Asian Journal of Computer Science and Technology
ISSN: 2249-0701 (P); 2583-7907 (E)
Vol.12 No.2, 2023, pp.48-65

DOI: https://doi.org/10.51983/ajcst-2023.12.2.3763

A Comprehensive Hybrid Model for Language-Independent Defect Prediction in Microservices Architecture

Yashwant Kumar[1] and Vinay Singh[2]
[1]Joint Director (IT, NIC), [2]Associate Professor,
[1&2]Department of Computing and Information Technology, Usha Martin University, Jharkhand, India
E-mail: yashwant.k@nic.in, vinay.singh@ushamartinacademy.org
(Received 15 September 2023; Revised 1 November 2023; Accepted 17 November 2023; Available online 23 November 2023)

***Abstract* - The transformation of software development from monolithic frameworks to microservices-based architectures, focusing on the challenges of creating a unified defect prediction model that spans various programming languages in practice of automating integration of code modification into a single codebase. It proposes a hybrid machine learning approach to enhance defect prediction accuracy by integrating different data sources and algorithms. The goal is to create a language and project-independent model. The hybrid model combines Bi-Directional LSTM (BiD-LSTM) networks and Attention mechanisms, static code metrics, and BERT-based language models. BiLSTM-Attention captures temporal dependencies within Abstract Syntax Trees (ASTs), static code metrics provide insights into software complexity, and BERT interprets textual context for a holistic understanding of code snippets. The research methodology involves quantitative techniques, starting with a literature review to establish the theoretical foundation. An empirical study follows, encompassing data gathering, feature crafting and pre-processing, model building, training and evaluation, validation and analysis and conclusions. The research's insights aim to improve defect prediction techniques, contributing to software engineering's pursuit of better quality and reliability.**
Keywords: **Software Defect Prediction, Machine Learning Approach, Predictive Accuracy, Hybrid Model, BiD-LSTM, BERT, ASTs**

I. INTRODUCTION

Software defect prediction model is an artefact of application of statistical and machine learning techniques. Defect prediction is an essential task during software development life cycle. Early detection of faults saves money and time for the companies. It is aimed at identifying the code which has a potential bug that helps to correct it during the development process itself.

The defect prediction model helps the developer and project managers to find the likelihood of defects in specific modules, codes, classes, components, processes or files. By early detection of defects, resource allocation can be efficiently managed. It also helps in prioritising test cases and ensures the software's quality. The implementation of best AI model in CICD pipeline is must for identifying the early bugs before successful deployment of any modules.

A. Background and Motivation

The background and motivation for creating a good model by utilizing Machine Learning methods for anticipating software flaws is to augment software development processes, reduce costs, enhance software quality, and deliver more reliable and robust software products [1]. Let's explore the key reasons behind this initiative.

1. Quality Improvement: Defect anticipating models are designed to spot possible defects or errors in software code at an early stage of the development process. By anticipating defects prior to their deployment and integration into the larger system, developers can take pre-emptive measures to address them, resulting in improved code quality and a reduced chance of defects making their way into the production environment [2].

2. Cost Reduction: Fixing defects in software can be costly, especially if they are detected late in the development lifecycle or, worse, after the software has been deployed to production. Defect prediction models help catch issues early, saving both time and resources required for bug fixing and maintenance.

3. Enhancing Developer Productivity: When developers have insights into potential problem areas in the code, they can focus their efforts on critical sections, optimize their work, and prioritize bug-fixing tasks more efficiently.

4. Risk Mitigation: Predicting defects can help project managers and stakeholders assess and manage project risks. It allows for better planning and resource allocation to address potential quality issues.

5. Continuous Integration and Delivery (CICD) Pipeline Improvement: Defect prediction models can be integrated into the CI/CD pipeline, allowing for automated checks during the build and deployment processes. This integration ensures that the software meets predefined quality criteria before being released. That aids in DevOps practice.

6. Software Security Enhancement: Certain defects can lead to security vulnerabilities, making the software susceptible to attacks. Defect prediction models can help identify such vulnerabilities early on and support the development of more secure software.

7. Data-Driven Decision Making: Machine Learning models, when trained on historical software data, can identify patterns and trends that human developers might miss. These models provide an additional data-driven perspective to aid in decision-making.

8. Benchmarking and Comparative Analysis: Defect prediction models can be used to benchmark different projects or teams based on their defect-proneness. This enables organizations to compare the quality of various projects and identify areas for improvement.

9. Research and Innovation: The creation of good defect prediction models using Machine Learning also fuels exploration and advancement in the realm of software engineering and data science. Researchers continuously explore new approaches, algorithms, and data sources to enhance the effectiveness and accuracy of the models [3].

Machine learning (ML) has risen as a potential tool across diverse domains, including software engineering, due to its ability to analyse large volumes of data, understand patterns, and do predictions [4]. In last few years, ML techniques had being applied to defect guessing with promising results. By leveraging historical data on software defects and associated code attributes, machine learning models can learn patterns and make predictions about the likelihood of defects in new or modified code. By leveraging ML techniques to build reliable and accurate models to anticipate software issues, software development teams can significantly enhance their development processes, reduce the occurrence of defects, improve software quality, and ultimately deliver better software products to users.

The scope of creating a source code-based model for predicting software defects and other static code metrics is vast and holds significant potential for improving software quality and development practices [5]. The NLP approach aims to leverage the natural language patterns present in source code, comments, and documentation to enhance defect prediction accuracy. The model aims to extract meaningful information from this unstructured text and use it as input for defect prediction tasks.

B. Problem Definition and Objective

The problem at hand is the shift in software development from single-language frameworks to multi-language microservices architecture. Microservices involve breaking down complex applications into smaller, independent services, each serving a specific business function and communicating through Web APIs. These services can be written in different programming languages like Java, PHP, Python, C# and JavaScript. The challenge lies in developing a unified software defect predictor model that can be used across these different programming languages within a CICD pipeline.

The objective was to create a proposed Hybrid model that utilizes Abstract Syntax Tree (AST), static code metric and source code from various cross-project and cross-version datasets as input features. This model was tested and demonstrated superior performance compared to alternatives.

C. Key Aspects of the Scope

1. Data Collection and Preparation: Collecting labelled datasets of source code and defects for training and evaluating the NLP model. Ensuring the data is representative of different software projects and domains.

2. Textual Data Analysis: The model will focus on processing and analysing textual data from various sources, such as source code, comments and documentation.

3. Feature Extraction: NLP techniques is applied to extract meaningful pattern from the textual data. This may include keyword extraction, sentiment analysis, topic modelling, and semantic analysis. [6]. The technique will also be applied to extract relevant features from source code, such as code tokens, API usage patterns, variable names, and comment sentiments, to be used as inputs to the NLP model.

4. Model Development: The model DL algorithms to spot the likelihood of issues based on the extracted features from the textual data, Designing and training the NLP model, which could also involve approaches like recurrent neural networks (RNNs), LSTM, transformer-based models like BERT.

5. Code Representation: Exploring methods to represent source code and other static code metrics in a format suitable for NLP models, such as embedding or tokenization techniques.

6. Model Evaluation: Performing thorough assessments to measure the effectiveness of the NLP-driven defect prediction model, utilizing metrics such as F1-score, precision, accuracy, recall and the AUC-ROC.

7. Generalization: The NLP-based model would be designed to integrate seamlessly with existing defect prediction systems or software development workflows. Assessing the generalization capabilities of the model by testing it on unseen datasets from different projects to determine its applicability beyond a specific context.

D. Proposed Model at a Glance

The proposed model is a Multi-Input Hybrid Model designed for accurate defect prediction in modern software engineering. It combines Source Code Metrics, Abstract Syntax Tree (AST) Tokens, and Bidirectional Encoder

Representations from Transformers (BERT) to enhance prediction accuracy. This model aims to depicts a detailed examination of its structure and components, with the goal of surpassing traditional defect prediction techniques by leveraging the contextual understanding capabilities of BERT for precision and robustness in defect prediction.

The upcoming sections will delve into the related work followed by detailed examination of proposed model's each component, explaining their importance and how they contribute to the model's main goal. The integration process will be described step by step, illustrating how these different elements work together to create a comprehensive and unified framework. Additionally, the article will discuss the potential advantages and real-world implications of the Multi-Input Hybrid Model, emphasizing its ability to significantly improve defect prediction accuracy.

In essence, the journey embarked upon in this article lays the groundwork for a paradigm shift in the realm of defect detection. By embracing the fusion of diverse data sources and cutting-edge technologies, the proposed model seeks to reshape the landscape of software quality assurance, offering an innovative and powerful tool for identifying defects and enhancing software reliability.

II. REVIEW OF LITERATURE

To improve defect prediction efficiency, researchers and practitioners continue to explore and develop more advanced and automated techniques, such as machine learning-based models, data mining, and AI-driven methods that can handle large-scale codebases and provide more accurate predictions with reduced human intervention as mentioned by Mendez, Padala and Burnett et al., [7] [8].

A. Some ML Algorithms

1. Statistical Techniques (Regression Models): Menzies; Butcher; Cok and Marcus et al. explained Regression models that are frequently employed in defect prediction within software development. These models, includes logistic regression, multiple linear regression and Poisson regression were examined in connection to software metrics and the presence of defects. Common predictors used in these models encompass code complexity, code churn, code size, and developer experience [9].

2. Decision Trees: Ibarguren, Perez, Mugerza, Rodriguezx, Harrison et al. used Decision Trees (DT) are hierarchical models that iteratively divide the data using the most influential predictors. They are commonly used in defect prediction due to their simplicity and interpretability. Decision trees exhibit versatility in software defect prediction tasks since they can accommodate both categorical and numerical data [10] [11].

3. Naive Bayes Classifier: Ahmet Okutan and Olcay Taner Yıldız explained Naive Bayes classifier which assumes feature independence for probabilistic classification. It has been applied to software defect prediction, where the features represent software metrics and the class labels represent defective or non-defective modules [12].

4. Neural Networks: Giray, Kwabena, Köksal, Babur, Tekinerdogan referred Neural networks and used in software defect prediction, addressing intricate metric-defect relationships and large dataset learning. Techniques like feedforward neural networks or recurrent neural networks have been explored for this purpose [3].

5. Genetic Algorithms: Nalini and Krishna used Genetic algorithms to optimize defect prediction models by seeking ideal feature combinations and model parameters for performance maximization [13]. These software defect prediction methods have undergone thorough extensively studied and have demonstrated potential benefits in numerous empirical investigations. The effectiveness of these methods may vary depending on factors such as project context, data quality, and specific defect prediction tasks across projects and languages.

B. Review of Deep Learning Related Studies on SDP

Zheng, Gao, Fengyu, Xun, Liu, Xiang Chen study assesses different deep learning models, including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, for software defect prediction. The authors analyse these models' real-world dataset performance and offer insights into their respective pros and cons [14].

Lang, Li and Kobayashi introduce a streamlined Convolutional Neural Network (CNN) for software defect prediction and conduct an empirical study across various software projects. They compare their CNN model to conventional machine learning methods, showcasing its effectiveness in defect prediction tasks [15].

The paper by Bahaweres, Jumral and Hermadi *et al.,* introduces a hybrid strategy, fusing Long Short-Term Memory (LSTM) neural networks with conventional machine learning techniques for software defect prediction. It demonstrates that this hybrid model can enhance prediction accuracy, surpassing individual methods [16].

Dada, Oyewola, Joseph and Dauda suggest an ensemble method that merges several machine learning algorithms like Random Forest, SVM, and Gradient Boosting for open-source software defect prediction. They assess the ensemble's performance across diverse datasets and analyse its effectiveness [17].

Giray *et al.,* in their work, researchers present a cost-sensitive Convolutional Neural Network (CNN) tailored for software defect prediction. They tackle the challenge of imbalanced data in defect prediction and demonstrate that the cost-sensitive CNN enhances predictive performance on imbalanced datasets [3].

Laradji, Alshayeb and Ghouti questioned the standalone effectiveness of various algorithms in software defect prediction, including Decision Trees, Artificial Neural Networks and Bayesian methods, and they noted that these methods performed sub-optimally with skewed and redundant defect datasets and deteriorated further when datasets contained incomplete or irrelevant features. Support Vector Machines (SVMs) often exhibit a bias towards the majority class, leading to high false negative rates due to the neglect of the minority class. To mitigate these issues, ensemble learning models are recommended as effective solutions [18].

Lei Qiao, Xuesong Li, Qasim Umer and Ping Guo discussed various Machine Learning Methods of Defect Prediction Techniques and its shortcomings. The current performance metrics, such as mean square error and squared correlation coefficient, require notable enhancements. This paper presents a deep learning-driven defect prediction model using regression to estimate defect counts in a module. The proposed method is evaluated for effectiveness in comparison to three state-of-the-art approaches Support Vector, Decision Tree Regression and FSVR [19].

Siers and Islam proposed technique was empirically assessed using six classifier algorithms on six widely used clean software defect prediction datasets. This cost-sensitive learning model was seen as a way to save money for software development groups, emphasizing the importance of avoiding false negative predictions, even if it means having some false positives. The study also explored a method for integrating oversampling into the Cost Sensitive Forest [20].

Cong and Shu-Wei applies a hybrid of Artificial Neural Network (ANN) and Quantum Particle Swarm Optimization (QPSO) for software fault-proneness prediction. QPSO reduces dimensionality, while ANN classifies software modules as fault-prone or not. The combination of ANN and QPSO results in an effective software fault-proneness prediction approach [21].

He, Li, Liu *et al.,* tallied metric occurrences in each prediction model and identified the top-k representative metrics as a universal feature subset for defect prediction across projects, making it suitable for projects lacking sufficient historical data. With more extensive training, the top-k feature subset becomes increasingly versatile [22].

To mitigate the unavailability of local historical data, Chen, Fang, Shang and Tang have embraced Cross-company defect prediction (CCDP). The approach involves preprocessing training data from external sources, re-weighting using a transfer method with data gravitation, and combining it with a small portion of local data to build prediction models [23].

Today's software industry often involves complex, configurable software, with a growing number of features leading to a vast configuration space. To understand the impact of various configurations on system performance, predictive models are employed. Shailesh, Nayak and Prasad in their paper suggested using a Neural Network model along with statistical techniques to predict system performance based on input configurations [24].

The study by Malhotra, Bahl, Sehgal and Priya in their paper compare 14 Machine Learning techniques for defect prediction using 9 open-source datasets in Weka and statistically analyses the models with SPSS. Results indicate Single Layer Perceptron as the most effective technique [25].

The paper by Tran, Hanh and Binh explores a method that combines feature selection and ensemble learning to tackle feature redundancy and class imbalance in software fault prediction. Additionally, it employs a deep learning model to enhance fault prediction model performance. The approach is tested on 12 NASA datasets [26].

The growing complexity of IoT applications poses a challenge for fault prediction in human-device interactions. Souri, Mohammed and Potrus introduce a hybrid fault prediction model using MLP and PSO, verified through behavioural modelling [27].

Elahi, Kanwal and Asif examined various ensemble methods to enhance prediction model performance and benchmarked model averaging using methods like voting and stacking [28].

Ge *et al.,* and Xu *et al.,* presents performance of SVM and RF in defect prediction was assessed, emphasizing RF's effectiveness in managing noisy data. Walunj *et al.,* Liang, Yu, Jiang, and Xie Presented an innovative deep learning method employing Long Short-Term Memory (LSTM) networks for defect prediction based on time series data [29] [30] [31] [32].

Alsolai and Roper examined how different feature selection techniques, such as Relief-F and Chi-Square, influence model accuracy in defect prediction [34]. The proposal suggests a hybrid method that combines static code metrics and process metrics to enhance the effectiveness of feature discrimination [33].

Pardalos, Rasskazova, and Vrahatis introduced LIME, a model-agnostic method for explaining predictions made by black-box machine learning models in the realm of defect prediction [35]. Mori and Uchihira created a rule-based classifier to offer clear and understandable defect predictions [36].

C. Research Gap

The field of software defect prediction through machine learning has seen significant research, encompassing diverse methods. Traditional machine learning algorithms and advanced models like CNNs and LSTMs have been explored.

Ensemble and cost-sensitive techniques address specific challenges. With the field's evolution, we can anticipate the

exploration of more advanced methods and larger datasets to enhance software defect prediction models' accuracy and applicability.

After conducting an extensive literature review, we have identified multiple research gaps and potential avenues for future exploration in the application of machine learning for software defect prediction.

1. Cross-Domain Transferability of Models: While the existing studies demonstrate promising results within specific domains, there is a lack of research on the transferability of defect prediction models across different industries and project types. Future research should focus on evaluating the effectiveness of models trained on one domain to predict defects in unrelated domains.

2. Real-Time Defect Prediction: Most studies focus on predicting defects on historical data, but there is a growing need for real-time defect prediction during software development. Investigating the feasibility and performance of machine learning models in a real-time setting can significantly impact software quality assurance practices.

3. Addressing Longitudinal Aspects: To better understand the evolution of software projects, it is essential to explore longitudinal defect prediction, considering how models adapt to changing project dynamics over time.

4. Integrating Traditional and Machine Learning Approaches: While machine learning techniques show promise, combining them with traditional defect prediction methods, such as code reviews and inspections, could lead to more robust and effective prediction systems.

5. Interpretability of Complex Models: With the growing popularity of deep learning models in defect prediction, it becomes crucial to create methods that provide insights into intricate model decisions, allowing stakeholders to have confidence in and comprehend the predictions. By presenting these research gaps, the article sets the stage for research to address important challenges and advancement in the field of software quality by predicting defects using proposed machine learning techniques.

III. METHODOLOGY

This section is a crucial part of the article, as it outlines the overall strategy and methodology adopted to conduct the study. This section provides a detailed explanation of the steps that is followed to carry out the research, ensuring transparency and replicability. Below is an elaboration of the contents for this section.

A. Data Collection and Preprocessing

Data gathering and pre-processing involved, collection of dataset that includes source code files, static code metrics, and corresponding labels indicating whether each module is defective or not (binary classification task). The dataset had a sufficient number of samples to build an effective predictive model. The dataset used for training and evaluation in the Cross-Language Software Defect Predictor project was collected from various open-source repositories on platforms like GitHub, PROMISE-backup-master, Bitbucket, or GitLab. These repositories contain code written in different programming languages, including but not limited to Java, JavaScript, C++, Python, and C-Sharp. The dataset was curated to include projects with labeled information about defective and defect-free code examples.

B. Data Preprocessing

The raw dataset obtained from the repositories underwent several pre-processing steps to prepare it for training and evaluation. The purpose of processing is to construct a multi-input neural network (Hybrid Model) that combines AST embeddings, static code metrics, and BERT embeddings for Source code with Bidirectional LSTM (BiLSTM) and Attention, The Keras, and Transformer's functional API were used. This architecture allowed to handle multiple inputs and build complex models with ease. The pre-processing steps include,

1. Language Identification: Since the repositories contain code from multiple programming languages, an initial language identification step was performed to categorize each file into its respective programming language using language-specific heuristics.

2. Lexical Analysis: After language identification next step was lexical analysis, also known as lexing. In this step, the source code was read character by character, and sequences of characters were identified as tokens based on predefined rules and grammar files. These rules are defined using regular expressions or finite state machines. The code files were tokenized using the language-specific lexer generated by ANTLR for each supported programming language. This process involved creating the custom code in Java for converting the source code into a stream of language-specific tokens, such as keywords, identifiers, literals, and operators.

3. AST Token Generation: The token streams were then passed through the corresponding language-specific parser generated by ANTLR to construct Abstract Syntax Trees (ASTs) or parse trees representing the hierarchical structure of the code.

Just like natural or formal languages, programming languages exhibit linguistic features such as syntax, semantics, pragmatics, and grammatical rules. Custom code is employed to utilize the language recognition tool ANTLR for transforming source code into a sequence of language-neutral tokens, as demonstrated below.

TABLE I EXTRACT LANGUAGE NEUTRAL TOKENS FROM THE SOURCE CODE OF JAVA FILE

Package Declaration, annotation, import Declaration, type Declaration, class Or Interface Modifier, class Declaration, enum Declaration, interface Declaration, annotation Type Declaration, modifier, class Or Interface Modifier, variable Modifier, type Parameters, type Type, type List, class Body, type Bound, enum Constants, enum Body Declarations, interface Body, member Declaration, method Declaration, method Body, type Type Or Void, generic Method Declaration, generic Constructor Declaration, constructor Declaration, field Declaration, const Declaration etc

From the generated ASTs Tokens, various language-independent features patterns were extracted automatically. These features may have included control flow information, variable usage patterns, function call patterns, and many more. The aim was to create a uniform set of tokens that could be used for defect prediction across different programming languages. Further these AST tokens have been converted into language-independent AST embeddings using deep learning.

4. Static Code Feature Selection: When developing models for software defect, it's essential to carefully select relevant and well-established metrics based on empirical evidence and prior research [37] [38]. The choice of metrics for predicting bugs in a dataset depends on various features, including the nature of the data, the characteristics of the software project, and the machine learning or statistical techniques being used. It's recommended to perform feature selection and experimentation to identify the most relevant metrics for predicting bugs in dataset. Machine learning techniques such as decision trees, logistic regression, random forests, and SVMs were often tried for this purpose.

However, some commonly used metrics [39] that have been used here and shown predictive power for bug prediction in Modules are listed below.

TABLE II LIST OF ADOPTED SOURCE CODE STATIC METRICS

Metric Abbreviation	Full Form	Descriptions with Rationale for Selection
WMC	Weighted Methods per Class	A measure of the complexity of the class, calculated as the sum of complexity weights of all methods in the class. (*High complexity classes may be more error-prone.*)
DIT	Depth of Inheritance Tree	The number of levels in the class's inheritance hierarchy. (*Deep inheritance hierarchies might increase complexity and lead to bugs.*)
NOC	Number of Children	The number of classes that inherit directly from this class. (*Classes with many direct subclasses may inherit defects.*)
CBO	Coupling Between Objects	The number of other classes to which this class is coupled. (*High coupling between classes might indicate potential bug propagation.*)
RFC	Response for a Class	The number of methods in the class that can be invoked in response to a message. (*Classes with high RFC might be more error-prone.*)
LCOM	Lack of Cohesion in Methods	A measure of the cohesion among methods in the class. (*Low cohesion could lead to more bugs.*)
CA	Afferent Couplings	The number of other classes that depend on this class. (*High afferent couplings may suggest higher potential for defects.*)
CE	Efferent Couplings	The number of other classes that this class depends on. (*High efferent couplings might indicate potential bug propagation.*)
NPM	Number of Public Methods	The number of public methods in the class. (*The number of public methods may influence defect-proneness.*)
LOC	Lines of Code	The total number of lines of code in the class. (*Larger classes may have more defects.*)
DAM	Data Access Metric	A measure of data access complexity in the class. (*High DAM may indicate more complex data access and potential bugs.*)
MOA	Measure of Aggregation	The number of data members in the class. (*High MOA might indicate more complex classes and higher defect-proneness.*)
MFA	Measure of Functional Abstraction	A measure of functional abstraction based on the methods in the class. (*High MFA might indicate more complex classes and higher defect-proneness.*)
CAM	Cohesion Among Methods of Class	A measure of cohesion among methods. (*High CAM might indicate more cohesive classes and lower defect-proneness.*)
IC	Inheritance Coupling	The number of parent classes that the class inherits from. (*High IC may suggest potential bug propagation.*)
CBM	Coupling Between Methods	The number of method calls to other methods within the class. (*High CBM might indicate higher inter-method dependencies and potential bug propagation.*)
AMC	Average Method Complexity	The average complexity of methods in the class. (*High AMC might indicate more complex methods and potential bugs.*)
MAX_CC	Maximum McCabe Cyclomatic Complexity	The highest complexity value among methods in the class. (*Classes with high Cyclomatic complexity might be more error-prone.*)
AVG_CC	Average McCabe Cyclomatic Complexity	The average complexity value of methods in the class. (*High average Cyclomatic complexity might indicate higher defect-proneness.*)

The "Number of Children (NOC)" metric holds significance for bug prediction and software maintenance, as it indicates the classes directly inheriting from a specific class, potentially leading to defects due to complexities and dependencies. LCOM measures cohesion within a class by counting method pairs not sharing instance variables, with higher values suggesting lower cohesion and a potential for defects. LCOM3 is a less commonly used variation. The study created a dataset with AST embeddings, static code metrics, and tokenized source code as inputs and bug labels as outputs. The hybrid model processes AST embeddings and static code metrics separately, combines BERT embeddings, and uses dense layers for defect prediction. The final sample representation of data set looked like below.

TABLE III SNAPSHOT REPRESENTATION OF DATASET

Package Identifier	Static Code Metrics	AST Tokens	Source Code Tokens	bug
org.apache.tools.ant.AntClassLoader	[{'wmc': 49, 'dit': 2, 'noc': 1, 'cbo': 24, 'rfc': 126, 'lcom': 926, 'ca': 18, 'ce': 8, 'npm': 31, 'lcom3': 0.883333333, 'loc': 1512, 'dam': 0.7, 'moa': 1, 'mfa': 0.441558442, 'cam': 0.163461538, 'ic': 1, 'cbm': 5, 'amc': 29.44897959, 'max_cc': 12, 'avg_cc': 1.9796}	compilationUnit,packageDeclaration, qualifiedNa...	package org.apache.tools.ant; \nimport java.io....	1
….	….	….	….	….

5. Data Cleaning: In data cleaning process duplicate and irrelevant samples were removed. special symbols in text data of AST and source code tokens were removed and sanitized. Applied text preprocessing techniques like lowercasing, removing punctuation, and handling special characters. For numerical static code metrics, imputed missing values using means or medians. For textual data like source codes and AST tokens, used padding or masking.

6. BERT Representations for Source Code: BERT, a transformer-based language model, excels in understanding contextual information in source code, effectively capturing semantic relationships and contextual meaning, enhancing its value in defect prediction. Fine-tuned a pre-trained BERT model on defect prediction task. Input the source code sequences as text into BERT, and then take the [CLS] token's output as a high-level representation of the code. Now the [CLS] token output can be used with the AST token embeddings and static code metrics to form a multi input representation.

The text tokens are converted into numerical representations. This was done using transformer-based embeddings (BERT). Also converted AST tokens into numerical representations. This was done using word embeddings. System represented the tokens as sequence of words of fixed vocabulary size and split it. Then it converted the text to sequence using Tokenizer. Finally, these tokens are converted into two dimensional Vectors. This Vectors are further processed in Embedding layer to converted into dense vector.

7. Feature Scaling: Normalized and standardized the static code metrics to ensure that they are on similar scales. This helps the model converge faster during training and prevents certain features from dominating others.

8. Label Encoding: Converted the bug labels into numerical format if they were not already. For binary labels, this could be mapping 'defective' to 1 and 'non-defective' to 0.

9. Data Integration: Depending on the structure of a multi-input deep learning model, it becomes necessary to combine various data components. This might entail establishing distinct input branches for static code metrics, source code, and AST tokens.

10. Data Splitting: Partitioned the dataset into training, validation, and test subsets, typically following a 70-75% training, 10-15% test, and 10-15% evaluation split. The data was shuffled to ensure impartiality during training.

```
BertForSequenceClassification(
 (bert): BertModel(
  (embeddings): BertEmbeddings(
   (word_embeddings): Embedding(30522, 768, padding_idx=0)
   (position_embeddings): Embedding(512, 768)
   (token_type_embeddings): Embedding(2, 768)
   (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
   (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
   (layer): ModuleList(
    (0-11): 12 x BertLayer(
     (attention): BertAttention(
      (self): BertSelfAttention(
       (query): Linear(in_features=768, out_features=768, bias=True)
       (key): Linear(in_features=768, out_features=768, bias=True)
       (value): Linear(in_features=768, out_features=768, bias=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
       (dense): Linear(in_features=768, out_features=768, bias=True)
       (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
     )
     (intermediate): BertIntermediate(
      (dense): Linear(in_features=768, out_features=3072, bias=True)
      (intermediate_act_fn): GELUActivation()
     )
     (output): BertOutput(
      (dense): Linear(in_features=3072, out_features=768, bias=True)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
     )
    )
   )
  )
  (pooler): BertPooler(
   (dense): Linear(in_features=768, out_features=768, bias=True)
   (activation): Tanh()
  )
 )
 (dropout): Dropout(p=0.1, inplace=False)
 (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

Fig. 1 Summary of Transformer BERT Model

11. Understand Class Imbalance: Class imbalance is a crucial consideration when dealing with software defect prediction or any binary classification task. Addressing class imbalance, where the representation of one class (defective code) is significantly lower than the other (non-defective code), is a crucial aspect of data preprocessing for training data.

Analysed the distribution of bug labels in training set. Calculate the ratio of defective to non-defective samples within the training dataset. If the defect class is notably smaller, a class imbalance problem is identified.

To address class imbalance, increased the count of minority class instances by duplicating samples or generating synthetic data points. Techniques like SMOTE or ADASYN were employed for this purpose.

12. Evaluation Metrics Selection: Chose appropriate evaluation metrics that consider class imbalance. Metrics like precision, recall, F1-score, and area under the ROC curve (AUC) are often more informative in imbalanced scenarios than simple accuracy.

a. Recall = $\frac{TP}{(TF+FN)}$

b. Specificity = $\frac{TN}{(TN+FN)}$

c. Precision = $\frac{TP}{(TP+FP)}$

d. 1-Specificity = $= \frac{FP}{(TN+FP)}$

e. Accuracy = $\frac{(TP+TN)}{(TP+TN+FP+FN)}$

f. F1 Score = 2 x $\frac{(Precision\ x\ Recall)}{(Precision + Recall)}$

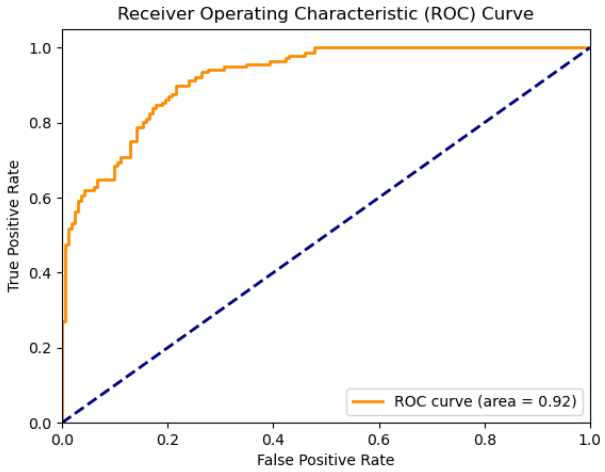


Fig. 2 AUC-ROC Curve

The ROC curve is a probability curve, and the AUC is a measure of how well a model can distinguish between defect and non-defect cases.

13. Adjust Thresholds: By default, many models predict the class with the highest probability. By modifying the classification threshold to attain a specific trade-off between precision and recall, considering the data distribution.

14. Hyper Parameter Tuning: Experiment with different hyper parameters, architectures, and techniques to find the best combination that handles class imbalance effectively.

15. Class Weights: Most deep learning frameworks allows to assign different weights to classes during training. Assigning higher weights to the minority class makes the model more sensitive to its predictions. This informs the model to assign higher importance to the minority class during training. This adjustment helps the model to focus on correctly identifying the defective instances despite their lower representation in the dataset.

Dealing with class imbalance isn't a universal fix; the optimal method depends on the unique characteristics of the training dataset. Evaluating the effect of class imbalance techniques on the model's performance is crucial, balancing the avoidance of false positives and false negatives.

C. Feature Selection and Engineering

Static code metrics, source codes, and language-independent AST tokens, a suitable model architecture for defect prediction that spans across different programming languages and software projects. could be a hybrid model that combines both the AST tokens and static code metrics while leveraging a powerful language model for understanding the textual context. One such architecture could be a combination of BERT-based sequence classification and a multi-input neural network [40]. Feature selection is the process of picking a subset of the most significant features from the initial feature set. The objective is to decrease data dimensionality while preserving or enhancing model performance.

In proposed model, there are three types of features: AST tokens, static code metrics and source codes.

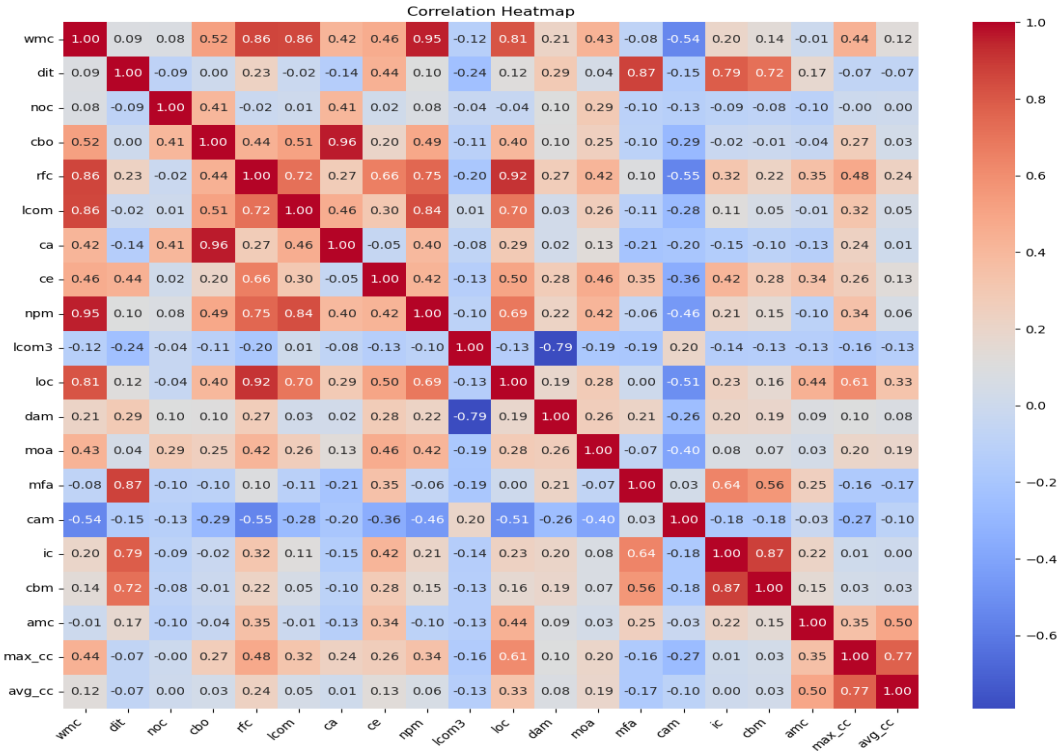


Fig. 3 Correlation heat map among Static Code Metrics

IV. SUGGESTED FRAMEWORK FOR PREDICTING SOFTWARE DEFECTS

In this study, a hybrid approach is chosen that leverages multiple sources of information for defect prediction. The selected machine learning algorithms include a combination of traditional techniques and advanced deep learning models. The chosen algorithms are as follows:

A. Bidirectional LSTM with Attention for AST Embeddings

The Bi - Directional Long short term memory (BiD-LSTM) and Attention is a deep learning architecture tailored for sequence data, such as Abstract Syntax Trees (AST) in source code. It captures intricate temporal dependencies and relationships present in code structures. AST representations are crucial for capturing structural information from source code. Bidirectional LSTM exploits both past and future contexts, while the Attention mechanism focuses on relevant parts of the code snippet.

B. Dense Layer for Static Code Metrics

A dense layer serves as a simple yet effective mechanism to process static code metrics. It aggregates numeric features to provide valuable insights into the software's complexity and maintainability. Static code metrics offer insights into software complexity, which can correlate with defect-prone areas. A dense layer effectively combines these features for further analysis.

C. BERT-based Language Model for Source Code

Bidirectional Encoder Representations from Transformers (BERT) is a powerful previously trained language model capable of capturing contextual information from textual data. Source code contains textual context that may be indicative of defects. BERT's ability to understand language context makes it suitable for extracting features from code snippets.

D. Model Architectures and Hyperparameter Tuning

The chosen algorithms are integrated into a hybrid model architecture, which combines the outputs of the Bidirectional LSTM, the dense layer for static code metrics, and the BERT-based language model. This combined approach aims to capture both structural and textual aspects of source code for enhanced defect prediction.

1. Bidirectional LSTM with Attention

a. An embedding layer converts discrete AST tokens into continuous vectors.
b. Spatial Dropout is applied to prevent overfitting by dropping entire channels of feature maps.
c. Bidirectional LSTM captures sequential patterns in the AST embeddings.
d. Attention mechanism highlights relevant parts of the code snippet.
e. Flatten layer converts the attention output into a suitable format for fusion.

2. Dense Layer for Static Code Metrics

a. A dense layer processes static code metrics, extracting high-level features.

3. BERT-based Language Model for Source Code

a. BERT tokenizer prepares code snippets for input.
b. BERT model generates contextual embeddings for the code snippets.

4. Hyperparameter Tuning Strategies

a. *Batch Size:* Tuning the batch size affects convergence and memory usage. Smaller batch sizes of 32 improved generalizations.
b. *Learning Rate:* The learning rate governs the step size during optimization. Grid search or random search has been applied to find an optimal learning rate.
c. *Dropout Rate:* Dropout is a regularization technique to mitigate overfitting. Hyperparameter search identified the dropout rate that balances overfitting and under fitting.
d. *LSTM Units:* The number of LSTM units influences model complexity. A grid search revealed the optimal number.
e. *Number of Dense Layers:* The number of dense layer units can be tuned for controlling model capacity.
f. *Embedding Dimensions:* For the embedding layer, different dimensions are explored to capture relevant features.

E. Model Summary

The Deep Learning layers of Hybrid Model Summary is given in the Table IV.

TABLE IV MODEL SUMMARY OF PROPOSED HYBRID MODEL FOR SDP

Layers	Shape	Param
InputLayer (AST Input)	1851	0
Embedding	1851 x 128	256000
SpatialDropout1D	1851x128	0
Bidirectional LSTM	1851X392	509600
Attention	1851X392	0
Input Layer (Static Metrics Input)	0	0
Flatten	725592	0
Dense	32	672
Input Layer (BERT Input)	512	0
Concatenate	726136	0
Dense	128	92945536
Dense	1	129
Total Parameters		93711937
Trainable Parameters		93711937
Non-trainable Parameters		0

The model consists of multiple input layers, each representing a different type of data: AST embeddings (ast_input`), static code metrics (static_metrics_input), and BERT embeddings (bert_input).

The model contains several layers that process and transform the input data. The Embedding layer converts AST tokens into continuous vectors with an output shape of (None, 1851, 128), indicating a sequence length of 1851 tokens and an embedding dimension of 128. The SpatialDropout1D layer applies spatial dropout to the embedded sequences, resulting in the same output shape as the embedding layer. The Bidirectional layer implements a bidirectional LSTM to capture sequential patterns in AST embeddings, resulting in an output shape of (None, 1851, 392). The Attention layer calculates attention scores over the bidirectional LSTM output, maintaining the same output shape. The Flatten layer flattens the attention output, transforming it into a vector of shape (None, 725592). The model also includes a Dense layer that processes the static code metrics, resulting in an output shape of (None, 32). All these outputs are concatenated using the `Concatenate` layer into a single feature vector of shape (None, 726136). The concatenated features are then passed through another Dense layer with 128 units, resulting in an output shape of (None, 128). Finally, the last Dense layer with a one unit and classifier sigmoid function produces the model's output with shape (None, 1), which corresponds to the binary classification of defect presence or absence.

The Total parameters count (93,711,937) represents the total number of adjustable parameters in the model. These parameters are acquired through training to enhance the model's performance for the specified task. The count of trainable parameters (93,711,937) signifies the quantity of parameters that will undergo updates during training via backpropagation.

The count of non-trainable parameters is zero that, implies that there are no fixed or pre-initialized parameters in this model. Non-trainable parameters are often associated with elements like embedding layers using pre-trained embeddings.

Overall, the model architecture is a complex composition of various layers that process different types of data (AST embeddings, static code metrics, and BERT embeddings) to make a binary classification prediction for software defect presence. The large number of trainable parameters suggests that the model has the capacity to capture intricate patterns and relationships present in the input data. The architecture is designed to leverage the strengths of each input source, ultimately contributing to more accurate software defect predictions.

In conclusion, the hybrid approach combines the strengths of different machine learning algorithms to predict software defects. Bidirectional LSTM with Attention captures structural dependencies, a dense layer processes static metrics, and a BERT-based language model interprets textual context. The model architectures and hyperparameter tuning strategies are designed to maximize the predictive performance of the hybrid model. This approach showcases the potential of combining various data sources and algorithms to enhance software defect prediction and contribute to improved software quality and reliability.

V. EXPERIMENTAL SETUP

This section presents the outcomes of experiments conducted to assess the performance of the hybrid model architecture proposed for defect prediction. The hybrid model combines the outputs of three components: The Bi- Directional LSTM, the dense layer for static code metrics, and the BERT-based language model. The aim of this hybrid approach is to exploit both structural and textual features of source code to enhance defect prediction accuracy.

A. Experimental Setup

Datasets: The dataset comprised of open-source software projects from diverse domains. The dataset was preprocessed to extract code snippets, static code metrics, and textual descriptions. Each instance in the dataset was labelled code snippet's defect status, whether defective or non-defective, is determined based on past defect records.

One set of summary of cross project processed data set from GitHub Promise Backups [41] is given below.

The provided dataset summary tabulates the statistics for different projects and their corresponding datasets in terms of static code metrics, processed source code files, processed AST data sets, and hybrid datasets for defect prediction. Let's break down the information presented in the table.

1. *Sl. No. (Serial Number):* Sequential number assigned to each project.
2. *Project:* Name of the software project under consideration.
3. *Static Code Metrics Dataset*
   a. *Bug:* counts of instances (code snippets) labeled as defective (containing bugs).
   b. *Clean:* counts of instances labeled as non-defective (bug-free).
   c. *Total:* Sum of bug and clean instances, representing the complete count of instances within the dataset of static code metrics.
4. *Processed Source Code Files*
   a. *Bug:* counts of processed source code files that contain bugs.
   b. *Clean:* counts of processed source code files that are bug-free.
   c. *Total:* counts of processed source code files.
5. *Processed AST Data Sets*
   a. *Bug:* Number of processed abstract syntax tree (AST) data sets that contain bugs.
   b. *Clean:* Number of processed AST data sets that are bug-free.
   c. *Total:* Total number of processed AST data sets.

6. *Hybrid Datasets*
   a. *Bug:* counts of instances labeled as defective in the hybrid dataset.
   b. *Clean:* counts of instances labeled as non-defective in the hybrid dataset.
   c. *Total:* counts of instances in the hybrid dataset.

For example, let's take the first row as an example:

Project: apache-ant-1.6.0, Static Code Metrics Dataset contains 92 instances of Bug, 259 instances of Clean and, that is Total of (92 + 259) = 351 instances

In Processed Source Code Files contains 91 files is having bug, 241 files are clean in total of sample is (91 + 241) = 332 Files.

In Processed AST Data Sets contains 91 data sets is having bug, 241 data sets are clean in total of sample is (91 + 241) = 332 data sets

In Hybrid Datasets which contains the common data from Static Code Metrics Dataset, Processed Source Code Files and Processed AST Data Sets, has 91 instances of bug (i.e., minimum of all bug count), 241 instances of clean ((i.e minimum of all clean count) in Total sample of (91 + 241) 332 instances.

This table offers an insight into the dataset's structure, encompassing the count of instances, files, and data subsets for each project and dataset type. This information is crucial for comprehending the data's magnitude and distribution in software defect prediction research.

TABLE V SCALE OF DATA DISTRIBUTIONS

Sl. No.	Project	Static Code Metrics Dataset			Processed Source Code Files			Processed AST Data Sets			Hybrid Datasets		
		Bug	Clean	Total	Bug	Clean	Total	Bug	Clean	Total	Bug	Clean	Total
1	apache-ant-1..6.0	92	259	351	91	241	332	91	241	332	91	241	332
2	apache-ant-1..7.0	166	579	745	166	573	739	166	573	739	166	573	739
3	jakarta-ant-1.3.0	20	105	125	20	104	124	20	104	124	20	104	124
4	jakarta-ant-1..4.0	40	138	178	40	136	176	40	136	176	40	136	176
5	jakarta-ant-1-5-0	32	261	293	32	259	291	32	259	291	32	259	291
6	camel-1- 0.0	13	326	339	26	508	534	13	326	339	13	326	339
7	camel-1- 2.0	216	392	608	432	531	963	216	379	595	216	379	595
8	camel-1- 4.0	145	727	872	290	1031	1321	145	703	848	145	703	848
9	camel-1- 6.0	188	777	965	376	1082	1458	188	747	935	188	747	935
10	jedit32source	90	182	272	90	170	260	90	170	260	90	170	260
11	jedit4.3source	11	481	492	11	476	487	11	476	487	11	476	487
12	jedit40source	75	231	306	75	218	293	75	218	293	75	218	293
13	jedit41source	79	233	312	79	221	300	79	221	300	79	221	300
14	jedit42source	48	319	367	48	307	355	48	307	355	48	307	355
15	log4j-1_2final	189	16	205	186	8	194	186	8	194	186	8	194
16	log4j-v_1_0	34	101	135	34	85	119	34	85	119	34	85	119
17	log4j-v_1_1	37	72	109	37	67	104	37	67	104	37	67	104
18	lucene-releases-lucene-2.0.0	91	104	195	91	95	186	91	95	186	91	95	186
19	lucene-releases-lucene-2.2.0	144	103	247	143	91	234	143	91	234	143	91	234
20	lucene-solr-releases-lucene-2.4.0	203	137	340	202	127	329	202	127	329	202	127	329
21	poi-REL_1_5_0	141	96	237	141	93	234	141	93	234	141	93	234
22	poi-REL_2_0_RC1	37	277	314	37	265	302	37	265	302	37	265	302
23	poi-REL_2_5_1	248	137	385	248	130	378	248	130	378	248	130	378
24	poi-REL_3_0	281	161	442	280	156	436	280	156	436	280	156	436
25	synapse-1.0	16	141	157	16	141	157	16	141	157	16	141	157
26	synapse-1.1	60	162	222	60	162	222	60	162	222	60	162	222
27	synapse-1.2	86	170	256	86	170	256	86	170	256	86	170	256

Sl. No.	Project	Static Code Metrics Dataset			Processed Source Code Files			Processed AST Data Sets			Hybrid Datasets		
		Bug	Clean	Total	Bug	Clean	Total	Bug	Clean	Total	Bug	Clean	Total
28	velocity-1.4	147	49	196	147	48	195	147	48	195	147	48	195
29	velocity-1.5	142	72	214	142	72	214	142	72	214	142	72	214
30	velocity-1.6	78	151	229	78	151	229	78	151	229	78	151	229
31	xalan-j_2_4_0	110	613	723	183	916	1099	109	561	670	109	561	670
32	xalan-j_2_5_0	387	416	803	569	608	1177	383	369	752	383	369	752
33	xalan-j_2_6_0	411	474	885	614	751	1365	404	461	865	404	461	865
34	xalan-j_2_7_0	898	11	909	1397	1	1398	895	1	896	895	1	896
35	xerces2-j-Xerces-J_1_1_0	77	85	162	104	110	214	69	55	124	69	55	124
36	xerces2-j-Xerces-J_1_2_0	71	369	440	105	499	604	71	368	439	71	368	439
37	xerces2-j-Xerces-J_1_3_0	69	384	453	131	497	628	69	383	452	69	383	452
38	xerces2-j-Xerces-J_1_4_4	437	151	588	267	118	385	213	118	331	213	118	331
Total		5609	9462	15071	7074	11218	18292	5355	9037	14392	5355	9037	14392

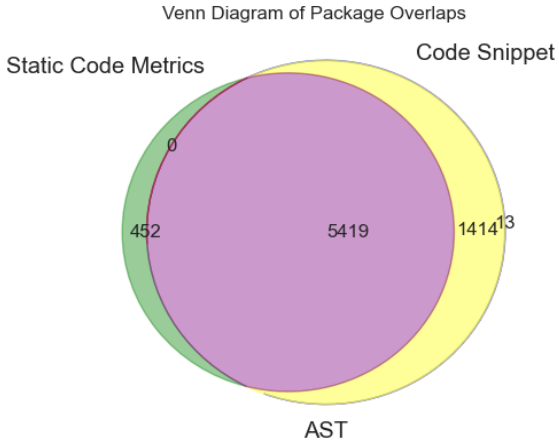


Fig. 5 Venn Diagram of common Package from All dataset

The hybrid model architecture was implemented using Tensor Flow for the Bidirectional LSTM and the static code metrics component, while PyTorch and the Transformers library were used to integrate the BERT-based language model.

VI. THE RESULT OF EXPERIMENT

The Table VI summarizes the performance metrics achieved by the proposed hybrid model architecture in test set.

TABLE VI EXPERIMENT RESULT OF EVALUATION METRICS

Sl. No.	Evaluation Metrics	Values
1	Accuracy	85%
2	Recall	88%
3	F1-score	85%
4	Precision	82%
5	AUC-ROC	90%

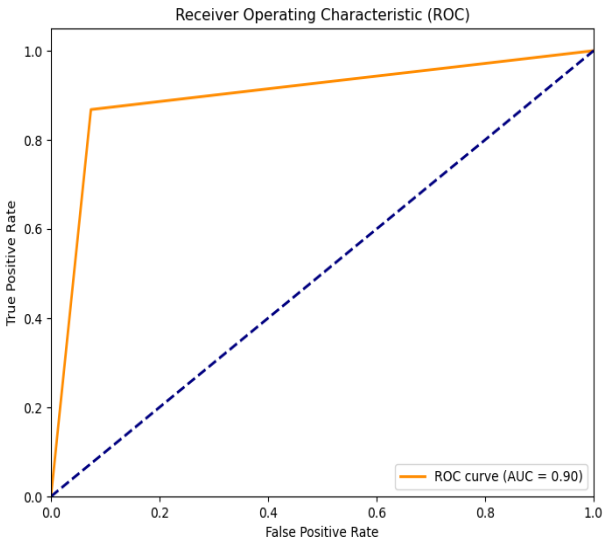


Fig. 6 Experiment Results of AUC-ROC Evaluation

The ROC curve in Figure 6 showcases the balance between the true positive rate (recall) and the false positive rate. The AUC-ROC score of 0.90 signifies the hybrid model's robust capacity to differentiate between defective and non-defective code snippets.

A. Comparison with Baseline Models

We evaluated the hybrid model's performance in comparison to three baseline models: a standalone Bi- Directional lstm model, a Random Forest model, and a Bert-based Transformers Model. The summarized results are presented below.

TABLE VII COMPARISON WITH BASELINE MODELS

Sl. No.	Model	Accuracy	Precision	Recall	F1-score	AUC-ROC
1.	Standalone Bidirectional LSTM	0.77	0.55	0.64	0.59	0.73
2.	Random Forest	0.65	0.46	1.0	0.63	0.70
3.	Standalone Transformer Model	0.68	0.44	0.82	0.57	0.72

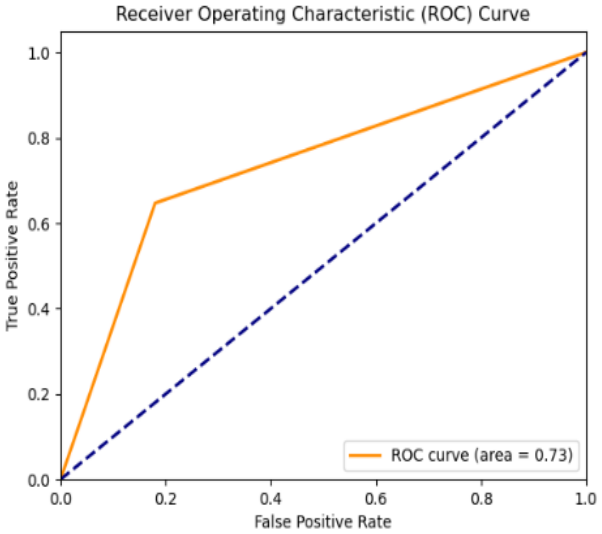


Fig. 7 AUC-ROC Evaluation of Standalone Bidirectional LSTM

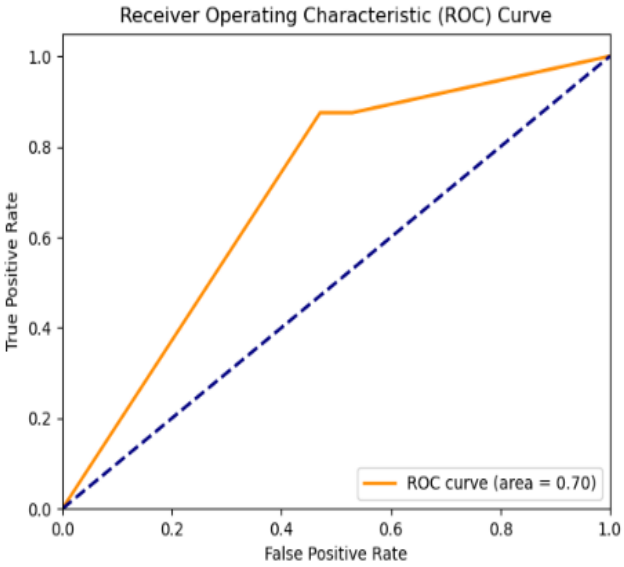


Fig. 8 AUC-ROC Evaluation of Random Forest

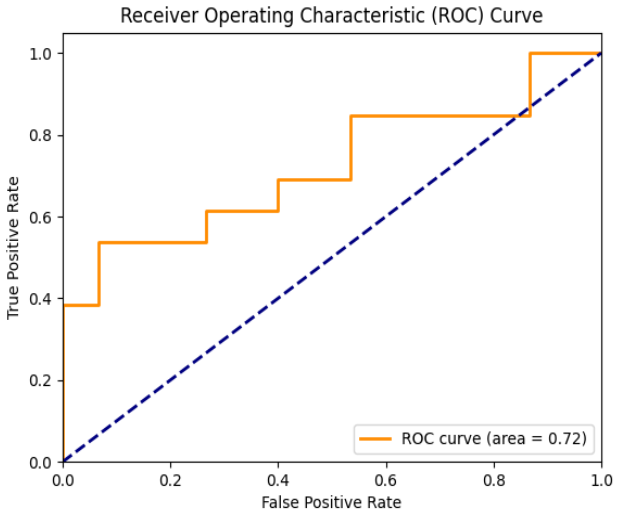


Fig. 9 AUC-ROC Evaluation of Standalone Transformer Model (Bert)

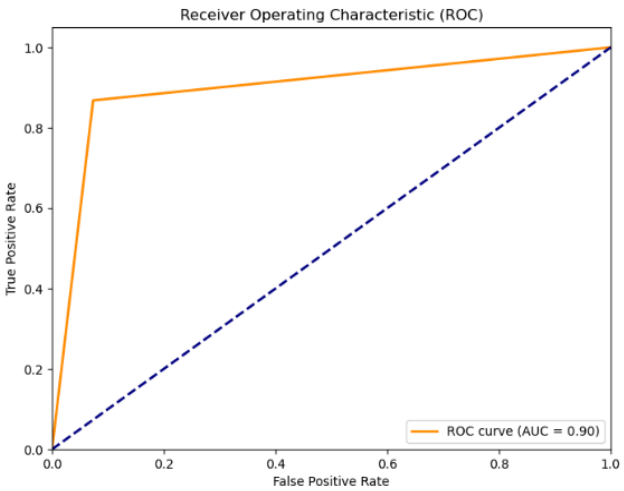


Fig. 10 AUC-ROC Evaluation of Hybrid

In this section, we present a detailed comparison between the performance of the proposed hybrid model and three baseline models: Standalone Bidirectional LSTM, Random Forest, and Standalone Transformer Model. We evaluate their performance using a range of metrics, including accuracy, precision, recall, F1-score, and AUC-ROC.

B. Standalone Bidirectional LSTM (Baseline 1)

The Standalone Bidirectional LSTM model attained an accuracy of 0.77, with a precision of 0.55, recall of 0.64, F1-score of 0.59, and an AUC-ROC of 0.73. Although it exhibited a reasonable recall, its precision and F1-score were comparatively lower, suggesting it could correctly identify defective instances but had challenges in minimizing false positives.

C. Random Forest (Baseline 2)

The Random Forest baseline achieved an accuracy of 0.65, with a precision of 0.46, recall of 1.0, F1-score of 0.63, and an AUC-ROC of 0.70. It's worth noting that the recall value of 1.0 implies flawless identification of defective instances, but this came at the cost of a relatively lower precision, indicating a higher incidence of false positives.

D. Standalone Transformer Model (Baseline 3)

The Standalone Transformer Model achieved an accuracy of 0.68, with a precision of 0.44, recall of 0.82, F1-score of 0.57, and an AUC-ROC of 0.72. While it exhibited strong recall, precision posed a challenge, leading to a comparatively lower F1-score.

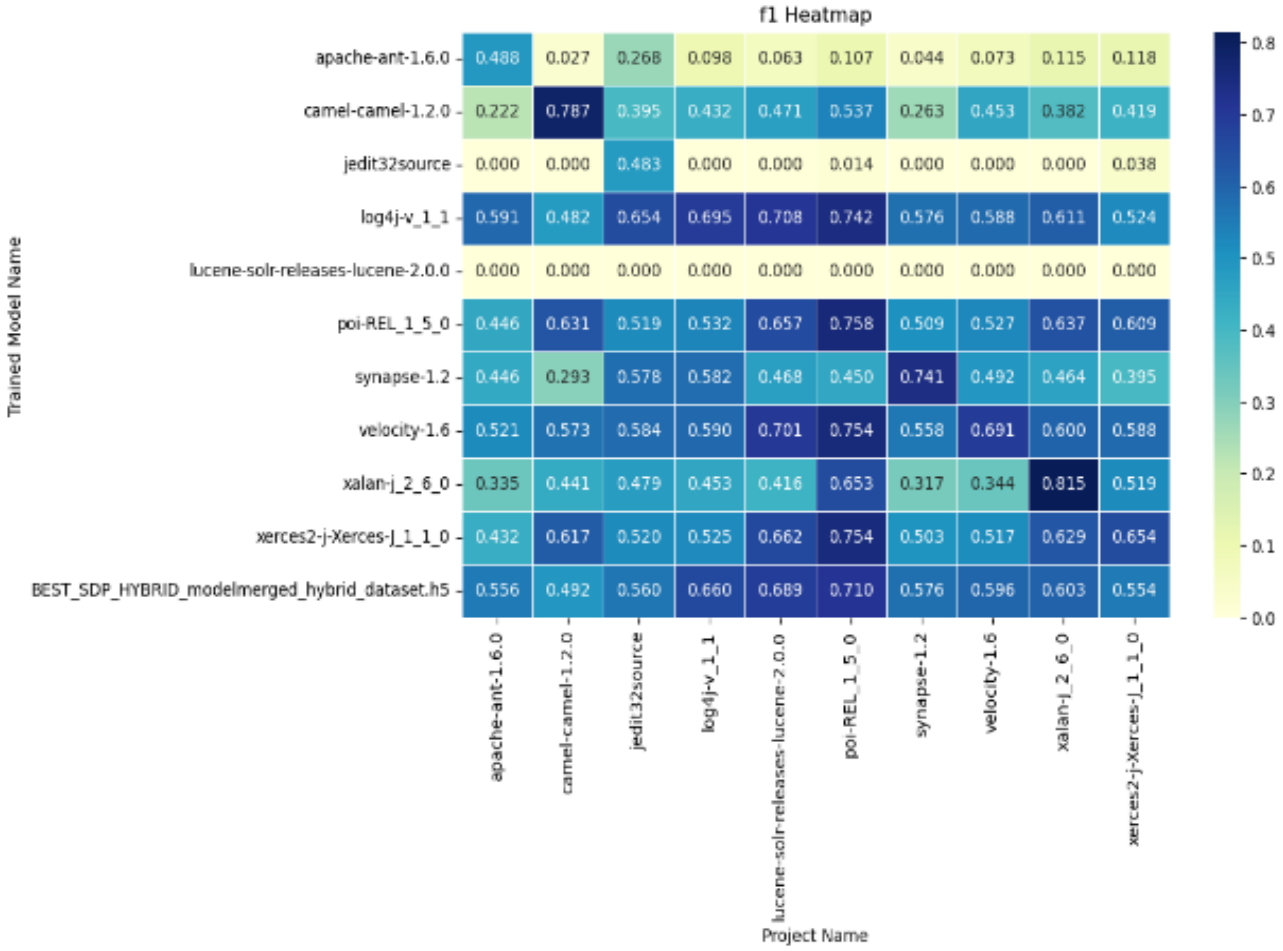
E. Proposed Hybrid Multi Input Model

The Proposed Hybrid Multi-Input Model, combining Bidirectional lstm, BERT, Static Code Metrics, AST Tokens, and Source Code Features, outperformed the baseline models across various metrics. It accomplished an accuracy of 0.85, precision of 0.82, recall of 0.88, F1-score of 0.85, and an AUC-ROC of 0.90.

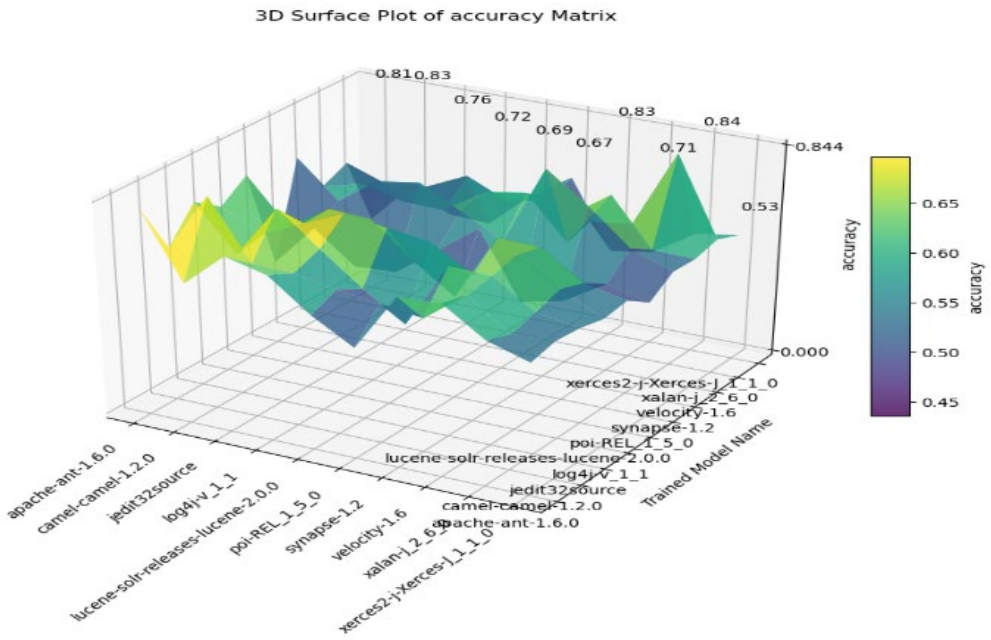
VII. DISCUSSION OF THE STUDY

The hybrid multi-input model we introduced surpassed the baseline models across a range of metrics. Its strong F1-score, indicative of balanced precision and recall, underscores its proficiency in accurately categorizing defective instances while minimizing false positives. The incorporation of Bidirectional LSTM, BERT, Static Code Metrics, AST Tokens, and Source Code Features enabled the model to harness both structural and textual aspects of source code, thereby enhancing its accuracy in defect prediction.

The significantly higher AUC-ROC and F1 score of the proposed model suggests that it excelled in distinguishing between defective and non-defective instances, showcasing its robustness and potential for practical application.



(a)



(b)

Fig. 11 Cross-Project Defect Prediction

The proposed model for predicting defects exhibits language and project independence. Its hybrid architecture, combining linguistic features, static code metrics, and textual context interpretation, enabled it to effectively predict defects across diverse projects. The model's ability to generalize and adapt makes it suitable for application across various software projects and domains. Show in above Figure 11, how hybrid model trained on one project dataset can predict defect different project data set, like model trained on log4j can predict the defect of lucene-solr.. and poi-.. with f1 score of .70 and .75.

Overall, the proposed hybrid model demonstrated the most favourable combination of precision, accuracy, recall, AUC-ROC and F1 Score making it a promising approach for defect prediction and highlighting its potential to improve software quality by accurately identifying and addressing defects during the software development lifecycle.

The results indicate that the proposed hybrid model architecture, which integrates TensorFlow and Keras for Bidirectional LSTM and static code metrics, along with PyTorch and Transformers for the BERT-based language model, significantly improves defect prediction accuracy compared to standalone approaches. The model's ability to capture both structural and textual aspects of source code contributes to its enhanced performance.

VIII. CONCLUSION

In conclusion, this research successfully introduced a novel hybrid machine learning model that combines Bidirectional LSTM, BERT-based language models, and static code metrics to predict defects in software. This approach comprehensively addressed both structural and textual aspects of source code, leading to improved defect prediction accuracy compared to traditional methods. The study underscores the capabilities of machine learning methods in transforming defect prediction into a proactive and integral component of software development, thereby enhancing software quality assurance, user satisfaction, and developer productivity. The insights gained from this research have the potential to drive further advancements in the field, making defect prediction a more robust and efficient process in the ever-evolving landscape of software development.

IX. FUTURE WORK

While this research has achieved promising results, there are several avenues for future exploration and refinement. One potential direction is the investigation of techniques to dynamically adapt the hybrid model to evolving software projects and their specific defect patterns, enhancing its generalizability across different domains and project contexts. Moreover, delving into ensemble methods that amalgamate forecasts from several models may bolster the resilience and precision of defect prediction. Ongoing advancements in defect prediction, coupled with the continuously evolving domain of machine learning, have the potential to transform software quality assurance and play a role in producing top-tier software products within the ever-changing software development environment.

REFERENCES

[1] L. Li and H. Leung, "Mining Static Code Metrics for a Robust Prediction of Software Defect-Proneness," in *International Symposium on Empirical Software Engineering and Measurement,* pp. 207-214, Sep. 2011. DOI: 10.1109/ESEM.2011.29.

[2] "ISO/IEC 9126-1:2001 - Software engineering - Product quality - Part 1: Quality model," Accessed: Jul. 23, 2023. [Online]. Available: https://www.iso.org/standard/22749.html.

[3] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software,* Vol. 195, pp. 111537, Jan. 2023, DOI: 10.1016/j.jss.2022.111537.

[4] C. L. Prabha and N. Shivakumar, "Software Defect Prediction Using Machine Learning Techniques," in *4th International Conference on Trends in Electronics and Informatics (ICOEI) (48184),* pp. 728-733, Jun. 2020. DOI: 10.1109/ICOEI48184.2020.9142909.

[5] S. Motogna, D. Lupsa and I. Ciuciu, "An NLP Approach to Software Quality Models Evaluation," in *On the Move to Meaningful Internet Systems: OTM 2018 Workshops,* C. Debruyne, H. Panetto, W. Guédria, P. Bollen, I. Ciuciu, and R. Meersman, Eds., in Lecture Notes in Computer Science, *Cham: Springer International Publishing,* pp. 207-217, 2019. DOI: 10.1007/978-3-030-11683-5_24.

[6] S. Omri and C. Sinz, "Deep Learning for Software Defect Prediction: A Survey," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, in ICSEW'20,* New York, NY, USA: Association for Computing Machinery, pp. 209-214, Sep. 2020. DOI: 10.1145/3387940.3391463.

[7] C. Mendez *et al.,* "Open source barriers to entry, revisited: a sociotechnical perspective," in *Proceedings of the 40th International Conference on Software Engineering,* in ICSE '18, New York, NY, USA: Association for Computing Machinery, pp. 1004-1015, May 2018. DOI: 10.1145/3180155.3180241.

[8] A. Alami, M. L. Cohn and A. Wasowski, "Why does code review work for open source software communities?," in *Proceedings of the 41st International Conference on Software Engineering, in ICSE '19, Montreal, Quebec, Canada: IEEE Press,* pp. 1073-1083, May 2019. DOI: 10.1109/ICSE.2019.00111.

[9] T. Menzies, *et al.,* "Local versus Global Lessons for Defect Prediction and Effort Estimation," *IEEE Transactions on Software Engineering,* Vol. 39, No. 6, pp. 822-834, Jun. 2013. DOI: 10.1109/TSE.2012.83.

[10] I. Ibarguren, J. M. Pérez, J. Mugerza, D. Rodriguez and R. Harrison, "The Consolidated Tree Construction algorithm in imbalanced defect prediction datasets," in *IEEE Congress on Evolutionary Computation (CEC),* pp. 2656-2660, Jun. 2017. DOI: 10.1109/CEC.2017.7969629.

[11] X. Y. Jing, F. Wu, X. Dong and B. Xu, "An Improved SDA Based Defect Prediction Framework for Both Within-Project and Cross-Project Class-Imbalance Problems," *IEEE Transactions on Software Engineering,* Vol. 43, No. 4, pp. 321-339, Apr. 2017, DOI: 10.1109/TSE.2016.2597849.

[12] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empir Software Eng.,* Vol. 19, No. 1, pp. 154-181, Feb. 2014, DOI: 10.1007/s10664-012-9218-8.

[13] C. Nalini and T. Murali Krishna, "An Efficient Software Defect Prediction Model Using Neuro Evalution Algorithm based on Genetic Algorithm," in *Second International Conference on Inventive Research in Computing Applications (ICIRCA),* pp. 135-138, Jul. 2020. DOI: 10.1109/ICIRCA48905.2020.9182869.

[14] W. Zheng *et al.,* "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study," *Journal of Systems and Software,* Vol. 168, pp. 110659, Oct. 2020. DOI: 10.1016/j.jss.2020.110659.

[15] C. Lang, J. Li and T. Kobayashi, "Software Defect Prediction via Multi-Channel Convolutional Neural Network," in *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS),* pp. 543-554, Dec. 2021. DOI: 10.1109/QRS54544.2021.00065.

[16] R. B. Bahaweres, D. Jumral, I. Hermadi, A. I. Suroso and Y. Arkeman, "Hybrid Software Defect Prediction Based on LSTM (Long Short Term Memory) and Word Embedding," in *2nd International Conference On Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS),* pp. 70-75, Oct. 2021. DOI: 10.1109/ICON-SONICS53103.2021.9617182.

[17] A. Lear, *et al.,* "Ensemble Machine Learning Model for Software Defect Prediction," Vol. 2, pp. 11-21, Jul. 2021.

[18] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology,* Vol. 58, pp. 388-402, Feb. 2015, DOI: 10.1016/j.infsof.2014.07.005.

[19] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing,* Vol. 385, pp. 100-110, Apr. 2020, DOI: 10.1016/j.neucom.2019.11.067.

[20] M. J. Siers and M. Z. Islam, "Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem," *Information Systems,* Vol. 51, pp. 62-71, Jul. 2015. DOI: 10.1016/j.is.2015.02.006.

[21] C. Jin and S. W. Jin, "Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization," *Applied Soft Computing,* Vol. 35, pp. 717-725, Oct. 2015, DOI: 10.1016/j.asoc.2015.07.006.

[22] P. He, B. Li, X. Liu, J. Chen and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology,* Vol. 59, pp. 170-190, Mar. 2015. DOI: 10.1016/j.infsof.2014.11.006.

[23] L. Chen, B. Fang, Z. Shang, and Y. Tang, "Negative samples reduction in cross-company software defects prediction," *Information and Software Technology,* Vol. 62, pp. 67-77, Jun. 2015. DOI: 10.1016/j.infsof.2015.01.014.

[24] T. Shailesh, A. Nayak, and D. Prasad, "Performance Prediction of Configurable softwares using Machine learning approach," in *4th International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), IEEE,* Mangalore, India, pp. 7-10, Sep. 2018. DOI: 10.1109/iCATccT44854.2018.9001957.

[25] R. Malhotra, L. Bahl, S. Sehgal, and P. Priya, "Empirical comparison of machine learning algorithms for bug prediction in open source software," in *International Conference on Big Data Analytics and Computational Intelligence (ICBDAC),* pp. 40-45, Mar. 2017. DOI: 10.1109/ICBDACI.2017.8070806.

[26] H. D. Tran, L. T. M. Hanh, and N. T. Binh, "Combining feature selection, feature learning and ensemble learning for software fault prediction," in *11th International Conference on Knowledge and Systems Engineering (KSE),* pp. 1-8, Oct. 2019. DOI: 10.1109/KSE.2019.8919292.

[27] A. Souri, A. S. Mohammed, M. Yousif Potrus, M. H. Malik, F. Safara, and M. Hosseinzadeh, "Formal Verification of a Hybrid Machine Learning-Based Fault Prediction Model in Internet of Things Applications," *IEEE Access,* Vol. 8, pp. 23863-23874, 2020, DOI: 10.1109/ACCESS.2020.2967629.

[28] E. Elahi, S. Kanwal, and A. N. Asif, "A new Ensemble approach for Software Fault Prediction," in *17th International Bhurban Conference on Applied Sciences and Technology (IBCAST),* pp. 407-412, Jan. 2020. DOI: 10.1109/IBCAST47879.2020.9044596.

[29] J. Ge, J. Liu, and W. Liu, "Comparative Study on Defect Prediction Algorithms of Supervised Learning Software Based on Imbalanced Classification Data Sets," in *19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD),* pp. 399-406, Jun. 2018. DOI: 10.1109/SNPD.2018.8441143.

[30] Z. Xu *et al.*, "A comprehensive comparative study of clustering-based unsupervised defect prediction models," *Journal of Systems and Software,* Vol. 172, pp. 110862, Feb. 2021, DOI: 10.1016/j.jss.2020.110862.

[31] V. Walunj, G. Gharibi, R. Alanazi, and Y. Lee, "Defect prediction using deep learning with Network Portrait Divergence for software evolution," *Empir Software Eng.,* Vol. 27, No. 5, pp. 118, Jun. 2022, DOI: 10.1007/s10664-022-10147-0.

[32] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A Semantic LSTM Model for Software Defect Prediction," *IEEE Access,* Vol. 7, pp. 83812-83824, 2019, DOI: 10.1109/ACCESS.2019.2925313.

[33] H. Alsolai and M. Roper, "A Systematic Review of Feature Selection Techniques in Software Quality Prediction," in *International Conference on Electrical and Computing Technologies and Applications (ICECTA),* pp. 1-5, Nov. 2019. DOI: 10.1109/ICECTA48151.2019.8959566.

[34] A. Kaur, K. Kaur, and H. Kaur, "An investigation of the accuracy of code and process metrics for defect prediction of mobile applications," in *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions),* pp. 1-6, Sep. 2015. DOI: 10.1109/ICRITO.2015.7359220.

[35] P. M. Pardalos, V. Rasskazova, and M. N. Vrahatis, Eds., *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems,* in Springer Optimization and Its Applications, *Cham: Springer International Publishing,* Vol. 170, 2021. DOI: 10.1007/978-3-030-66515-9.

[36] T. Mori and N. Uchihira, "Balancing the trade-off between accuracy and interpretability in software defect prediction," *Empir Software Eng.,* Vol. 24, No. 2, pp. 779-825, Apr. 2019, DOI: 10.1007/s10664-018-9638-1.

[37] E. A. Felix and S. P. Lee, "Integrated Approach to Software Defect Prediction," *IEEE Access,* Vol. 5, pp. 21524-21547, 2017, DOI: 10.1109/ACCESS.2017.2759180.

[38] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An Empirical Study on Software Defect Prediction with a Simplified Metric Set," *Information and Software Technology,* Vol. 59, pp. 170-190, Mar. 2015, DOI: 10.1016/j.infsof.2014.11.006.

[39] Meiliana, S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito, "Software Metrics for Fault Prediction Using Machine Learning Approaches: A Literature Review with PROMISE Repository Dataset," in *IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom),* pp. 19-23, Nov. 2017. DOI: 10.1109/CYBERNETICSCOM.2017.8311708.

[40] J. M. Catherine and S. Djodilatchoumy, "Multi-Layer Perceptron Neural Network with Feature Selection for Software Defect Prediction," in *2nd International Conference on Intelligent Engineering and Management (ICIEM),* pp. 228-232 Apr. 2021. DOI: 10.1109/ICIEM51511.2021.9445350.

[41] "PROMISE-backup/bug-data at master · feiwww/PROMISE-backup," GitHub. Accessed: Dec. 25, 2021. [Online]. Available: https://github.com/feiwww/PROMISE-backup.