** The number of software versions until a specific point in time.

2. **NumberOfFixesUntil:** The total number of bug fixes applied until a specific point in time.

3. **NumberOfRefactoringsUntil:** The number of code refactoring actions performed until a specific point in time. This metric helps assess the frequency of code improvements and restructuring, indicating efforts to enhance code quality and maintainability over time.

4. **NumberOfAuthorsUntil:** The total number of authors who have contributed to the code until a specific point in time. This metric helps identify the level of collaboration and team involvement in the project. A higher number of contributors may indicate a more distributed development effort or a larger team working on the codebase.

5. **LinesAddedUntil:** The total number of lines of code added until a specific point in time.

6. **MaxLinesAddedUntil:** The maximum number of lines of code added in a single change until a specific point in time.

7. **AvgLinesAddedUntil:** The average number of lines of code added per change until a specific point in time.

8. **LinesRemovedUntil:** The total number of lines of code removed until a specific point in time, provides insight into the ongoing maintenance efforts, highlighting areas where obsolete or inefficient code has been cleaned up to improve performance or readability.

9. **MaxLinesRemovedUntil:** The maximum number of lines of code removed in a single change until a specific point in time.

10. **AvgLinesRemovedUntil:** The average number of lines of code removed per change until a specific point in time.

11. **CodeChurnUntil:** Represents the cumulative changes in the codebase over time, providing insights into development activity. It helps track the evolution of the code by measuring both additions and deletions leading up to a given point.

12. **MaxCodeChurnUntil:** The maximum code churn (total sum of lines added and lines removed) in a single change until a specific point in time.

13. **AvgCodeChurnUntil:** The average code churn per change until a specific point in time.

14. **AgeWithRespectTo:** The age of the code with respect to a specific point in time, to determine how long the code has been in active use or development, which can significantly impact its reliability and future maintenance needs.

15. **WeightedAgeWithRespectTo:** The age of the code, weighted by its significance or frequency of changes, with respect to a specific point in time, helps in identifying parts of the system that are more prone to defects. This metric is particularly useful for assessing the stability and maintainability of code over its lifecycle, offering insights into areas that may require more attention during maintenance phases.

	classname	numberOfVersionsUntil:	numberOfFixesUntil:
0	org::eclipse::jdt::internal::core::search::ind...	65	4
1	org::eclipse::jdt::internal::compiler::codegen...	2	0
2	org::eclipse::jdt::internal::compiler::ast::AS...	120	10
3	org::eclipse::jdt::internal::compiler::lookup:...	28	4
4	org::eclipse::jdt::internal::eval::CodeSnippet...	93	17
...	...	...	...

FIGURE 3.5: Process Metrics File Columns

numberOfRefactoringsUntil:	numberOfAuthorsUntil:	linesAddedUntil:	maxLinesAddedUntil:
0	8	608.0	158.0
0	2	10.0	10.0
0	12	1361.0	99.0
0	5	138.0	39.0
0	8	1870.0	684.0
...	...	...	...

FIGURE 3.6: Process Metrics File Columns

3.2.1.3 Product Metrics File

The product metrics file of this dataset contains 20 columns; some of them are displayed in figures 3.7 and 3.8. This file also have some extra columns like class name etc. and unnamed columns.

As mentioned earlier, product metrics, also known as code metrics, are derived from the software's coding structure. These metrics offer valuable insights into the code's properties, size, complexity, and overall structure. These metrics include the number of lines of code, the complexity of the code, how different parts

of the software are interconnected (coupling), how well individual parts of the software work together (cohesion), and other similar factors.Analyzing these attributes allows us to gain a better understanding of the software's quality and identify potential areas for improvement. Seventeen different product metrics were incorporated into our experiments in this research. The following is a list of all seventeen metrics, with their names and brief descriptions:

1. **CBO(Coupling between objects):** Measures the number of classes to which a particular class is coupled. Higher coupling indicates greater dependency, which can affect maintainability and flexibility of the code.

2. **DIT(Depth Of Inheritance Tree):** Indicates the depth of a class in the inheritance hierarchy. A deeper tree often suggests increased complexity and potential for more specialized behavior.

3. **Fan In:** Counts the number of other classes that reference a given class, reflecting its level of reuse and dependency within the system.

4. **Fan Out:** Counts the number of other classes directly referenced by a given class.

5. **LCOM (Lack of Cohesion Methods):** Measuring the quantity of unconnected method pairs in a class. A higher LCOM value suggests lower cohesion, which may result in a more fragmented class structure, making the code harder to understand and maintain.

6. **NOC (Number of Children):** Represents the number of immediate subclasses inheriting from a class.

7. **NumberOfAttributes:** Indicates the total number of attributes in a class, reflecting the complexity and dimensionality of the data associated with that class.

8. **NumberOfAttributesInherited:** Counts the number of attributes inherited from parent classes, highlighting the extent of reuse and inheritance within the class hierarchy.

9. **NumberOfLinesOfCode:** Measures the total number of lines of code in a class

10. **NumberOfMethods:** Determines the total number of methods in a class. This metric provides insight into the class's complexity and functionality, reflecting how many distinct actions or behaviors the class can perform and manage.

11. **NumberOfMethodsInherited:** Counts the number of methods inherited from the parent classes. This metric can help assess the reusability of code and the potential impact on the subclass's behavior.

12. **NumberOfPrivateAttributes:** Determines the number of private attributes in a class.

13. **NumberOfPrivateMethods:** Counts the number of private methods in a class, which can offer insights into the level of encapsulation and modularity within the code. A higher number of private methods may indicate more internal complexity, but also better adherence to object-oriented design principles by limiting external access to the class's internal functionality and methods of class.

14. **NumberOfPublicAttributes:** Determines the number of public attributes in a class, which can reflect the class's exposure to external manipulation. A higher count may suggest weaker encapsulation, potentially increasing the risk of unintended interactions or making the class attributes more difficult to maintain.

15. **NumberOfPublicMethods:** Counts the number of public methods in a class.

16. **RFC (Response for Class):** The number of methods that can be called in response to a message received by one of the class's objects, indicating the complexity of the class's interface. A higher RFC value suggests increased potential for interactions, which could lead to greater testing and maintenance challenges.

17. **WMC (Weighted Methods per Class):** Calculates the complexities of all methods in a class.

	classname	cbo	dit	fanIn	fanOut	lcom	noc
0	org::eclipse::jdt::internal::core::search::ind...	9	2	1	9	15	0
1	org::eclipse::jdt::internal::compiler::codegen...	1	1	1	0	0	0
2	org::eclipse::jdt::internal::compiler::ast::AS...	114	1	102	18	190	6
3	org::eclipse::jdt::internal::compiler::lookup:...	5	6	1	4	10	0
4	org::eclipse::jdt::internal::eval::CodeSnippet...	23	2	1	22	820	0
...	...	...	...	...	...	...	...

FIGURE 3.7: Product Metrics File Columns

numberOfPublicAttributes	numberOfPublicMethods	rfc	wmc
1.0	5	34.0	20.0
2.0	1	1.0	1.0
3.0	19	156.0	176.0
0.0	4	18.0	12.0
7.0	1	174.0	115.0
...	...	...	...

FIGURE 3.8: Product Metrics File Columns

3.2.2 Data Pre-Processing

Data preprocessing is an important step in getting raw data ready for analysis or machine learning tasks. It entails a variety of operations and transformations to clean, format, and organize data in order to prepare it for further analysis or model

training. This dataset was then used to train machine learning and deep learning models for predicting software faults. Below Figure 3.9 lists the preprocessing steps that were performed on the datasets. Without proper preprocessing, models may struggle to capture relevant patterns, leading to poor performance and unreliable results.

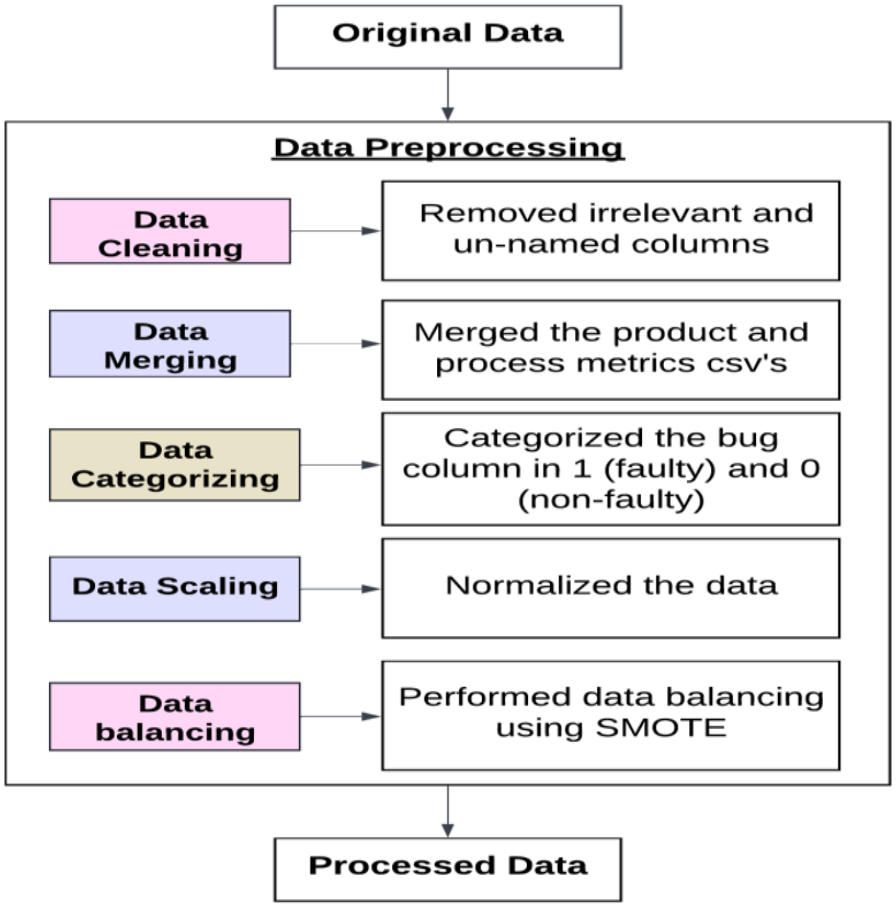

FIGURE 3.9: Preprocessing Steps

The preprocessing steps that the dataset undergone are explained below in detail with the code that was implemented for the required results.

3.2.2.1 Removing Irrelevent Columns From All Files

To prepare data for analysis, we cleaned it by removing irrelevant columns using the below function and standardized the remaining data.

```
def remove_unnamed_cols(data):
   return data.loc[:,~data.columns.str.contains('^Unnamed')]
```

Removing Unnamed columns from target file:

```
data_bugs = pd.read_csv('/kaggle/input/dataset/eclipse/eclipse
/bug-metrics.csv', sep='\s;\s*', engine='python')
data_bugs_new=data_bugs.drop(["classname","numberOfBugsFoundUntil:",
"numberOfNonTrivialBugsFoundUntil:","numberOfMajorBugsFoundUntil:",
"numberOfCriticalBugsFoundUntil:","numberOfHighPriorityBugsFoundUntil:",
"nonTrivialBugs","majorBugs","criticalBugs",
"highPriorityBugs"], axis='columns')
data_bugs_new = remove_unnamed_cols(data_bugs_new)
data_bugs_new
```

Removing unnamed columns from other files; as shown in the code below.

```
data_change = pd.read_csv('/kaggle/input/dataset/eclipse/eclipse/
change-metrics.csv', sep='\s;\s*',
engine='python')
data_single_version_ck_oo = pd.read_csv('/kaggle/input/dataset/eclipse/
eclipse/single-version-ck-oo.csv', sep='\s;\s*', engine='python')
data_change=data_change.drop(["nonTrivialBugs",
"majorBugs","criticalBugs","highPriorityBugs"], axis='columns')
data_single_version_ck_oo=data_single_version_ck_oo.
drop(["nonTrivialBugs","majorBugs","criticalBugs",
"highPriorityBugs"], axis='columns')
```

3.2.2.2 Merging All Files

For the product metrics dataset, we merged the product metrics file with the target file. For the combined metrics dataset, we merged the product metrics, change-metrics and target file, aim is to use the merged file for training models.

```
# merge data
data = data_change.merge(data_single_version_ck_oo, how='left')\
.merge(data_bugs_new['bugs'],how='left')
#remove unnamed columns
eclipse = remove_unnamed_cols(data)
eclipse
```

3.2.2.3 Dropping Duplicates

```
eclipse = eclipse.apply(lambda x: x.astype(str).str.lower()).
drop_duplicates( keep='first')
eclipse
```

3.2.2.4 Converting To Int Type

```
 eclipse['bugs']= eclipse['bugs'].astype(int)
```

3.2.2.5 Categorizing Bug Column To 0 And 1

Our dataset was originally a regression dataset, but we converted it into a binary classification dataset. The bug column was initially categorized into more than 2 classes, as shown below, which required conversion to fit the binary classification model.

```
Class  Bugs
0       791
1       138
2       31
3       15
4        8
6        4
7        3
8        3
5        2
9        2
Name: count, dtype: int64
```

We categorized the bug column as faulty (1) or non-faulty (0) to performed binary classification for software fault prediction.

We chose binary classification because it has been commonly used in the literature, and we want to compare our results with those of other studies. This approach also simplifies the evaluation of model performance and allows for more straightforward benchmarking against established methodologies.

```
for i in range(len(eclipse)):
   if eclipse['bugs'].values[i] > 0:
       eclipse['bugs'].values[i] = 1
```

Bug column after Categorization:

```
Class  Bugs
0       791
1       206
Name: count, dtype: int64
```

3.2.2.6 Removing Class Name Column

The class name column was unused, so it was removed from the merged file to streamline the dataset and focus on relevant features.

```
eclipse=eclipse.drop(['classname'], axis=1)
```

3.2.2.7 Final Preprocessed And Merged File

The figures below display some columns from the final preprocessed and merged dataset. These columns illustrate the data after cleaning and modifications.

	numberOfVersionsUntil:	numberOfFixesUntil:	numberOfRefactoringsUntil:	numberOfAuthorsUntil:
0	65	4	0	8
791	2	0	0	2
1582	120	10	0	12
1720	28	4	0	5
2511	93	17	0	8
...	...	...	...	...

FIGURE 3.10: Final Merged File

	avgLinesRemovedUntil:	codeChurnUntil:	maxCodeChurnUntil:	avgCodeChurnUntil:
0	8.70769	42.0	15.0	0.646154
791	0.0	10.0	10.0	5.0
1582	6.4	593.0	66.0	4.94167
1720	4.39286	15.0	23.0	0.535714
2511	20.2258	-11.0	18.0	-0.11828
...	...	...	...	...

FIGURE 3.11: Final Merged File

	numberOfAttributes	numberOfAttributesInherited	numberOfLinesOfCode	numberOfMethods
0	1	8	122.0	6.0
791	2	0	4.0	1.0
1582	131	249	484.0	20.0
1720	0	61	33.0	5.0
2511	7	416	673.0	41.0
...	...	...	...	...

FIGURE 3.12: Final Merged File

	numberOfPublicMethods	rfc	wmc	bugs
0	5	34.0	20.0	0
791	1	1.0	1.0	0
1582	19	156.0	176.0	1
1720	4	18.0	12.0	0
2511	1	174.0	115.0	0
...	...	...	...	...

FIGURE 3.13: Final Merged File

3.2.3 Data Balancing

Figure 3.14 shows the data imbalance ratio. Since the data was imbalanced, we needed to balance it because it is an important challenge in software fault prediction is handling imbalanced datasets.

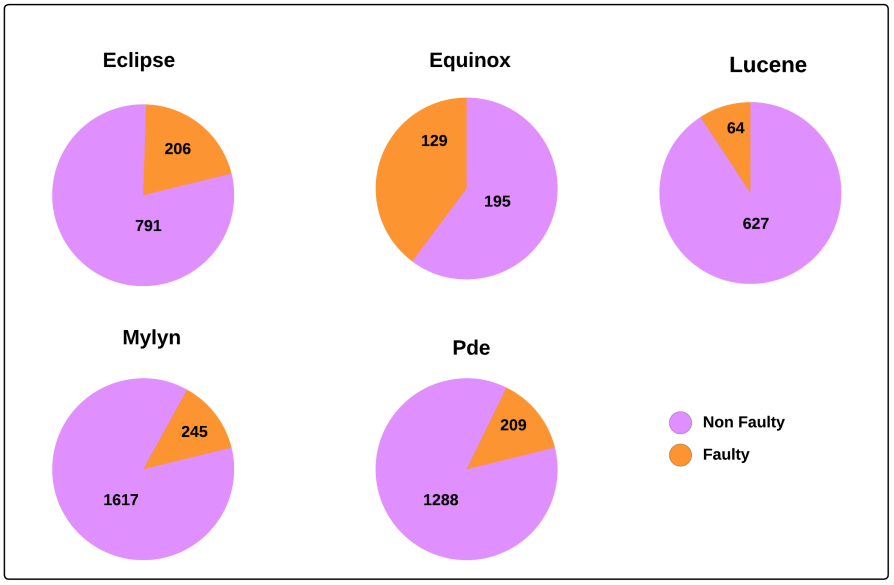

FIGURE 3.14: The Datasets Imbalanced Ratio

When one class (such as software instances with faults) fails to appear frequently in comparison to another (such as non-faulty instances), the dataset is said to be imbalanced. This significant difference may cause models to be biased in favor of the majority class, making it difficult to accurately identify the minority class. [59] As a result, the model might miss many important instances of the minority class, making it less effective.

To improve results, it's important to balance the dataset through techniques that are designed to handle class imbalances, and we have used **Synthetic Minority Over-sampling Technique (SMOTE)** to balance the data because it is the most commonly used method in the literature. Instead of oversampling with replacement, SMOTE creates synthetic examples, increasing the number of instances for the minority class (faulty modules in our study). This approach ensures a more robust training process, allowing the model to better learn patterns.

3.3 Deep Learning Models

This section describes the DL model's that we used for experimentation in our research. We observed from the literature that CNN, LSTM and BILSTM have been producing excellent results in the classification of datasets using numeric data, so we decided to use them in our study.

3.3.1 Convolutional Neural Networks (CNN)

Neurons in a neural network receive inputs, and before moving on to the next neuron, the weighted sum of those inputs passes through an activation function.

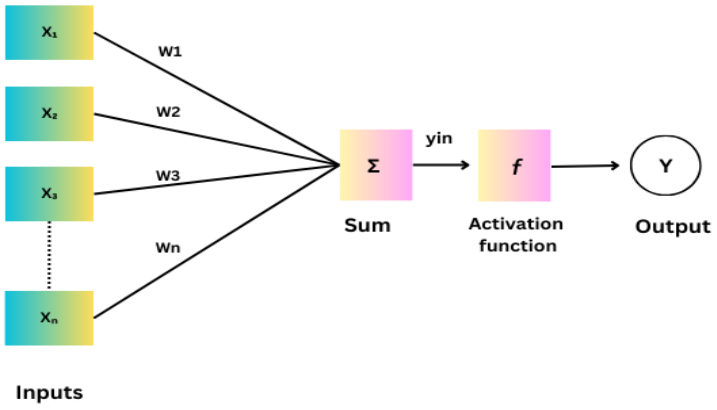

FIGURE 3.15: Basic Neural Network Structure

A convolutional neural network differs from a neural network in that it processes a large number of inputs. Figure 3.15 shows the basic structural design of a Convolutional Neural Network (CNN).

In our research study, we used a 1D convolution instead of a 2D convolution; since you have 1D data, a 1D convolution is more suitable. As 1D CNN performs well at identifying patterns in numerical data over time, that is why we selected it. Because it can directly learn valuable features from the data, it can better predict software faults by understanding the sequential changes in the data. Figure 3.16 shows the proposed CNN architecture.

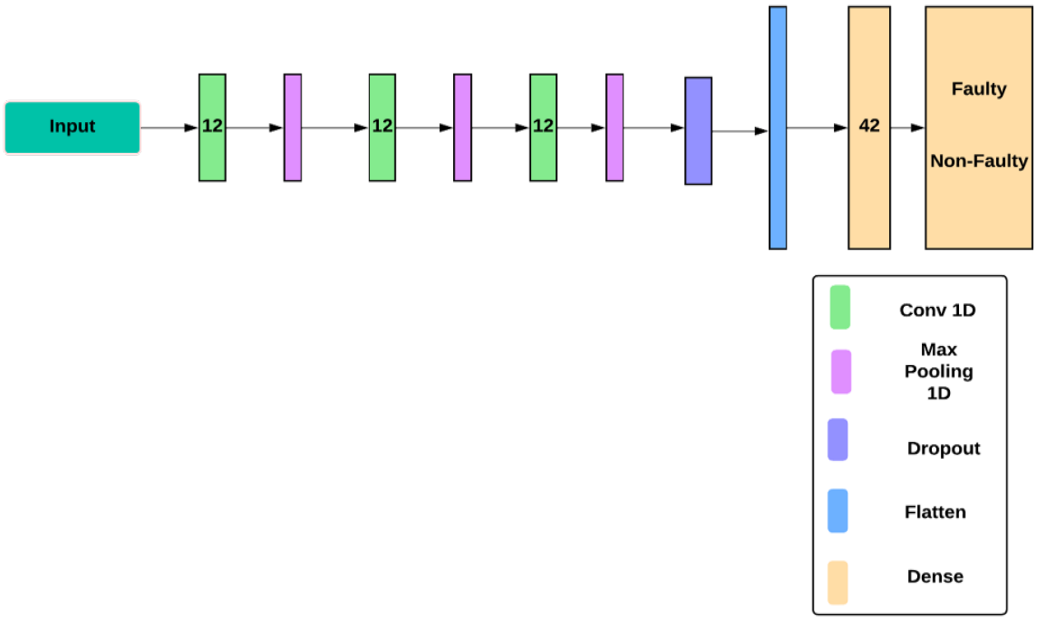

FIGURE 3.16: Proposed CNN Architecture

For **Product metrics**, the 1D Convolutional Neural Network (CNN) for binary classification begins with an AveragePooling1D layer, which is followed by a Conv1D layer with 12 filters, a kernel size of 2, ReLU activation, and L2 regularization. A deep convolutional structure is produced by repeating this combination twice more.

To avoid overfitting, a Dropout layer with a 20% dropout rate is added after the convolutional layers. The binary classification result is then obtained by flattening the output and passing it via a Dense layer with 42 units and ReLU activation, along with L2 regularization. Lastly, a Dense output layer with 1 unit and sigmoid activation produces the output.

For **Combined metrics**, the ReLU activation function is used to extract features from the input data using three Conv1D layers, each configured with 12 filters and a kernel size of 2. To decrease the feature maps' spatial dimensions, a MaxPooling1D layer with a pool size of two is added after each convolutional layer.

A 20% dropout rate dropout layer is added immediately after the pooling layers to help avoid overfitting.

3.3.2 Long Short-Term Memory (LSTM)

Long-Short Term Memory (LSTM) Networks are a subset of RNNs used in deep learning to identify patterns in data sequences. Memory blocks in the LSTM architecture are linked together via recurrent sub-networks. For software fault prediction, LSTMs outperform traditional machine learning models in terms of accuracy and interpretability. We chose LSTM because it is designed to capture long-range dependencies and temporal patterns in sequential data. Its architecture effectively manages and retains information over extended sequences, making it ideal for analyzing trends and patterns in software metrics that span across multiple time steps.

3.3.3 LSTM Architecture

An LSTM network consists of an input layer, memory unit, gates (forget gate, input gate, cell state, output gate), and an output layer. Their details are mentioned below. The basic structure of LSTM is shown in Figure 3.17.

1. **Input layer:** This layer receives a set of software metrics, including code complexity, coupling, cohesion, and other relevant characteristics. These metrics are extracted from the software codebase using a variety of tools and methods. The accuracy of these metrics is crucial for ensuring the model's performance and reliability.

2. **LSTM Layer:** The LSTM layer is the network's core component that learns long-term dependencies in the input sequence. It consist of memory cells, input gates, output gates, and forget gates.

   (a) Memory Cells: LSTMs have a unique structure called memory cells, which help retain information over long periods. These cells have built-in mechanisms called gates to regulate the flow of information.

   (b) Forget Gate: Selects the data that should be removed from the previous cell state.

(c) Input Gate:It controls what information from previous output and current data enters the memory cell.

(d) Output Gate: Based on the state of the cell, controls the output. Together, these gates enable LSTMs to effectively maintain and update data and identifying in the data any long-term dependencies.

3. **Output Layer:** This layer takes the hidden state from the LSTM layer and computes a probability score that indicates the likelihood of a fault.

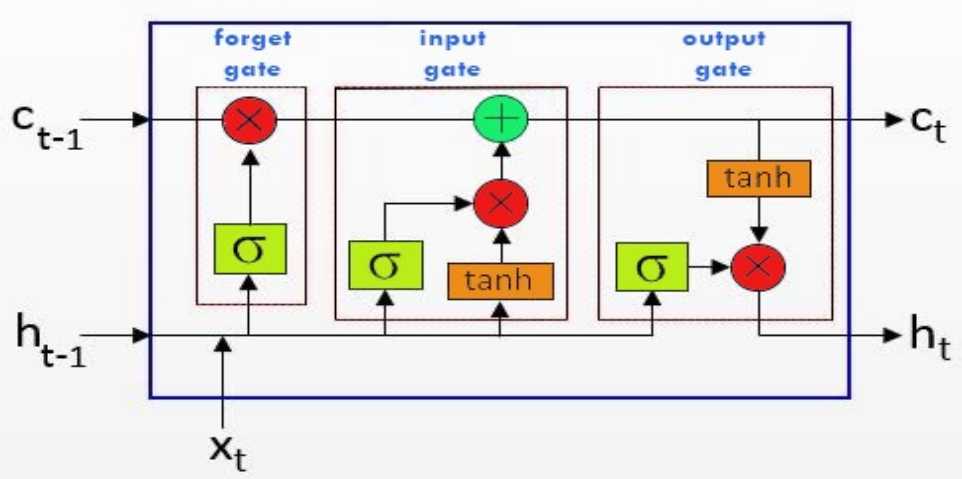

FIGURE 3.17: Basic Structure of LSTM Network

1. **Forget Gate**:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.1}$$

2. **Input Gate**:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3.2}$$

3. **Cell State**:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{3.3}$$

4. **Output Gate**:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)h_t = o_t * \tanh(C_t) \tag{3.4}$$

The proposed LSTM architecture is shown in Figure 3.18.

For **Product metrics**, in order to capture temporal dependencies within the input data, a 50-unit LSTM layer is first added, then dropout layer is added. To produce a binary classification result, a dense layer with a sigmoid activation function is then included.

For **Combined metrics**, dense and LSTM layers are used for adding dense layers and LSTM for creating a linear stack of layers, respectively. We then initialize a sequential model. The input shape of the 50-unit LSTM layer is set to match the time steps and features of the training data. Subsequently, a sigmoid activation function and a single unit dense layer are added to produce a probability appropriate for binary classification.

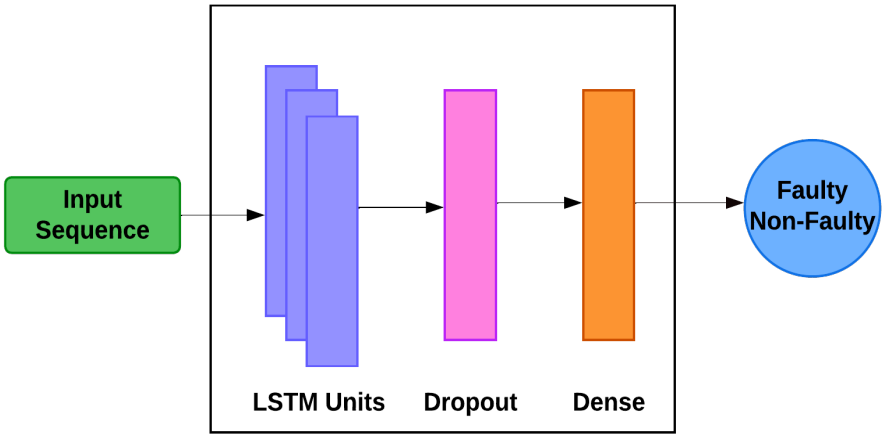

FIGURE 3.18: Proposed LSTM Architecture

3.3.4 Bi-directional LSTM (BILSTM)

Bidirectional Long-Short Term Memory (BI-LSTM) networks are a variant of traditional LSTM networks. They are intended to improve the learning of sequential data by taking into account both past and future contexts. They excel at capturing dependencies in sequential data, which is crucial for analyzing complex patterns in code and identifying potential faults effectively. This feature makes BI-LSTMs especially useful for tasks like software fault prediction, where understanding the full context of code changes and metrics can lead to more accurate

predictions.BI-LSTM networks capture contextual information in both directions, resulting in a more complete understanding of the input sequence. BILSTM was chosen because it can process sequences both forward and backward, giving us a more complete understanding of the data. This bidirectional approach enhances the model's ability to capture context from both past and future data points, improving its accuracy in software fault prediction. Figure 3.19 shows the proposed BILSTM architecture.

A BI-LSTM network is comprised of the following components:

1. **Input Layer:** This layer receives a set of software metrics, including code complexity, coupling, cohesion, and other relevant characteristics. These metrics are extracted from the software codebase using a variety of tools and methods.

2. **Forward LSTM Layer:** The forward LSTM layer gathers historical data by sequentially processing the input sequence in a forward direction (from the beginning to the end).

3. **Backward LSTM Layer:** In order to gather data from the future, the backward LSTM layer processes the input sequence backwards—that is, from the beginning to the end.

4. **Concatenation Layer:** Concatenating the results of the unidirectional and bidirectional LSTM layers in a single output that provides a complete understanding of the sequence.

5. **Output Layer:** The output layer takes the concatenated output and calculates a probability score that indicates the likelihood of a software fault occurring.

BI-LSTM networks provide a powerful approach for software fault prediction by taking advantage of their ability to consider both past and future contexts in sequential data. This capability results in more accurate and reliable fault predictions, which contribute to the development of high-quality software systems.

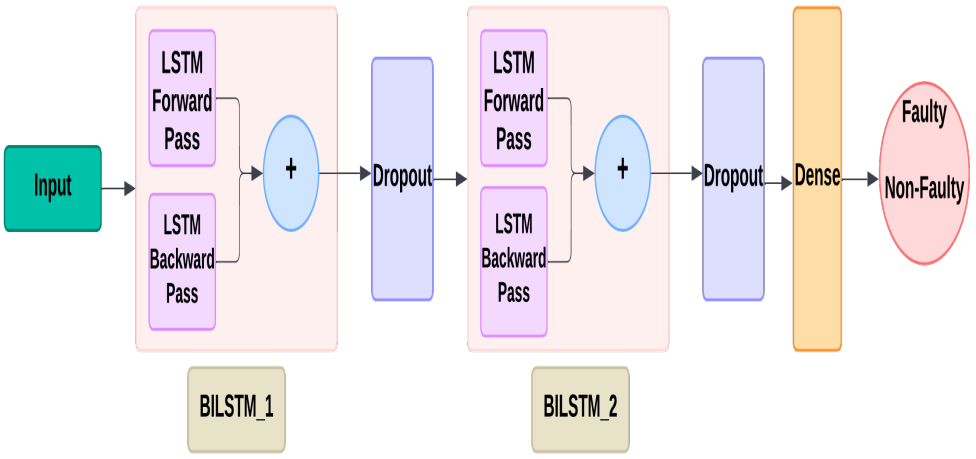

FIGURE 3.19: Proposed Structure of BILSTM Network

For **Product metrics**, Bi-directional Long Short Term Memory runs on LSTM forward and backward pass with 128 units that is configured to return sequences. To avoid overfitting, this layer is followed by a dropout layer with a 20% dropout rate.

Subsequently, an additional 64-unit Bidirectional LSTM layer with a sigmoid activation function and another Dropout layer with an identical dropout rate are implemented. The final layer, which is dense and features a single unit with a sigmoid activation function, generates a probability for binary classification.

For **Combined metrics**, the required layers and models, such as Dense, LSTM, Activation, Dropout, Embedding, Bidirectional, and Sequential, are first imported. A bidirectional LSTM with 128 units is the first layer added to the model. It processes the input sequence both forward and backward and returns sequences to the next layer.

Next, a dropout layer with a 20% dropout rate randomly sets input units to zero during training to help avoid overfitting. Next, a sigmoid activation function and 64 units of a second bidirectional LSTM layer are added. Another dropout layer with a 20% dropout rate comes after this. After that, a sigmoid activation function and a dense layer with one unit are added to produce a probability appropriate for binary classification.

3.4 Comparisons

3.4.1 Metrics Comparison

The graphical representation of metrics based comparison is clearly shown in Figure 3.20. This process begins by gathering metrics from the software product, such as code complexity, design, and functionality, as well as process metrics like defect history, developer experience, and team size.

Data preprocessing prepares information for machine learning by cleaning (removing inconsistencies, duplicates, and missing values) and scaling (normalizing features). The dataset is then divided into two parts: a training set for model training and a test set for performance assessment. The chosen DL model, such as CNN is trained using the training dataset. The model modifies its parameters to reduce prediction errors. The model's performance on the test set is measured using metrics such as accuracy, precision, recall, and F1-score. The results are visualized using bar charts and confusion matrices, which show the model's performance and prediction accuracy.

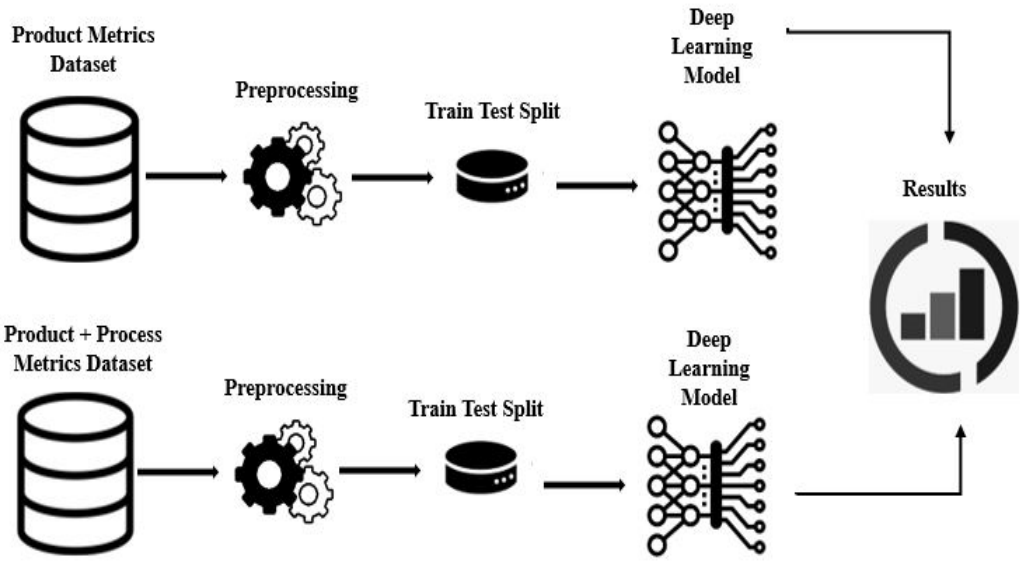

FIGURE 3.20: Metrics Comparison

3.4.2 Model's Comparsion

The graphical representation of model's performance based comparison is shown in Figure 3.21. The process begins with gathering information about the software, such as product metrics (code complexity, size, and number of lines of code) and process metrics (team size and development methodology). These metrics create the dataset used to train the model. Preprocessing is the process of cleaning and preparing data, which includes handling missing values, removing outliers, generating new features, and normalizing the data. The preprocessed dataset is then divided into two sets: one for training the model and one for testing its performance. We then use ML and DL models to validate the findings. After training, The efficacy of the model is evaluated by the utilization of accuracy, precision, recall, and the F1-score. This approach aids in the identification of code areas that are likely to contain faults, allowing teams to better allocate resources and improve software quality. The choice between traditional and deep learning models is determined by data complexity and accuracy requirements.

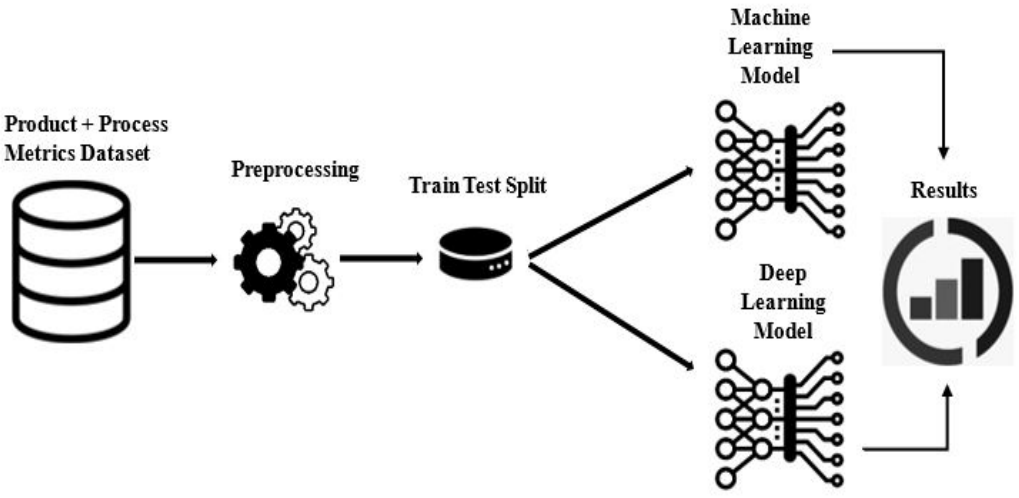

FIGURE 3.21: Model's Performance Comparsion

Our analysis of the literature that is presented in Chapter 2 showed that machine learning has produced remarkable results. We examined machine learning models from the literature and chose to compare our proposed approach with KNN, Naive Bayes, and LR because these models are commonly used and have achieved better

results. This will enable us to evaluate whether our deep learning-based approach performs better than these conventional techniques.

3.4.2.1 K-Nearest Neighbours

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 (neighbors) to a new data point using a selected distance metric, typically Euclidean distance [60]. 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. For regression, the algorithm predicts a numerical value by averaging (or weighted averaging) the values of the nearest neighbors.

Due to its simplicity and ease of understanding, KNN is a popular choice for beginners in ML and for tasks where interpretability and simplicity are important. But in addition, its effectiveness can vary based on the choice of K and the quality of the distance metric used [61].

3.4.2.2 Naive Bayes

Naive Bayes is a probabilistic classification method that uses Bayes' theorem, assuming that features are independent. Despite its simplicity, it has demonstrated considerable effectiveness in various machine learning applications, including text classification, spam detection, and software fault prediction.

The naive Bayes classifier is based on the Bayes theorem, which is stated as follows.

$$P(A \mid B) = \frac{P(A) \cdot P(B \mid A)}{P(B)} \tag{3.5}$$

- $P(A \mid B)$ is the posterior probability of A given B.
- $P(B \mid A)$ is the likelihood of B given A.
- $P(A)$ is the prior probability.

- $P(B)$ is the evidence probability.

The Naive Bayes(NB) method treats defect prediction as a binary classification problem. It builds a predictive model by analyzing historical data of software modules. Using this model, it determines whether a new module is predicted to have defects or not based on its features and the learned patterns from past data [62].

3.4.2.3 Logistic Regression

To predict the likelihood that an observation will belong to a certain class (often represented as 1 or 0), statistical methods such as logistic regression are employed in binary classification. Logistic regression, despite its name, is not a regression model but a linear model for classification. LR can be calculated as:

$$\hat{y} = b_1 x_1 + b_2 x_2 + \cdots + b_k x_k + a \tag{3.6}$$

- $\hat{y}$ is the predicted value (dependent variable)
- b1,b2,b3. . . ..bk are the coefficients for independent variables.
- x1,x2,x3. . . .xk are the independent variables.
- a is the intercept term.

The logistic model is based on the logistic function, which is unique because it maps any input from negative to positive infinity to an output range between 0 and 1.

$$\sigma(x) = \frac{1}{1 + e^{-z}} \tag{3.7}$$

The outcome of applying the logistic function to the upper regression equation at this point is:

$$\hat{y} = \sigma(b_1 x_1 + b_2 x_2 + \cdots + b_k x_k + a) = \frac{1}{1 + e^{-(b_1 x_1 + b_2 x_2 + \cdots + b_k x_k + a)}} \tag{3.8}$$

3.4.3 Evaluation Metrics

Performance measures in DL and ML are important benchmarks for understanding and improving model performance. These metrics help to guide model selection and fine-tuning by quantifying aspects such as predictive accuracy, robustness, and efficiency. They help in the detection of overfitting and underfitting by revealing how well models generalize to previously unseen data. Metrics also make it easier to compare different models and algorithms, ensuring that the approach chosen is appropriate for the project's goals and constraints. In this research study we have used Accuracy, Precision, Recall and F1 score as evaluation metrics.

3.4.3.1 Accuracy

Accuracy is a crucial metric for evaluating performance in both ML and DL, indicating the proportion of correctly classified instances in the dataset. It is calculated as the proportion of true positives (correctly predicted instances) and true negatives (correctly rejected instances) to the total number of instances.

Accuracy is a simple and intuitive metric that provides an overview of a model's performance. Its drawbacks, however, are most clear when working with unbalanced datasets, where a model might obtain excellent accuracy by only predicting the majority class. Despite this, accuracy is still widely used, especially in applications like image processing, NLP, and bioinformatics.

3.4.3.2 Precision

Precision is a crucial performance metric in machine learning and deep learning that calculates how accurate positive predictions are. The ratio of true positive predictions to the sum of true positive and false positive predictions is how it is defined. In fields like fraud detection or medical diagnosis, where the cost of false positives is significant, precision is extremely helpful.

Precision is an important metric for applications where false positives can have serious consequences because it emphasizes the accuracy of positive predictions.

3.4.3.3 Recall

Recall, also known as sensitivity or true positive rate, is an important evaluation metric in machine learning and deep learning that evaluates a model's capacity to find every relevant occurrence inside a dataset. The ratio of true positive predictions to the sum of true positives and false negatives is how it is defined. A high recall value means that most real positive instances are accurately captured by the model, which is particularly significant in situations when it is expensive to miss positive cases, such as disease screening or security threat detection. By emphasizing the completeness of positive class predictions, recall ensures that the model does not overlook significant instances, providing a thorough assessment of its ability to recognize all relevant cases.

3.4.3.4 F1 Score

The F1-score is a popular evaluation metric in ML and statistics, especially in tasks where precision and recall are important. It combines these two metrics to create a single value that balances them, making it useful for evaluating a classifier's overall performance. The harmonic mean of recall and precision yields the F1 score, that indicates how well a model's predictions match true positive cases while accounting for both false positives and negatives.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3.9}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.10}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.11}$$

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{3.12}$$

Chapter 4

Results and Discussion

A detailed explanation of the proposed methodology can be found in Chapter 3. This chapter is dedicated to presenting the results obtained through the application of proposed methodology.

4.1 Tools and Technologies

4.1.1 Python Programming Language

Python is an immensely popular programming language, renowned for its versatility and widespread use. It's easy to read and use, making it great for both beginners and experts. One of its big advantages is the large number of libraries available, which provide powerful tools for ML and DL. Furthermore, its strong community support ensures continuous improvements and a wealth of resources for problem-solving and learning.

4.1.2 Kaggle Jupyter Notebook

Kaggle Jupyter Notebook is a web-based platform that allows you to write and execute Python code. It's widely used because of its user-friendly interface and

comprehensive tools for data science and easy access to a wide variety of datasets and kernels.

4.1.3 TensorFlow And Keras Libraries

TensorFlow and Keras are leading libraries in DL. TensorFlow is a robust, low-level library that offers essential tools for building complex DL models, known for its flexibility and strength in supporting a wide range of ML and DL algorithms. In contrast, Keras is a high-level library designed to simplify TensorFlow, making it easier to create and train DL models. Keras is particularly valued for its user-friendly interface, which caters to DL practitioners of all experience levels.

4.2 Implementation of Models

The dataset was split into training, validation, and test sets, which were then fed into three different models.

4.2.1 CNN Model

Table 4.1: CNN Parameters

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

The first model is a Convolutional Neural Network (CNN),and its implementation details are in above Table 4.1. It begins with a 1D convolutional layer (‘Conv1D‘) with 64 filters, a kernel size of 3, and ReLU activation, processing input shaped (number of features, target class). This layer is followed by a max pooling layer (‘MaxPooling1D‘) with a pool size of 2 to reduce dimensionality. The output is first converted into a 1D array using the ’Flatten’ layer. This is followed by adding a ’dense’ layer with 100 units and a ReLU activation function. To reduce the risk of overfitting, a Dropout layer with a rate of 0.5 is then applied. With a sigmoid activation function for binary classification, the final dense layer consists of one unit. A performance parameter called accuracy is used along with the Adam optimizer, the binary cross-entropy loss function, and the model’s compilation. Using a batch size of 128 and 200 epochs of training data, it is verified using validation data.

4.2.2 LSTM Model

TABLE 4.2: LSTM Parameters

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

The second model is a LSTM network, and its implementation details are in above Table 4.2. It begins with an LSTM layer with 128 units, set to return sequences, and expects input with the shape (number of features, target class). A dropout layer with a rate of 0.2 follows to prevent overfitting by randomly dropping 20% of the units during training. The second LSTM layer has 64 units and uses the sigmoid activation function. Another dropout layer with a rate of 0.2 is added. The final layer is a dense layer with 1 unit and a sigmoid activation function, making it suitable for binary classification. The binary cross-entropy loss function, the Adam optimizer, and accuracy as a performance indicator are used in the compilation of the model. Using the training set, it runs 200 training epochs with a batch size of 128 to confirm how well it performed using the validation data. This configuration aims to balance model complexity and performance, ensuring robust results through rigorous training and evaluation.

4.2.3 BILSTM Model

TABLE 4.3: BILSTM Parameters

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

The third model is a Bidirectional LSTM (BILSTM) network, and its implementation details are in above Table 4.3. It begins with a Bidirectional LSTM layer with 128 units, set to return sequences, and processes input with the shape (number of features, target class). This is followed by a dropout layer with a rate of 0.2 to prevent overfitting.

Another bidirectional LSTM with 64 units and a sigmoid activation function makes up the second layer. There is also another dropout layer with a 0.2 rate. With a sigmoid activation function and one unit, the final layer is dense, making it suitable for binary classification. The model is compiled using the Adam optimizer, the binary cross-entropy loss function, and accuracy as a performance measure. With a batch size of 128 and 200 epochs, it is trained on the training data and verified on the validation data.

4.2.4 Models Evaluation

All the models were evaluated on the test data using accuracy, precision, recall and F1 score.

4.3 Results Of Experiments

To validate our approach we performed two type of comparisons: first, we compared product metrics with combined metrics using deep learning, and second, we compared ML techniques with DL methods using combined metrics across five datasets: Eclipse, Equinox, Lucene, Mylyn, and Pde. Based on a literature review, we selected Logistic Regression, Naive Bayes, and K-Nearest Neighbors for ML, and LSTM, CNN, and BILSTM for DL. F1 scores, accuracy, precision, and recall were used to evaluate the models. Our results indicates that deep learning models consistently outperformed machine learning models, with the best results achieved using the combined dataset. Specifically, the deep learning models demonstrated superior accuracy, higher precision, better recall, and improved F1 scores across all datasets. This highlights the potential of deep learning techniques in enhancing

the performance of predictive models for software metrics. Furthermore, incorporating process metrics along with product metrics yields remarkable results.

4.4 Metrics Comparison

In this section, we discuss the results of metrics based comparison. We used two types of metrics: product metrics and combined metrics (product + process), with details provided in section 3.2.1. The analysis and comparison are performed using various deep learning models to evaluate their effectiveness in handling both types of metrics. The results offer insights into the models' performance across different datasets.

4.4.1 Accuracy

Accuracy is the frequency with which a model predicts the outcome correctly. The accuracy of all five datasets are arranged in below Table 4.4 in accordance with the DL models results.

This table highlights the comparative performance of each model across the product and combined metrics datasets, emphasizing the differences in their predictive power.

Table 4.4: Accuracy of Metrics Dataset

	Product Metrics			**Combined Metrics**		
	CNN	LSTM	BILSTM	CNN	LSTM	BILSTM
ECLIPSE	0.77	0.86	0.86	**0.93**	0.83	0.85
EQUINOX	0.69	0.64	0.69	0.80	**0.88**	0.87
LUCENE	0.79	0.83	0.83	0.90	0.83	**0.91**
MYLYN	0.81	0.87	0.86	**0.93**	0.86	**0.89**
PDE	0.74	0.88	0.89	0.86	0.88	**0.90**

4.4.2 Precision

Precision quantifies the proportion of true positive instances among all the instances that the model predicted as positive. Below Table 4.5 shows the results of precision on all five datasets utilized in our research.

TABLE 4.5: Precision of Metrics Dataset

	Product Metrics			**Combined Metrics**		
	CNN	LSTM	BILSTM	CNN	LSTM	BILSTM
ECLIPSE	0.83	0.88	0.88	**0.93**	0.83	0.84
EQUINOX	0.66	0.60	0.73	0.81	0.88	**0.89**
LUCENE	0.65	0.69	0.69	0.78	0.69	**0.80**
MYLYN	0.79	0.85	0.80	**0.87**	0.81	0.84
PDE	0.71	0.82	0.84	0.82	0.81	**0.84**

4.4.3 Recall

The frequency with which a model correctly identifies true positives from among all of the real positive samples in the dataset is measured by recall. The recall of all five datasets are arranged in below Table 4.6 in accordance with the DL models results.

TABLE 4.6: Recall of Metrics Dataset

	Product Metrics			**Combined Metrics**		
	CNN	LSTM	BILSTM	CNN	LSTM	BILSTM
ECLIPSE	0.75	0.88	0.87	**0.97**	0.88	0.84
EQUINOX	0.73	0.78	0.57	0.87	0.85	**0.89**
LUCENE	0.99	0.93	0.95	0.97	0.93	**1**
MYLYN	0.79	0.88	0.94	0.98	0.92	**0.94**
PDE	0.72	0.95	0.95	0.90	0.96	**0.99**

4.4.4 F1 Score

The F1 score is the harmonic mean of the precision and recall of a classification model. The F1 Score of all five datasets are arranged in below Table 4.7 in accordance with the DL models results.

TABLE 4.7: F1 Scores of Metrics Dataset

	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**

4.5 Model's Comparison

In this section, we discuss the results of model-based comparisons. We employed both machine learning (ML) and deep learning (DL) models, with details provided in Sections 3.3 and 3.4. The analysis is focused on the combined metrics dataset, where we compare the performance of each model to assess their accuracy and effectiveness across various evaluation criteria.

4.5.1 Accuracy

Accuracy measures how closely the predictions of machine learning and deep learning models match the true outcomes when using a combined dataset. The accuracy of all five datasets are arranged in below table in accordance with the machine learning and deep learning models results using combined metrics dataset. Comparing accuracy helps determine which model more correctly predicts results.

TABLE 4.8: Accuracy of Combined Dataset

	ML Models			**DL Models**		
	KNN	NB	LR	CNN	LSTM	BILSTM
ECLIPSE	0.91	0.65	0.81	**0.93**	0.83	0.85
EQUINOX	0.76	0.87	0.87	0.80	0.87	**0.88**
LUCENE	0.86	0.72	0.75	0.90	0.83	**0.91**
MYLYN	0.91	0.70	0.75	**0.93**	0.86	0.89
PDE	0.87	0.70	0.75	0.86	**0.88**	**0.90**

4.5.2 Precision

Precision evaluates how often the model's correctly identify positive instances among all their positive predictions. Precision for all datasets is in the table below, from ML and DL models with combined metrics.

TABLE 4.9: Precision of Combined Metrics Dataset

	ML Models			**DL Models**		
	KNN	NB	LR	CNN	LSTM	BILSTM
ECLIPSE	0.91	0.85	0.88	**0.93**	0.83	0.84
EQUINOX	0.77	0.88	0.85	0.81	0.88	**0.89**
LUCENE	0.73	0.65	0.62	0.78	0.69	**0.80**
MYLYN	0.86	0.74	0.69	**0.87**	0.81	0.84
PDE	0.79	0.70	0.75	0.82	0.81	**0.84**

4.5.3 Recall

Recall measures how effectively each model identifies all relevant positive instances. The recall of all five datasets are arranged in below table in accordance with the ML and DL models results using combined metrics dataset.

Table 4.10: Recall of Combined Metrics Dataset

	ML Models			**DL Models**		
	KNN	NB	LR	CNN	LSTM	BILSTM
ECLIPSE	0.96	0.49	0.79	**0.97**	0.88	0.84
EQUINOX	0.73	0.84	0.89	0.87	0.85	**0.89**
LUCENE	0.98	0.50	0.82	0.97	0.93	**1**
MYLYN	0.97	0.56	0.83	**0.98**	0.92	0.94
PDE	0.98	0.65	0.71	0.90	0.96	**0.99**

4.5.4 F1 Score

F1 Score balances precision and recall into a single metric. It demonstrates the model's accuracy in positive predictions and its ability to capture relevant instances. The F1 score of all five datasets are arranged in below table in accordance with the ML and DL models results using combined metrics dataset.

Table 4.11: F1 Scores of Combined Metrics Dataset

	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**

4.6 Analysis of Results

We are utilizing bar charts to analyze the results in this section. We started by evaluating the performance of the product metrics and combined metrics datasets with deep learning models. Next, we evaluated both ML and DL models on the combined metrics datasets to conduct a more comprehensive analysis.

4.6.1 Analysis of Results for Product and Combined Metrics

In the figures provided below, from 4.1 to 4.5, provides detailed analysis of F1 Scores across different DL models on five datasets indicate several trends, highlighting the comparative performance of each model.

For the **Eclipse** dataset reveals that the CNN model achieves higher F1 Score with the combined dataset. The LSTM model shows a marginal increase in F1 Score with the combined dataset. The BILSTM model's score are nearly identical for both datasets.

For the **Equinox** dataset, the CNN model shows a significant increase with combined dataset.The LSTM model also benefits from the combined dataset, with a noticeable improvement in F1 Score. The BILSTM model similarly shows a slight enhancement in F1 Score with the combined dataset, supporting the trend that additional process data positively impacts model performance for this dataset.

In the case of the **Lucene** dataset, the CNN model interestingly performs better with combined dataset, Conversely, the LSTM model shows almost identical performance on both datasets, The BI-LSTM model shows a marginal improvement with the combined dataset, suggesting that it can make better use of the additional process data to enhance its predictions.

In the **Mylyn** dataset, the CNN demonstrates a marginal increase with the combined dataset, while both the LSTM and BILSTM models perform equally well on the two datasets, with no noticeable differences in F1 Scores. This suggests that for Mylyn, the inclusion of process data does not significantly impact the performance of these two models, indicating that the product metrics alone are sufficient for effective classification in this instance.

Finally, **PDE** dataset, the CNN model shows a improvement using combined dataset (Product + Process) compared to the Product dataset alone, suggesting that the process metrics provide useful additional information. The LSTM model exhibits nearly identical F1 Scores for both datasets, indicating that this model's

performance is relatively unaffected by the inclusion of process metrics in this specific case. The BI-LSTM model also shows a small improvement when using the combined dataset, which aligns with the overall trend of enhanced performance with combined datasets.

Overall, the variations in the results are due to the performance of the dataset being influenced by its characteristics, such as the imbalance ratio and the number of instances. Each project dataset has unique characteristics, including its code base, commit history, bug reports, and development practices, which impact deep learning 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 Convolution Neural Network model's.

This underscores the importance of incorporating both product metrics 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.

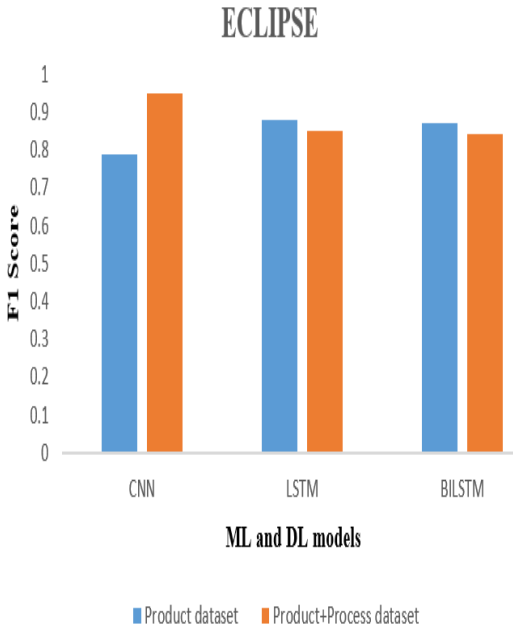

FIGURE 4.1: Eclipse Product and Combined Metrics Results

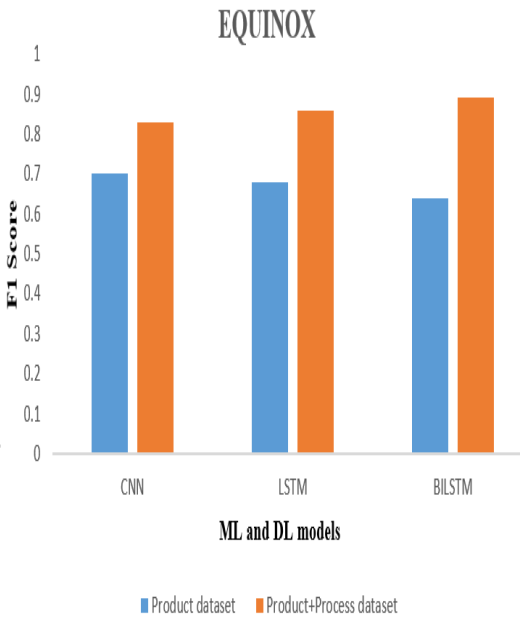

FIGURE 4.2: Equinox Product and Combined Metrics Results

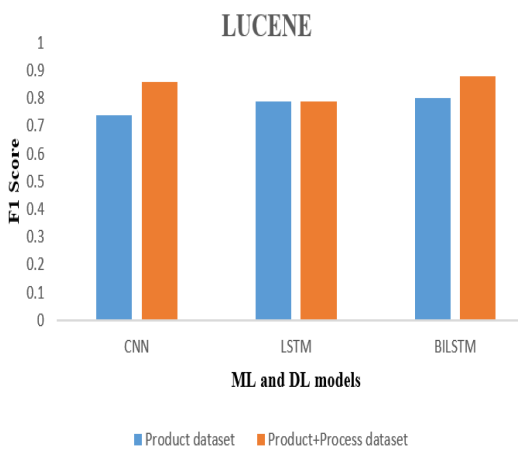

FIGURE 4.3: Lucene Product and Combined Metrics Results

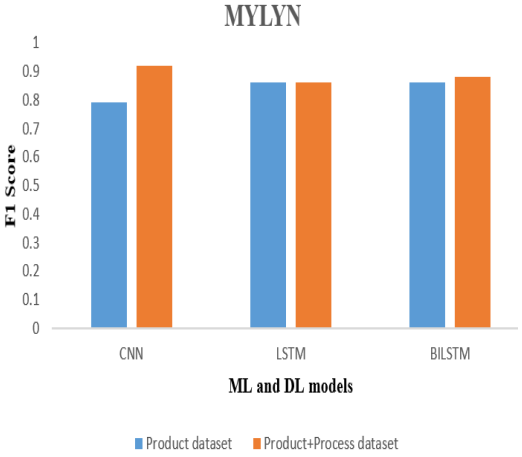

FIGURE 4.4: Mylyn Product and Combined Metrics Results

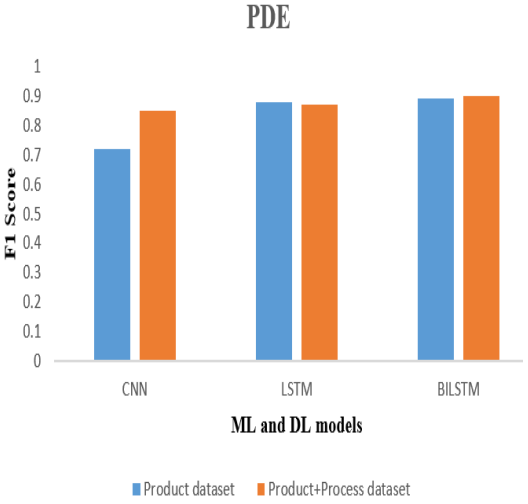

FIGURE 4.5: Pde Product and Combined Metrics Results

4.6.2 Analysis of Results for ML and DL on Combined Metrics

Figure 4.6 provides a comparative analysis of the F1 scores achieved by various ML and DL models across five different datasets. The models evaluated include KNN, NB, LR, CNN, LSTM, and BILSTM. The results indicate that Logistic Regression (LR), CNN, LSTM, and BILSTM generally exhibit higher and more consistent performance across different datasets compared to KNN and NB. Specifically, BILSTM and CNN tend to outperform other models, showing slightly better results in most datasets.

The analysis reveals significant variability in model performance based on the dataset used. For instance, Naive Bayes performs poorly on the Lucene dataset but fares better on the Eclipse and PDE datasets. In general, models achieve higher F1 scores on the Eclipse and PDE datasets than on Lucene. This suggests that the characteristics of the dataset play a crucial role in the effectiveness of a model. Overall, deep learning 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 software fault prediction. This underscores the importance of selecting the appropriate model and tuning it to the specific dataset to achieve optimal results.

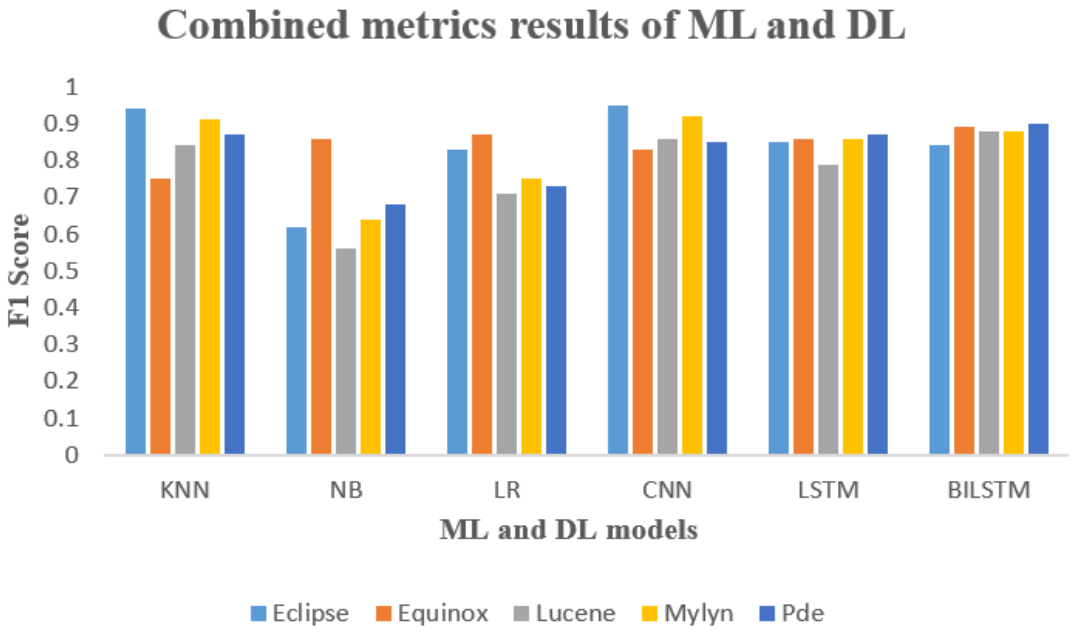

FIGURE 4.6: Combined Metrics Dataset Results on ML and DL

4.6.3 Performance Analysis of DL with Combined Metrics

In the previous section, we observed that deep learning algorithms performed better according to the combined metrics. To further analyze and illustrate these results, we have visualized them using validation curves.

A validation curve is a graphical tool that illustrates how a model's performance varies with different values of a specific hyperparameter. It graphs the validation score versus the hyperparameter values, enabling the visualization of how

adjustments in the hyperparameter impact the model's performance. This curve is useful for identifying the best hyperparameter setting that optimizes the model's validation score.

Figures 4.7, 4.8, 4.9 demonstrates consistent improvements in performance as the number of training epochs increases. For all three DL models; CNN, LSTM and BILSTM, the validation accuracy tends to rise steadily, indicating that the models are effectively learning from the data without significant overfitting up to 200 epochs. The CNN validation curve shows that the Lucene dataset achieves the highest performance, especially after 100 epochs, followed closely by PDE and the other datasets. The LSTM model's validation curves reveal similar trends, with Lucene again showing strong performance, but with a slightly tighter clustering of results across datasets. BILSTM displays a similar pattern to LSTM, with all datasets gradually improving and converging towards higher accuracy, suggesting robust learning. Across all models, Lucene and PDE datasets tend to perform better, while Equinox and Mylyn demonstrate slightly lower validation accuracy, particularly in the earlier epochs. Overall, the upward trend across these curves indicates that the combined dataset is effective in improving model performance, with the models generalizing well across different software projects as training progresses.

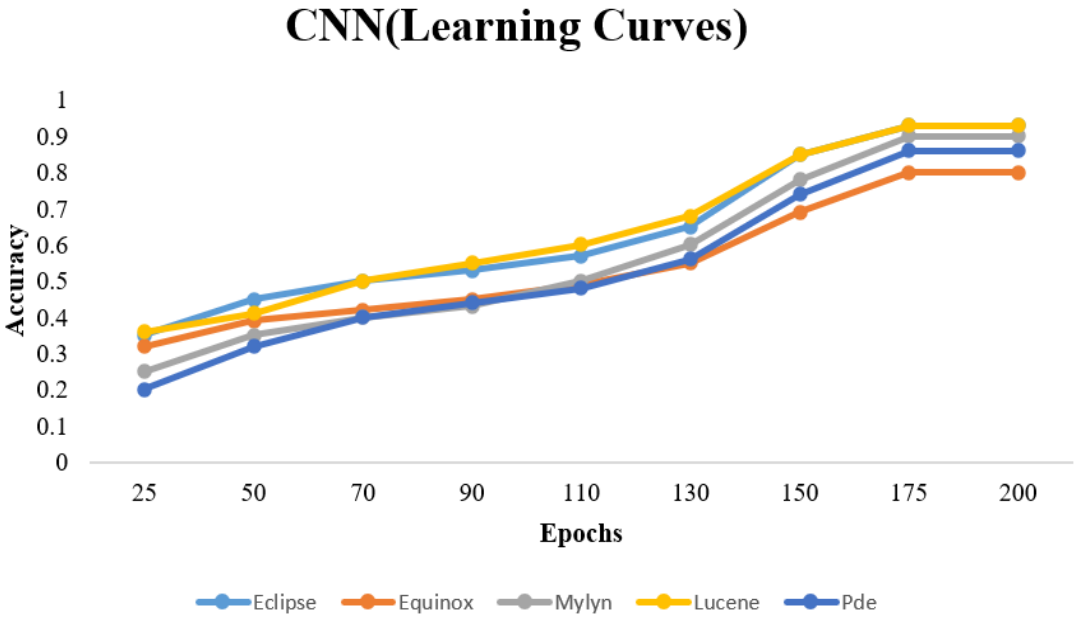

FIGURE 4.7: Learning Curve of CNN

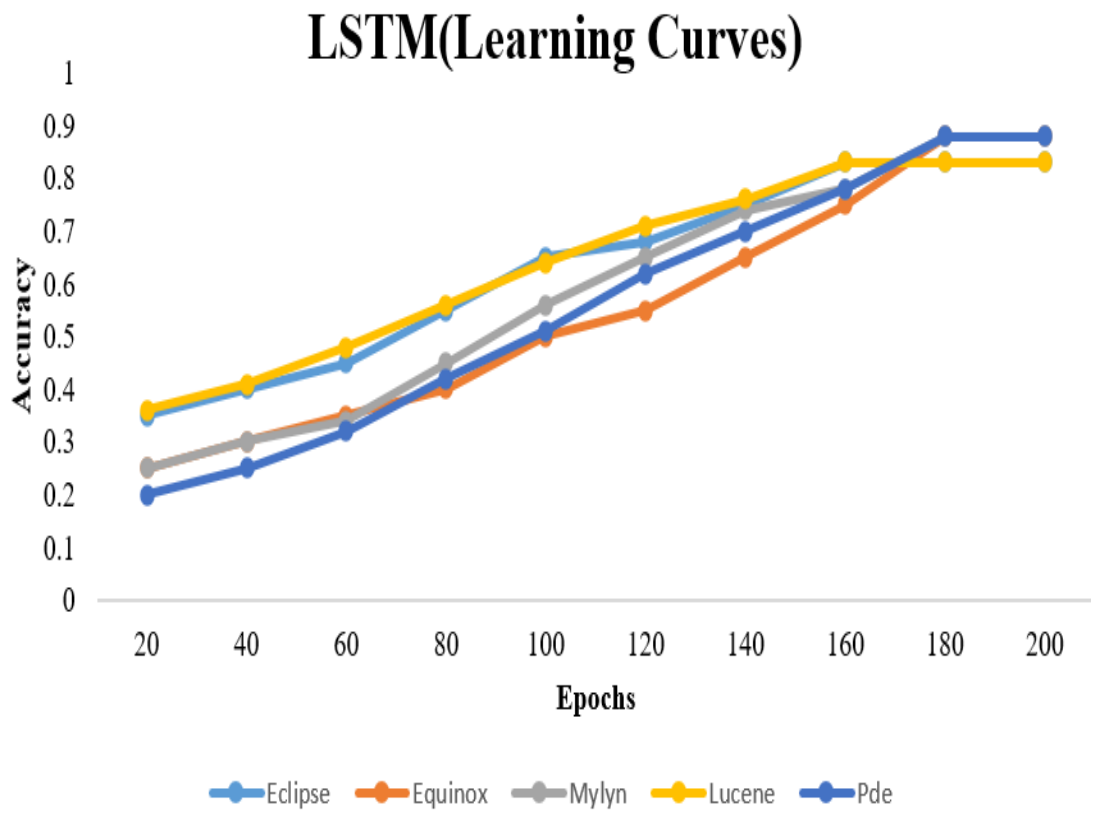

FIGURE 4.8: Learning Curve of LSTM

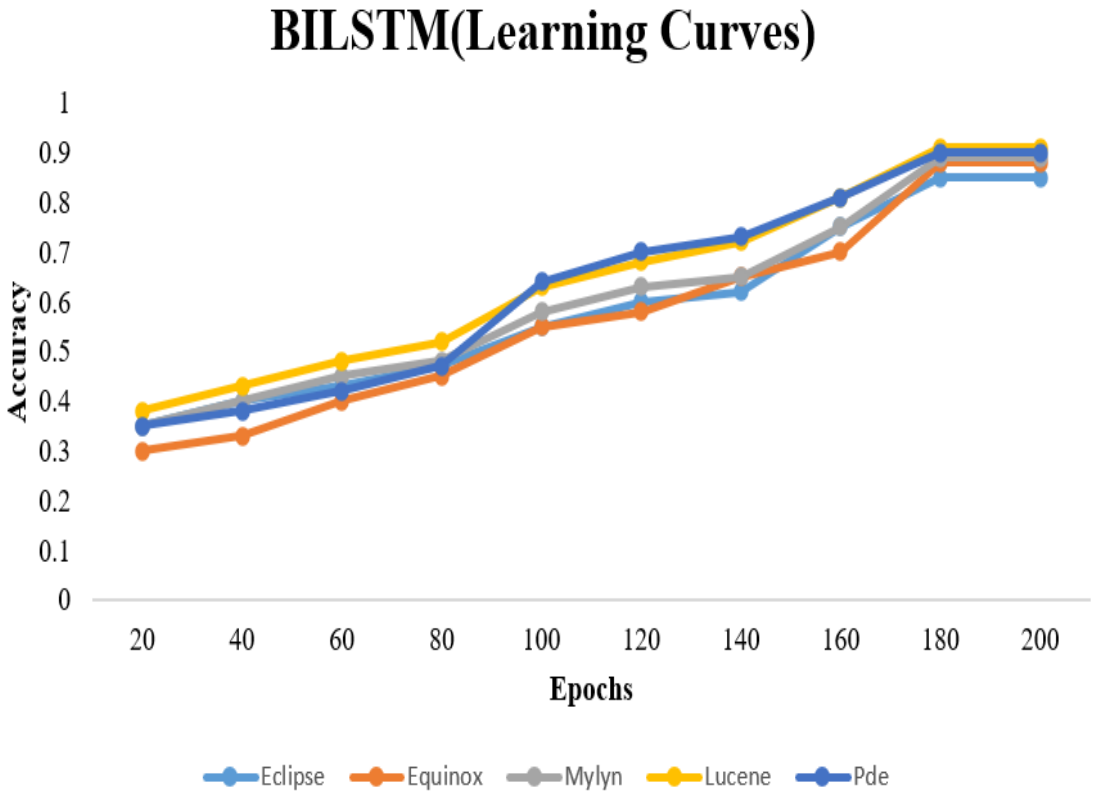

FIGURE 4.9: Learning Curve of BILSTM

4.7 Discussion

Software metrics are vital in predicting software faults, with both product and process metrics being important. Researchers have mainly focused on product metrics, also known as product metric. However, our research highlights that process metrics are significant also, we have analyzed that combining product metrics with change metrics improves performance, as we discussed earlier in section 4.4 that proposed combined metrics yield better results indicate that for most datasets, models trained on the combined metrics dataset exhibit higher accuracy compared to those trained on product metrics alone. For instance, in the Eclipse dataset, the CNN model shows a significant increase in results from 0.77 with Product Metrics to 0.93 with Combined Metrics. Similarly, in the Equinox dataset, the LSTM model's accuracy improves substantially from 0.64 with Product Metrics to 0.88 with Combined Metrics. These trends suggest that the inclusion of process metrics alongside product metrics generally enhances the predictive accuracy of the models. The BILSTM model also shows consistent improvements across all datasets when using Combined Metrics, with the PDE dataset reaching the highest accuracy of 0.90.

Although Prior research has analyzed product metrics using DL and ML, but the use of process metrics has not explored. So, we have used Deep Learning and Machine Learning to explore the combination of product and process metrics, and we found that deep learning produced better results. Our results show that when deep learning techniques are used, combined metrics perform better than product metrics alone as discussed in section 4.5 that we compared various ML and DL models on software fault prediction across five datasets: Eclipse, Equinox, Lucene, Mylyn, and PDE. The results demonstrate that deep learning models, particularly CNN and BILSTM, tend to outperform Machine Learning models across the different datasets. For example, the CNN model achieves the highest accuracy of 0.93 on both the Eclipse and Mylyn datasets, surpassing all ML models. Similarly, the BILSTM model reaches an accuracy of 0.91 on the Lucene dataset, which is higher than the accuracy obtained by the ML models. These observations indicate that DL models are generally more effective in leveraging the Combined Metrics dataset

to predict software faults with higher accuracy, highlighting their robustness and suitability for complex predictive tasks in software fault prediction. Overall, the tables underscore the importance of using a combined dataset and DL models to achieve superior performance in software fault prediction tasks.

As a result, we draw the conclusion that deep learning models works well with combined metrics. By combining process and product metrics, the advantages of each are combined to give the deep learning model access to a larger dataset for analysis. With the help of this improved dataset, the model is better able to predict software faults because it can find patterns and relationships that would not be obvious when relying just on product metrics.

Furthermore, following are the answers to our research questions that were mentioned in Chapter 1, identified after literature review and experimentation.

- **RQ1: Do combined metrics (Product + Process) provide better results than solely using product metrics when employing deep learning algorithms?**

  By evaluating product metrics and combined metrics with DL models, to determine weather product metrics performance is promising over the combined metrics (Product + Process). The results achieved that the combined metrics outperformed product metrics as shown in section 4.4 where the enhanced performance of the combined metrics underscores their effectiveness in improving model accuracy and reliability.

- **RQ2: Does deep learning produce better results on combined metrics compared to machine learning algorithms?**

  Based on our results, it has been observed that combined metrics can enhance fault prediction using Machine Learning and Deep Learning. The results indicates that Deep Learning outperformed Machine Learning, highlighting the superior capability of deep learning models in leveraging combined metrics for more accurate predictions. As a result, Deep Learning-based approaches offer greater potential for improving prediction accuracy in fault detection tasks. The detailed results are shown in section 4.5.

4.8 Threats to Validity

Our experiment's results enable us to use combined metrics(Product + Process) metrics to Software Fault Prediction. However, we would need to take into account any threats to the validity of the outcome before we could accept it.

1. We have merged the datasets to make combined metrics and evaluated the performance with deep learning and machine learning using small dataset. The results could be enhanced if we conducted experiments on a larger dataset. This is due to the fact that a larger dataset helps the model recognize patterns and relationships, which improves its capacity to predict software faults. The validity of the current dataset is threatened by its small size,though, as it might not fully capture the variability required for reliable predictions.

2. Regarding datasets project size, we selected projects of an adequate and manageable size. Projects that were either very large or very small were excluded due to the lack of available source code or fault information for such projects.

Chapter 5

Conclusion and Future Work

In Chapter 4, we described the details of proposed methodology results and discussion. This chapter summarizes our work, presents the research findings, and identifies other possibilities for further investigation into this area.

5.1 Conclusion

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 the Eclipse dataset 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, Naive Bayes, and LR) and deep learning 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 deep learning 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.

5.2 Future Work

The next stage of progress in our research involve the following future directions.

1. Firstly, exploring more advanced deep learning models or ensembles could potentially improve prediction accuracy beyond the models we studied in this research.These advanced models may better capture complex patterns in the data, leading to more accurate and reliable predictions of software faults.

2. Validating the combined metrics approach on diverse software datasets beyond Eclipse is crucial to ensure that the findings are not specific to a single dataset. This can test its robustness and generalizability across different software environments and fault types.

3. Investigating dynamic metrics for real-time fault prediction involves developing methods to continuously monitor and analyze software changes as they occur. By incorporating dynamic metrics instead of staic metrics, we can improve the timeliness and accuracy of fault predictions, enabling more proactive maintenance and faster response to emerging problems.

Bibliography

[1] S. A. Sherer, "Software fault prediction," *Journal of Systems and Software*, vol. 29, no. 2, pp. 97–105, 1995.

[2] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, and V. E. Misilov, "A survey on software defect prediction using deep learning," *Mathematics*, vol. 9, no. 11, p. 1180, 2021.

[3] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[4] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[5] J. Alzubi, A. Nayyar, and A. Kumar, "Machine learning from theory to algorithms: an overview," in *Journal of physics: conference series*, vol. 1142. IOP Publishing, 2018, p. 012012.

[6] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of big data*, vol. 2, pp. 1–21, 2015.

[7] J. Kumar Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of computer science and technology*, vol. 25, pp. 1016–1029, 2010.

[8] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[9] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, 1991, pp. 197–211.

[10] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[12] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[13] D. Sharma and P. Chandra, "Towards recent developments in the methods, metrics and datasets of software fault prediction," *International Journal of Computational Systems Engineering*, vol. 6, no. 1, pp. 14–45, 2020.

[14] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[15] D. Sharma and P. Chandra, "Towards recent developments in the methods, metrics and datasets of software fault prediction," *International Journal of Computational Systems Engineering*, vol. 6, no. 1, pp. 14–45, 2020.

[16] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. T. M. Phuong, and P. H. Thong, "Empirical study of software defect prediction: a systematic mapping," *Symmetry*, vol. 11, no. 2, p. 212, 2019.

[17] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th international conference on predictive models in software engineering*, 2010, pp. 1–10.

[18] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 2010, pp. 31–41.

[19] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 9–9.

[20] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, pp. 540–578, 2009.

[21] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[22] A. Iqbal, S. Aftab, U. Ali, Z. Nawaz, L. Sana, M. Ahmad, and A. Husen, "Performance analysis of machine learning techniques on software defect prediction using nasa datasets," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 5, 2019.

[23] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[24] F. Xing, P. Guo, and M. R. Lyu, "A novel method for early software quality prediction based on support vector machine," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE, 2005, pp. 10–pp.

[25] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

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

[28] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[29] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[30] C. Catal, "Software fault prediction: A literature review and current trends," *Expert systems with applications*, vol. 38, no. 4, pp. 4626–4636, 2011.

[31] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2011.

[32] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[33] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[34] S. S. Rathore and S. Kumar, "A study on software fault prediction techniques," *Artificial Intelligence Review*, vol. 51, pp. 255–327, 2019.

[35] M. Caulo, "A taxonomy of metrics for software fault prediction," in *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 1144–1147.

[36] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "Machine learning based methods for software fault prediction: A survey," *Expert Systems with Applications*, vol. 172, p. 114595, 2021.

[37] M. A. Khan, N. S. Elmitwally, S. Abbas, S. Aftab, M. Ahmad, M. Fayaz, and F. Khan, "Software defect prediction using artificial neural networks: A systematic literature review," *Scientific Programming*, vol. 2022, no. 1, p. 2117339, 2022.

[38] S. Pandey and K. Kumar, "Software fault prediction for imbalanced data: a survey on recent developments," *Procedia Computer Science*, vol. 218, pp. 1815–1824, 2023.

[39] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. T. M. Phuong, and P. H. Thong, "Empirical study of software defect prediction: a systematic mapping," *Symmetry*, vol. 11, no. 2, p. 212, 2019.

[40] G. Abaei and A. Selamat, "A survey on software fault detection based on different prediction approaches," *Vietnam Journal of Computer Science*, vol. 1, pp. 79–95, 2014.

[41] S. R. Aziz, T. A. Khan, and A. Nadeem, "Efficacy of inheritance aspect in software fault prediction—a survey paper," *IEEE Access*, vol. 8, pp. 170 548–170 567, 2020.

[42] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[43] I. Batool and T. A. Khan, "Software fault prediction using deep learning techniques," *Software Quality Journal*, vol. 31, no. 4, pp. 1241–1280, 2023.

[44] M. Nevendra and P. Singh, "Software defect prediction using deep learning," *Acta Polytechnica Hungarica*, vol. 18, no. 10, pp. 173–189, 2021.

[45] S. K. Pandey, A. Haldar, and A. K. Tripathi, "Is deep learning good enough for software defect prediction?" *Innovations in Systems and Software Engineering*, pp. 1–16, 2023.

[46] K. Nehéz and N. A. A. Khleel, "A new approach to software defect prediction based on convolutional neural network and bidirectional long short-term memory," *Production Systems and Information Engineering*, vol. 10, no. 3, pp. 1–18, 2022.

[47] P. Tadapaneni, N. C. Nadella, M. Divyanjali, and Y. Sangeetha, "Software defect prediction based on machine learning and deep learning," in *2022 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, 2022, pp. 116–122.

[48] A. Ho, N. Nhat Hai, and B. Thi-Mai-Anh, "Combining deep learning and kernel pca for software defect prediction," in *Proceedings of the 11th International Symposium on Information and Communication Technology*, 2022, pp. 360–367.

[49] K. Sekaran and L. S. P. Annabel, "A deep learning based model for defect prediction in intra-project software," in *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, 2023, pp. 1148–1155.

[50] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Computers & Electrical Engineering*, vol. 67, pp. 15–24, 2018.

[51] L. Šikić, P. Afrić, A. S. Kurdija, and M. ŠIlić, "Improving software defect prediction by aggregated change metrics," *IEEE access*, vol. 9, pp. 19 391–19 411, 2021.

[52] Y. A. Alshehri, K. Goseva-Popstojanova, D. G. Dzielski, and T. Devine, "Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–7.

[53] Q. Yu, S. Jiang, J. Qian, L. Bo, L. Jiang, and G. Zhang, "Process metrics for software defect prediction in object-oriented programs," *IET Software*, vol. 14, no. 3, pp. 283–292, 2020.

[54] L. Jiang, S. Jiang, L. Gong, Y. Dong, and Q. Yu, "Which process metrics are significantly important to change of defects in evolving projects: an empirical study," *IEEE Access*, vol. 8, pp. 93 705–93 722, 2020.

[55] F. Lomio, S. Moreschini, and V. Lenarduzzi, "A machine and deep learning analysis among sonarqube rules, product, and process metrics for fault prediction," *Empirical Software Engineering*, vol. 27, no. 7, p. 189, 2022.

[56] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 432–441.

[57] O. Al Qasem, M. Akour, and M. Alenezi, "The influence of deep learning algorithms factors in software fault prediction," *IEEE Access*, vol. 8, pp. 63 945–63 960, 2020.

[58] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 2010, pp. 31–41.

[59] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, "Investigation on the stability of smote-based oversampling techniques in software defect prediction," *Information and Software Technology*, vol. 139, p. 106662, 2021.

[60] R. Jindal, R. Malhotra, and A. Jain, "Analysis of software project reports for defect prediction using knn," in *Proceedings of the World Congress on Engineering*, vol. 1, 2014.

[61] G. Kaur, J. Pruthi, and P. Gandhi, "Machine learning based software fault prediction models," *Karbala International Journal of Modern Science*, vol. 9, no. 2, p. 9, 2023.

[62] Ö. F. Arar and K. Ayan, "A feature dependent naive bayes approach and its application to the software defect prediction problem," *Applied Soft Computing*, vol. 59, pp. 197–209, 2017.