Software defect prediction at file level: A hybrid deep learning approach combining source code metrics and semantic features

Esrael GEREMEW, Mekonnen WAGAW*

Software Engineering Program, Bahir Dar Institute of Technology, Bahir Dar University, Bahir Dar, Ethiopia

monalitha@gmail.com (*Corresponding author)

Abstract: The necessity for trustworthy Software Defect Prediction (SDP) models is highlighted by the increasing complexity of contemporary software systems. These models facilitate the effective use of scarce testing resources by early detection of potentially defective modules. Although deep learning has demonstrated promise in learning characteristics from source code, the efficacy of current methods is generally limited by their reliance on a particular sort of information, such as hand-crafted code metrics or semantic features from code structure. One of the biggest challenges is still integrating several data types into a single, discriminative feature set. In order to forecast file-level defects, this study presents a unique approach that blends semantic characteristics with source code metrics. In addition to Combined Defect Data Modelling (CDDM) we suggest Learning Hybrid Feature Representation (LHFR), a deep neural network model. LHFR combines a Multi-Layer Perceptron (MLP) to learn from manually constructed metrics with a Bidirectional Long Short-Term Memory (Bi-LSTM) network to extract semantic features from Abstract Syntax Trees (ASTs). With an average F-measure of 69.08%, LHFR outperforms models based solely on metrics or semantic characteristics when tested on 12 open-source Java projects. A new combined dataset, an improved feature set and a hybrid representation strategy that significantly enhances fault detection performance are among the contributions.

Keywords: Software Defect Prediction, Deep learning, Source code metrics, Semantic features, Abstract Syntax Trees, Feature representation learning

1. Introduction

The objective of high program dependability, which requires substantial testing and debugging expenses, is a fundamental aspect of modern software engineering (Son, Kim & Kim, 2019). Given the time and money constraints, it is imperative that these activities be properly prioritized. SDP (Software Defect Prediction) addresses this need by developing models that classify software modules (such files and classes) as defective or non-defective or predict their defect likelihood (Jayanthi, Florence & Arya, 2017). By enabling developers and testers to focus on high-risk components, SDP enhances software quality and reduces development costs (Allamanis et al., 2018; Qiu et al., 2025).

A typical SDP process consists of three key steps: (1) gathering characteristics from software artifacts, including source code; (2) utilizing machine learning (ML) techniques to create classification models and (3) using the learned model for prediction (Jayanthi & Florence, 2017). According to Di Nucci et al. (2018), SDP research has frequently followed two paths: one focused on building ML and DL models for better classification, while the other focuses on inventing novel features or inventive combinations of traditional features for better fault characterisation.

In the past, SDP has prioritized manually developed code metrics, such as Halstead's software science metrics, McCabe's cyclomatic complexity and the Chidamber and Kemerer (CK) object-oriented metrics suite (Chidamber & Kemerer, 1994). These measurements provide quantifiable insights into code attributes including size, complexity and coherence. However, they often fail to capture the rich contextual information included in source code, such as its grammatical structure and semantic meaning (Li, Zhang & Xie, 2017).

In parallel, many machine-learning models, including Random Forests, Decision Trees and Logistic Regression, have been applied to SDP (Zhou, Zhang & Sun, 2019). In more recent times, DL models have emerged as efficient techniques for automatically extracting features from challenging data. Their ability to model highly non-linear connections makes them particularly well-suited for learning from source code representations (Ferenc, Zoltan & Gabor, 2020).

https://doi.org/10.33436/v36i1y202606

Unlike real language, source code contains a precisely detailed grammar that may be converted into an Abstract grammar Tree (AST). ASTs have shown promise in software engineering activities such as defect prediction (Fan et al., 2019) and code completion (Li et al., 2017a) because they capture the structural and semantic essence of code. DL models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have been used in recent research (Nivetha & Kavitha, 2019; Pan, Li & Yang, 2019) to automatically learn semantic features from ASTs, showing performance gains over traditional metric-based techniques. Previous AST-based methods focused on manually designed structural elements.

Despite these advancements, there is still a significant disparity. Most state-of-the-art DL-based SDP methods employ either code metrics or semantic features, but not both, in a completely integrated manner. Getting a high-quality Combined Defect Data (CDD) collection and using this combined data to create an effective feature representation are two important challenges. We suggest that code metrics and semantic features capture complementary aspects of source code: metrics quantify visible properties, while semantics uncovers the underlying logic and structure. A model utilizing both should yield a more robust and discriminative representation.

As a result, our experiment design focused on the following research topics: (a) How does the quality of the dataset obtained using CDDM differ from that of the original PROMISE dataset? (b) How does the proposed combination technique (SDP-CMSF) improve defect prediction performance compared to baseline methods using single characteristics? (c) How does SDP-CMSF compare to state-of-the-art combined feature-based fault prediction methods? (d) What effects do different parameter settings (such as feature representations and class imbalance treatment) have on the prediction performance of SDP-CMSF?

This work makes the following noteworthy contributions: in order to learn a unified feature representation from the combined metric and semantic data, we design a novel neural network architecture that combines an MLP and a Bi-LSTM. Additionally, we propose a method to create a file-level defect dataset that combines AST-based semantic features extracted directly from Java source files with static code metrics from the PROMISE repository. In order to assess our method against many baselines and show its efficacy in enhancing prediction performance, we also carried out comprehensive tests on twelve open-source Java projects.

The rest of the paper is organized in this manner: Section 2 provides background information and a review of pertinent literature; Section 3 describes our proposed method, which involves both data collection and model building; Section 4 displays our experimental setup and results; Section 5 discusses the findings, validity issues and suggestions for more study.

2. Literature review

2.1. Software defect prediction

Predicting the defect-proneness of software modules (such as files, classes, and procedures) is a classification job in software engineering called SDP (Zhou, Zang & Sun, 2019; Jiang et al., 2026). The process, as shown in Figure 1, typically involves:

1. **Data Extraction:** Mining historical repositories to collect software modules and label them as defective or clean based on bug reports.
2. **Feature Extraction:** Calculating features for each module, which can be static code metrics, process metrics or code change metrics.
3. **Model Training**: Using ML algorithms to train a classifier on the labeled data with extracted features.
4. **Prediction:** Applying the trained model to new, unlabeled modules to identify potential defects.

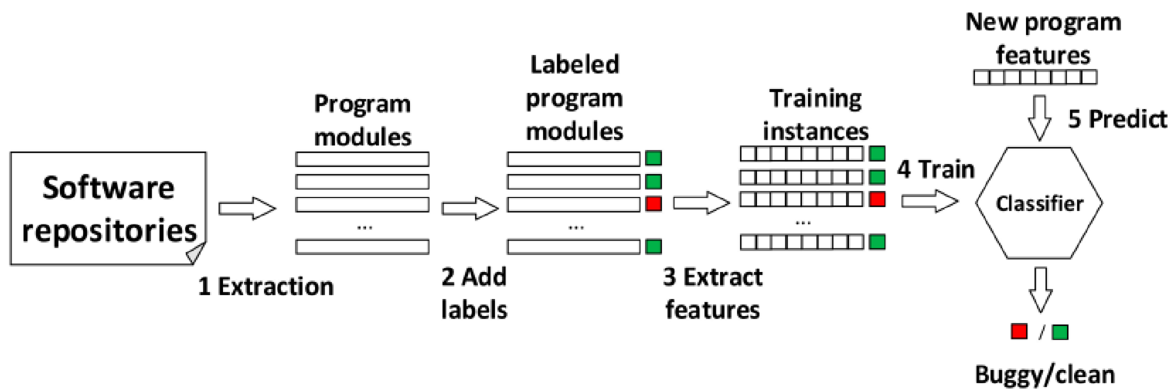


Figure 1. A typical software defect prediction process based on machine learning (Pan et al., 2021)

Static software metrics including LOC, Halstead, McCabe, and CK metrics are the mainstay of metric-based SDP techniques. To learn higher-level representations from these manually constructed measures, deep neural networks—such as autoencoders, deep belief networks and fully connected neural networks—have become popular. Tong, Zang & Xu (2018), for instance, used ensemble learning in conjunction with stacked denoising autoencoders to improve classification performance and reduce class imbalance. To enhance discriminative feature learning Xu *et al.* (2019) presented hybrid loss functions that include weighted cross-entropy and triplet loss.

Static metrics are unable to capture syntactic structure and semantic linkages buried in program logic, therefore even if these approaches show increased prediction performance, they are fundamentally flawed in their comprehension of source code. As a result, metric-based approaches frequently have trouble generalizing to other projects and programming paradigms.

Metric-Based SDP: uses static code characteristics like Lines of Code (LOC) and Cyclomatic Complexity (CCN) as well as object-oriented metrics like WMC, DIT and NOC (Chidamber & Kemerer, 1994). These approaches are predicated on the idea that complex code is more prone to mistakes. Table 1 compiles common object-oriented metrics utilized in SDP.

Table 1. Common object-oriented metrics for defect prediction

Metric	Description
WMC	Weighted Methods per Class: Number of methods in a class.
DIT	Depth of Inheritance Tree: Length of the longest path to the root.
NOC	Number of Children: Count of immediate subclasses.
CBO	Coupling Between Objects: Number of classes coupled to a given class.
RFC	Response For a Class: Number of unique methods invoked.
LCOM	Lack of Cohesion in Methods: Measures the dissimilarity of methods.
LOC	Lines of Code: Total lines in a class.

Context-Based SDP: extracts feature from ASTs that are produced by the code's structure and meaning using DL models. These techniques aim to find patterns and contexts that are hidden by traditional measures (Fan et al., 2019; Liang et al., 2019).

2.2. Semantic-based deep learning approaches

Recent research has concentrated on learning semantic representations directly from source code utilizing abstract syntax trees (ASTs), token sequences and control flow structures in order to get beyond the drawbacks of handmade metrics. For this, CNNs, LSTM networks and RNNs have all been used extensively.

In order to extract long-range relationships from token, sequences produced from AST, (Fan et al., 2019) used attention-based RNN models. Similarly, to learn semantic representations, (Liang, Zang & Li, 2019) used LSTM networks in conjunction with continuous bag-of-words embedding. Traditional metric-based methods were much outperformed by CNN-based approaches (Li et al., 2017a; Wehr et al., 2019) in capturing hierarchical structural patterns from ASTs.

Despite these developments, semantic-only methods sometimes ignore useful quantitative signs offered by static code metrics, such size and complexity measurements, which are still reliable indicators of problem proneness.

2.2.1. Software defect prediction

DL models have gained popularity in SDP due to their ability to automatically learn features from raw or semi-processed data (Ferenc, Zoltan & Gabor, 2020). We categorize them according to the feature sources of DL-based SDP research.

2.2.2. Defect prediction with hand-crafted features

These investigations use DL models to obtain higher-level representations from traditional code metrics. For instance, Tong, Zang & Xu (2018) demonstrated that utilizing Stacked Denoising Auto-encoders to train deep representations from NASA MDP datasets may enhance prediction. Similarly, Xu *et al.* (2019) used a deep network with a hybrid loss function to learn discriminative features from PROMISE and AEEEM datasets in order to overcome class imbalance.

2.2.3. Defect prediction with deep semantic features

In this area of research, DL is used to directly learn features from code representations like ASTs. Wang, Lo & Jiang (2016) used Deep Belief Networks (DBNs) to AST sequences for cross-version and within-project prediction. Li et al. (2017a) suggested a CNN-based model that outperforms previous DBN models by learning from AST tokens. Dam, Tran & Pham (2018) and Tufano et al. (2018) looked at LSTM and tree-based LSTM models employing ASTs for fault prediction. Fan et al. (2019) demonstrated significant improvements in F1-score and AUC by using a Bi-LSTM model with an attention mechanism to learn syntactic and semantic properties.

2.2.4. Defect prediction with hybrid features

Hybrid feature learning techniques that include semantic features and static code metrics have been studied in more recent studies. Significant gains were made when Nivetha & Kavitha (2019) combined certain static measurements with bidirectional RNN-based semantic features. A transfer CNN approach was presented by Qiu et al. (2019) that employs kernel-based distribution adaption to combine handmade measurements with deep semantic features.

Existing methods usually use feature concatenation, which greatly increases feature dimensionality and introduces redundancy, resulting in increased computational complexity and possible overfitting, even if these hybrid models show improved predictive accuracy. Furthermore, the effectiveness of many current hybrid approaches in integrating disparate feature representations is limited by the absence of explicit feature alignment procedures.

Recognizing the limitations of using a single feature type, several studies have combined hand-crafted metrics with deep semantic features. (Li, Zhang and Xie, 2017b) and Qiu et al. (2019) merged traditional measurements with CNN-trained features for classification using a merge operator and logistic regression. Although they did not always improve accuracy, they saw a decline in false positives. Nivetha and Kavitha (2019) proposed a Bidirectional RNN Language Model (BRNNLM) that showed improvements in precision by merging conventional metrics using cross-entropy with semantic information from code.

Our work expands on this third area. However, unlike earlier work that often employs simple concatenation or late fusion, our LHFR model uses a deep, integrated network to learn a non-linear, unified representation from both metric and semantic variables from the beginning, aiming for a more synergistic combination.

We provide a novel deep feature embedding and fusion approach that uses combined optimization of classification and embedding loss to map heterogeneous features into a common latent space in order to overcome these restrictions. In contrast to current techniques, our method

combines deep feature alignment and fusion instead of shallow concatenation, which improves generalization, reduces dimensionality and facilitates efficient representation learning.

2.3. Deep neural network architectures

Several important DL architectures are used in our suggested model. A simple neural network in which each neuron is connected to every other neuron in the layer above is called a Fully Connected Network (FCN) or Multilayer Perceptron (MLP).

The output of a neuron is given by:

$$yi=f(\sum j=1nWijxj+b) \quad (1)$$

where xj are inputs, Wij are weights, b is a bias and f is a non-linear activation function (e.g., ReLU, Sigmoid). MLPs are capable of learning complex, non-linear relationships in data (Zhang et at., 2019).

An RNN version called Long Short-Term Memory (LSTM) is designed to find long-range associations in sequential data. It uses a gating system (input, forget and output gates) to regulate the information flow. The main computations performed by an LSTM unit are (Hochreiter & Schmidhuber, 1997):

$$it = \sigma (Wixt + Uiht-1 + bi);$$
$$ft = \sigma (Wfxt + Ufh\,t-1 + bf);$$
$$ot = \sigma (Woxt + Uoht-1 + bo);$$
$$ut = \tanh (Wuxt + Uuht-1 + bu);$$
$$cst = it \odot ut + ft \odot ct-1;$$
$$ht = ot \odot \tanh (cst); \quad (2)$$

where it, ft and ot denote input, forget and output gates, respectively. There are four input weights Wu, Wi, Wf and Wo, corresponding to the unit input, the input gate, forget and output gates, respectively. There are also four recurrent weights Ui, Uu, Uf and Uo, as well as four bias terms bu, bi, bf and bo, respectively. The cst is a built-in memory cell of an LSTM node, which can maintain its internal state over multiple time steps; ht is a hidden state and ut is an input node. The weight matrix W is the weight between the input and the current hidden layer, while U is the weight between the current and previous hidden layer. The symbols σ and *tanh* represent the non-linear sigmoid function and Hyperbolic Tangent function, respectively. The ⊙ signifies element-wise multiplication.

Bidirectional LSTM (Bi-LSTM): This approach runs two separate LSTMs, one processing the sequence forward and the other backward, allowing the network to collect contextual information from both previous and future phases in the sequence (Fan et al., 2019). The final hidden state is often created by concatenating the outputs from both directions:

$$h_t' = \mathcal{Z}(\overline{h_t}, \overline{h_t}) \quad (3)$$

Three crucial research needs are evident from the data above:

1. Shallow Feature Integration: The mainstay of current hybrid models is direct feature concatenation, which does not ensure semantic alignment across representations based on metrics and semantics.
2. High-Dimensional Feature Burden: Feature fusion techniques greatly raise the dimensionality of inputs, which makes training more difficult and limits generalization.
3. Limited Latent Feature Learning: Only a small number of research investigate unified latent embedding spaces that can learn discriminative and compact joint representations.

We provide a novel deep feature embedding and fusion approach that uses combined optimization of classification and embedding loss to map heterogeneous features into a common

latent space in order to overcome these restrictions. In contrast to current techniques, our method combines deep feature alignment and fusion instead of shallow concatenation, which improves generalization, reduces dimensionality, and facilitates efficient representation learning.

2.4. Comparison with existing hybrid Software Defect Prediction models

Previous research on Software Defect Prediction (SDP) has investigated a variety of feature representation techniques, from deep semantic learning to static code metrics. In order to improve defect prediction using handmade software metrics, Tong, Zang & Xu (2018) used stacked denoising autoencoders and ensemble learning; nevertheless, their method lacked semantic information taken from source code, which limited its representational richness (Tong, Zang & Xu, 2018). In a similar vein, Xu *et al.* (2019) suggested deep neural networks with hybrid loss functions to enhance discriminative metric-based feature learning; however, their architecture did not make use of syntactic or semantic program structures and instead depended entirely on static metrics Xu *et al.* (2019).

Several researches concentrated on applying deep learning approaches to extract semantic representations from Abstract syntax Trees (ASTs) in order to get around the drawbacks of metric-based models. In order to improve predictive performance, Fan et al. (2019) developed attention-based Recurrent Neural Networks (RNNs) to extract long-range semantic dependencies from AST token sequences. However, their model failed to take into account complementary quantitative insights offered by conventional software metrics (Fan et al., 2019). Similarly, (Li, Zhang and Xie, 2017b) showed better performance than traditional methods by proposing a Convolutional Neural Network (CNN)-based framework to learn hierarchical structural patterns from AST representations; nevertheless, its prediction robustness was limited by the lack of metric integration (Li et al., 2018).

By combining static measures and semantic representations, more recent research investigated hybrid feature learning techniques. Although Nivetha & Kavitha's fusion technique depended on shallow feature concatenation, which raised dimensionality and caused feature redundancy, they were able to improve classification accuracy by combining bidirectional RNN-based semantic features with specific static code metrics (Nivetha & Kavitha, 2019). The direct concatenation of heterogeneous features imposed a high computational burden and increased the risk of overfitting (Qiu et al., 2019). Similarly, Qiu et al. (2019) proposed a transfer learning-based CNN framework that fused handcrafted metrics and deep semantic features via kernel-based distribution adaptation, achieving improved cross-project defect prediction.

As an alternative to these current methods, the suggested approach presents a feature fusion framework based on deep latent embeddings that aligns semantic representations and static code metrics in a common latent space, allowing for efficient feature integration while drastically lowering dimensionality and redundancy. By enabling the simultaneous optimization of classification and embedding loss, this architectural approach creates feature representations that are compact, discriminative and semantically aligned, which eventually improves prediction performance, robustness and scalability.

3. Methods and materials

3.1. Combined defect data modeling

Building a high-quality file-level dataset with both metric and semantic attributes is the initial stage.

3.1.1. Data collection

We employed twelve open-source Java apps from the publicly available PROMISE repository (PROMISE, 2016), a benchmark often used in SDP research. This repository contains 20 static code metrics (including WMC, DIT, NOC, CBO and LOC) and defect labels for various

software versions. We obtained the pertinent Java source code files from the associated repositories in order to extract ASTs.

3.1.2. Data preprocessing and integration

We developed CDDM, a Python-based framework, to integrate many data sources. Its procedure is as follows:

1. **Metric Data Extraction -** By parsing the PROMISE CSV files, we were able to extract the file name, the 20 static code metrics and the binary defect label for each instance of each project version.
2. **AST Sequence Data Extraction** - To locate every.java file, we searched through each project's source code folders. The Javalang parser was used to parse each file into an AST. After employing a Depth-First Traversal (DFT) technique to traverse the AST, 29 typical node types (such as MethodDeclaration, IfStatement, and VariableDeclarator) were gathered to create a series of tokens that maintain structural and semantic information. 10 displays a sample of the retrieved sequencing data.
3. **Data Integration and Tokenization** - We compared the recovered Java source files with the file names from the PROMISE dataset. We merged the metric data with the AST token sequence for matching files. A number sequence was then produced by mapping the textual AST tokens to integer indices. We used padding and truncation to a preset length (L, established by empirical analysis) in order to accommodate different sequence lengths. A project's final, integrated dataset consists of a set of instances, each of which is a tuple (File Name, 20 Metric Values, AST Token Sequence, Defect Label).

The final mapped dataset contained 13,885 file-level instances.

3.2. Learning Hybrid Feature Representation

The core of our approach is the LHFR model, a deep neural network designed to construct a unified representation from the combined defect data. The general architecture of SDP-CMSF (Software Defect Prediction using Code Metric and Semantic Features) is depicted in Figure 2.

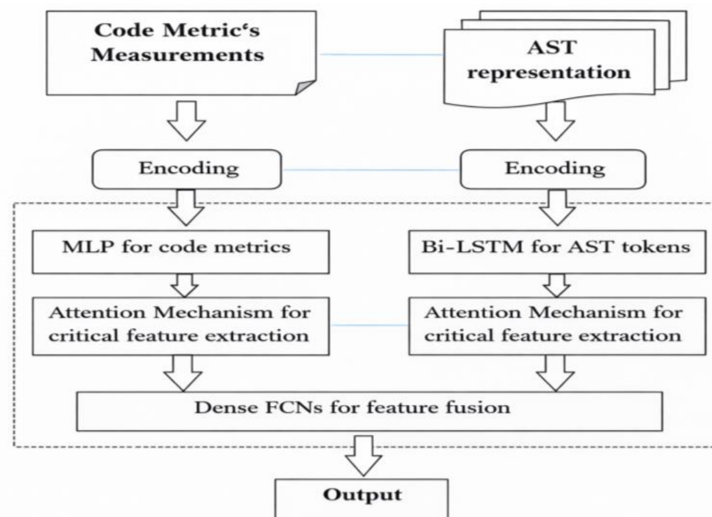


Figure 2. Overview of the proposed SDP using source code metrics and semantic features (SDP-CMSF) (Authors' own research)

3.2.1. Input representations

Metric Data Processing: The 20-dimensional vector of static code metrics in each file is normalized using Z-score normalization. This normalized vector serves as the raw input for the metric-learning branch of the network.

Semantic Data Processing: Each token in the string of integer tokens that make up the AST is transformed into a dense, 100-dimensional vector via an embedding layer. This embedding layer is initialized using a Word2Vec model that was trained on the whole corpus of AST sequences, capturing the semantic connections between code tokens. The raw input for the semantic-learning branch is the created embedding sequence.

3.2.2. Neural network architecture

The LHFR model consists of three main components:

Component A: Bi-LSTM learning with semantic characteristics. This branch processes the AST embedding sequence. The code structure is made up of several Bi-LSTM layers in order to extract long-range contextual correlations. An attention approach (Global Max Pooling) is used on top of the Bi-LSTM outputs to highlight the most crucial features for fault prediction. This component produces v_{sem}, a 128-dimensional semantic feature vector.

Component B: Metric Feature Learning (MLP). This branch processes the normalized metric vector. It is a standard MLP with many fully connected (dense) layers and non-linear activation functions (ReLU). Its objective is to use the manually created measurements to learn a non-linear transformation into a more discriminative feature space. This component produces v_{met}, a 15-dimensional metric feature vector.

Component C: FCN and Unified Representation Learning. A single 143-dimensional vector called $v_{concat}=[v_{sem};v_{met}]$ is created by concatenating the feature vectors from Components A and B. In order to learn a unified, high-level representation, v_{hybrid}, that combines information from both data modalities, this concatenated vector is then fed into another FCN (a sequence of dense layers). The last layer generates a probability between 0 and 1 that indicates the possibility that the file is flawed using a sigmoid activation function.

The model is trained end-to-end using the binary cross-entropy loss function:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i) \right] \quad (4)$$

where p_i is the expected probability for the i-th sample and y_i is the real label. To address the issue of class imbalance that is frequently present in defect datasets, we utilize class weights and stochastic gradient descent (SGD) using mini-batches.

4. Experiments and results

4.1. Experimental setup

The twelve Java projects that CDDM processed were used by us. For every project, we created a within-project prediction setting, utilizing the most recent version for testing and previous revisions for training. The dataset was split into 20% for validation and 80% for training during the model-building process.

4.1.1. Baseline models and evaluation metrics

To evaluate our approach, we compared it against several baselines:

- FBMF: Random Forest (RF) trained on Base Metric Features (all files from PROMISE).
- FMF: RF trained on Matched Metric Features (only matched .java files).
- LSF: The Bi-LSTM language model (Component A of LHFR) trained directly on AST sequences for prediction.
- FSF: RF trained on the 128-dimensional semantic features extracted from the pre-trained Bi-LSTM (Component A).

- FHCF: RF trained on Hand-Crafted Feature Combination (20 metrics + AST sequence length).
- FConcatF: RF trained on the simple concatenation of metric features and deep semantic features (Vconcat).

We used a Random Forest classifier with 8000 estimators as the primary conventional ML model for baselines due to its outstanding performance in SDP (Zhou, Zhang & Sun, 2019). To investigate the impact of feature representation depth, we report results from many levels of Component C for the LHFR model. Accuracy, Precision, Recall, F1-Score and Area are five popular measures. Each model was evaluated using the Area Under the ROC Curve (AUC).

4.1.2. Hyper-parameter selection and optimization strategy

Instead of choosing the hyper-parameters at random, a methodical tuning process was employed to choose all of the deep learning models used in this investigation. In the beginning, a broad search area was established using empirical best practices and previous SDP research. The training set was then subjected to a grid search technique and k-fold cross-validation ($k = 5$) in order to determine the best hyper-parameter configurations.

For example, the number of hidden layers was [2–4], the number of neurons per layer was [64, 128, 256, 512], the learning rate was [0.0001, 0.001, 0.01], the batch size was [16, 32, 64], the dropout rate was [0.2–0.5] and the embedding dimension was [50, 100, 200]. The validation AUC score with the greatest value was used to choose the final parameter configuration.

By avoiding overfitting and ensuring fair comparisons between models, this methodical tuning approach enhances the robustness and repeatability of the experimental findings.

4.2. Results and analysis

The mean and standard deviation of all assessment measures were given and each experiment was conducted ten times independently using different random seeds to guarantee the robustness and dependability of the data.

Additionally, 5-fold cross-validation was used to provide more accurate generalization estimates and lessen bias brought on by data partitioning. To determine if the observed improvements of the suggested strategy over baseline procedures were statistically significant, paired t-tests were performed at a 95% confidence level ($p < 0.05$). Furthermore, 95% CIs were calculated for the F1-score and AUC, which shed more light on the prediction models' consistency and stability. The reliability and repeatability of the experimental results are greatly enhanced by these statistical validation techniques.

We looked at the dataset produced by CDDM. Our approach successfully matched 13,885 Java source files to the pertinent defect occurrences in the PROMISE repository with a matching accuracy of 96.45%. Analysis revealed that 0.07% of the 470 mismatched instances were non-Java files and 3.2% were files that were no longer included in the current codebase. This implies that CDDM effectively eliminates extraneous or noisy data. A somewhat better-balanced learning dataset was also shown by the mapped dataset's defect rate (class imbalance), which was 34.44%, a 3.01% improvement over the original PROMISE dataset's 31.41%. The optimal AST sequence length $L = 1480$, which covers 98% of the problematic files while limiting computational cost, was also determined using cost-sensitive analysis. In summary, CDDM produces an improved, language-specific defect dataset that is more refined and perhaps more suited for training DL models than the raw PROMISE data.

The average results across all 12 projects for the different approaches are summarized in Table 2.

Table 2. Average defect prediction performance (%) across all projects

Model	Features Used	Accuracy	Precision	Recall	F1-Score	AUC
F_{BMF}	Base Metrics	65.97	76.60	65.97	66.81	69.73
F_{MF}	Matched Metrics	65.46	76.59	65.46	66.53	68.32
L_{SF}	Semantic (Bi-LSTM)	61.74	57.57	61.74	52.93	48.90
F_{SF}	Deep Semantic Features	66.12	76.42	66.12	67.02	70.42
F_{HCF}	Metrics + SeqLen	66.15	**77.07**	66.15	67.30	68.86
$F_{ConcatF}$	Metrics ⊕ Semantics	66.34	76.19	66.34	67.25	70.93
LHFR (Ours)	**HybridF (Unified)**	**67.57**	77.31	**67.57**	**69.08**	69.80

Metric Features: Models (FBMF) and (FMF) performed similarly, with (FBMF) having a little edge. This implies that static code measurements provide a rather strong basis.

Semantic Features: In this case, the end-to-end Bi-LSTM model (LSF) performed poorly for direct classification, particularly in Precision and AUC. But when an RF classifier (FSF) was trained utilizing its learned features, performance significantly increased, surpassing the metric-only models in Accuracy, Recall, F1-Score, and AUC. This demonstrates the value of deep semantic features, but only when combined with a trustworthy classifier like RF.

The simple hand-crafted combination FHCF showed a notable improvement in accuracy with the best score of 77.07%. This shows that even a basic contextual feature like AST sequence length may improve traditional measurements. With the highest AUC (70.93%), the simple concatenation of deep features (FConcatF) produced the best overall ranking of defective and non-defective files.

Our proposed LHFR model, which learns a unified representation (HybridF), had the best results in Accuracy (67.57%), Recall (67.57%), and F1-Score (69.08%). The F1-Score, which balances recall and accuracy, is often considered the most significant metric for imbalanced classification. The 1.83% absolute gain in F1-Score over (FConcatF) and the 2.27% improvement over the best individual-feature model (FSF), both of which are statistically significant, demonstrate the superiority of our deep feature fusion approach.

We assessed the efficacy of the LHFR model by gathering data from several levels of Component C (the FCN) and utilizing two class imbalance strategies: Cost-Sensitive (CS) learning and Balanced (using class weights). The results are shown in Table 3.

Table 3. Performance of LHFR using different feature representations and class-imbalance handling strategies

Features Rep.	Source Layer (Dim)	Strategy	Accuracy	Precision	Recall	F1-Score	AUC
THF	MLP Output (15)	CS	66.29	**77.44**	66.29	68.15	69.02
ConcatF	Concatenation (143)	CS	**66.34**	76.19	**66.34**	67.25	**70.93**
HybridF-1	FCN Layer 1 (64)	CS	67.48	77.29	67.48	69.00	69.80
HybridF-2	**FCN Layer 2 (32)**	**CS**	**67.13**	**77.18**	**67.13**	**68.89**	**69.59**

The unified representations (HybridF) consistently outperformed the basic concatenation (ConcatF) and modified metric features (THF) in terms of accuracy, recall and F1-score. This illustrates how the non-linear modification of the FCN efficiently generates a more powerful feature set for classification. The greatest F1-Score was obtained with a 64-dimensional unified representation.

The Cost-Sensitive (CS) technique regularly yielded results that were either equal to or slightly better than the Balanced strategy in almost every feature representation and statistic. This implies that explicitly quantifying the cost of misclassification during training is advantageous for SDP.

Figure 3 shows a radar chart of the average performance of the three main feature representations (THF, ConcatF, and HybridF), which unequivocally demonstrates that HybridF provides the best and balanced performance across most parameters.

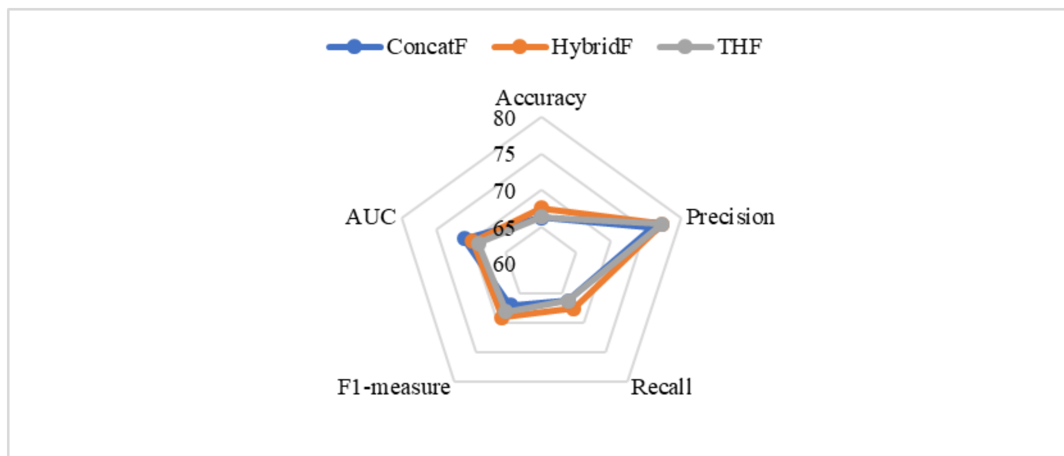


Figure 3. Radar charts of average indicator values for LCFR under combined features with three types of features transformation (Author's own research)

5. Discussion and conclusion

The most significant finding from our experiments is that a deep, trained combination of source code metrics and semantic features yields the best fault prediction model (HybridF). Semantics and metrics by themselves don't provide a complete view of the code. Metrics quantify "what" the code is (size, complexity), whereas semantics captures "how" the code is structured and operates. Our LHFR model effectively integrates these viewpoints. LHFR outperforms F_ConcatF, demonstrating that a non-linear, trained fusing of features is more efficient than a simple, linear concatenation.

Although the Bi-LSTM's raw semantic features were not particularly helpful for direct classification (LSF), they were highly beneficial when fed into a conventional classifier (FSF) or fused with metrics (LHFR). The FCN in Component C serves as an advanced feature fusion engine. This highlights how important it is to maintain the distinction between feature learning and classification, while the effectiveness of the cost-sensitive learning technique highlights the need to particularly address the intrinsic class imbalance in defect datasets.

5.1. Threats to validity

To reduce the possibility of errors in our implementation, we employed peer code review and standard libraries (Keras, Scikit-learn). The baseline model parameters were determined using either creative research or conventional methods.

Our study made use of twelve Java programs. While this provides a solid basis, more validation is required before implementing our findings in projects involving other programming languages or domains.

We used standard evaluation metrics including Accuracy, Precision, Recall, F1 and AUC that are present in SDP literature. Other metrics, such G-mean or effort-aware measures, may be investigated in future studies.

5.2. Conclusions

In this study we presented an all-encompassing approach that forecasts software problems at the file level by combining semantic features with source code metrics. We introduced the CDDM framework for building a high-quality, integrated defect dataset and the LHFR model, a hybrid deep learning network that learns a unified feature representation. Our empirical evaluation on a benchmark of Java systems demonstrates that our method consistently beats techniques based on individual features or simple feature combinations.

We plan to investigate more advanced AST representation learning techniques, such as Graph Neural Networks, which can more naturally model the tree structure of code, expand the LHFR framework for cross-project defect prediction by adding transfer learning techniques, incorporate a wider range of features, such as code change histories and developer activity metrics and investigate semi-supervised learning to leverage the abundance of unlabeled source code available in software repositories.

5.3. Limitations and future research directions

The suggested hybrid deep learning model performs admirably, however there are still a number of drawbacks. Initially, open-source Java programs from the PROMISE repository were used for the evaluation. Despite being extensively utilized in SDP research, these datasets could not accurately reflect industrial-scale software systems, which could restrict how broadly the suggested method can be applied.

Second, deep neural networks' computational complexity raises the hardware and training time requirements, which may make them impractical for use in contexts with limited resources. In order to increase scalability and efficiency, future research will look at lightweight architectures and model compression strategies.

Third, AST-based semantic extraction is the foundation of the existing feature combination system. Defect prediction performance may be further improved by using additional program representations, such as control-flow graphs (CFGs) and program dependency graphs (PDGs), even though AST collects rich structural and semantic information.

Lastly, there was a lack of thorough investigation of the effects of cross-project variability. Future research will use sophisticated domain adaption approaches and expand the suggested framework to cross-project defect prediction (CPDP) scenarios.

Resolving these issues would enhance the suggested model's scalability, robustness and industrial usability even further.

Author contributions

Conceptualization: E.G. and M.W.; Data Curation: E.G. and M.W.; Supervision: M.W. and E.G.; Validation: E.G. and M.W.; Writing—original draft: E.G. and M.W.; Writing—review and editing: M.W. and E.G. All authors have read and agreed to the published version of the manuscript.

Acknowledgements

The authors would like to acknowledge Bahir Dar University for supporting us by provision of resources.

Submited: 08 Decemer 2025; Revised: 02 Feruary 2026; Accepted: 03 Feruary 2026; Published: 31 March 2026.

REFERENCES

Ali, M., Mazhar, T., Al-Rasheed, A. et al. (2024) Enhancing software defect prediction: a framework with improved feature selection and ensemble machine learning. *PeerJ Computer Science*. 10, p. e1860.

Allamanis, M., Barr, E.T., Devanbu, P. & Sutton, C. (2018) A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*. 51(4), pp. 1–37.

Chidamber, S. R. & Kemerer, C. F. (1994) A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*. 20(6), 476–493.

Dam, H. K., Tran, T. & Pham, H. (2018) Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*. 44(10), 930–942.

Di Nucci, D., Oliveto, R., Panichella, A. et al. (2018) Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence*. 2(3), 1–12.

Fan, G., Diao, X., Yu, H., Yang, K. & Chen, L. (2019) Software defect prediction via attention-based recurrent neural network. *Scientific programming*. 2019(1), p.6230953.

Ferenc, R., Zoltan, P. & Gabor, B. (2020) Deep learning in static, metric-based bug prediction. *Array*. 6, 100021. https://doi.org/10.1016/j.array.2020.100021

Hochreiter, S. & Schmidhuber, J. (1997) Long short-term memory. *Neural Computation*. 9(8), 1735–1780.

Jayanthi, R. & Florence, L. (2017) A review on software defect prediction techniques using product metrics. *International Journal of Advanced Research in Computer Science*. 8(5), 123–130.

Jiang, F., Yu, X., Hu, Q. et al. (2026) An ensemble method using neighbourhood granular combination entropy for software defect prediction. *Information Processing & Management*. 63(2), 104483.

Li, J., Wang, Y., Lyu, M.R. and King, I. (2017) Code completion with neural attention and pointer networks. *arXiv* [preprint] arXiv:1711.09573.

Li, J., Zhang, Z. & Xie, T. (2017) 'Software defect prediction via convolutional neural network', in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 25–29 July 2017, Prague, Czech Republic*. Piscataway, NJ: IEEE, pp. 318–328.

Liang, H., Zhang, Y. & Li, H. (2019) A semantic LSTM model for software defect prediction. *IEEE Access*. 7, 83812–83824.

Nivetha, R. & Kavitha, S. (2019) Bidirectional recurrent neural network language model: Cross entropy churn metrics for defect prediction modelling. *International Journal of Advanced Research in Computer Science*. 10(3), 45–52.

Pan, C., Li, W. & Yang, Y. (2019) An improved CNN model for within-project software defect prediction. *Applied Sciences*. 9(10), Article 2076.

Pan, C., Lu, M. & Xu, B. (2021) An empirical study on software defect prediction using codebert model. *Applied Sciences*. 11(11), 4793.

PROMISE Repository (2016) *Open science software defect datasets*. Available at: http://openscience.us/repo [Accessed: 18th May 2025].

Qiu, S., E, B., He, J. and Liu, L. (2025) 'Survey of software defect prediction features', *Neural Computing and Applications*, 37(4), pp. 2113–2144.

Qiu, S., Xu, H., Deng, J., Jiang, S. & Lu, L. (2019) Transfer convolutional neural network for cross-project defect prediction. *Applied Sciences*. 9(13), p. 2660.

Son, J., Kim, S. & Kim, J. (2019) Empirical study of software defect prediction: A systematic mapping. *Information and Software Technology*. 106, 83–96.

Tong, H., Zhang, H. & Xu, Z. (2018) Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology*. 96, 94–111.

Tufano, M., Watson, C., Bavota, G. et al. (2018) Deep learning similarities from different representations of source code. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. ACM. pp. 226–236.

Wang, S., Lo, D. & Jiang, L. (2016) Automatically learning semantic features for defect prediction. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. pp. 297–308.

Wehr, D., Chen, Y., Lyu, M.R. et al. (2019) Learning semantic vector representations of source code via a Siamese neural network. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. pp. 1–10.

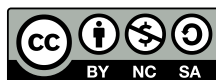
Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J. and Tang, Y. (2019) LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software*. 158, p. 110402.

Zhang, A., Lipton, Z.C., Li, M. & Smola, A.J. (2023) *Dive into deep learning*. Cambridge: Cambridge University Press.

Zhou, T., Zhang, L. & Sun, J. (2019) Improving defect prediction with deep forest. *Empirical Software Engineering*. 24, 3213–3243.

Esrael GEREMEW is a research assistant who completed his graduate studies in Software Engineering at the Bahir Dar Institute of Technology, Bahir Dar University, Ethiopia. He is actively engaged in academic research initiatives and has contributed to scholarly work focused on intelligent systems and data-driven solutions. His research interests include applications of artificial intelligence, data analytics, software systems development, computational modeling and machine learning.

Mekonnen WAGAW is an Assistant Professor and Head of the Software Engineering Program at the Bahir Dar Institute of Technology, Bahir Dar University, Ethiopia. He holds a Ph.D. in Information Systems and has extensive research, leadership and academic experience. His research interests include data analytics, software testing, software project management, cybersecurity, artificial intelligence and intelligent systems. In addition to supervising numerous postgraduate theses, he has published extensively in highly respected international journals and conferences. He actively pursues transdisciplinary research using cutting-edge computing technologies to adress both local and global challenges.


This is an open access article distributed under the terms and conditions of the
Creative Commons Attribution-NonCommercial 4.0 International License.