Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 12, No. 03, September 2026.

FR-CNN: SOFTWARE FAULT PREDICTION USING A NOVEL HYBRID HIERARCHICAL MODEL

Mehrasa M. Jouybari, Alireza Tajary, Mansoor Fateh and Esmaeel Tahanian

(Received: 16-May-2026, Revised: 29-Jul.-2026, Accepted: 19-Aug.-2026)

ABSTRACT

Developing high-quality software products is directly related to the absence of unseen faults. The software fault prediction (SFP) methods have been considered to reduce the overall testing costs and discover probable faults. Various machine-learning and deep-learning algorithms have been used to predict faults. However, both deliver varying results regarding SFP. This research proposes a novel hybrid model (FR-CNN) based on a fine-tuned fully connected deep neural network, a random forest and a convolutional neural network. Harnessing the strengths of combining these techniques, the proposed model aims to enhance prediction performance while reducing overfitting, resulting in a robust framework. We conduct extensive empirical studies to illustrate the effectiveness of the proposed model in predicting faults on six projects of the BugHunter dataset. The empirical results demonstrate that FR-CNN achieves competitive fault-prediction performance while requiring lower computational complexity and training time than recent deep-learning approaches. In particular, it improves the best previously reported traditional machine-learning result on the BugHunter benchmark by up to 19.95% in F1-score.

KEYWORDS

Software-fault prediction, Class imbalance, Deep learning, Ensemble learning, Machine learning, Hyper-parameters.

1. INTRODUCTION

The size and complexity of software in various applications are increasing and software engineers are tasked with developing high-quality software products. The existence of faults is one of the main issues that threaten the software quality and reliability [1] [2]. A software fault indicates a defect or deficiency in the system that leads to the final product's failure [3, 4]. Hidden and unforeseen faults are unavoidable, as they grow within the proper platform facilitated by modern software [5]. Over 50% of the costs of software development can be attributed to the detection and elimination of faults, as well as the removal of adverse consequences resulting from system failures [6].

Software Fault Prediction (SFP) is an approach that estimates the probability of faults in software units without running and testing the code. This strategy can effectively contribute to the production of higher-quality software [7]. Anticipating and early elimination of possible faults in different phases of development provides higher levels of system quality and improves maintenance effort [8]. SFP process allocates more effort to the most faulty modules and pays attention to diminishing the overall cost of testing [4], [9].

Various machine learning (ML) algorithms including Logistic Regression (LR) [10], Support Vector Machine (SVM) [11], K-Nearest Neighbors (KNN) [12], Decision Tree (DT) [13], Genetic Algorithm (GA) [14], Random Forest (RF) [15], Naive Bayes (NB) [16] and neural network [17], have been studied and investigated for SFP [12]. Nevertheless, these methods produce varying results in SFP. Deep learning (DL), as a subfield of ML, in a supervised and unsupervised manner, has achieved commendable success in various domains, including speech recognition and image processing [18]. DL is a proper approach to dealing with complex problems [19], demonstrating exemplary performance across both large and small datasets [20]. The DL model automatically extracts essential information using the features in the known project and after learning knowledge from them, it is used to predict faults in the unknown project. This approach affects the fault-prediction process. Convolution Neural Network (CNN) is a popular architecture of neural networks in SFP, capable of acquiring semantic features from software metrics [18].

Recent studies have further demonstrated that hybrid deep-learning architectures have become one of the most promising directions for software-fault prediction. Abdu et al. [21] proposed a hybrid deep-

M. M. Jouybari (Corresponding Author), A. Tajary, M. Fateh and E. Tahanian are with Faculty of Computer Engineering, Shahrood University of Technology, Shahrood, Iran. Emails: {mehrasa.modanlou; tajary; mansoor_fateh; e.tahanian}@shahroodut.ac.ir

learning framework that fuses semantic and traditional software features to improve defect prediction performance. Kaliraj et al. [22] introduced a dynamic classification framework for software-fault prediction that improves prediction robustness through adaptive classification strategies. More recently, Fei et al. [23] presented a multivariate heterogeneous hybrid deep-learning architecture that combines multiple complementary learning mechanisms for software-defect prediction, while Malhotra and Singh [24] proposed a dual-attention hierarchical BiGRU architecture that enhances software-defect prediction through hierarchical feature learning and attention mechanisms. These recent developments indicate that integrating complementary machine-learning and deep-learning models is becoming an increasingly effective strategy for improving software-fault prediction performance, which strongly motivates the proposed FR-CNN architecture.

The efficiency of the SFP on ML and DL techniques is also influenced by different fault datasets. The majority of well-known datasets for SFP, such as PROMISE [25] and NASA MDP [26], have relatively small sizes. Neural networks, particularly deep neural networks, are data-hungry models that their ability to learn increases with a larger amount of training data [27]. In this research, the BugHunter dataset [28], a recently introduced fault dataset comprising a sufficient number of data instances, has been used to satisfy the data requirements of DL-based models. This dataset consists of 15 popular open-source Java projects widely used by software engineers.

Although BugHunter is a recently introduced large-scale software fault-prediction dataset that provides a valuable benchmark for evaluating software fault-prediction models, its adoption in the literature remains relatively limited compared with widely used benchmark datasets, such as PROMISE and NASA. Existing approaches primarily rely on individual machine-learning models, conventional ensemble techniques or standalone deep-learning architectures, each of which has inherent limitations in exploiting complementary feature representations. Consequently, comprehensive evaluations of advanced hierarchical hybrid machine-learning and deep-learning models for method-level software fault prediction on the BugHunter dataset remain scarce. Motivated by recent advances in machine learning and deep learning, together with the availability of the large-scale BugHunter dataset, this study investigates whether a hierarchical hybrid architecture can improve software-fault prediction performance. Accordingly, we propose the FR-CNN model, which integrates a fine-tuned FC-DNN, Random Forest and a CNN meta-learner to exploit the complementary strengths of machine learning and deep learning for method-level software-fault prediction on six selected BugHunter projects.

The objective of this study is to develop and evaluate the proposed FR-CNN model for method-level software-fault prediction on six selected BugHunter projects and to compare its performance with the developed traditional machine-learning baseline models and the implemented recent state-of-the-art approaches under identical experimental conditions. The main contributions of this paper are as follows:

- We apply the random over-sampling technique on the BugHunter dataset to address the class-imbalance issue and enhance the training accuracy on both fault classes.
- We propose a new Fully Connected Deep Neural Network (FC-DNN) structure for predicting faults in six projects of the BugHunter dataset. We construct and evaluate multiple architectures for each project, investigating the effect of diverse hyper-parameters on improving the model performance until we achieve the optimal one.
- We propose the FR-CNN, a novel hierarchical hybrid model that combines a fine-tuned FC-DNN and the best-performing ML algorithm (Random Forest) using a CNN-based meta-learner. We further investigate the influence of kernel size, batch size and number of filters on the model performance.
- The proposed FR-CNN overcomes the limitations of existing CNN-based, ensemble-learning and hybrid software fault-prediction approaches through two key architectural innovations: (a) a hierarchical stacking architecture in which a fine-tuned FC-DNN and Random Forest (RF) serve as complementary base learners—each model learns different patterns from the software metrics and our feature importance analysis (Sub-section 4.6.2) confirms that they focus on diverse aspects of the feature space; and (b) a one-dimensional CNN as the meta-learner, enabling the learning of a combination rule over the base-learner outputs. An ablation study (Sub-section 4.6.7) shows that this learned combination outperforms parameter-free averaging on average across the six projects and that the CNN performs comparably to alternative learnable meta-classifiers (DNN, Logistic Regression) while remaining substantially more lightweight than the base-learner networks that it

combines. To the best of our knowledge, this specific FR-CNN hierarchical combination has not previously been investigated for
method-level software-fault prediction on the BugHunter dataset.

- We conduct comprehensive empirical studies by comparing the proposed FR-CNN with seven traditional machine-learning models and five recent state-of-the-art approaches using prediction performance, training time, testing time and model complexity. The results demonstrate that FR-CNN achieves competitive prediction performance while providing a favorable balance between computational efficiency, model complexity and training stability.

The paper proceeds as follows: Section 2 reviews previous works and contains the activities carried out in the SFP. Section 3 presents the proposed method. Section 4 pertains to the results and addresses the research questions. Section 5 deals with threats to validity of the proposed approach. Section 6 provides conclusions and describes future work.

2. RELATED WORKS

Numerous studies are available in the literature to classify software components as faulty or non-faulty, leveraging different techniques. These techniques were applied to various datasets at different granularity levels. Kumar et al. [8] collected and categorized various ML techniques, including NB, Decision Tree Regression and LR, for predicting fault with several performance evaluation measures. French et al. [28] developed a novel dataset called "BugHunter" to predict software faults using 11 traditional ML techniques from the Weka library, including RandomForest, J48, RandomTree, DecisionTable, SimpleLogistic and Logistic algorithms. Among the results presented for this dataset, the best F-measure at the method level was 75.73%, obtained by the RF algorithm for the ANTLR v4 project.

Cynthia et al. [29] developed a Feature Transformation for Improved Software Bug Detection Models that utilize a genetic algorithm to find the weighted transformation of the extracted features and identify buggy and non-buggy classes using ML models. They've reported their findings at the Class granularity and applied the undersampling technique to balance the dataset. They concluded that their proposed method can effectively separate bug data from non-bug data in the low-dimensional space of the transformed features.

Reference [30] introduced the Static Execute After (SEA)-based metrics instead of traditional software metrics to predict software faults at the method level of the BugHunter dataset. We will compare our proposed method with this model to understand how well our approach predict faulty methods. This research utilized random under-sampling to address the imbalance of two bug classes. The experimental results on the most frequently ML algorithms illustrated that the proposed SEA metrics are even suitable to predict future bugs of the project independently, if sufficient learning data is available.

So far, several studies have been carried out for SFP using ensemble learning-based techniques [31] [32] [33] [34]. These techniques are utilized to surpass the accuracy threshold of individual learners and increase the model's global predictive performance [35] [36]. According to [15], EL models, which are created by combining multiple ML classifiers, perform significantly better than individual classifiers such as decision trees, naïve Bayes or a multilayer perceptron for the fault prediction. Laradji et al. [37] formed a software fault-prediction method using the Average Probability Ensemble learning (APE) module with seven basic classifiers to predict faulty components.

Wang et al. [38] presented a Multiple Kernel ensemble-learning (MKEL) approach using EL, considering a multiple kernel classifier to improve software module classification and prediction power with faults. The experimental results showed the effectiveness of the proposed model on 12 NASA datasets. Pandey et al. [39] proposed an effective BPDET model for SFP using EL and deep representation techniques. The proposed approach was carried out in two phases, one was DL and the other was two layers of EL (TEL). The results of experiments on 12 public NASA datasets showed that BPDET is stable and the combination of deep representation and EL is more efficient and effective than the base methods and other traditional methods.

Although conventional ensemble-learning methods have demonstrated significant improvements over individual classifiers, recent research has shifted toward hybrid deep-learning frameworks that combine complementary feature representations. Abdu et al. [21] integrated semantic and traditional software

metrics using a hybrid deep-learning architecture and demonstrated that feature fusion significantly improves software-defect prediction. Likewise, Kaliraj et al. [22] proposed a holistic software fault-prediction framework incorporating dynamic classification mechanisms to improve prediction robustness. These studies indicate that combining different learning paradigms provides more informative feature representations than relying on a single prediction model.

DL techniques using convolutional layers, multilayer perceptron and Long Short-Term Memory (LSTM) automatically extract semantic features of the dataset [40]. Thus, these techniques create potent models that perform significantly in diverse domains, such as natural-language processing, computer vision and speech recognition. Reference [41] used the DL technique for fault prediction and presented the Seml framework. In this technique, the long short-term memory model performed fault prediction by automatically learning the semantic information of programs. The evaluation results on 13 Java open-source projects showed that the proposed model performs more promisingly compared to the state-of-the-art fault-prediction approaches. Alenezi et al. [42] employed character n-gram embedding with deep learning to improve source-code representation for software-vulnerability detection, achieving better prediction performance than previous approaches.

The latest research continues this trend. Fei et al. [23] proposed a multivariate heterogeneous hybrid deep-learning model that combines multiple deep-learning components for software-defect prediction, demonstrating improved prediction accuracy on complex software projects. Similarly, Malhotra and Singh [24] developed a dual-attention hierarchical gated BiGRU architecture that improves software defect prediction through hierarchical recurrent feature learning and attention mechanisms. Samal [43] proposed a hybrid optimization-based fault-prediction approach integrating particle-swarm optimization with fuzzy time series modeling. Collectively, these studies demonstrate that hybrid architectures capable of learning complementary representations have become one of the dominant research directions in software fault prediction.

In our research, we aim to combine these techniques to harness the advantages of ML and DL approaches while leveraging the strength of the ensemble model.

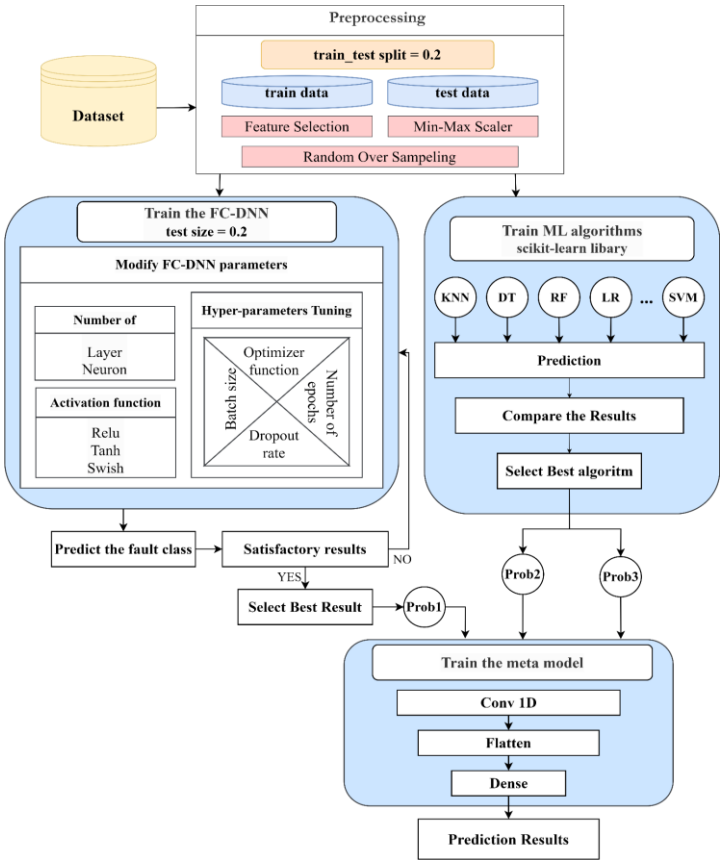

Figure 1. Overall workflow of the proposed hierarchical FR-CNN framework, illustrating the pre-processing stage, the FC-DNN and RF base learners and the CNN meta-learner for software-fault prediction.

3. PROPOSED APPROACH

This section introduces the proposed FC-DNN and FR-CNN models. The framework's overview is shown in Figure 1. To predict the number of faults at the method level of six Java projects from the BugHunter dataset, we employ the following steps. First, we conduct the pre-processing by choosing six datasets with various fault percentages, selecting and normalizing software metrics and addressing the class-imbalance problem. Second, we design multiple architectures for fault-prediction models using a DL structure and fine-tune them.

We also train various ML classifiers to be utilized in our hybrid model. Then, we develop an FR-CNN hybrid model by combining the optimized deep neural network and the best-performing ML model and feed them into a CNN structure. Our proposed approach includes two key phases: Data Pre-processing and Model Building.

3.1 Data Pre-processing

Preprocessing the dataset using different techniques improved the effectiveness of the model for learning [44]. To accomplish this task while strictly preventing data leakage, we applied a carefully ordered sequence of preprocessing steps. All data transformations and oversampling were performed using parameters learned exclusively from the training data after the train-test split, with the test data being transformed using the same parameters (e.g., min-max scaling), but never used to fit any transformations or oversampling, ensuring unbiased performance evaluation. The complete pre-processing workflow is as follows:

3.1.1 Feature Selection

The selected dataset contains a set of software metrics that describe information about each instance to predict the fault class at the method level. However, some of the included software metrics within the dataset, such as the 'Hash', 'LongName' and 'Migration14 Rules' metrics, do not possess sufficient value as reliable indicators for predicting faults [45]. Therefore, these metrics should be excluded, so that the selected metrics represent the original ones [46]. Additionally, we removed constant features (those with identical minimum and maximum values), as they provide no discriminative information. If the corresponding project at the method level is indicated by M and the fault class is denoted by Y, then the fault-prediction issue can be modeled according to Equation (1):

$$Y = f(M) \tag{1}$$

Some software metrics may have data with different scales, redundant features or useless information for prediction, which require pre-processing. Equation (2) shows the chosen features, where the M project being represented as a set of s_m, with each s_m denoting a software metric.

$$M = < sm_1, sm_2, \ldots, sm_n > \tag{2}$$

3.1.2 Null-value Verification

We verified that the dataset contains no null values to ensure data integrity before proceeding with further pre-processing steps.

3.1.3 Train-Test Split

To prevent data leakage, we first split the dataset into training and test sets using train_test_split from sklearn with a test size of 0.2. This ensures that the test data remains completely isolated and is never used for learning any transformations or model parameters throughout all subsequent pre-processing and model-building stages.

3.1.4 Binary Classification of the Output Class

In the selected dataset, the "Number of Bugs" software metric is considered as the label and exhibits a maximum of seven classes, which determine the count of faults within the output class. During the phase of prediction, the classification of the output class is considered as the presence or absence of faults for each individual sample. Hence, it is necessary to convert the label from the multi-class classification mode to the binary class classification. Therefore, instances characterized by the class label of 0 reflect the absence of faults, while instances associated with class labels ranging from 1 to 6 indicate the

presence of faults. Based on this, all labels have been modified to either 0 or 1. This conversion was applied to both training and test sets separately using the same mapping rule.

3.1.5 Data Normalization

Some ML techniques perform better on numerical datasets that have been scaled in a specific range. We performed the metric normalization on the dataset to ensure a fair balance in the range of data and equalize the effect of all dimensions. To achieve this, we applied the min-max scaler exclusively to all metrics of the training data, excluding the label. The same scaler fitted on the training data was then used to transform the test data. This was done using Equation (3), which rescaled the values of each software metric to a new range of 0 to 1.

$$\text{Normalization } (sm_i) = (sm_i - \min(sm)) / (\max(sm) - \min(sm)) \quad (3)$$

where sm_i indicates the corresponding software metric before applying normalization, max (sm) and min (sm) indicate the maximum and minimum values of sm_i and Normalization (sm_i) indicates the normalized values of sm_i. After normalization, the method M is defined according to Equation (4), which includes the set of sm_i' software metrics.

$$M = < sm_1', sm_2', \dots, sm_n' > \quad (4)$$

3.1.6 Addressing the Imbalance Issue

Unbalanced classes of the dataset reduce the classifier performance for the minority class. To overcome this, oversampling and undersampling techniques are applied to change the data space [47]. These techniques use strategies that either increase the instances of the minority class or reduce the instances of the majority class [48] to advance the final efficiency of the model [49]. In this article, the Random Over Sampling (ROS) technique was applied exclusively on the training instances to handle the imbalance distribution of the class labels, prevent the model from bias and enhance the performance of the learning process. The test data was not oversampled to maintain its original distribution and provide a realistic evaluation of model performance. Figure 2 and Figure 3 illustrate the count of two class instances in the BroadleafCommerce project, depicting the dataset's state prior to and following the usage of ROS on the training data. First, the number of instances was 826 for class 1 and 2941 for class 0. After applying ROS, the count of two class instances became equal.

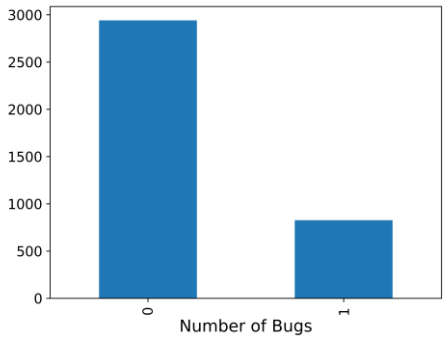

Figure 2. Class distribution of the BroadleafCommerce training data before applying Random oversampling (ROS), illustrating the original class imbalance.

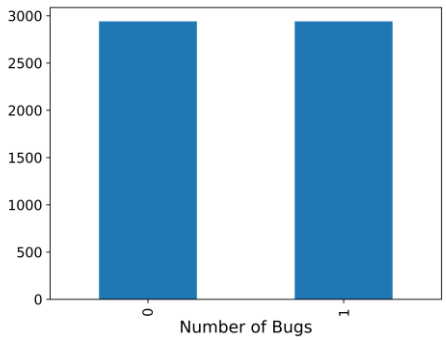

Figure 3. Class distribution of the BroadleafCommerce training data after applying Random oversampling (ROS), showing the balanced class distribution used for training.

It is important to emphasize that this sequential pre-processing workflow strictly prevents data leakage: the train-test split is performed before any transformations, the min-max scaler is fitted only on the training data and then applied to transform the train and test data and oversampling is applied exclusively to the training data. Consequently, no information from the test data is used during pre-processing or model training, ensuring an unbiased evaluation of model performance.

3.2 Model Building

3.2.1 Building the FC-DNN

The proposed FC-DNN, depicted in Figure 4, is built by a structure of dense and dropout layers. This fully connected feedforward deep neural network has an input layer, an output layer and between 1 and 4 hidden layers. There is a dropout layer after each layer except the last layer that prevents the model from overfitting by randomly dropping out the incoming and outgoing connections of units [50]. This regularization technique, combined with early stopping during training of the FR-CNN model, ensures robust generalization performance, as evidenced by the training and validation loss curves presented in Sub-section 4.6.4.

The model input is fixed according to the number of software metrics for each project which falls within the range of 56 to 59.

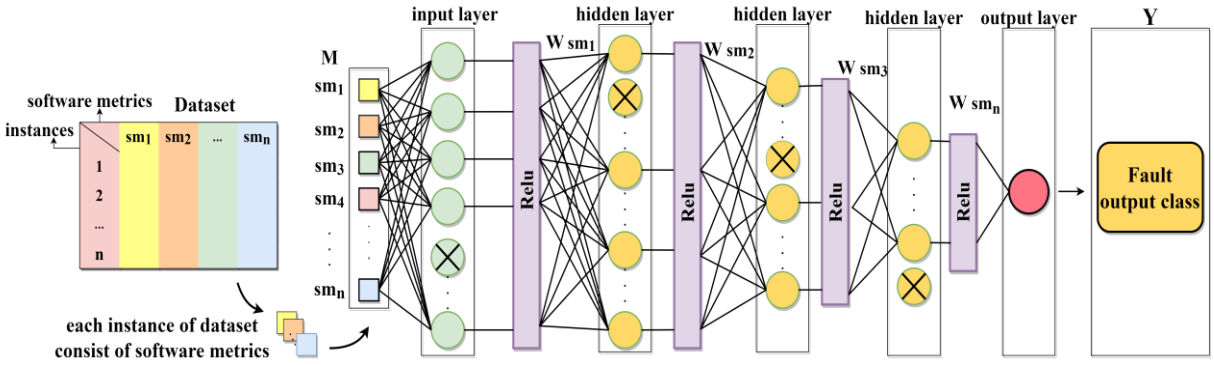

Figure 4. Architecture of the proposed FC-DNN base learner used for software-fault prediction.

The network output is considered in binary mode, with one neuron in the last layer. The model adjusts its weights using the optimizer function to reduce the error in the backpropagation stage [19]. The weights of layers are represented using $W_{sm_1}, W_{sm_2}, W_{sm_3}, \dots, W_{sm_n}$ in Figure 4. In the Result Section, we will conduct many experiments to design prediction models on each project, modify their parameters and record the best results. The impact of modifying parameters has an essential role in prediction performance [50]. The number of layers is a critical criterion in the architecture of the artificial neural networks and can influence the performance of the models [51]. We will modify and evaluate various hyper-parameters, different numbers of layers and activation functions to obtain the optimal design of the network and more accurate prediction models. Tuning the model hyper-parameters has been carried out by manipulating the number of epochs, batch sizes, dropout rates and optimizer functions until we reach satisfactory results. We will use three optimizer functions, Adam (Adaptive Moment Estimation), AdaGrad (Adaptive Gradients) and SGD (Stochastic Gradient Descent). We will utilize the Sigmoid activation function for the last layer of the network and test the ReLU, Tanh and Swish activation functions for other layers to choose the most suitable ones.

3.2.2 Building the FR-CNN Hybrid Model

In this Sub-section, the FR-CNN hybrid model is presented to improve the fault-prediction model's performance. The structure of the FR-CNN is depicted in Figure 1. This model is constructed by combining the FC-DNN, which is optimized with hyper-parameters and the RF, which is identified as the best-performing ML algorithm. Subsequently, a single meta-learner based on CNN is employed, comprising a convolutional layer, a flatten layer and a dense layer, serving as the final model. This model extracts semantic features from the combined features of both base models and determines the fault class for the entire hybrid model. To build the effective FR-CNN, we analyzed various

combinations of models. For instance, we investigated the potential of combining one or more ML classifiers with an FC-DNN serving as the base models, along with an ML, an FC-DNN or an CNN as a meta-learner. Through our evaluations, we found that the most effective approach for detecting software faults was achieved by combining an FC-DNN with an RF and putting a CNN at the end of it. Among various ML models that we established, the standalone RF exhibited superior performance in the hybrid model. To leverage its strengths, we combined it with a FC-DNN model as the base models. By combining these two models of different natures and distinct characteristics, we aimed to cover some of their individual weaknesses and enhance overall performance. In the proposed FR-CNN, the RF is constructed by aggregating the result of 50 decision trees. Given the existence of numerous outliers in the dataset under investigation, it is advantageous to adopt a clustering-oriented perspective during data analysis. Through the process of partitioning data among the decision trees and enabling independent decision-making by each tree, the RF algorithm exhibits a clustering-like behavior. Moreover, RF stands out as a robust EL-based algorithm that has an outlier-detection mechanism [52]. Additionally, we incorporated a fully connected DL-based algorithm into our methodology, leveraging its capability to automatically learn and capture distinguishing features and discriminative characteristics in the dataset. The utilization of this type of model has the potential for contributing towards the advancement of more accurate fault-prediction models [53]. In the Result Section, we will demonstrate the effectiveness of the presented FR-CNN on the BugHunter dataset. We also investigate the effect of different kernel sizes, batch sizes and numbers of filters on the performance of the FR-CNN. The workflow illustrated in Figure 1 and summarized in Algorithm 1 is mathematically formulated in Equations (5)–(8). Specifically, the formulation describes how the FC-DNN and Random Forest (RF) base learners generate prediction probabilities that are concatenated into meta-features and subsequently processed by the CNN meta-learner to produce the final software-fault prediction.
Let $x_i \in \mathbb{R}^d$ denote the feature vector of the i-th sample, where d is the number of software metrics after feature selection and $i = 1, 2, \dots, N$, with N representing the total number of samples. Let $f_{DNN}(\cdot)$, $f_{RF}(\cdot)$ and $f_{CNN}(\cdot)$ denote the trained FC-DNN, Random Forest and CNN models, respectively.

The FC-DNN estimates the probability that the i-th sample belongs to the faulty class as:

$$p_i^{DNN} = f_{DNN}(x_i), \qquad p_i^{DNN} \in [0,1], \tag{5}$$

where, p_i^{DNN} is the probability assigned by the FC-DNN to the faulty class.

The Random Forest predicts the probabilities of both classes as:

$$p_i^{RF} = [\, p_i^{(RF,0)}, p_i^{(RF,1)} \,] = f_{RF}(x_i), \tag{6}$$

where, $p_i^{(RF,0)}$ and $p_i^{(RF,1)}$ represent the predicted probabilities of the non-faulty and faulty classes, respectively, predicted by the Random Forest.

The outputs of the two base learners are concatenated to construct the meta-feature vector

$$z_i = [\, p_i^{DNN}, p_i^{RF} \,], \tag{7}$$

where, $z_i \in \mathbb{R}^3$ is the meta-feature vector, consisting of one probability generated by the FC-DNN and two class probabilities generated by the Random Forest and $[\cdot]$ denotes the vector concatenation operator.

The meta-feature vector is then provided as the input to the CNN meta-learner, which produces the final fault prediction according to o

$$\hat{y}_i = f_{CNN}(z_i), \qquad \hat{y}_i \in [0,1], \tag{8}$$

where, $\hat{y}_i$ denotes the final predicted probability that the i-th sample is faulty. A decision threshold of 0.5 is subsequently applied to assign the sample to either the faulty or non-faulty class.

Accordingly, for each software module, the FC-DNN predicts the probability of the faulty class, whereas the Random Forest estimates the probabilities of both classes. These three probability values are concatenated to form the meta-feature vector, which serves as the input to the CNN meta-learner. The CNN extracts higher-level interactions among these complementary predictions through its convolutional operation and produces the final software-fault prediction.

Algorithm 1 summarizes the complete workflow of the proposed FR-CNN stacking architecture. First,

the FC-DNN and Random Forest are independently trained using the same pre-processed training data. Their probability outputs are concatenated to construct the meta-feature representation, which is subsequently used to train the CNN meta-learner. During testing, the same sequence of operations is applied to unseen samples to generate the final fault prediction.

Algorithm 1. Pseudocode of the proposed FR-CNN stacking architecture

Input:
Preprocessed training set (X_{train}, Y_{train})
Preprocessed test set X_{test}

Output:
Predicted labels $\hat{Y}$

/ Train base learners */*
Train FC-DNN using (X_{train}, Y_{train})
Train Random Forest using (X_{train}, Y_{train})
/ Construct meta-features for training */*
$P_{\mathrm{DNN}_{train}} \leftarrow$ FC-DNN.predict (X_{train}) ▷ *Probability of the faulty class*
$P_{\mathrm{RF}_{train}} \leftarrow$ RF.predict_proba (X_{train}) ▷ *Probabilities of both classes*
$Z_{train} \leftarrow$ Concatenate $(P_{\mathrm{DNN}_{train}}, P_{\mathrm{RF}_{train}})$
/ Train meta-learner */*
Train CNN using (Z_{train}, Y_{train})
/ Predict on unseen instances */*
$P_{\mathrm{DNN}_{test}} \leftarrow$ FC-DNN.predict (X_{test})
$P_{\mathrm{RF}_{test}} \leftarrow$ RF.predict_proba (X_{test})
$Z_{test} \leftarrow$ Concatenate $(P_{\mathrm{DNN}_{test}}, P_{\mathrm{RF}_{test}})$
$\hat{Y} \leftarrow$ CNN.predict (Z_{test})
Return $\hat{Y}$

4. EVALUATION OF RESULTS

This Section presents a comprehensive evaluation of the proposed approach. It describes the dataset, software metrics, performance evaluation measures and implementation details. It also discusses the research questions, experimental results and corresponding findings.

4.1 Dataset

We have used the BugHunter, a new and reliable fault dataset collected by Ferenc R. et al. [28]. This dataset was created automatically using the OpenStaticAnalyzer tool and is publicly available. The BugHunter dataset assigns source code metrics to the code elements and it encompasses a larger amount of data compared to previous fault-prediction datasets. It includes a selection of 15 Java projects sourced from the GitHub repository, all of which are open source in nature. The projects of this dataset are the most common software, such as BroadleafCommerce, orientDB and Netty. BugHunter includes many instances at the method, class and file levels and contains four filters in addition to a basic and unfiltered case. Since the best-obtained results on this dataset were reported using the subtract filter, we selected the subtract-filter instances for six projects at the method level in our experiments and compared our results with the best-reported results in reference [28]. Every project in the BugHunter dataset includes 75 software metrics at the method level, which have been reduced after pre-processing step. Table 1 illustrates the dataset description for each project before and after the pre-processing. It also shows information about the quantity of software metrics, the imbalance ratio and the overall count of instances.

Table1. Description of the BugHunter projects before and after pre-processing (prep.).

Name of the project in the BugHunter	Total # of instances before prep.	# of faulty instances before prep.	# of non-faulty instances before prep.	Faulty ratio (%)	Imb. ratio	# of software metrics after prep.	Total # of instances after prep.	# of faulty instances after prep.	# of non-faulty instances after prep.
Netty	11171	2434	8737	21/78	3/58	57	16207	7470	8737
Neo4j	7030	1841	5189	26/18	3/14	57	9704	4515	5189
OrientDB	9445	2589	6856	27/41	2/64	59	12911	6055	6856
BroadleafCommerce	4709	1025	3684	21/76	3/59	59	6824	3140	3684
Ceylon-IDE-Eclipse	2087	508	1579	24/34	3/10	56	2972	1393	1579
MapDB	1456	480	976	32/96	2/03	57	1842	866	976

4.2 Software Metrics

The software metrics presented in the BugHunter dataset consist of static source-code metrics, code-smell metrics and code-clone metrics. Among 29 object-oriented metrics used at the method level of this dataset, for shortness, we listed some of them in Table 2. Table 3 presents the code-duplication metrics or basic clone-related metrics at the method level of the BugHunter.

4.3 Performance Evaluation Measures

To assess the results and compare the performance of the fault-prediction models, we employed measures including Accuracy, Precision, Recall and F1-score, Matthews Correlation Coefficient (MCC), Area Under the Receiver Operating Characteristic Curve (AUC-ROC), Area Under the Precision–Recall Curve (AUC-PR), Balanced Accuracy and G-Mean. Accuracy shows the proximity between an observed value and the true value, the proportion of the total count of correct predictions. Precision measures the degree of related results and recall measures how many related results are correct. F1-score represents a harmonic average between the precision and recall of a particular class. MCC provides a balanced evaluation by considering all four elements of the confusion matrix, making it particularly suitable for imbalanced datasets. AUC-ROC measures the model's ability to discriminate between defective and non-defective modules across different decision thresholds, while AUC-PR summarizes the trade-off between Precision and Recall. Balanced Accuracy computes the average of sensitivity and specificity and G-Mean evaluates the geometric mean of sensitivity and specificity, encouraging balanced-classification performance. These measures are defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \times 100 \tag{9}$$

$$\text{Precision} = \frac{TP}{TP + FP} \times 100 \tag{10}$$

$$\text{Recall} = \frac{TP}{TP + FN} \times 100 \tag{11}$$

$$\text{F1-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{12}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{13}$$

$$Balanced\ Accuracy = \frac{Sensitivity + Specificity}{2} \tag{14}$$

where,

$$Sensitivity = \frac{TP}{TP + FN}, \qquad \text{Specificity} = \frac{TN}{TN + FP}$$

$$G - Mean = \sqrt{Sensitivity \times \text{Specificity}} \tag{15}$$

The AUC-ROC and AUC-PR metrics are threshold-independent measures computed from the Receiver

Operating Characteristic (ROC) curve and the Precision–Recall (PR) curve, respectively. Higher values indicate better discrimination capability, with a value of 1 representing perfect classification.

where TP is used to represent True Positives, FP signifies False Positives, FN denotes False Negatives and TN represents True Negatives.

In addition to the prediction performance metrics, we also evaluate the computational efficiency of the proposed framework using training time and testing time. Training time is defined as the total time required to fit a model using the training dataset, whereas testing time represents the total time required to generate predictions for the testing dataset. These measurements provide an indication of the computational cost and practical applicability of the evaluated models.

Table2. Object-oriented features in the Bughunter dataset [28].

Abbreviation	Full name
CLOC	Comment Lines of Code
LOC	Lines of Code
LLOC	Logical Lines of Code
NL	Nesting Level
NLE	Nesting Level Else-If
NII	Number of Incoming Invocations
NOI	Number of Outgoing Invocations
CD	Comment Density
DLOC	Documentation Lines of Code
TCD	Total Comment Density
NOS	Number of Statements

Table3. Basic clone-related features in the Bughunter dataset [28].

Abbreviation	Full name
CC	Clone Coverage
CCL	Clone Classes
CCO	Clone Complexity
CI	Clone Instances
CLC	Clone Line Coverage
CLLC	Clone Lines of Code
LDC	Lines of Duplicated Code
LLDC	Logical Lines of Duplicated Code

4.4 Implementation Details

This Sub-section pertains to the implementation details required for the experiments; we will specify the execution environment and set the parameter settings for both the presented model and the established baseline models. We initially divided each dataset into 80% training data and 20% independent test data during the pre-processing stage. We emphasize that the train-test split was performed before any data normalization or oversampling to prevent data leakage. Subsequently, during the model-training phase, 20% of the training data was reserved as a validation set to monitor model performance, tune hyper-parameters and prevent overfitting. Consequently, the effective data partition consisted of approximately 64% training, 16% validation and 20% test data. The validation set was derived exclusively from the training data and remained completely independent of the final test set, further minimizing any potential data leakage. For designing the FC-NN models, 3 to 6 layers were tested and evaluated. The Relu, Tanh and Swish activation functions were used for all layers except the last layer. For the output layer, however, the Sigmoid activation function was utilized. To train the FC-DNN, we used Adam, Adagrad and SGD optimizer functions. The learning rate employed for these optimizers was set to 0.001 and the binary cross entropy was adopted as the loss function. We tested the epochs from 1000 to 20000 to determine the appropriate epoch for network convergence. Different batch sizes from 10 to 14000 have been considered and different dropout rates have been investigated from 0.1 to 0.6 by increments of 0.1. In CNN, we tested the number of filters with values of 2, 4, 8, 15 and 32 and evaluated the length of each filter from 1 to 3. Additionally, we used the batch size values of 32, 64 and 128. The proposed method was implemented using TensorFlow 2.7, Keras 2.7 and Python 3.7. The Pandas, Numpy and Scikitlearn libraries were used in this work. In addition, the Matplotlib and Imblearn libraries were used for visualization and addressing the class-imbalance problem, respectively. Many experiments have been conducted using GPU and CPU. System-processor specifications include

an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz.

For reproducibility, the CNN meta-learner of the proposed FR-CNN model employed a fixed architecture across all six projects to ensure a consistent stacking strategy. It consists of a one-dimensional convolutional layer with 8 filters (kernel size = 1) and ReLU activation, followed by a Flatten layer and a single-node output layer with a Sigmoid activation function. The model was trained using the Adam optimizer with binary cross-entropy loss for up to 100 epochs and a batch size of 64. Early stopping with a patience of 40 epochs based on the validation F1-score was applied to mitigate overfitting. To assess the effectiveness of the proposed model, we compared it with established ML and state-of-the-art baseline models (sub-sections 4.6.3 and 4.6.4). To ensure a fair comparison, all baseline models were implemented and evaluated under the same experimental conditions as the proposed method, including the same datasets, pre-processing pipeline, train-validation-test partitioning strategy, evaluation metrics and execution environment. The parameter settings for each baseline model are summarized below.

- The implementation of the traditional ML baseline models involved configuring them with default parameter values, utilizing the scikit-learn library.
- CBIL [54] includes an embedding layer with a dimension of 30, followed by a CNN layer with 20 filters, where each filter has a length of 10. Then, there is a max pooling layer, a BiLSTM layer with 32 LSTM units and a dense layer with the Sigmoid activation function. The tanh activation is used in both CNN and BiLSTM layers.
- The DL-based model [53] consists of 4 layers. The first layer has 20 nodes with tanh activation and the kernel initializer is uniform. The two hidden layers consist of 10 and 6 nodes, respectively, by using the relu activation. The output layer has a single node and the kernel initializer in the last three layers is normal.
- In 1D-CNN [18], 64 filters with a kernel size of 1 are chosen in each of two convolutional 1D layers. One max pooling layer with a pool size of 1 and two dense layers with activation ReLU and Sigmoid (last dense layer) were also used.
- The MLPs [50] consist of five layers, an input layer and three hidden layers employing the ReLU activation, as well as an output layer employing the Sigmoid activation. Following each of the first four layers, a dropout layer using the rate of 0.2 is included.

4.5 Research Questions

In evaluating the effectiveness of the presented model, the following research questions were formulated.

RQ1: Does tuning the FC-DNN and FR-CNN hyper-parameters increase the performance?

RQ2: Does proposed FR-CNN hybrid model achieve better performance compared to the individual learners?

RQ3: How effective is the proposed model in improving the prediction performance compared to the ML baseline models and the existing performance on the BugHunter dataset?

RQ4: Does proposed FR-CNN achieve better performance compared to the state-of-the-art models across all given projects in the BugHunter dataset?

RQ5: How does the proposed model compare with traditional machine-learning and state-of-the-art deep-learning approaches in terms of prediction performance, computational efficiency and model complexity on the BugHunter dataset?

RQ6: Does the proposed FR-CNN provide statistically significant improvements over existing software fault-prediction approaches?

RQ7: Does the proposed FR-CNN achieve an effective balance between prediction performance and computational efficiency while demonstrating the contribution of each architectural component?

4.6 Results and Discussion

The results of the proposed FC-DNN and FR-CNN models, along with the answers to RQ1 - RQ4, have been presented and discussed in the following sub-sections.

4.6.1 RQ1: Effect of Tuning the Hyper-parameters

To address the first research question, we present the experiments in subsections A and B.

A. Hyper-parameter Tuning Process for FC-DNN

To systematically evaluate the impact of hyper-parameter tuning on the performance of the proposed FC-DNN, we employed a comprehensive grid search strategy across a wide range of hyper-parameter configurations. The tuning process was conducted independently for each of the six projects.

Search Strategy and Parameter Ranges: We began with a base architecture and iteratively manipulated individual hyper-parameters while keeping others constant. Once the optimal value for a given hyper-parameter was identified, it was fixed and the tuning cycle was repeated for the next hyper-parameter. This iterative refinement process ensured that each hyper-parameter converged to its optimal value. The parameter ranges explored were as follows: number of epochs from 1,000 to 20,000 (with increments of 500-1000); batch size from 10 to 14,000 (with smaller increments for smaller batch sizes and larger steps for larger batch sizes); dropout rate from 0.1 to 0.6 (in increments of 0.1); number of hidden layers from 3 to 6; activation functions including ReLU, Tanh and Swish for hidden layers (with Sigmoid fixed for the output layer); and optimizer functions including Adam, Adagrad and SGD. This exhaustive exploration resulted in over 200 distinct experimental configurations being evaluated across all projects.

Validation Procedure and Selection Criteria: During the hyper-parameter tuning phase, a fixed validation split of 20% from the training data was employed to monitor training progress and prevent overfitting. For each hyperparameter configuration, the model was trained on the training sub-set and evaluated on the validation sub-set, with Accuracy and F1-score considered as the primary evaluation metrics. The final selected model was subsequently evaluated once on the independent test set. Table 4 summarizes the performance of the FC-DNN models obtained using the optimal hyper-parameter configurations for each project in terms of Accuracy, Precision, Recall and F1-score. These results demonstrate that different hyper-parameter configurations yield different prediction performances for the proposed FC-DNN model across the six BugHunter projects.

B. Hyper-parameter Tuning for FR-CNN Meta-learner

For the FR-CNN hybrid model, we investigated the impact of three key hyper-parameters of the CNN meta-learner: the number of filters (ranging from 2 to 32), kernel size (ranging from 1 to 3) and batch size (ranging from 32 to 128). Similar to the FC-DNN tuning process, we evaluated each configuration using the same data split and performance metrics. The tuning was performed by modifying one hyper-parameter at a time while keeping others fixed until reaching the highest performance. The average performance in terms of accuracy and F1-score was calculated and visualized on bar graphs (Figures 5-7). The results indicate that variations in these parameters do not yield significant performance enhancement, suggesting that the FR-CNN is relatively robust to these hyper-parameter choices.

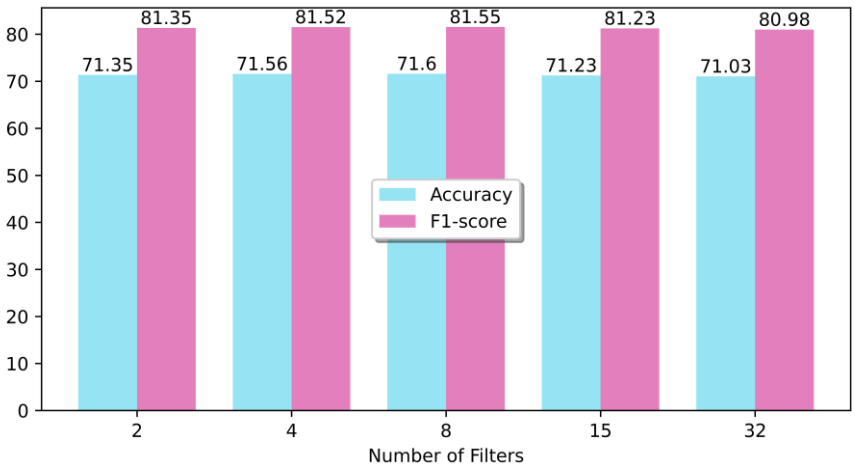

Figure 5. Effect of the number of CNN filters on the average Accuracy and F1-score of the proposed FR-CNN.

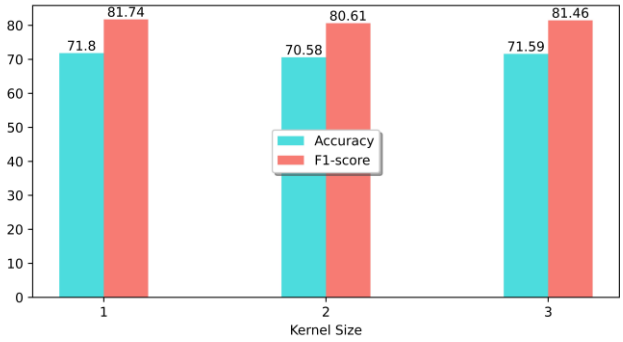

Figure 6. Effect of CNN kernel size on the average Accuracy and F1-score of the proposed FR-CNN.

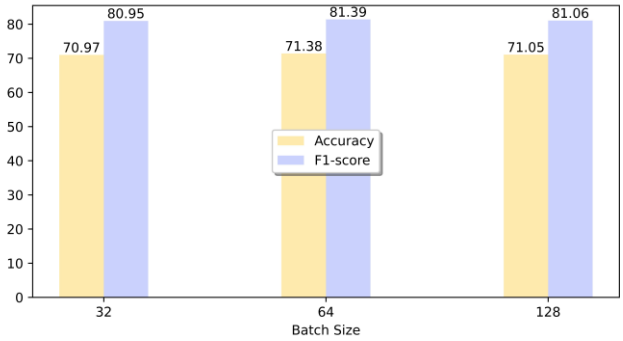

Figure 7. Effect of CNN batch size on the average Accuracy and F1-score of the proposed FR-CNN.

4.6.2 RQ2: Performance Comparison between the Presented FR-CNN and Individual Learners

In Tables 5 and 6, we present the results of the optimal FC-DNN model from the previous sub-section, the FR-CNN hybrid model and the RF model on six projects. The highest values of the average accuracy and F1-score are highlighted in bold. By comparison, the proposed FR-CNN hybrid model outperformed the proposed FC-DNN on the Neo4j orientDB, BroadleafCommerce, Ceylon-IDE-Eclipse and Netty projects, demonstrating improvements of up to 4.78% in accuracy (from 65.31 to 70.09) and up to 4.04% in F1-score (77.44 to 81.48). Compared to the RF, the FR-CNN improved the prediction performance on all six projects, especially on Ceylon-IDE-Eclipse. The FR-CNN demonstrated an improvement of up to 5.26% in accuracy (from 64.83% to 70.09%) and enhanced the F1-score by up to 4.35% (from 77.13% to 81.48%) on RF. We can conclude that the FR-CNN is a robust model that can enhance the prediction accuracy of any individual learner participating in the model, in terms of accuracy and F1-score, by covering their weaknesses. Compared to the other projects, the presented FC-DNN model outperformed the presented FR-CNN model on MapDB, exhibiting a 4.45% increase in accuracy (from 60.61 to 65.06) and a 5.3% improvement in F1-score (from 72.42 to 77.72). This behavior can be attributed to the hierarchical stacking mechanism adopted by the proposed FR-CNN. The CNN-based meta-learner combines the outputs of both the FC-DNN and the RF; therefore, its effectiveness depends on the complementary predictive information provided by the two base learners. The feature importance analysis reveals that RF and FC-DNN focus on different aspects of the feature space, confirming that their complementary strengths justify their use as base learners in the hierarchical FR-CNN architecture. However, on the MapDB dataset, the RF exhibits comparatively weaker predictive performance than the FC-DNN, limiting the amount of complementary information available to the meta-learner. As a result, the hybrid model is unable to further improve upon the stronger standalone FC-DNN on this particular dataset. This observation does not indicate a limitation of the stacking strategy itself; rather, it highlights that the effectiveness of hierarchical ensembles depends on the quality and diversity of their constituent base learners. When one base learner already captures most of the discriminative information while the other contributes relatively weaker predictions, the potential performance gain of the ensemble naturally becomes limited. Our analysis further reveals that the important features identified by the FC-DNN on the MapDB dataset are more discriminative than those selected by the RF, enabling the FC-DNN to capture the underlying fault patterns more effectively on this project. In contrast, on the remaining BugHunter projects, the complementary strengths of the two base learners enable the proposed FR-CNN to outperform the individual models. This interpretation is further supported by the

feature-importance analysis presented in Figures 8–10. The FC-DNN consistently identifies more discriminative features with lower uncertainty than the RF on the MapDB project, indicating that the neural network captures the underlying characteristics of this relatively small dataset more effectively. Consequently, the additional information contributed by the RF is insufficient to further improve the ensemble prediction on this project, whereas on the remaining BugHunter projects the complementary strengths of the two base learners enable FR-CNN to outperform the individual models.

4.6.3 RQ3: Performance Comparison of the Presented Model and the Traditional Baseline Models

To answer the third research question, we conduct the performance comparison between the proposed model and the seven established baseline classification models: KNN, NB, DT, LR, RF, GA and SVM. We also evaluate the performance of our model against the best performance achieved by [28] using 11 traditional ML algorithms, including RandomForest, DecisionTable, J48 (C4.5), RandomTree and SimpleLogistic and present the results in Tables 5 and 6. Additionally, we conduct a comparison between the best baseline model when using the scikit-learn library, which we used and the best ML algorithm when using the Weka library, as utilized by [28].

On average (Tables 5 and 6), the presented model enhanced the accuracy and F1-score of seven classification baseline models by 12.24% and 15.1%, respectively. In terms of F1-score (Table 6), the proposed model outperformed the best traditional ML algorithms [28] by 20.09%. By comparison, our proposed model increased the performance value from 53.95, using the RandomTree algorithm [28], to 81.48 on the Ceylon-IDE-Eclipse project, improving the performance by 27.53%. Also, the best baseline model used in this study, the RF, improved the average F1-score of the best traditional ML algorithms used in [28] by 18.28% on the selected projects.

4.6.4 RQ4: The Performance of the Presented Model against State-of-the-Art Approaches

To answer the fourth research question, we compare the proposed FR-CNN with five recent state-of-the-art software fault-prediction approaches: Hybrid model of Convolutional neural network and Bidirectional Long short-term memory (CBIL) [54], Deep Learning-based model (DL-based model) [53], 1-Dimensional Convolutional Neural Network (1D-CNN) [18] and Multi-layer perceptron's (MLPs) [50] and SEA-based metrics [30]. The comparative results are presented in Tables 7 and 8. Overall, the proposed FR-CNN demonstrates competitive prediction performance across the six BugHunter projects. Although no single method consistently achieves the highest performance on every project, FR-CNN provides a favorable balance between predictive effectiveness and computational efficiency, as further supported by the complexity analysis presented in the following sub-section.

Compared with the CBIL model, FR-CNN achieves competitive prediction performance across the evaluated BugHunter projects. In particular, it outperforms CBIL on the BroadleafCommerce project, achieving improvements of up to 8.17% (from 77.07 to 85.24) in accuracy and 4.53% (from 86.36 to 90.89) in F1-score, whereas CBIL achieves higher F1-scores on several of the remaining projects. The average performance of the proposed model closely approximates that of the CBIL model. Surprisingly, in the rest of the projects, the CBIL outperforms our proposed approach, which can be related to the specific structure of the CBIL model, comprising a combination of CNN and BiLSTM layers, resulting in varying performance across different projects. Nevertheless, it is important to highlight that, we took measures to prevent overfitting and minimize error during the training phase of our model to enhance its performance. Figure 11 presents the comparison of the model loss between our proposed approach and CBIL across BroadleafCommerce dataset, with the left graph corresponding to our proposed approach and the right graph corresponding to CBIL. Graphs show the same trend and fluctuation for other projects. Compared with the published optimization behavior of CBIL, the training and validation losses of which exhibit more noticeable oscillations across the BugHunter projects, the proposed FR-CNN consistently converges more smoothly. This stable optimization behavior demonstrates that the proposed hierarchical architecture achieves competitive prediction performance without sacrificing training stability or exhibiting signs of overfitting. We can conclude that our proposed approach achieved the lowest training and evaluation errors across various datasets, indicating excellent performance during the training phase without any overfitting issues. We included dropout layers in the proposed FC-DNN layers and used an early stopping process during training the FR-CNN model to ensure this. However, the prediction performance of the proposed approach needs improvement in the rest of the projects and we will consider it for future work.

Compared to the DL-based model, our proposed approach successfully improved the prediction performance in five projects: Ceylon-IDE-Eclipse, BroadleafCommerce, Netty, MapDB and OrientDB. The most significant increase in prediction performance was observed in the Ceylon-IDE-Eclipse project, with a 9.57% increase (from 60.52 to 70.09) in accuracy and a 7.15% increase (from 74.33 to 81.48) in F1-score. This was followed by BroadleafCommerce, which exhibited a 9.13% increase (from 76.11 to 85.24) in accuracy and a 6.36% increase (from 84.53 to 90.89) in F1-score.

Compared to the 1D-CNN, the proposed method improves the prediction performance in terms of accuracy and F1-score on the BroadleafCommerce, Ceylon-IDE-Eclipse and Netty projects. Our proposed method has better accuracy on the BroadleafCommerce and Netty, by 9.87% (from 75.37 to 85.24) and 8.59% (from 67.74 to 76.33), respectively. In terms of F1-score, the proposed method again shows superior performance on the Netty and BroadleafCommerce by 6.8% (from 78.73 to 85.53) and 6.76% (from 84.13 to 90.89), respectively. These findings indicate that the presented model is more efficient and effective than 1D-CNN, in terms of classification accuracy and F1-score on the aforementioned projects.

Compared to MLPs, the presented model improved the average accuracy and f1-score by 7.43% and 6.61%, respectively, on the Neo4j, Ceylon-IDE-Eclipse and MapDB projects.

Compared to SEA-based metrics, the proposed method improves the prediction performance in five projects: Neo4j, BroadleafCommerce, Ceylon-IDE-Eclipse, Netty and MapDB. In terms of F1-score, the proposed method shows superior performance on the Ceylon-IDE-Eclipse and BroadleafCommerce by 19.95% (from 61.53 to 81.48) and 15.77% (from 75.12 to 90.89), respectively.

4.6.5 Additional Performance Evaluation and Computational Analysis

To answer the fifth research question, we evaluate the effectiveness of the proposed approach and state-of-the-art approaches against traditional ML algorithms on the selected projects. In terms of accuracy, the presented model and state-of-the-art models demonstrate superiority against the ML baseline models with 69.09%. In terms of F1-score, they again outperformed ML models with 78.96%. On average, the proposed model and state-of-the-art models improved the accuracy and F1-score of the traditional ML baseline algorithms by 9.26% and 12.75%, respectively.

It is also worth clarifying the comparatively low average MCC values reported in Figure 12 relative to Balanced Accuracy, AUC-ROC and G-Mean. Unlike Balanced Accuracy, which is bounded below by 0.5 for a random classifier and therefore tends to remain in a moderate range (0.5–0.6) even for weak models, MCC is a chance-corrected measure in which a value of 0 indicates performance statistically indistinguishable from random guessing and it penalizes any imbalance among the four confusion-matrix cells multiplicatively rather than additively. Consequently, on projects where a classifier's predictions are only marginally better than chance for the minority (faulty) class, Balanced Accuracy can still appear moderately above 0.5, while the corresponding MCC collapses toward zero. This effect is most evident on the two smallest projects in our study, Ceylon-IDE-Eclipse and MapDB, where FR-CNN obtains an MCC of only 0.051 and 0.040, respectively (Balanced Accuracy of 0.520 and 0.519), compared with an MCC of 0.496 (Balanced Accuracy of 0.731) on BroadleafCommerce. Notably, near-zero or even negative MCC values on Ceylon-IDE-Eclipse are observed across nearly all compared models—not only FR-CNN—including NB (−0.062), SVM (−0.053), KNN (−0.050), 1D-CNN (−0.036) and CBIL (−0.043), indicating that this is a dataset-level property related to the limited size and difficulty of these two projects rather than a weakness specific to the proposed model. Since Figure 12 reports a simple (unweighted) average across the six BugHunter projects, the near-zero MCC values on these two smaller, harder projects disproportionately pull down the six-project average MCC relative to the other metrics, which are individually less sensitive to this near-chance behavior. The per-project MCC and Balanced Accuracy values underlying this analysis were verified by the authors against the raw confusion-matrix outputs for all six projects and all twelve models.

The practical applicability of the proposed framework is further demonstrated through computational complexity and execution-time analyses, as presented in Figs. 13 and 14. Compared with recent deep-learning approaches, FR-CNN requires substantially fewer trainable parameters and shorter training time while maintaining a testing time below one second on average. The proposed model employs only 41 trainable parameters in its CNN meta-learner, whereas 1D-CNN, CBIL and the DL-based model require approximately 22.7K, 236K and 1.5K trainable parameters, respectively. It should be noted that

these parameters correspond only to the lightweight CNN meta-learner, since the FC-DNN base learners are pre-trained offline using their optimized hyper-parameters and subsequently reused with fixed weights during deployment. Consequently, practitioners are not required to retrain the FC-DNN components, reducing computational overhead while preserving the representational capability of deep learning. Overall, the complexity analysis demonstrates that FR-CNN achieves competitive prediction performance while requiring substantially fewer trainable parameters and shorter training time than the recent deep-learning-based approaches evaluated in this study. These results indicate that the proposed hierarchical framework provides a favorable balance between predictive performance and computational efficiency, making it suitable for practical software-engineering environments where computational resources, training time and model complexity are important considerations.

4.6.6 Statistical Significance Analysis

To verify that the performance improvements of FR-CNN are statistically meaningful rather than due to random variation, we conducted non-parametric significance testing across the six BugHunter projects. Results were presented in Table 9. First, we applied the Friedman test, which evaluates whether performance differences exist among all compared models across the six projects. The test yielded a highly significant result for both accuracy ($\chi^2 = 40.31$, $p < 0.00001$) and F1-score ($\chi^2 = 41.42$, $p < 0.00001$), confirming that the observed differences among models are unlikely to have occurred by chance. We then conducted pairwise Wilcoxon signed-rank tests comparing FR-CNN against each baseline model (KNN, NB, DT, LR, RF, GA, SVM) across the six projects, applying the Benjamini-Hochberg false-discovery rate (FDR) correction to account for multiple comparisons. FR-CNN significantly outperformed all seven traditional ML baselines on both accuracy and F1-score (adjusted $p = 0.036$ for all comparisons), indicating consistent superiority across projects rather than dataset-specific gains. The comparison between FR-CNN and the standalone FC-DNN did not reach statistical significance (accuracy: $p = 0.31$; F1-score: $p = 0.44$), consistent with the MapDB result discussed in Sub-section 4.6.2, where the FC-DNN outperformed the hybrid model due to the comparatively weaker RF performance on that project. We note that, with six paired projects, the Wilcoxon test has a minimum achievable two-sided p-value of 0.031; while this constrains the granularity of the reported significance levels, the consistent direction of the effect across all projects, combined with the significant Friedman omnibus result, supports the robustness of the observed improvement.

4.6.7 Ablation Study: Meta-learner Comparison and Component Contribution

To directly assess the design choices underlying the proposed FR-CNN architecture, we conducted two complementary ablation studies across all six BugHunter projects.

A. Meta-learner Comparison: Holding the base learners and the meta-feature vector $z_i = [p_{DNN}, p_{RF},0, p_{RF},1]$ fixed, we replaced the CNN meta-learner with three alternatives: a feed-forward DNN of matched capacity (Dense(8, ReLU) → Dense(1, Sigmoid)), a Logistic Regression classifier and a parameter-free unweighted average of p_{DNN} and $p_{RF},1$. The CNN and DNN variants, being stochastic, were each trained over five random seeds; we report the mean, with the standard deviation across seeds not exceeding 0.49 F1 points for any project, confirming that the training procedure converges reliably. Table 10 reports the per-project and average F1-score, together with the number of trainable parameters and training time for each meta-learner.

The results show that the three learnable meta-learners (CNN, DNN, Logistic Regression) perform comparably, with average F1-scores within 0.3 points of one another (80.74–81.00%) and on average outperform simple unweighted averaging by approximately one point of F1-score and accuracy across the six projects. This margin is not uniform across every project: on MapDB and to a lesser extent OrientDB, simple averaging performs on par with or slightly better than the learnable meta-learners, whereas on the remaining four projects the learnable meta-learners provide a clearer advantage of 1.4–2.4 F1 points. This indicates that the principal benefit of the meta-learning stage comes from learning a combination of the base-learner outputs rather than from the specific functional form of the meta-learner. We retain the CNN meta-learner in the proposed FR-CNN, because it matches or exceeds the simpler learnable alternatives while requiring only 41 trainable parameters and a training time under 34 seconds on average (Table 10), consistent with the lightweight-inference goal already demonstrated in Sub-section 4.6.5 (Figs. 13–14). We note that Logistic Regression achieves a marginally higher average F1-score (81.00% *vs.* 80.83%) at a fraction of the parameter and training cost; we report this transparently,

as the meaningful architectural conclusion is that a learned combination rule is beneficial, while the specific choice among CNN, DNN and Logistic Regression has a comparatively minor effect on predictive performance.

B. Component Contribution: To isolate the contribution of each element of the hierarchical architecture, we compared FC-DNN alone, RF alone and the FC-DNN+RF combination under three different combiners: simple averaging, a Logistic Regression meta-learner and the proposed CNN meta-learner. Table 11 reports the per-project and average F1-score for each configuration.

Combining FC-DNN and RF, regardless of the combiner used, improves over FC-DNN alone on five of the six projects, with gains of up to 4.7 F1 points on Netty and matches or slightly exceeds RF alone on all six projects. The only exception is MapDB, where every combined configuration performs worse than the standalone FC-DNN. This is consistent with and independently corroborates our analysis in Sub-section 4.6.2, which attributes this behavior to RF's comparatively weaker predictive performance on the MapDB project, which limits the complementary information available to any meta-learner regardless of its architecture. Taken together, Tables 10 and 11 confirm that (i) each base learner contributes meaningfully to the hybrid architecture except where one base learner is markedly weaker than the other and (ii) the specific meta-learner used to combine them is a secondary, though non-negligible, design choice.

5. Threats to Validity

To maintain scientific rigor, it is important to acknowledge potential limitations across four key areas: internal, external, construct and conclusion validity.

5.1 Internal Validity

A primary internal threat is the data pre-processing pipeline. We removed nominal features, such as Hash and LongName, and also filtered out strictly constant features (i.e., those with no variance across all instances). While this is a standard and necessary step, it assumes that any feature with zero variance is entirely non-informative. This assumption is reasonable, but simplistic, as such features could theoretically have predictive value in rare cases.

5.2 External Validity

External validity concerns the generalizability of our results. Our experiments were conducted on six selected projects from the BugHunter dataset at the method level. Furthermore, the results may not generalize to other widely used datasets, such as PROMISE or NASA, which have different characteristics and features. The choice of granularity level (method, class or file) could also influence the outcomes, as fault patterns may vary across these levels.

5.3 Construct Validity

The primary construct we aim to predict is "fault-proneness," which we operationalize as a binary outcome (i.e., whether a module contains bugs or not). This simplification is common in the field, but does not distinguish between the severity or count of bugs. Additionally, the selected software metrics (such as coupling, cohesion and code smells) are indirect indicators of complexity and maintainability and may not fully capture all factors contributing to fault-proneness.

5.4 Conclusion Validity

We mitigated conclusion-validity threats by evaluating our models using multiple metrics (Accuracy, Precision, Recall, F1-score and MCC) rather than relying on a single measure. To ensure reproducibility and reduce variability, a fixed random seed (42) was used for all stochastic processes. Furthermore, an early stopping mechanism was applied during training to prevent overfitting. The use of training/validation loss curves further supports the stability of our model's training process. While the FR-CNN model shows competitive performance, it does not always consistently outperform all baseline models and we have reported these comparisons transparently in the Results Section.

We also note a limitation regarding the stability of the headline improvement figures reported in this study (e.g., the 19.95% F1-score improvement over SEA-based metrics on Ceylon-IDE-Eclipse, Sub-

section 4.6.4 and the comparisons against the traditional ML and state-of-the-art baselines in Sub-sections 4.6.3 and 4.6.4). These headline results, like the fixed-seed setting described above, are reported from a single run per model (seed = 42) rather than averaged across multiple random seeds, since several of the compared baseline results are literature-reported values for which repeated-seed statistics are not available. We therefore treat these point-estimate improvements as indicative rather than as precise, variance-quantified effect sizes. Where repeated-seed evaluation was feasible — i.e., for the FR-CNN and DNN meta-learners in the ablation study of Sub-section 4.6.7 — Table 10 reports the mean F1-score across five random seeds for each project and we additionally state in the accompanying text of Sub-section 4.6.7 that the observed standard deviation across seeds did not exceed 0.49 F1 points for any project, suggesting that the training procedure itself is stable, even though the per-project standard deviations themselves are not tabulated. We nonetheless flag the single-seed nature of the remaining headline comparisons as a limitation and recommend that future replications average results over multiple seeds, particularly for the comparisons against externally reported baselines.

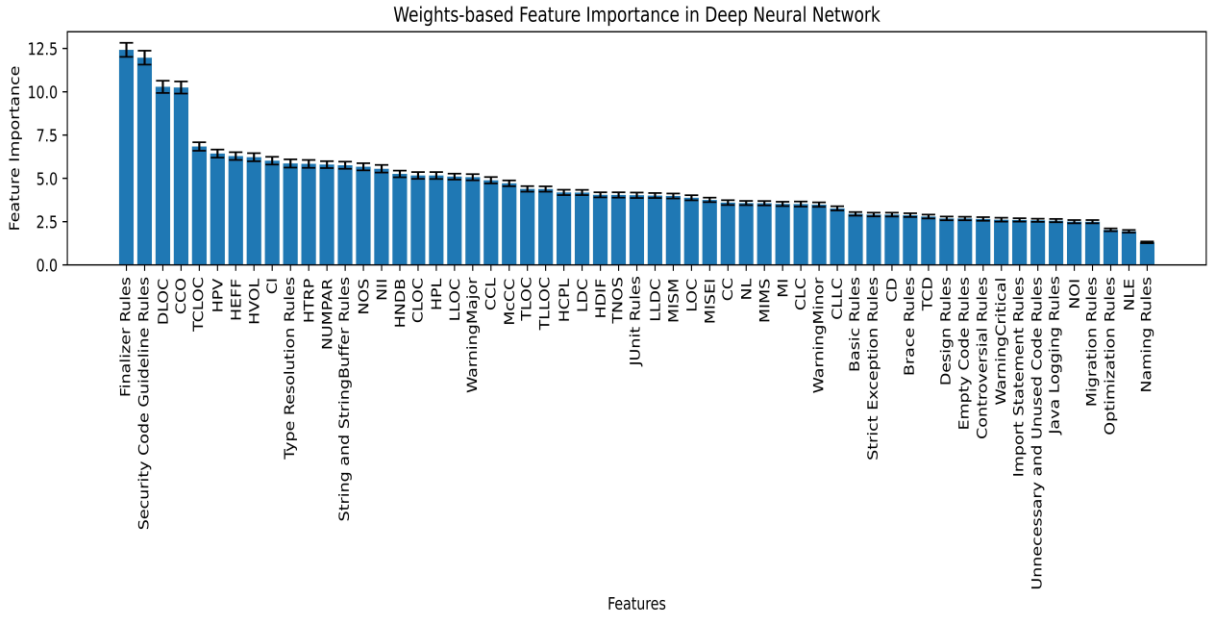

Figure 8. Comparison of the feature importance identified by the FC-DNN, illustrating its complementary feature representations in the proposed FR-CNN. Error bars represent the standard deviation of input layer weights, indicating uncertainty in the feature importance.

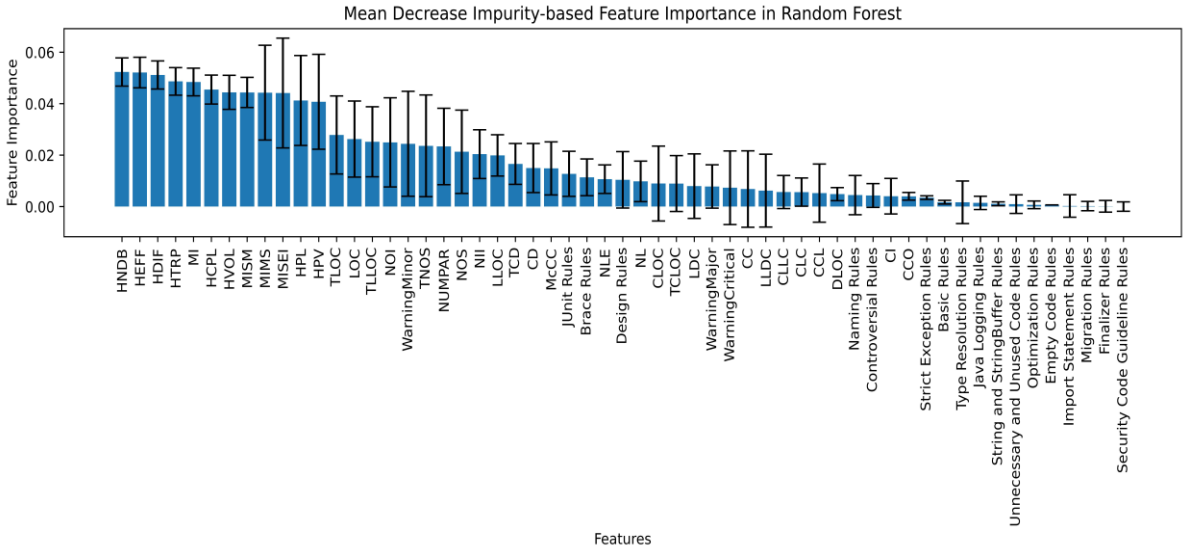

Figure 9. Comparison of the feature importance identified by the RF, illustrating its complementary feature representations in the proposed FR-CNN. Error bars represent the standard deviation of feature-importance scores obtained from individual trees within the RF ensemble, indicating uncertainty in the feature importance.

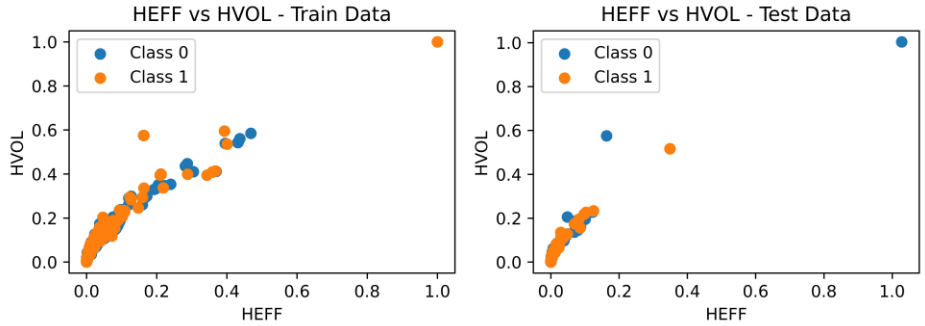

Figure 10. Faulty or non-faulty class association with HEFF and HVOL which are common among the 10 most important features shared with both the RF and FC-DNN.

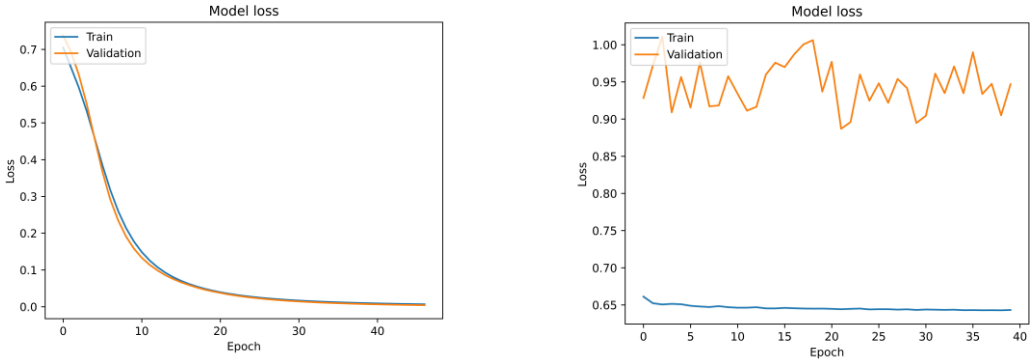

Figure 11. Training and validation loss curves of a) the proposed FR-CNN model and b) CBIL state-of-the-art model, illustrating the convergence behavior during training.

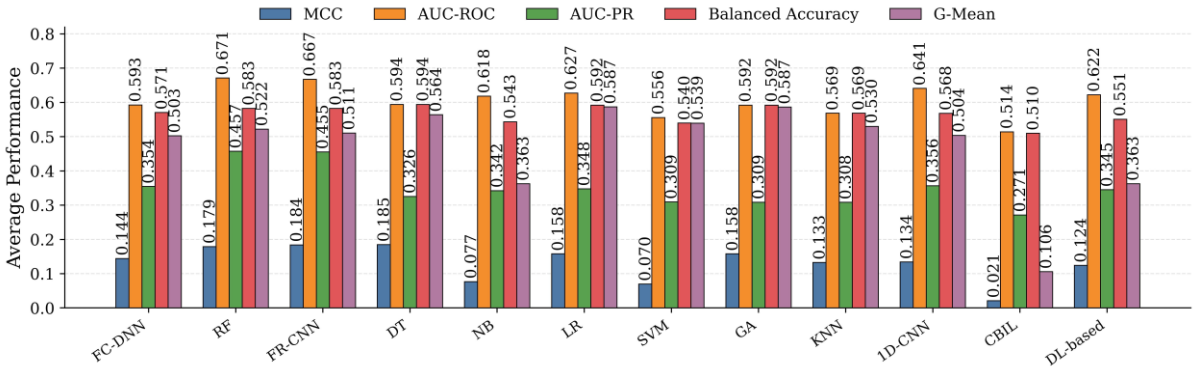

Figure 12. Comparison of the average performance of the proposed FR-CNN and baseline models using additional evaluation metrics (MCC, AUC-ROC, AUC-PR, Balanced Accuracy and G-Mean) across the six selected BugHunter projects.

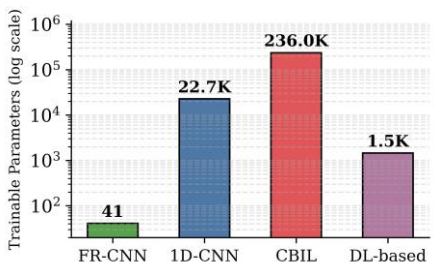

Figure 13. Comparison of the trainable parameters of the proposed FR-CNN and recent deep-learning approaches. The reported parameters for FR-CNN correspond only to the lightweight CNN meta-learner, while the pre-trained FC-DNN base learners are reused with fixed weights during deployment.

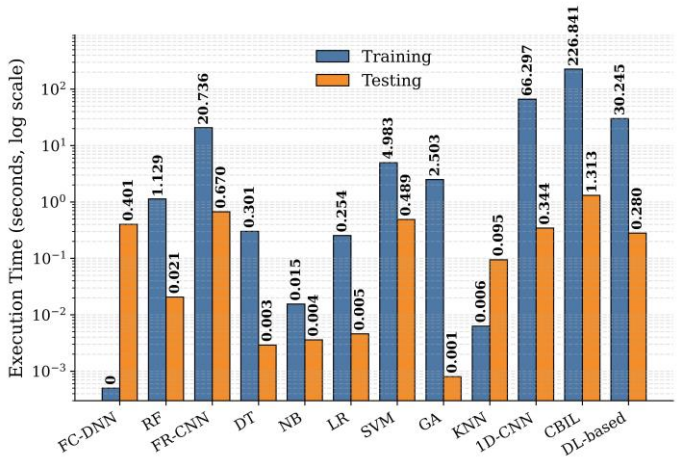

Figure 14. Comparison of the average training time and testing time of the proposed FR-CNN, traditional machine-learning algorithms and recent deep-learning approaches across the six selected BugHunter projects.

Table 4. Optimal hyper-parameter for each project achieved by more than 200 experiments during FC-DNN model tuning.

Project name	#epochs	Batch size	Dropout rate	optimizer	#layers	Activation function	Acc.	Prec.	Rec.	F1.
BroadleafCommerce	9500	2000	0.1	Adam	4	Relu	81.52	88.15	88.39	88.27
Ceylon-IDE-Eclipse	5000	2600	0.4	Adam	3	Relu	65.31	82.45	73.02	77.44
Neo4j	8000	8500	0.2	Adam	3	Relu	67.70	81.15	76.58	78.80
OrientDB	7500	1000	0.6	Adam	3	Swish	67.33	81.56	74.82	78.05
Netty	5000	7000	0.1	Adam	3	Swish	71.90	82.35	81.88	82.11
MapDB	2000	1000	0.6	Adam	3	Tanh	65.06	88.55	69.26	77.72

Table 5. Accuracy (%) of the proposed FC-DNN and FR-CNN models *vs.* traditional ML baseline models.

Name of project	FC-DNN	FR-CNN	KNN	NB	DT	LR	RF	GA	SVM
Neo4j	67.70	71.76	61.73	65.29	67.14	61.37	69.77	61.02	60.31
OrientDB	67.33	68.39	61.14	45.47	66.11	57.54	66.59	57.64	58.28
BroadleafCommerce	81.52	85.24	73.24	21.76	81.42	64.86	84.71	64.75	66.34
Ceylon-IDE-Eclipse	65.31	70.09	54.54	28.46	61.96	55.02	64.83	54.30	55.26
Netty	71.90	76.33	64.42	60.40	72.08	58.34	75.21	59.59	57.76
MapDB	65.06	60.61	52.05	33.21	58.56	57.87	59.24	57.53	56.16
Average	**69.80**	**72.07**	61.18	42.43	67.87	59.16	**70.05**	59.13	59.01

Table 6. F1-score (%) of the proposed FC-DNN and FR-CNN models *vs.* traditional ML baseline models.

Name of project	FC-DNN	FR-CNN	[28]	KNN	NB	DT	LR	RF	GA	SVM
Neo4j	78.80	82.06	60.86	71.71	74.92	77.59	69.98	80.35	69.65	68.82
OrientDB	78.05	79.23	62.36	70.56	41.60	76.41	64.48	77.58	64.66	65.13
Broadleaf-Commerce	88.27	90.89	73.66	81.60	2.38	88.10	73.45	90.47	73.13	75.93
Ceylon-IDE-Eclipse	77.44	81.48	53.95	66.19	6.26	74.06	66.06	77.13	66.07	64.38
Netty	82.11	85.53	64.12	75.17	71.21	81.93	68.19	84.72	69.40	67.15
MapDB	77.72	72.42	56.10	60.89	9.30	69.21	68.21	70.47	67.19	65.59
Average	**80.39**	**81.93**	61.84	71.02	34.27	77.88	68.39	**80.12**	68.35	67.83

Table 7. Accuracy (%) of the proposed FC-DNN and FR-CNN models *vs.* state-of-the-art approaches.

Name of project	FC-DNN	FR-CNN	CBIL [54]	DL-based model [53]	1D-CNN [18]	MLPs [50]
Neo4j	67.70	71.76	73.54	72.54	73.32	63.39
OrientDB	67.33	68.39	71.04	67.28	71.20	-
BroadleafCommerce	**81.52**	**85.24**	77.07	76.11	75.37	-
Ceylon-IDE-Eclipse	65.31	70.09	72.00	60.52	61.96	62.67
Netty	71.90	76.33	77.98	71.90	67.74	-
MapDB	65.06	60.61	68.15	62.32	67.12	58.56
Average	69.80	**72.07**	**73.29**	68.44	69.45	61.54

Table 8. F1-score (%) of the proposed FC-DNN and FR-CNN models *vs.* state-of-the-art approaches.

Name of project	FC-DNN	FR-CNN	CBIL [54]	DL-based model [53]	1D-CNN [18]	MLPs [50]	SEA-based metrics
Neo4j	78.80	82.06	84.61	82.43	84.38	75.36	73.84
OrientDB	78.05	79.23	83.03	77.54	83.17	-	82.30
BroadleafCommerce	**88.27**	**90.89**	86.36	84.53	84.13	-	75.12
Ceylon-IDE-Eclipse	77.44	81.48	83.72	74.33	75.11	74.83	61.53
Netty	82.11	85.53	87.63	82.57	78.73	-	73.58
MapDB	77.72	72.42	81.05	74.88	78.85	71.25	66.41
Average	80.39	**81.93**	**84.4**	79.38	80.72	73.81	72.13

Table 9. Statistical-significance testing results comparing FR-CNN against baseline models across all six BugHunter projects.

Comparison (FR-CNN *vs.*)	**Accuracy p-value**	**Accuracy (FDR-adj.)**	**F1-score p-value**	**F1-score (FDR-adj.)**	**Significant?**
FC-DNN	0.3125	0.3125	0.4375	0.4375	No
KNN	0.0312	0.0357	0.0312	0.0357	Yes
Naive Bayes	0.0312	0.0357	0.0312	0.0357	Yes
Decision Tree	0.0312	0.0357	0.0312	0.0357	Yes
Logistic Regression	0.0312	0.0357	0.0312	0.0357	Yes
Random Forest	0.0312	0.0357	0.0312	0.0357	Yes
Genetic Algorithm	0.0312	0.0357	0.0312	0.0357	Yes
SVM	0.0312	0.0357	0.0312	0.0357	Yes

Table 10. F1-score (%) of the proposed CNN meta-learner *versus* simpler alternatives, using the identical base-learner outputs across the six BugHunter projects.

Project	**CNN (proposed)**	**DNN**	**Logistic Regression**	**Simple averaging**
BroadleafCommerce	90.33	90.20	90.28	88.56
Ceylon-IDE-Eclipse	79.88	79.88	79.70	77.45
MapDB	70.39	69.97	70.10	73.21
Neo4j	81.29	81.24	81.53	79.89
Netty	85.02	85.04	85.68	83.21
OrientDB	78.05	78.13	78.74	78.18
Average	**80.83**	80.74	81.00	80.08
Trainable parameters	41	41	4	0
Average training time (s)	33.63	33.16	0.03	0.00

Table 11. F1-score (%) contribution of each component of the hierarchical architecture across the six BugHunter projects.

Project	FC-DNN alone	RF alone	FC-DNN+RF, averaging	FC-DNN+RF, LR meta-learner	FC-DNN+RF, CNN meta-learner (FR-CNN)
BroadleafCommerce	87.61	90.08	88.56	90.28	90.22
Ceylon-IDE-Eclipse	76.28	78.53	77.45	79.70	79.76
MapDB	77.38	69.80	73.21	70.10	70.10
Neo4j	78.49	80.11	79.89	81.53	81.30
Netty	80.34	84.83	83.21	85.68	85.02
OrientDB	77.39	77.71	78.18	78.74	78.10
Average	79.58	80.18	80.08	81.00	80.75

6. Conclusion and Future Work

In this study, we proposed FR-CNN, a hierarchical hybrid framework for method-level software-fault prediction on the BugHunter dataset. The framework integrates a fine-tuned FC-DNN, Random Forest and a CNN-based meta-learner following comprehensive pre-processing and hyper-parameter optimization. Extensive experiments on six Java projects demonstrate that FR-CNN consistently outperforms the developed traditional machine-learning baselines and achieves competitive performance against recent state-of-the-art approaches. While some methods, such as CBIL, achieve higher F1-scores on several individual projects, FR-CNN provides a more favorable balance between predictive performance, computational efficiency and model complexity. The additional efficiency analysis further shows that the proposed framework requires fewer trainable parameters, shorter training time and stable optimization behavior without overfitting, making it well suited for practical software fault-prediction applications.

The proposed FR-CNN is also consistent with the recent evolution of software fault-prediction research toward hybrid deep-learning architectures. Recent studies have shown that integrating complementary feature representations, dynamic classification strategies, attention mechanisms and heterogeneous deep learning models can substantially improve software fault prediction performance. In contrast to these recent approaches, the proposed FR-CNN employs a hierarchical stacking architecture that combines a fine-tuned FC-DNN and Random Forest using a CNN-based meta-learner, enabling the model to exploit complementary strengths of machine learning and deep learning for large-scale method-level software-fault prediction on the BugHunter dataset [21]-[24].

For future work, specific combinations of boosting or bagging algorithms with different structures of deep neural networks can be evaluated to potentially enhance the performance of the hybrid model. Also, it is valuable to conduct empirical studies on the rest of the projects at the method-level of the BugHunter dataset or other granularity levels, such as file-level and class-level for SFP. Moreover, we can consider the structure of CNN, LSTM, Dense and Dropout layers to detect software faults.

Data Availability

The datasets used in this study are publicly available through BugHunter (BugHunter Dataset-Mendeley Data).

Code Availability

To enhance the transparency, reproducibility and practical applicability of the proposed framework, the complete implementation of the FR-CNN approach has been made publicly available through an online GitHub repository (FR-CNN Hybrid approach). The repository includes the complete experimental workflow, encompassing data pre-processing, implementation of the proposed FR-CNN framework and implementations of the baseline deep-learning and traditional machine-learning models evaluated in this study. In addition, the publicly available BugHunter datasets used in the experiments, the optimal hyper-parameter configurations identified for each of the six projects and the corresponding pre-trained FC-DNN models are provided. To facilitate practical reuse, each pre-trained FC-DNN model is available in three complementary formats: the complete pre-trained model, the model architecture (JSON configuration) and the corresponding model weights. The implementation supports both direct loading of the complete pre-trained model and reconstruction of the model from its architecture and weights using TensorFlow/Keras. Since the FC-DNN base learners have already been trained using the optimal hyper-parameter configurations reported in Table 4, users can directly load the pre-trained models without repeating the computationally intensive training process. Consequently, reproducing the proposed FR-CNN framework requires zero additional training time for the FC-DNN stage, substantially reducing the computational cost while enabling researchers and practitioners to efficiently reproduce the reported experiments under the same experimental settings.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Funding Statement

The authors declare that this paper was not supported by any funding or financial assistance.

REFERENCES

[1] A. Arshad, S. Riaz, L. Jiao and A. Murthy, "Semi-supervised Deep Fuzzy C-mean Clustering for Software Fault Prediction," IEEE Access, vol. 6, pp. 25675–25685, 2018.

[2] F. Li, P. Yang, J. W. Keung, W. Hu, H. Luo and X. Yu, "Revisiting 'Revisiting Supervised Methods for Effort-aware Cross-project Defect Prediction'," IET Software, vol. 17, no. 4, pp. 472–495, 2023.

[3] E. N. Akimova et al., "A Survey on Software Defect Prediction Using Deep Learning," Mathematics, vol. 9, no. 11, p. 1180, 2021.

[4] S. K. Pandey et al., "Software Defect Prediction Using K-PCA and Various Kernel-based Extreme Learning Machine: An Empirical Study," IET Software, vol. 14, no. 7, pp. 768–782, 2020.

[5] C. Jin, "Cross-project Software Defect Prediction Based on Domain Adaptation Learning and Optimization," Expert Systems with Applications, vol. 171, p. 114637, 2021.

[6] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran and A. Abraham, "A Systematic Literature Review on Software Defect Prediction Using Artificial Intelligence: Datasets, Data Validation Methods, Approaches and Tools," Engineering Applications of Artificial Intelligence, vol. 111, p. 104773, 2022.

[7] Y. Hassouneh et al., "Boosted Whale Optimization Algorithm with Natural Selection Operators for Software Fault Prediction," IEEE Access, vol. 9, pp. 14239–14258, 2021.

[8] S. Kumar and S. S. Rathore, Software Fault Prediction: A Road Map, eBook ISBN978-981-10-8715-8, Springer, 2018.

[9] Z. Li, X.-Y. Jing and X. Zhu, "Progress on Approaches to Software Defect Prediction," Iet Software, vol. 12, no. 3, pp. 161–175, 2018.

[10] X. Chen, Y. Zhao, Q. Wang and Z. Yuan, "MULTI: Multi-objective Effort-aware Just-in-time Software Defect Prediction," Information and Software Technology, vol. 93, pp. 1–13, 2018.

[11] K. Wang, L. Liu, C. Yuan and Z. Wang, "Software Defect Prediction Model Based on LASSO–SVM," Neural Computing and Applications, vol. 33, no. 14, pp. 8249–8259, 2021.

[12] S. K. Pandey, R. B. Mishra and A. K. Tripathi, "Machine Learning Based Methods for Software Fault Prediction: A Survey," Expert Systems with Applications, vol. 172, p. 114595, 2021.

[13] F. Khan et al., "Hyper-parameter Optimization of Classifiers, Using an Artificial Immune Network and Its Application to Software Bug Prediction," IEEE Access, vol. 8, pp. 20954–20964, 2020.

[14] C. Arun and C. Lakshmi, "Genetic Algorithm-based Oversampling Approach to Prune the Class Imbalance Issue in Software Defect Prediction," Soft Computing, vol. 26, no. 23, pp. 12915–12931, 2022.

[15] F. Matloob et al., "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review," IEEE Access, vol. 9, pp. 98754-98771, 2021.

[16] Ö. F. Arar and K. Ayan, "A Feature Dependent I Bayes Approach and Its Application to the Software Defect Prediction Problem," Applied Soft Computing, vol. 59, pp. 197–209, 2017.

[17] M. Singh and D. S. Salaria, "Software Defect Prediction Tool Based on Neural Network," Int. J. of Computer Applications, vol. 70, no. 22, 2013.

[18] Z. M. Zain et al., "Software Defect Prediction Harnessing on Multi 1-dimensional Convolutional Neural Network Structure," Computers, Materials and Continua, vol. 71, no. 1, p. 1521, 2022.

[19] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, [Online], Available: http://www.deeplearningbook.org, MIT Press, 2016.

[20] M. M. Ahsan, T. E Alam, T. Trafalis and P. Huebner, "Deep MLP-CNN Model Using Mixed-data to Distinguish between COVID-19 and Non-COVID-19 Patients," Symmetry, vol. 12, no. 9, p. 1526, 2020.

[21] A. Abdu et al., "Semantic and Traditional Feature Fusion for Software Defect Prediction Using Hybrid Deep Learning Model," Scientific Reports, vol. 14, no. 1, p. 14771, 2024.

[22] S. Kaliraj, V. G. P. Sahasranth and V. Sivakumar, "A Holistic Approach to Software Fault Prediction with Dynamic Classification," Automated Software Engineering, vol. 31, no. 2, p. 70, 2024.

[23] Q. Fei et al., "A Software Defect Prediction Method Using a Multivariate Heterogeneous Hybrid Deep Learning Algorithm," Computers, Materials, & Continua, vol. 82, no. 2, p. 3251, 2025.

[24] R. Malhotra and P. Singh, "DHG-BiGRU: Dual-attention based hierarchical gated BiGRU for software defect prediction," Information and Software Technology, vol. 179, p. 107646, 2025.

[25] J. S. Shirabad and T. J. Menzies, "The PROMISE Repository of Software Engineering Databases," School of Information Technology and Engineering, University of Ottawa, Canada, vol. 24, p. 3, 2005.

[26] M. Shepperd, Q. Song, Z. Sun and C. Mair, "Data Quality: Some Comments on the NASA Software Defect Datasets," IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208–1215, 2013.

[27] E. N. Akimova et al., "PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction," Proc. of the 2021 28th Asia-Pacific Software Engineering Conf. (APSEC), pp. 141–151, Taipei, Taiwan, 2021.

[28] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth and T. Gyimóthy, "An Automatically Created Novel Bug Dataset and Its Validation in Bug Prediction," J. of Systems and Software, vol. 169, p. 110691, 2020.

[29] S. T. Cynthia et al., "Feature Transformation for Improved Software Bug Detection Models," Proc. of the 15th Innovations in Software Engineering Conf. (ISEC '22), Article no. 16, pp. 1–10, 2022.

[30] J. Jász, "The Effectiveness of Hidden Dependence Metrics in Bug Prediction," IEEE Access, vol. 12, pp. 77214–77225, 2024.
[31] S. S. Rathore and S. Kumar, "Linear and Non-linear Heterogeneous Ensemble Methods to Predict the Number of Faults in Software Systems," Knowledge-based Systems, vol. 119, pp. 232–256, 2017.
[32] S. S. Rathore and S. Kumar, "Towards an Ensemble Based System for Predicting the Number of Software Faults," Expert Systems with Applications, vol. 82, pp. 357–382, 2017.
[33] S. Wang and X. Yao, "Using Class Imbalance Learning for Software Defect Prediction," IEEE Transactions on Reliability, vol. 62, no. 2, pp. 434–443, 2013.
[34] Z. Sun, Q. Song and X. Zhu, "Using Coding-based Ensemble Learning to Improve Software Defect Prediction," IEEE Trans. on Systems, Man and Cybernetics, Part C, vol. 42, no. 6, pp. 1806–1817, 2012.
[35] S. S. Rathore and S. Kumar, "An Empirical Study of Ensemble Techniques for Software Fault Prediction," Applied Intelligence, vol. 51, no. 6, pp. 3615–3644, 2021.
[36] S. Moustafa et al., "Software Bug Prediction Using Weighted Majority Voting Techniques," Alexandria Engineering Journal, vol. 57, no. 4, pp. 2763–2774, 2018.
[37] I. H. Laradji, M. Alshayeb and L. Ghouti, "Software Defect Prediction Using Ensemble Learning on Selected Features," Information and Software Technology, vol. 58, pp. 388–402, 2015.
[38] T. Wang, Z. Zhang, X. Jing and L. Zhang, "Multiple Kernel Ensemble Learning for Software Defect Prediction," Automated Software Engineering, vol. 23, no. 4, pp. 569–590, 2016.
[39] S. K. Pandey et al., "BPDET: An Effective Software Bug Prediction Model Using Deep Representation and Ensemble Learning Techniques," Expert Systems with Applications, vol. 144, p. 113085, 2020.
[40] T.-Y. Yu, C.-Y. Huang and N. C. Fang, "Use of Deep Learning Model with Attention Mechanism for Software Fault Prediction," Proc. of the 2021 8th IEEE Int. Conf. on Dependable Systems and Their Applications (DSA), pp. 161–171, Yinchuan, China, 2021.
[41] H. Liang, Y. Yu, L. Jiang and Z. Xie, "Seml: A Semantic LSTM Model for Software Defect Prediction," IEEE Access, vol. 7, pp. 83812–83824, 2019.
[42] M. Alenezi, M. Zagane and Y. Javed, "Efficient Deep Features Learning for Vulnerability Detection Using Character N-gram Embedding," Jordanian Journal of Computers and Information Technology (JJCIT), vol. 7, no. 01, pp. 25-38, 2021.
[43] U. Samal, "Hybrid Approach to Software Fault Prediction Using Particle Swarm Optimization and Fuzzy Time Series," Quality and Reliability Engineering International, vol. 41, no. 5, pp. 2005–2018, 2025.
[44] H. Turabieh, M. Mafarja and X. Li, "Iterated Feature Selection Algorithms with Layered Recurrent Neural Network for Software Fault Prediction," Expert Systems with Appl., vol. 122, pp. 27–42, 2019.
[45] H. Ji et al., "A New Weighted I Bayes Method Based on Information Diffusion for Software Defect Prediction," Software Quality Journal, vol. 27, no. 3, pp. 923–968, 2019.
[46] K. Zhu et al., "Software Defect Prediction Based on Enhanced Metaheuristic Feature Selection Optimization and a Hybrid Deep Neural Network," J. of Sys. and Software, vol. 180, p. 111026, 2021.
[47] B. S. Raghuwanshi and S. Shukla, "SMOTE Based Class-specific Extreme Learning Machine for Imbalanced Learning," Knowledge-Based Systems, vol. 187, p. 104814, 2020.
[48] N. Nnamoko and I. Korkontzelos, "Efficient Treatment of Outliers and Class Imbalance for Diabetes Prediction," Artificial Intelligence in Medicine, vol. 104, p. 101815, 2020.
[49] N. Limsettho et al., "Cross Project Defect Prediction Using Class Distribution Estimation and Oversampling," Information and Software Technology, vol. 100, pp. 87–102, 2018.
[50] O. Al Qasem, M. Akour and M. Alenezi, "The Influence of Deep Learning Algorithms Factors in Software Fault Prediction," IEEE Access, vol. 8, pp. 63945–63960, 2020.
[51] M. Adil, R. Ullah, S. Noor and N. Gohar, "Effect of Number of Neurons and Layers in an Artificial Neural Network for Generalized Concrete Mix Design," Neural Computing and Appl., pp. 1–9, 2022.
[52] J. Zhang et al., "Random-forests-based Network Intrusion Detection Systems," IEEE Trans. on Systems, Man and Cybernetics, Part C (Applications and Reviews), vol. 38, no. 5, pp. 649–659, 2008.
[53] L. Qiao, X. Li, Q. Umer and P. Guo, "Deep Learning Based Software Defect Prediction," Neurocomputing, vol. 385, pp. 100–110, 2020.
[54] A. B. Farid, E. M. Fathy, A. S. Eldin and L. A. Abd-Elmegid, "Software Defect Prediction Using Hybrid Model (CBIL) of Convolutional Neural Network (CNN) and Bidirectional Long Short-term Memory (Bi-LSTM)," PeerJ Computer Science, vol. 7, p. e739, 2021.

"FR-CNN: Software Fault Prediction Using a Novel Hybrid Hierarchical Model", M. M. Jouybari, A. Tajary, M. Fateh and E. Tahanian.

ملخص البحث:

يـرتبط تطـوير منتجـاتٍ برمجيـةٍ عاليـة الجـودة ارتباطـاً مباشـراً بخلوّهـا مـن الأخطـاء غيـر المكتشـفة. وقـد اسـتُخدمت أسـاليب التّنبُّـؤ بأخطـاء البرمجيـات لتقليـل التّكـاليف الإجماليـة للاختبـار والكشـف عـن الأخطـاء المحتملـة. واسـتُخدمت خوارزميـات تعلّـم آلـي متنوعـةٌ لهذا الغرض. إلّا أنّ أداءَها متفاوتٌ فيما يتعلّق بأخطاء البرمجيات.

يقتـرح هـذا البحـث نموذجـاً هجينـاً مبتكـراً يعتمـد علـى دمـج ثـلاث تقنيـاتٍ: شـبكة عصـبية عميقـة متّصـلة بالكامـل (تـم ضـبط معاملاتهـا بدقّـة)، وخوارزميـة الغابـة العشـوائية، وشـبكة عصـبية تلافيفيـة. ومـن خـلال الاسـتفادة مـن نقـاط القـوّة النّاتجـة عـن دمـج هـذه التّقنيـات، يهـدف هـذا النّمـوذج إلـى تحسـين أداء التّنبّـؤ بأخطـاء البرمجيـات مـع الحـدّ مـن مشكلة الإفراط في التّخصيص، ممّا ينتج عنه إطار عملٍ قويٌ وفعّال.

وقـد أجرينـا دراسـاتٍ تجريبيـة موسّـعة لتوضـيح فعاليـة النّظـام المقتـرح فـي التّنبّـؤ بالأخطـاء عبـر سـتة مشـاريع مـن مجموعـة بيانـات (BugHunter). وأظهـرت النّتـائج التّجريبيـة أنّ النّمـوذج المقتـرح يحقّـق أداءً تنافسـياً فـي التّنبّـؤ بالأخطـاء، مـع تطلّبـه لتعقيـد حسـابي أقـلّ وزمـن تـدريب أقصـر. وقـد تفـوّق علـى أسـاليب الـتّعلُّم الآلـي التّقليديـة فـي عـددٍ من مؤشّرات الأداء على مجموعة البيانات المذكورة.