Journal of Harbin Engineering University
ISSN: 1006-7043
Vol 44 No. 9
September 2023

An Inclusive Hybrid Approach for Predicting Defects in Microservices Architecture Across Languages

Yashwant Kumar [1], Vinay Singh[2], Love Kumar[3]

[1,2]Faculty of Computing & Information Technology, Usha Martin University, Ranchi, Jharkhand
[3]Department of Computer Engineering and Applications, Mangalayatan University, Aligarh, U.P.
Email: vinaysinghuma@gmail.com, love.mittal@mangalayatan.edu.in

Abstract
In the evolving landscape of software development, where monolithic frameworks are giving way to microservices-based architectures, a significant challenge lies in crafting a unified defect prediction model that transcends the boundaries of diverse programming languages, all within the context of continuous integration and continuous delivery (CI/CD) pipelines. This paper introduces a novel hybrid machine learning approach aimed at elevating the accuracy of defect prediction by seamlessly amalgamating disparate data sources and employing a diverse set of algorithms. The ultimate objective is the creation of a defect prediction model that is both language-agnostic and project-independent.

This hybrid model amalgamates Bidirectional Long Short-Term Memory (BiLSTM) networks with Attention mechanisms, static code metrics, and BERT-based language models. BiLSTM-Attention adeptly captures temporal dependencies residing within Abstract Syntax Trees (ASTs), while static code metrics furnish crucial insights into software complexity. Simultaneously, BERT lends its prowess in comprehending the textual context, thus facilitating a holistic comprehension of code snippets.

The research methodology encompasses a rigorous quantitative approach, commencing with an exhaustive literature review to establish a solid theoretical foundation. Subsequently, an empirical study unfolds, encompassing the entire gamut of activities ranging from data collection, preprocessing, and feature engineering, to model development, training, evaluation, analysis, validation, and the eventual derivation of conclusions. The insights derived from this research endeavour aspire to advance defect prediction techniques, thereby contributing significantly to the overarching goals of software engineering—namely, the pursuit of enhanced software quality and reliability.

Keywords: Software development, microservices, defect prediction, unified model, hybrid machine learning, LSTM, Attention, static metrics, BERT, quality, reliability.

1. 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 CI/CD pipeline is must for identifying the early bugs before successful deployment of any modules.

1.1 Background and motivation

The background and motivation for creating a good model of software defect prediction using Machine Learning techniques is to improve software development processes, reduce costs, enhance software quality, and deliver more reliable and robust software products.(Li & Leung, 2011) Let's explore the key reasons behind this initiative:

1.1.1 Quality Improvement

Defect prediction models anticipate software issues early, enabling proactive resolution for improved code quality and reduced production defects.(*ISO/IEC 9126-1:2001 - Software Engineering — Product Quality — Part 1: Quality Model*, n.d.)

1.1.2 Cost Reduction

Defect prediction models save time and resources by detecting software issues early, minimizing costly late-stage or post-production fixes.

1.1.3 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.

1.1.4 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.

1.1.5 Continuous Integration and Delivery (CI/CD) Pipeline Improvement
Integrating defect prediction models in the CI/CD pipeline automates checks for quality assurance, enhancing DevOps practices.

1.1.6 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.

1.1.7 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.

1.1.8 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.

1.1.9 Research and Innovation
The creation of good defect prediction models using Machine Learning also drives research and innovation in the field of Software Engineering and Data Science. Researchers continuously explore new approaches, algorithms, and data sources to improve the accuracy and effectiveness of these models.(Giray et al., 2023)

Machine learning has emerged as a powerful tool in various domains, including software engineering, due to its ability to analyse large volumes of data, identify patterns, and make predictions.(Prabha & Shivakumar, 2020) Recent years have seen the successful application of machine learning in software defect prediction by analysing historical data and code attributes. This approach enhances development processes, reduces defects, improves software quality, and results in better products. Building defect prediction models using source code and static metrics holds significant potential for improving software quality and development practices.(Motogna et al., 2019). The NLP approach seeks to improve defect prediction accuracy by harnessing natural language patterns found in source code, comments, and documentation. It extracts valuable insights from unstructured text and employs them for defect prediction tasks.

1.2. Problem
Definition and Objective
The challenge addressed here is the transition in software development towards multi-language microservices architecture, where services can be written in various programming languages. The goal is to develop a unified defect prediction model for use in a CI/CD pipeline. The proposed Hybrid model incorporates diverse data sources like Abstract Syntax Tree tokens, source code, and static code metrics from cross-project and cross-version datasets as input features, demonstrating superior performance compared to other approaches.

1.2 Key Aspects of the Scope

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

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

1.2.3 Feature Extraction
NLP techniques is applied to extract relevant features from the textual data. This may include keyword extraction, sentiment analysis, topic modelling, and semantic analysis. (Omri & Sinz, 2020). 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.

1.2.4 Model Development
The model deep learning algorithms to predict the likelihood of defects 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.

1.2.5 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.

1.2.6 Model Evaluation
Conducting comprehensive evaluations to measure the performance of the NLP-based defect prediction model, using metrics like precision, recall, F1-score, and area under the receiver operating characteristic curve (AUC-ROC).

1.2.7 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.

1.3 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 provide a comprehensive overview of its architecture 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 prediction. 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.

2. Related Work

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. (Mendez et al., 2018) (Alami et al., 2019)

2.1 Some Machine Learning Techniques

(Menzies et al., 2013) Regression models, such as multiple linear, logistic, and Poisson regression, are commonly used in software development to analyse the relationship between software metrics like code complexity, code churn, code size, and developer experience and the occurrence of defects.

(Ibarguren et al., 2017) (Jing et al., 2017) Decision trees, known for their hierarchical and interpretable nature, are widely used in defect prediction as they recursively partition data using key predictors, accommodating both categorical and numerical data, making them versatile for software defect prediction applications.

(Okutan & Yıldız, 2014) The Naive Bayes classifier, based on the assumption of feature independence given the class label, is employed in software defect prediction using software metrics as features and defective/non-defective modules as class labels.

(Giray et al., 2023) Neural networks, including feedforward and recurrent types, are utilized in software defect prediction due to their capacity to capture intricate relationships between software metrics and defects, with the ability to learn from extensive datasets.

(Nalini & Murali Krishna, 2020) Genetic algorithms are search-based optimization techniques that have been used to tune the parameters of defect prediction models. They aim to find the best combination of features and model parameters to maximize prediction performance.

2.2 Review of Deep Learning Related Studies on SDP

(Zheng et al., 2020) study compares various deep learning models, such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, for software defect prediction. The authors evaluate the performance of these models on real-world datasets and provide insights into the strengths and weaknesses of each approach.

(Lang et al., 2021) authors propose a simplified Convolutional Neural Network (CNN) for software defect prediction. They conduct an empirical study on multiple software projects, comparing their CNN model with traditional machine learning algorithms. The study demonstrates the effectiveness of the CNN model for defect prediction tasks.

(Bahaweres et al., 2021) paper presents a hybrid approach that combines Long Short-Term Memory (LSTM) neural networks with traditional machine learning algorithms for software defect prediction. The authors show how the hybrid model can improve the prediction accuracy compared to individual methods.

(Lear et al., 2021) authors propose an ensemble approach that combines multiple machine learning algorithms, including Random Forest, Support Vector

Machine (SVM), and Gradient Boosting, for defect prediction in open-source software. The study evaluates the performance of the ensemble on different datasets and discusses its effectiveness.

(Giray et al., 2023) research introduces a cost-sensitive Convolutional Neural Network (CNN) for software defect prediction. The authors address the issue of imbalanced data in defect prediction and show that the cost-sensitive CNN improves the prediction performance on imbalanced datasets.

(Laradji et al., 2015) criticized the success of research on software defect prediction by many algorithms if used alone such as Decision trees , Bayesian methods, and Artificial Neural Networks Multilayer Perceptron (ANN-MLPs). The authors also stated that, these methods were sub-optimal in the case of skewed and redundant defect datasets. The prediction performance of these methods get worse when the defect datasets contain incomplete or irrelevant features. Classifiers such as Support Vector Machines (SVMs) and ANN-MLPs, biased towards the dominant class i.e majority class, which tend to ignore the minority class that results in high false negative rates in confusion matrix. Therefore, it is suggested that ensemble learning models were very adequate to address the data issues mentioned earlier.

(Qiao et al., 2020) discussed various Machine Learning Methods of Defect Prediction Techniques and its shortcomings. Their Performances (e.g., mean square error and squared correlation coefficient) requires significant improvements. This paper introduces a deep learning-based defect prediction model that employs regression to estimate the number of defects in a module. The proposed approach is compared to three state-of-the-art approaches (SVR, FSVR, and DTR) to evaluate its effectiveness.

(Siers & Islam, 2015) was empirically evaluated the proposed technique comparing them with six(6) classifier algorithms on six(6) publicly available clean datasets that are commonly used in the research on software defect prediction. The proposed model was said to be cost sensitive learning in SDP can result in monetary savings for software development group. As per the authors it is better to have several false positive predictions in order to avoid a single false negative prediction. It also investigated a potential technique for incorporating over-sampling into Cost Sensitive Forest (CSForest) model.

(Jin & Jin, 2015) presents the application of hybrid artificial neural network (ANN) and Quantum Particle Swarm Optimization (QPSO) in software fault-proneness prediction. QPSO is applied for reducing dimensionality. ANN is used for classifying software modules into fault-proneness or non-fault-proneness categories. By the hybrid ANN and QPSO, an effective software fault-proneness prediction approach is proposed in this paper.

(He et al., 2015a) recorded the number of occurrences of different metrics in each prediction model, and then use the Top-k representative metrics as a universal feature subset to predict defects for all projects. This approach is said to be suitable for the projects without sufficient historical data for WPDP because of its generality. As the prediction model undergoes more extensive training, the feature subset ranked in the top-k becomes increasingly general.

(Chen et al., 2015) Local historical training data from the same company is not always available or is difficult to collect in practice. To address this problem, researchers have turned to Cross-company defect prediction (CCDP) as an alternative solution in the last few years. Its main steps were as follows: First, training datasets from other sources are preprocessed by the NN filter and data oversampling (SMOTE). Next, these preprocessed data were re-weighted by the first transfer method with data gravitation. Finally, limited amounts of labeled WC data (i.e., 10% of the total WC data) are mixed with re-weighted data to build the prediction model using the transfer boosting learning algorithm.

(Shailesh et al., 2018) In the current software industry most of the complex software are configurable. There is an enormous increase in configuration space as the number of features increases. Hence, there is a need to study the impact of different configuration on the system performance. Predictive models offer solutions to analyse system performance for a given configuration set. In this paper author propose a Neural network model with statistical techniques for predicting system performance for input configuration.

(Malhotra et al., 2017) study aims at comparison of 14 Machine Learning (ML) Techniques for development of effective defect prediction models. The study validates 9 open source data sets to assess and compare performance of 14 ML techniques using Weka tool. The models are further statistically compared using F-Test on SPSS software. The results of the study show that Single Layer perceptron is the best technique amongst all the techniques used in this study for development of defect prediction models.

(Tran et al., 2019) paper studies a combination of feature selection and ensemble learning to address the feature redundancy and class imbalance problems in software fault prediction. Also, a deep learning model is used to generate deep representation from defect data to improve the performance of fault prediction models. The proposed method, is evaluated on 12 NASA datasets.

(Souri et al., 2020) By increasing the complexity of the Internet of Things (IoT) applications, fault prediction become an important challenge in interactions between human, and smart devices. This paper presents a behavioural modeling and formal verification of a hybrid machine learning-based fault prediction model with Multi-Layer Perceptron (MLP) and Particle Swarm Optimization (PSO) algorithms.

(Elahi et al., 2020) Authors have investigated and compared the effect of different ensemble methods in improving the performance of prediction model. It also benchmarked the model averaging method using existing ensemble methods such as voting and stacking.

(Ge et al., 2018) Mainly describes the basic concepts, software metrics, feature selection methods of software defect prediction, and proposes a simplified software defect prediction framework suitable for engineering applications.

(Ge et al., 2018) and (Xu et al., 2021) compared the performance of Support Vector Machines (SVM) and Random Forest (RF) in defect prediction and highlighted the superiority of RF in handling noisy data. (Walunj et al., 2022) & (Liang et al., 2019) introduced a novel deep learning approach using Long Short-Term Memory (LSTM) networks for time series-based defect prediction.

(Alsolai & Roper, 2019) investigated the impact of various feature selection methods, including Relief-F and Chi-Square, on model accuracy for defect prediction. (Kaur et al., 2015) proposed a hybrid approach combining static code metrics and process metrics to improve the discriminative power of features.

(Pardalos et al., 2021) introduced LIME, a model-agnostic approach, to explain the predictions of black-box machine learning models in the context of 5defect prediction. (Mori & Uchihira, 2019) developed a rule-based classifier to provide transparent and interpretable defect predictions.

2.3 Research Gap

Software defect prediction through machine learning has been a dynamic research domain, encompassing a range of methodologies, including conventional machine learning algorithms, deep learning models such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, as well as ensemble methods and cost-sensitive strategies to address specific challenges. The evolution of this field is expected to involve the exploration of more advanced techniques and larger datasets to enhance the accuracy and applicability of software defect prediction models. A thorough examination of existing literature has revealed research gaps and opportunities for further exploration in this area.

The field of software defect prediction faces several critical areas of research and development. Firstly, there is a significant gap in understanding the transferability of defect prediction models across different industries and project types, as current studies primarily focus on specific domains. Future research should concentrate on assessing the efficacy of models trained in one domain when applied to unrelated domains. Additionally, while many studies concentrate on historical data for defect prediction, there is an emerging demand for real-time defect prediction during software development, which necessitates investigation into the feasibility and performance of machine learning models in this context. Furthermore, longitudinal defect prediction, which considers how models adapt to changing project dynamics over time, is essential for a comprehensive understanding of software project evolution. To enhance prediction systems, there is potential in integrating machine learning techniques with traditional defect prediction methods such as code reviews and inspections. Finally, as complex deep learning models gain traction in defect prediction, it is imperative to develop techniques that provide interpretability for these models, enabling stakeholders to trust and comprehend the decision-making process behind the predictions. Addressing these areas can significantly advance the field of software defect prediction.

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.

3. 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.

3.1 Data Collection and Preprocessing

In the Cross-Language Software Defect Predictor project, data collection and preprocessing involved gathering a dataset comprising source code files, static code metrics, and binary labels indicating module defectiveness. The dataset, sourced from open repositories like GitHub, PROMISE-backup-master, Bitbucket, and GitLab, covered a variety of programming languages like Java, C++, Python, JavaScript, and C#. It was carefully curated to include projects with labeled information regarding both defective and defect-free code instances. This dataset offered a sufficient number of samples, facilitating the construction of effective predictive models for binary classification tasks.

3.2 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:

3.2.1 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.

3.2.2 Lexical Analysis

After language identification, the lexical analysis, or lexing, phase involved reading the source code character by character and identifying tokens based on predefined rules and grammar files, typically defined using regular expressions or finite state machines. ANTLR-generated custom code in Java was used to tokenize code files into language-specific tokens like keywords, identifiers, literals, and operators for supported programming languages.

3.2.3 AST Token Generation

The token streams, obtained through the ANTLR-generated custom code, were subsequently processed by language-specific parsers to create Abstract Syntax Trees (ASTs) or parse trees that captured the hierarchical structure of the code. Just as natural or formal languages possess linguistic attributes like syntax, semantics, pragmatics, and grammatical rules, programming languages also exhibit such characteristics. Custom code was developed to leverage ANTLR as a language recognition tool, enabling the conversion of source code into sequences of language-neutral tokens, facilitating further analysis and processing.

Table 3.1 Extract Language Neutral Tokens from the Source Code of Java File

packageDeclaration, annotation, importDeclaration, typeDeclaration, classOrInterfaceModifier, classDeclaration, enumDeclaration, interfaceDeclaration, annotationTypeDeclaration, modifier, classOrInterfaceModifier, variableModifier, typeParameters, typeType, typeList, classBody, typeBound, enumConstants, enumBodyDeclarations, interfaceBody, memberDeclaration, methodDeclaration, methodBody, typeTypeOrVoid, genericMethodDeclaration, genericConstructorDeclaration, constructorDeclaration, fieldDeclaration, constDeclaration etc

ASTs were used to extract language-independent features like control flow, variable usage, and function call patterns, with the aim of establishing a consistent feature set for cross-language defect prediction; in addition, deep learning was employed to transform AST tokens into language-independent AST embeddings.

3.2.4 Static Code Feature Selection

When building models for software defect prediction, it's essential to carefully select relevant and well-established metrics based on empirical evidence and prior research. (Felix & Lee, 2017) (He et al., 2015b). Selecting predictive metrics for bug prediction depends on data, project characteristics, and analysis techniques. Experimentation helps identify relevant metrics, often using machine learning methods like logistic regression, decision trees, random forests, and support vector machines.

However, some commonly used metrics (Meiliana et al., 2017) that have been used here and shown predictive power for bug prediction in Modules are listed below:

Table 3.2 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.*)

Metric Abbreviation	**Full Form**	**Descriptions with Rationale for Selection**
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 above steps created dataset consisting of AST embeddings, static code metrics, and tokenized source code as inputs and bug labels as output. The Proposed Hybrid model would be using processed AST embeddings, while the Dense layer handles static code metrics. The BERT embeddings are combined with these processed inputs using a concatenation layer. Finally, dense layers are used for further processing before the output layer, which performs defect prediction through binary classification.

The final sample representation of data set looked like below:

Table 3.3 Snapshot Representation of Dataset

packageidentfier	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
....	....	....	....	....

3.2.5 Data Cleaning

Data cleaning involved removing duplicates, irrelevant samples, and special symbols, while preprocessing text data included lowercasing, punctuation removal, and handling special characters. Numeric data had missing values imputed with means or medians, while padding or masking was applied to textual data like source code and AST tokens.

3.2.6 BERT representations for Source code

Bidirectional Encoder Representations (BERT) from Transformers is a powerful language model capable of understanding the contextual information of the source code. It can capture semantic relationships and contextual meaning effectively, making it valuable for 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.

3.2.7 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.

3.2.8 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.

3.2.9 Data Integration
Depending on the architecture of multi-input deep learning model, there is a need to integrate the different data components. This could involve creating separate input branches for static code metrics, source codes, and AST tokens.

3.2.10 Data Splitting
Split dataset into training, validation, and test sets. A common split was 70% training, 15% validation, and 15% test. Shuffled the data to prevent any potential biases during training.

3.2.11 Understand Class Imbalance
In software defect prediction, addressing class imbalance is vital. It involves analysing the distribution of defective and non-defective code samples in the training set. If a significant class imbalance is detected, techniques like random duplication or synthetic data generation (e.g., SMOTE or ADASYN) are employed to balance class proportions.

3.2.12 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.

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

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

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

4. 1-Specificity = $= \frac{FP}{(TN+FP)}$

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

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

ROC is a probability curve and AUC represents degree or measure of separation. It tells how much model is capable of distinguishing between defect and non-defect cases.

3.2.13 Adjust Thresholds
Classifiers can be fine-tuned by adjusting the classification threshold to balance precision and recall, rather than relying on default probability-based predictions.

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

3.2.15 Class Weights
In deep learning, assigning higher weights to the minority class during training makes the model more sensitive to it, helping identify defects despite class imbalance; the approach depends on dataset specifics, requiring evaluation to balance false positives and negatives.

3.3 Feature Selection and Engineering
Static code metrics, source codes, and language-independent AST tokens, a suitable model architecture for cross-language and cross-project defect prediction 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.

(Catherine & Djodilatchoumy, 2021) Feature selection involves selecting a subset of the most relevant features from the original feature set. The goal is to reduce the dimensionality of the input data while maintaining or even improving model performance.

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

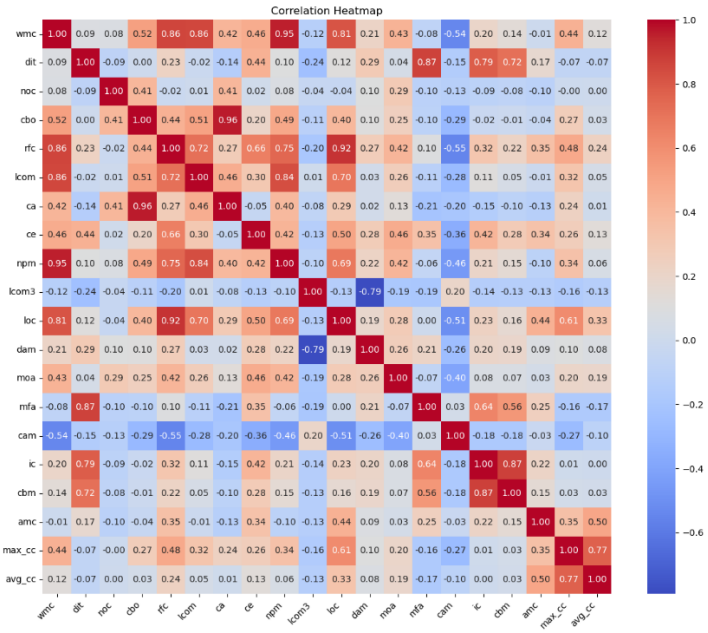

Figure 3.1 Correlation heat map among Static Code Metrics

4. Proposed Software Defect Prediction Framework

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:

4.1 Bidirectional LSTM with Attention for AST Embeddings

Bidirectional LSTM with Attention is designed for sequence data like ASTs in source code, capturing temporal dependencies and structural information.

4.2 Dense Layer for Static Code Metrics

Dense layers in deep learning efficiently process static code metrics, offering insights into software complexity and defect prediction.

4.3 BERT-based Language Model for Source Code

BERT (Bidirectional Encoder Representations from Transformers) is a powerful pre-trained language model capable of capturing contextual information from text 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.

4.5 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.

4.5.1 Bidirectional LSTM with Attention

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

4.5.2 Dense Layer for Static Code Metrics

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

4.5.3 BERT-based Language Model for Source Code

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

4.5.4 Hyperparameter Tuning Strategies

- **Batch Size:** Tuning the batch size affects convergence and memory usage. Smaller batch sizes of 32 improved generalizations.
- **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.
- **Dropout Rate**: Dropout is a regularization technique to mitigate overfitting. Hyperparameter search identified the dropout rate that balances overfitting and under fitting.

- **LSTM Units:** The number of LSTM units influences model complexity. A grid search revealed the optimal number.
- **Dense Layer Units**: The number of units in the dense layer can be tuned to control model capacity.
- **Embedding Dimensions:** For the embedding layer, different dimensions are explored to capture relevant features.

4.6 Model Summary

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 single unit and a sigmoid activation function produces the model's output with shape (None, 1), which corresponds to the binary classification of defect presence or absence.

The Total params count (93,711,937) represents the total number of trainable parameters in the model. These parameters are learned during the training process to optimize the model's performance on the given task. The Trainable params count (93,711,937) indicates the number of parameters that will be updated during training through backpropagation.

The Non-trainable params count (0) indicates that there are no non-trainable parameters in this model. Non-trainable parameters typically refer to parameters that are fixed or pre-initialized, such as in the case of embedding layers with 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.

5. Hybrid Model Evaluation and Results

In this section, the results of the experiments are presented that was conducted to evaluate the performance of the proposed hybrid model architecture for defect prediction. The hybrid model combines the outputs of three components: The Bidirectional LSTM, the dense layer for static code metrics, and the BERT-based language model. The objective of this hybrid approach is to leverage both structural and textual features of source code to enhance defect prediction accuracy.

5.1 Experimental Setup

5.1.1 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 as either a defective or non-defective code snippet based on historical defect records.

One set of summary of cross project processed data set from GitHub Promise Backups (PROMISE-Backup/Bug-Data at Master · Feiwww/PROMISE-Backup, n.d.) 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. **SNo** (Serial Number): Sequential number assigned to each project.
2. **Project**: Name of the software project under consideration.
3. **Static Code Metrics Dataset**

a. **Bug**: Number of instances (code snippets) labeled as defective (containing bugs).
b. **Clean**: Number of instances labeled as non-defective (bug-free).
c. **Total**: Sum of bug and clean instances, representing the total number of instances in the static code metrics dataset.

4. **Processed Source Code Files**
   a. **Bug**: Number of processed source code files that contain bugs.
   b. **Clean**: Number of processed source code files that are bug-free.
   c. **Total**: Total number 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**: Number of instances labeled as defective in the hybrid dataset.
   b. **Clean**: Number of instances labeled as non-defective in the hybrid dataset.
   c. **Total**: Total number of instances in the hybrid dataset.

For instance, 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 provides an overview of the dataset's composition, including the number of instances, files, and data sets for each project and dataset type. It's essential for understanding the scale and distribution of data used in the research on software defect prediction.

Table 5.1 Scale of Data Distributions

SNo	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	20	105	125	20	104	124	20	104	124	20	104	124
4	jakarta-ant-1.4	40	138	178	40	136	176	40	136	176	40	136	176
5	jakarta-ant-1.5	32	261	293	32	259	291	32	259	291	32	259	291
6	camel-camel-1.0.0	13	326	339	26	508	534	13	326	339	13	326	339
7	camel-camel-1.2.0	216	392	608	432	531	963	216	379	595	216	379	595
8	camel-camel-1.4.0	145	727	872	290	1031	1321	145	703	848	145	703	848
9	camel-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

SNo	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
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
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

SNo	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
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**

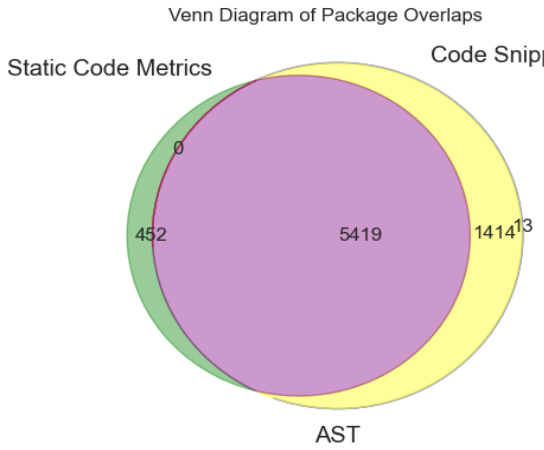

Figure 5.1 Venn Diagram of common Package from All dataset.

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.

6. The Result of Experiment

The following table summarizes the performance metrics achieved by the proposed hybrid model architecture in test set:

Table 6.1 Experiment Result of Evaluation Metrics

Sl No	Metric	Value
1	Accuracy	0.85
2	Precision	0.82
3	Recall	0.88
4	F1-score	0.85
5	AUC-ROC	0.90

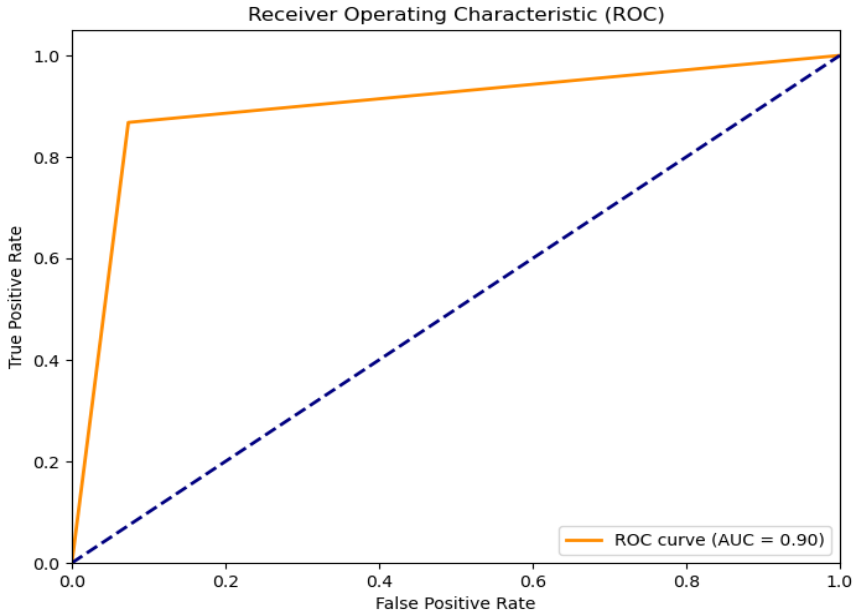

Figure 6.1 Experiment Results of AUC-ROC Evaluation

The ROC curve in Figure 6.1 illustrates the trade-off between the true positive rate (recall) and the false positive rate. The AUC-ROC value of 0.90 indicates a strong discriminative ability of the hybrid model in distinguishing between defective and non-defective code snippets.

6.1 Comparison with Baseline Models

We compared the performance of the hybrid model with three baseline models: a standalone Bidirectional LSTM model, Random Forest model and Bert based Transformers Model. The results are summarized below:

Table 6.2 Comparison with Baseline Models

SlNo.	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

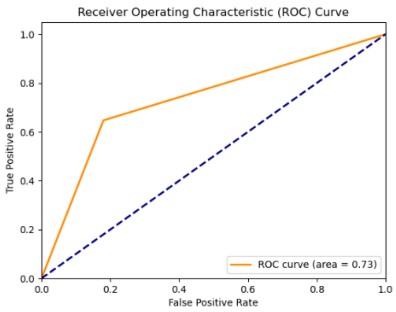

Figure 6.2 AUC-ROC Evaluation of Standalone Bidirectional LSTM

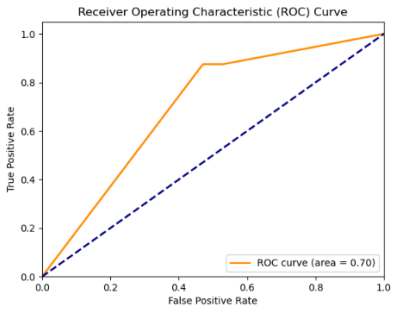

Figure 6.3 AUC-ROC Evaluation of Random Forest

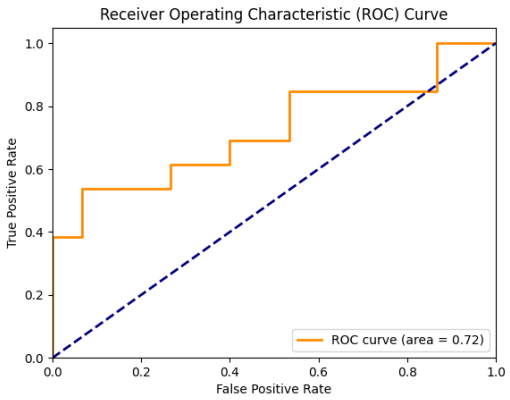

Figure 6.4 AUC-ROC Evaluation of Standalone Transformer Model (Bert)

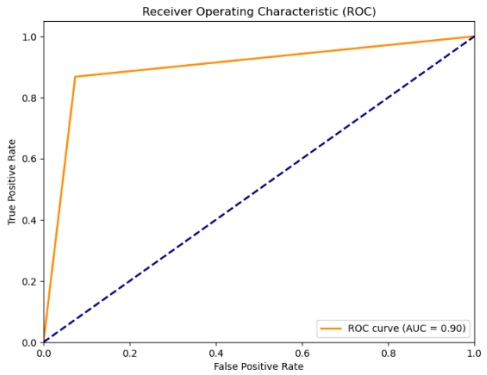

Figure 6.5AUC-ROC Evaluation of Hybrid

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.

6.1.1 Standalone Bidirectional LSTM (Baseline 1)

The Standalone Bidirectional LSTM model achieved an accuracy of 0.77, precision of 0.55, recall of 0.64, F1-score of 0.59, and AUC-ROC of 0.73. While it demonstrated reasonable recall, its precision and F1-score were relatively lower, indicating that it could correctly classify defective instances but struggled with minimizing false positives.

6.2.2 Random Forest (Baseline 2)

The Random Forest baseline achieved an accuracy of 0.65, precision of 0.46, recall of 1.0, F1-score of 0.63, and AUC-ROC of 0.70. Notably, the recall value of 1.0 suggests perfect identification of defective instances; however, this was accompanied by relatively lower precision, indicating a high rate of false positives.

6.2.3 Standalone Transformer Model (Baseline 3)

The Standalone Transformer Model attained an accuracy of 0.68, precision of 0.44, recall of 0.82, F1-score of 0.57, and AUC-ROC of 0.72. It demonstrated good recall but struggled with precision, resulting in a lower F1-score.

6.2.4 Proposed Hybrid Multi Input Model

The Proposed Hybrid Multi Input Model, which integrates Bidirectional LSTM, BERT with Static Code Metrics and AST Tokens, and Source Code Features, outperformed the baseline models across multiple metrics. It achieved an accuracy of 0.85, precision of 0.82, recall of 0.88, F1-score of 0.85, and AUC-ROC of 0.90.

7. Discussion

The proposed hybrid multi input model outperformed the baseline models across various metrics. Its balanced precision and recall, as indicated by the high F1-score, demonstrate its effectiveness in correctly classifying defective instances while minimizing false positives. The integration of Bidirectional LSTM, BERT, Static Code Metrics, AST Tokens, and Source Code Features allowed the model to leverage both structural and textual aspects of source code, leading to its enhanced defect prediction accuracy.

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

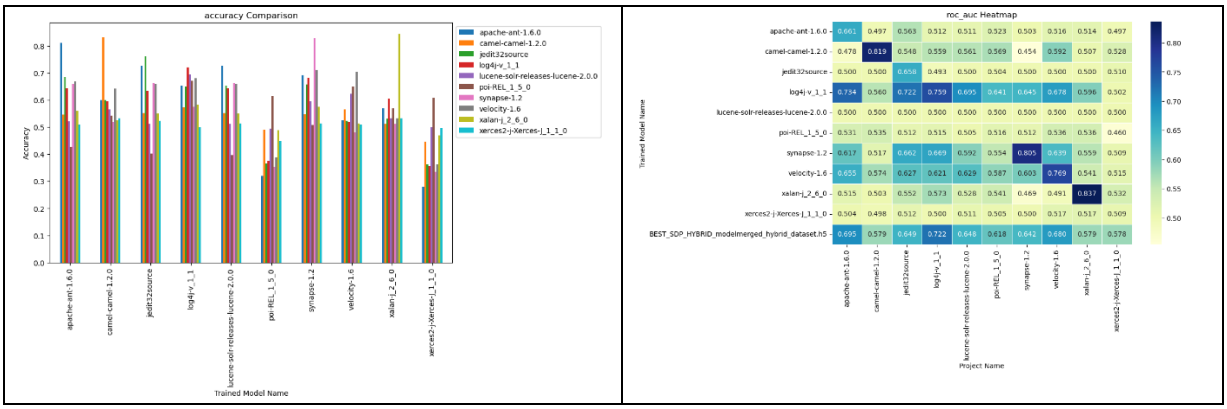

Figure 7.1 Cross-Project Defect Prediction

The proposed defect prediction model 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 1, 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, recall, F1 Score and AUC-ROC, 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.

8. Conclusion

In summary, this study successfully introduced an innovative hybrid machine learning model that combines Bidirectional LSTM, BERT-based language models, and static code metrics for software defect prediction. This approach effectively addressed both the structural and textual aspects of source code, resulting in superior defect prediction accuracy compared to conventional methods. The research highlights the capacity of machine learning techniques to revolutionize defect prediction, contributing to its proactive integration into software development and subsequently enhancing software quality, user satisfaction, and developer efficiency. The findings from this study hold the potential to drive further advancements in the field, making defect prediction a more robust and efficient process in the constantly evolving realm of software development.

9. Future Work

While this study has shown promising results, there are areas for future research and improvement. One avenue is to explore methods for dynamically adapting the hybrid model to changing software projects and their unique defect patterns, increasing its applicability across diverse domains and project scenarios. Ongoing advancements in defect prediction, coupled with the evolving field of machine learning, have the potential to transform software quality assurance and facilitate the development of high-quality software in the ever-changing software development environment.

References

[1] Alami, A., Cohn, M. L., & Wasowski, A. (2019). Why does code review work for open source software communities? *Proceedings of the 41st International Conference on Software Engineering*, 1073–1083. https://doi.org/10.1109/ICSE.2019.00111

[2] Alsolai, H., & Roper, M. (2019). A Systematic Review of Feature Selection Techniques in Software Quality Prediction. *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, 1–5. https://doi.org/10.1109/ICECTA48151.2019.8959566

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

[4] Catherine, J. M., & Djodilatchoumy, S. (2021). Multi Layer Perceptron Neural Network with Feature Selection for Software Defect Prediction. *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*, 228–232. https://doi.org/10.1109/ICIEM51511.2021.9445350

[5] Chen, L., Fang, B., Shang, Z., & Tang, Y. (2015). Negative samples reduction in cross-company software defects prediction. *Information and Software Technology*, *62*, 67–77. https://doi.org/10.1016/j.infsof.2015.01.014

[6] Elahi, E., Kanwal, S., & Asif, A. N. (2020). A new Ensemble approach for Software Fault Prediction. *2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 407–412. https://doi.org/10.1109/IBCAST47879.2020.9044596

[7] Felix, E. A., & Lee, S. P. (2017). Integrated Approach to Software Defect Prediction. *IEEE Access*, *5*, 21524–21547. https://doi.org/10.1109/ACCESS.2017.2759180

[8] Ge, J., Liu, J., & Liu, W. (2018). Comparative Study on Defect Prediction Algorithms of Supervised Learning Software Based on Imbalanced Classification Data Sets. *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 399–406. https://doi.org/10.1109/SNPD.2018.8441143

[9] Giray, G., Bennin, K. E., Köksal, Ö., Babur, Ö., & Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software*, *195*, 111537. https://doi.org/10.1016/j.jss.2022.111537

[10] He, P., Li, B., Liu, X., Chen, J., & Ma, Y. (2015a). An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, *59*, 170–190. https://doi.org/10.1016/j.infsof.2014.11.006

[11] He, P., Li, B., Liu, X., Chen, J., & Ma, Y. (2015b). An Empirical Study on Software Defect Prediction with a Simplified Metric Set. *Information and Software Technology*, *59*, 170–190. https://doi.org/10.1016/j.infsof.2014.11.006

[12] Ibarguren, I., Pérez, J. M., Mugerza, J., Rodriguez, D., & Harrison, R. (2017). The Consolidated Tree Construction algorithm in imbalanced defect prediction datasets. *2017 IEEE Congress on Evolutionary Computation (CEC)*, 2656–2660. https://doi.org/10.1109/CEC.2017.7969629

[13] *ISO/IEC 9126-1:2001—Software engineering—Product quality—Part 1: Quality model*. (n.d.). Retrieved July 23, 2023, from https://www.iso.org/standard/22749.html

[14] Jin, C., & Jin S.-W.(2015). Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization. *Applied Soft Computing*, *35*, 717–725. https://doi.org/10.1016/j.asoc.2015.07.006

[15] Jing, X.-Y., Wu, F., Dong, X., & Xu, B. (2017). An Improved SDA Based Defect Prediction Framework for Both Within-Project and Cross-Project Class-Imbalance Problems. *IEEE Transactions on Software Engineering*, *43*(4), 321–339. https://doi.org/10.1109/TSE.2016.2597849

[16] Kaur, A., Kaur, K., & Kaur, H. (2015). An investigation of the accuracy of code and process metrics for defect prediction of mobile applications. *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, 1–6. https://doi.org/10.1109/ICRITO.2015.7359220

[17] Lang, C., Li, J., & Kobayashi, T. (2021). Software Defect Prediction via Multi-Channel Convolutional Neural Network. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 543–554. https://doi.org/10.1109/QRS54544.2021.00065

[18] Laradji, I. H., Alshayeb, M., & Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, *58*, 388–402.

[19] Nagendram, S., Singh, A., Harish Babu, G., Joshi, R., Pande, S.D., Ahammad, S.K.H., Dhabliya, D., Bisht, A. Stochastic gradient descent optimisation for convolutional neural network for medical image segmentation (2023) Open Life Sciences, 18 (1), art. no. 20220665, . https://doi.org/10.1016/j.infsof.2014.07.005

[20] Lear, A., Dada, E., Oyewola, D., Joseph, S., Dauda, A., Bassi, S., & Baba, A. (2021). *Ensemble Machine Learning Model for Software Defect Prediction*. *2*, 11–21.

[21] Li, L., & Leung, H. (2011). Mining Static Code Metrics for a Robust Prediction of Software Defect-Proneness. *2011 International Symposium on Empirical Software Engineering and Measurement*, 207–214. https://doi.org/10.1109/ESEM.2011.29

[22] Liang, H., Yu, Y., Jiang, L., & Xie, Z. (2019). Seml: A Semantic LSTM Model for Software Defect Prediction. *IEEE Access*, *7*, 83812–83824. https://doi.org/10.1109/ACCESS.2019.2925313

[23] Malhotra, R., Bahl, L., Sehgal, S., & Priya, P. (2017). Empirical comparison of machine learning algorithms for bug prediction in open source software. *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, 40–45. https://doi.org/10.1109/ICBDACI.2017.8070806

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

[25] Mendez, C., Padala, H. S., Steine-Hanson, Z., Hilderbrand, C., Horvath, A., Hill, C., Simpson, L., Patil, N., Sarma, A., & Burnett, M. (2018). Open source barriers to entry, revisited: A sociotechnical perspective. *Proceedings of the 40th International Conference on Software Engineering*, 1004–1015. https://doi.org/10.1145/3180155.3180241

[26] Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., Turhan, B., & Zimmermann, T. (2013). Local versus Global Lessons for Defect Prediction and Effort Estimation. *IEEE Transactions on Software Engineering*, *39*(6), 822–834. https://doi.org/10.1109/TSE.2012.83

[27] Mori, T., & Uchihira, N. (2019). Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering*, *24*(2), 779–825. https://doi.org/10.1007/s10664-018-9638-1

[28] Motogna, S., Lupsa, D., & Ciuciu, I. (2019). A NLP Approach to Software Quality Models Evaluation. In C. Debruyne, H. Panetto, W. Guédria, P. Bollen, I. Ciuciu, & R. Meersman (Eds.), *On the Move to Meaningful Internet Systems: OTM 2018 Workshops* (pp. 207–217). Springer International Publishing. https://doi.org/10.1007/978-3-030-11683-5_24

[29] Nalini, C., & Murali Krishna, T. (2020). An Efficient Software Defect Prediction Model Using Neuro Evalution Algorithm based on Genetic Algorithm. *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*, 135–138. https://doi.org/10.1109/ICIRCA48905.2020.9182869

[30] Okutan, A., & Yıldız, O. T. (2014). Software defect prediction using Bayesian networks. *Empirical Software Engineering*, *19*(1), 154–181. https://doi.org/10.1007/s10664-012-9218-8

[31] Omri, S., & Sinz, C. (2020). Deep Learning for Software Defect Prediction: A Survey. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 209–214. https://doi.org/10.1145/3387940.3391463

[32] Pardalos, P. M., Rasskazova, V., & Vrahatis, M. N. (Eds.). (2021). *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems* (Vol. 170). Springer International Publishing. https://doi.org/10.1007/978-3-030-66515-9

[33] Prabha, C. L., & Shivakumar, N. (2020). Software Defect Prediction Using Machine Learning Techniques. *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, 728–733. https://doi.org/10.1109/ICOEI48184.2020.9142909

[34] *PROMISE-backup/bug-data at master · feiwww/PROMISE-backup*. (n.d.). GitHub. Retrieved December 25, 2021, from https://github.com/feiwww/PROMISE-backup

[35] Qiao, L., Li, X., Umer, Q., & Guo, P. (2020). Deep learning based software defect prediction. *Neurocomputing*, *385*, 100–110. https://doi.org/10.1016/j.neucom.2019.11.067

[36] Shailesh, T., Nayak, A., & Prasad, D. (2018). Performance Prediction of Configurable softwares using Machine learning approach. *2018 4th International Conference on Applied and Theoretical Computing and Communication Technology (ICATccT)*, 7–10. https://doi.org/10.1109/iCATccT44854.2018.9001957

[37] Siers, M. J., & Islam, M. Z. (2015). Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. *Information Systems*, *51*, 62–71. https://doi.org/10.1016/j.is.2015.02.006

[38] Souri, A., Mohammed, A. S., Yousif Potrus, M., Malik, M. H., Safara, F., & Hosseinzadeh, M. (2020). Formal Verification of a Hybrid Machine Learning-Based Fault Prediction Model in Internet of Things Applications. *IEEE Access*, *8*, 23863–23874. https://doi.org/10.1109/ACCESS.2020.2967629

[39] Tran, H. D., Hanh, L. T. M., & Binh, N. T. (2019). Combining feature selection, feature learning and ensemble learning for software fault prediction. *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, 1–8. https://doi.org/10.1109/KSE.2019.8919292

[40] Walunj, V., Gharibi, G., Alanazi, R., & Lee, Y. (2022). Defect prediction using deep learning with Network Portrait Divergence for software evolution. *Empirical Software Engineering*, *27*(5), 118. https://doi.org/10.1007/s10664-022-10147-0

[41] Xu, Z., Li, L., Yan, M., Liu, J., Luo, X., Grundy, J., Zhang, Y., & Zhang, X. (2021). A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software*, *172*, 110862.

[42] Kamble, S.D., Saini, D.K.J.B., Jain, S., Kumar, K., Kumar, S., Dhabliya, D. A novel approach of surveillance video indexing and retrieval using object detection and tracking (2023) Journal of Interdisciplinary Mathematics, 26 (3), pp. 341-350. https://doi.org/10.1016/j.jss.2020.110862

[43] Zheng, W., Gao, J., Wu, X., Liu, F., Xun, Y., Liu, G., & Chen, X. (2020). The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software*, *168*, 110659. https://doi.org/10.1016/j.jss.2020.110659