
Production Systems and Information Engineering
Volume 10 (3), pp. 1–18.
http://doi.org/10.32968/psaie.2022.3.1

A NEW APPROACH TO SOFTWARE DEFECT PREDICTION BASED ON CONVOLUTIONAL NEURAL NETWORK AND BIDIRECTIONAL LONG SHORT-TERM MEMORY

NASRALDEEN ALNOR ADAM KHLEEL
University of Miskolc
Hungary Institute of Information Technology
nasr.alnor@uni-miskolc.hu

KÁROLY NEHÉZ
University of Miskolc
Hungary Institute of Information Technology
aitnehez@uni-miskolc.hu

Abstract. Software defect prediction (SDP) plays an important role in improving software quality and reliability while reducing software maintenance cost. The problem in the field of SDP is how to determine the defective source code with high accuracy. To build more accurate predictor models, a lot of features are presented, e.g., static code features, social network features, and process features, etc. Several machine learning (ML) and deep learning (DL) algorithms have been developed and adopted to identify and remove defects from the source code, where previous studies have proved that DL algorithms are promising techniques for predicting software defects. The aim of this study is to investigate the prediction performance of two DL algorithms namely, Convolutional Neural Network (CNN) and Bidirectional Long short-term memory (BI-LSTM) in the domain of SDP. To establish the effectiveness of the proposed approach, the experiments were conducted on the available benchmark datasets which obtained from open-source java projects GitHub repository and the models were evaluated by applying seven evaluation metrics which are accuracy, precision, recall, f-measure, matthews correlation coefficient (MCC), area under the ROC curve (AUC), mean square error (MSE). We found out that the best accuracy obtained on training dataset is 81% by using CNN model, while the best accuracy obtained on validation dataset is 80% by using BI-LSTM model. The best AUC obtained on training dataset is 88% by using CNN model, while the best AUC obtained on validation dataset is 83% by using the both models. It is nearly impossible to rule which model is better than the other so every model can be analyzed separately and the best model according to the problem at hand can be used, therefore, based on the problem of this study, The evaluation results show the effectiveness of our proposed models based on standard performance evaluation criteria.

Keywords: Software defect prediction, Software metrics, Deep learning, Convolutional neural network, Bidirectional Long short-term memory

1. Introduction

These days, software quality assurance is overall the most necessary activity for software developing organizations. A software defect can be defined as errors in the software development process which may cause many problems for users and developers aside and may lead to the failure of the program to meet the desired expectations, which lead to raised development and maintenance costs and reduced customer satisfaction. Where the cost of finding and repairing defects represents one of the costliest software development activities. Early detection of defects helps practitioners allocate additional resources [1, 2]. There are many activities during the software life cycle to identify source code defects such as design review, code inspection, integration testing, functions and units testing, etc. Various research studies have analysed and applied SDP approaches to help prioritize software testing and correction. SDP refers to the techniques that use the historical defect data to build the relationship between software metrics and software defects. Earlier work in domain of SDP concentrated on static source code metrics e.g., size, complexity, and object-oriented metrics etc. as the main predictors of software defects [2, 3]. SDP is a process depends on three main components: dependent variables, independent variables and a model. Dependent variables are the defect data for the particular piece of code (defective or non-defective), which can be binary or ordinal variables. Independent variables are the metrics (inputs) that scores the software code. The model contains the rules or algorithms which predict the dependent variable from the independent variables. To determine the effectiveness of the classifier, the inputs (variables) are split into test and training data sets. Where the training data set is used to create the classifier and then this classifier is used to predict potential defects in the test data set and evaluate these predictions using different performance measures to determine if they are correct or not [4]. DL is a new and very successful area in artificial intelligence, it applies deep neural networks. Recently, DL algorithms have been adopted to improve research tasks in software engineering, especially in the field of SDP [5]. DL is a type of ML that allows computational models consisting of multiple processing layers to learn data representations with multiple levels of abstraction [6]. DL architecture has been widely applied in many fields and used to solve many detections, classification, and prediction problems. DL has drawn more and more attention, because of its powerful feature learning capability, and has been successfully used in many domains, such as speech recognition, image classification, etc. [7, 8] This study selects dataset from the GitHub repository for experimental purposes. Although some experiments in the previous studies [9] are conducted based on this dataset using ML techniques, very few of them are based on DL. Even there is no experiment using CNN and BI-LSTM in the literature. To bridge these gaps, the novelty and main contributions of our work are summarized as follows:

1. In this study, we propose a novel approach based on CNN and BI-LSTM to predict software defects.

2. This study evaluates the effectiveness and efficiency of the proposed approach based on different performance measures.

The structure of this paper is organized as follows. Section 2 presents a discussion on related work. Section 3 presents background on the topics of DL algorithms, CNN, Long-Short Term Memory (LSTM) Networks and BI-LSTM Networks. After that, our research methodology is presented in Section 4. Section 5 presents the experimental results and discussion followed by conclusions in the last section.

2. Related work

The efforts of previous studies toward building accurate prediction models can be categorized into the two following approaches: The first approach is manually designing new features or new combinations of features to represent defects more effectively, and the second approach involves the application of new and improved ML based classifiers. Many research studies in literature apply ML techniques to predict software's defects [4, 9, 10]. Hoa Khanh Dam et al. [3] presented a novel approach based on LSTM architecture to predict source code defects. Abstract syntax tree which representing a source code was used as input for the proposed prediction model to predict if the source code is defective or non-defective. The experiments were evaluated based on two different datasets presented by Samsung and the PROMISE repository. The results showed that the approach was accurate and significant enough to predict source code defects. Rudolf Ferenc et al. [5] proposed a methodology of how to adapt DNNs s for bug prediction. The methodology was applied on a large bug dataset (containing 8780 bugged and 38,838 not bugged Java classes). The results demonstrate that DL with static metrics can indeed boost prediction accuracies. Amirabbas Majd et al. [7] proposed SLDeep using LSTM as learning model, a technique for statement-level SDP based on more than 100,000 C/C ++ programs. The evaluation results show that the proposed model seems to be effective at statement-level SDP and can be adopted. Mohamed Samir et al. [8] proposed a new method using DNN to predict software defects. The method has been compared with some ML algorithms. The results of the experiment showed that the proposed method has a slight improvement over the other methods. Sonali Agarwal and Divya Tomar [11] proposed a new method called the feature selection based Linear Twin Support Vector Machine (LSTSVM) model to predict software defects. The experiment was performed on four PROMISE datasets. The method was evaluated and compared with other existing ML models. The experimental results showed the effectiveness of the proposed method. Jiehan Deng et al. [12] proposed a novel LSTM method to perform SDP, their method can automatically learn semantic and contextual information from the program's ASTs. The experiment was performed on several open-source projects, the results showed that the proposed LSTM method is superior to the state-of-the-art methods. Xin Ye et al. [13] proposed a classification model using a LSTM-network to classify bugs based on 9,000 bug reports from three software projects. The results of the evaluation and comparison

show that the proposed model achieves the best results. Hani Bani-Salameh et al. [14] proposed a framework using LSTM for automatically assigning bugs. The proposed model has been validated on five real projects. The performance of the model was compared with two ML algorithms. The results show that LSTM predicts and assigns the priority of the bug more accurately and effectively. HONGLIANG LIANG et al. [15] proposed Seml, a novel framework for defect prediction using LSTM network based on eight open-source projects. The evaluation results show that the proposed model outperforms three state-of-the-art defect prediction approaches on most of the datasets. Ahmed Bahaa Farid et al. [16] proposed a hybrid model using BI-LSTM and a CNN to predict software defects. The proposed model was evaluated using seven open-source Java projects from the PROMISE dataset. The results show that the proposed model is accurate for predicting software defects. Xuan Zhou and Lu Lu [17] developed a LSTM network based on bidirectional and tree structure (LSTM-BT) to predict software defects based on 8 pairs of Java open-source projects. The evaluation results show that the proposed model performs better compared to several state-of-the-art defect prediction models. Cong Pan et al. [18] proposed an improved CNN model for within project defect prediction (WPDP) and compare the results of the experiment with those of existing CNN studies. The experiment was performed based on a 30-repetition holdout validation and a 10 * 10 cross-validation. The results showed that the CNN model outperformed the state-of-the-art ML models significantly for WPDP. JIEHAN. Kun Zhu et al. [19] proposed a novel just-in-time defect prediction model named DAECNN-JDP based on denoising autoencoder and CNN. The model was evaluated based on six large open-source projects and compared with 11 baseline models. The experimental results show that the DAECNN-JDP model outperforms these baseline models. Jian Li et al. [20] proposed a framework based on the programs' Abstract Syntax Trees called Defect Prediction via CNN (DP-CNN). the model was evaluated based on seven open-source projects in terms of F-measure. The experimental results show that on average, the DP-CNN model improves the state-of-the-art method by 12%. Ashima Kukkar et al. [21] proposed a novel DL model for multiclass severity classification called Bug Severity classification using a CNN and Random Forest with Boosting based on five open-source projects. The results prove that the proposed model enhances the performance of bug severity classification over state-of-the-art techniques.

3. Background

In this section, we present the background about the topics of DL algorithms, CNN, Long-Short Term Memory (LSTM) Networks and BI-LSTM Networks.

3.1. Deep learning algorithms in software defect prediction

Deep learning (DL) algorithms have received extensive attention in the field of software engineering for a considerable period. Therefore, recently DL algorithms

have been adopted to enhance research tasks in the field of SDP. One effort to create effective DL models valuable for the software engineering community is the classification and regression based on source code metrics, but the most popular DL algorithms used to predict software defects are classification algorithms. Even though these efforts share the fundamentals of analyzing code metrics, they also vary in terms of accuracy, complexity, and the input data they require to predict a defect [2, 9]. Commonly, there are three classes of prediction models that are used to predict defective software modules, binary class classification of defects, number or density of defects prediction, and severity of defect prediction. The binary class is the most frequently used in SDP. Researchers have provided different classification techniques for binary class classification of defects, including statistical techniques, supervised techniques, semi supervised techniques, and unsupervised techniques. Most of the studies in the literature have used statistical and supervised learning techniques [22].

3.2. Convolutional Neural Network

Convolutional Neural Network (CNN) is a special type of deep neural network or a class of convolutional feedforward neural network used to process data that has a known, grid-like topology. It is constructed to mimic the visual perception of biological processes and can be used for both supervised learning and unsupervised learning. CNN has been tremendously successful in practical applications, including speech recognition, image classification, and natural language processing [1, 6]. CNN model is inspired by the typical CNN architecture used in image classification and consists of a feature extraction part and a classification part as shown in the Figure 1. These parts consist of multiple layers are convolution, batch normalization, and maximum merge layers. These layers constitute the hidden layer of the architecture. The convolutional layer performs convolution operations based on the specified filter and kernel parameters and calculates the network weights to the next layer, while the maximum pooling layer achieves a reduction in the dimension of the feature space. Batch normalization is used to mitigate the effect of different input distributions for each training mini-batch for the purpose of improving training. Activation functions enabling the training of CNN model in a fast and accurate manner. There are many activation functions used in CNN such as Sigmoid, Rectified Linear Unit (ReLU) and hyperbolic tangent (Tanh) [18]. In this study, we used two activation functions, the ReLU function for the input and hidden layers and the Sigmoid function for the output layer as shown in equations below.

$$h_i{}^m = ReLU(W_i{}^{m-1} \times V_i{}^{m-1} + b^{m-1}) \quad (1)$$

Where $h_i{}^m$ represents convolutional layer, $W_i{}^{m-1}$ represents the weights of neuron, $V_i{}^{m-1}$ represents the nodes, and b^{m-1} represents the bias layer.

$$S(x) = \frac{1}{1+ e^{-\sum_k W_i+X_i+b}} \quad (2)$$

Where X_i represents the input, W_i is the weight of the input and b is the bias.

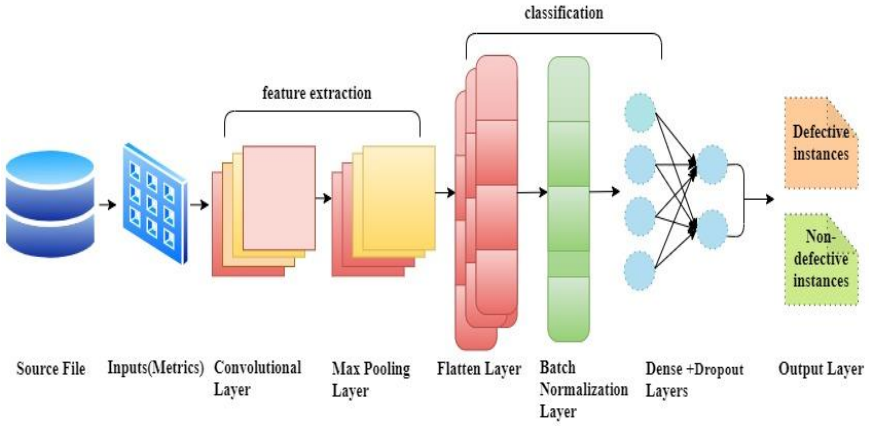

Figure 1. CNN Model for SDP

3.3. Long-Short Term Memory (LSTM) Networks and BI-LSTM Networks

Long-Short Term Memory (LSTM) Networks are a special type of RNN used in the field of DL, which are designed to recognize patterns in data sequences. LSTM Networks were introduced to avoid or handling long term dependencies problem without being affected by an unstable gradient. This problem frequently occurs in regular RNN when connecting previous information to new information [13, 23]. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. Due to the ability of the LSTM network to recognize longer sequences of time-series data, LSTM models can provide high predictive performance in SDP. [24]. More recently, Bidirectional long-short term memory (BI-LSTM) are a new way to train data by expanding the capabilities of LSTM networks, it uses two separate hidden layers to train the input data twice in the forward and backward directions as shown in Figure 2. With the regular LSTM Networks, the input flow in one direction, either backwards or forward. BI-LSTM Networks are the process of making any neural network have the sequence information in both directions (a sequence processing model that consists of two LSTM): one taking the input in a forward direction (past to future), and the other in a backwards direction (future to past). [14, 23, 24]. we build a BI-LSTM Network, because the defective source code is closely related to its previous and subsequent source code segments. The idea behind BI-LSTM Networks is to exploit

spatial features to capture bidirectional temporal dependencies from historical data to overcome the limitations of traditional RNN. Standard RNNs take sequences as inputs, and each step of the sequence refers to a certain moment. For a certain moment t, the output o_t not only depends on the current input x_t but is also influenced by the output from the previous moment $t-1$. The output of moment (t) can be formulated as the following equations:

$$h_t = f(U \times x_t + W \times h_{t-1} + b) \tag{3}$$

$$o_t = g(V \times h_t + c)$$

Where U, V, and W denote the weights of the RNN, b and c denote the bias, f and g are the activation functions of the neurons. The cell state carries the information from the previous moments and will flow through the entire LSTM chain, which is the key that LSTM can have long-should be filtered from the previous moment, the output of forget gate can be formulated as the following equation:

$$f_t = \sigma(W_f . [h_{t-1}, x_t] + b_f) \tag{4}$$

Where σ denotes the activation function, W_f and b_f denote the weights and bias of the forget gate, respectively. The input gate determines what information should be kept from the current moment, and its output can be formulated as the following equation:

$$i_t = \sigma(W_i . [h_{t-1}, x_t] + b_i) \tag{5}$$

Where σ denotes the activation function, W_i and b_i denote the weights and bias of the input gate, respectively. With the information from forget gate and input gate, the cell state C_{t-1} is updated through the following formula:

$$\check{C}_t = \tanh(W_c . [h_{t-1}, x_t] + b_c) \tag{6}$$

$$\check{C}_t = f_t \times C_{t-1} + i \times \check{C}_t)$$

$\check{C}_t$ is a candidate value that is going to be added into the cell state and C_t is the current updated cell state. Finally, the output gate decides what information should be outputted according to the previous output and current cell state.

$$o_t = \sigma(W_o . [h_{t-1}, x_t + b_o] \tag{7}$$
$$h_t = o_t \times \tanh(C_t).$$

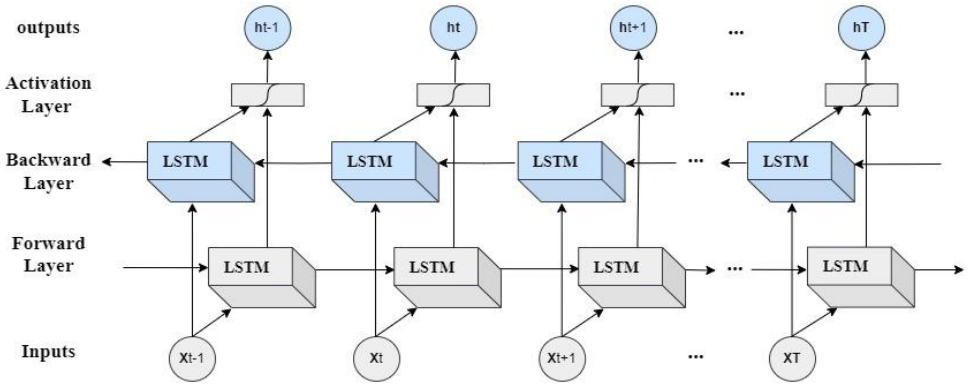

Figure 2. Interacting layers of the repeating module in a BI-LSTM Network

4. The proposed methodology

The aim of this study is to build SDP models that would outperform other defect prediction models. This study proposed a method to train and test SDP model based on high-performance DL algorithms, namely CNN and BI-LSTM. A series of steps have been taken and described such as data modelling and collection, data pre-processing and features selection, models building and evaluation. Figure 3 illustrates the overview of the proposed approach for SDP where each step is described in the following sections.

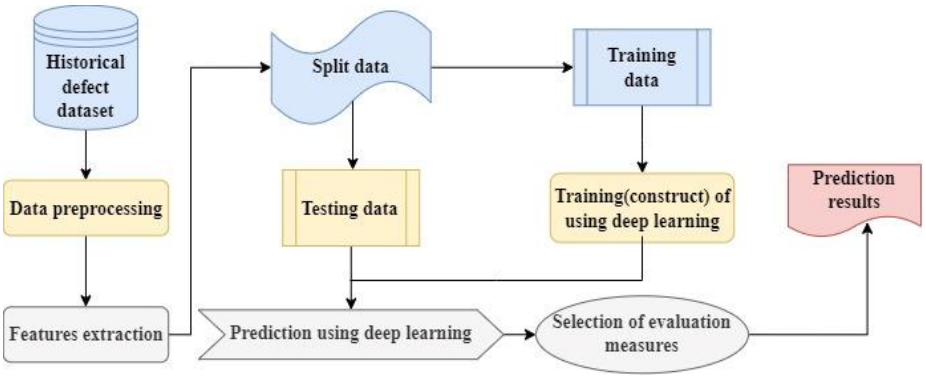

Figure 3. Proposed SDP approach

4.1. Data modelling and collection

Dataset selection is an important task in the problem of DL, and classification models perform better if the dataset is more relevant to the problem. Having a large dataset is fundamental to train DL models and allow generalization of the obtained results. Software defect database consists mostly of three components: collection of software metrics, defect details (Defective or non-defective code) and meta information [25].

In this study, we select public benchmark datasets for SDP which obtained from the GitHub repository (GHPR Dataset) [9]. The GHPR dataset contains 21 static metrics for the total 6052 instances (3026 defective instances and 3026 non-defective instances), which is the data used for the baseline approaches. This dataset focuses on class-level and metric-level code metrics in Java projects that computed by CK-open-source tool. Table 1 shows the description of the metrics. Dataset links are: https://github.com/feiwww/GHPR_dataset

Table 1. Class-level and metric-level code metrics calculated by the CK-open-source tool [9]

Metrics	**Description**
Coupling Between Objects (CBO)	Counts the number of dependencies a class has.
Weight Method Class or McCabe's complexity (WMC)	It counts the number of branch instructions in a class.
Depth Inheritance Tree (DIT)	It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object).
Response for a Class (RFC)	Counts the number of unique method invocations in a class.
Lack of Cohesion of Methods (LCOM)	Calculates LCOM metric.
Total Methods	Counts the number of methods.
Total Fields	Counts the number of fields.
NOSI	Number of static invocations. Counts the number of invocations to static methods.
Lines of code (LOC)	It counts the lines of count, ignoring empty lines.
Quantity of returns (Return Qty)	The number of return instructions.
Quantity of loops (Loop Qty)	The number of loops (i.e., for, while, do while, enhanced for).
Quantity of comparisons (Comparisons Qty)	The number of comparisons (i.e., == and! =).
Quantity of try/catches (Try Catch Qty)	The number of try/catches.
Quantity of parenthesized expressions (Parenthesized Exps Qty)	The number of expressions inside parenthesis.
String Literals Qty	The number of string literals (e.g., "John Doe").
Quantity of Number (Numbers Qty)	The number of numbers (i.e., int, long, double, float) literals.
Quantity of Variables (Assignments Qty)	Number of declared variables.
Quantity of Math Operations (Math Operations Qty)	The number of math operations (times, divide, remainder, plus, minus, left shit, right shift).
Quantity of Variables (Variables Qty)	Number of declared variables.
Max nested blocks (Max Nested Blocks)	The highest number of blocks nested together.
Number of unique words (Unique Words Qty)	Number of unique words in the source code.

4.2. Data Pre-processing and Features Selection

Pre-processing the collected data is one of the important stages before constructing the model. To generate a good model, the quality of data needs to be considered. Not all data collected is suitable for training and model building. Anyhow the inputs will greatly impact the performance of the model and later moreover affect the output. Data pre-processing is known as a group of techniques that are applied to the data to improve the quality of the data before model building for the purpose of removing noise and unwanted outliers from the data set, dealing with missing values, feature type conversion, etc. [21, 26, 27]. The data set used in this study is a clean copy. Normalization is necessary to convert the values into scaled values (scaling of the data in numeric variables in the range of 0 to 1) to increase the efficiency of the model. Therefore, the data set was normalized using Min–Max normalization. The formula for calculating normalized score can be described by (8). Feature selection is a crucial step to select the most discriminative features from the list of features using appropriate feature selection methods. The goal of feature selection is to select the features which are more relevant to the target class from high-dimensional features and remove the features which are redundant and uncorrelated. Feature extraction facilitates the conversion of pre-processed data into a form that the classification engine can use [14, 28]. In this study, no feature was removed, all features in the data sets were identified as independent variables of the models and feature scaling technique was applied to make the output the same standard. Figure 4 shows a heat map of the data features.

$$x_i = (x_i - X\,min)/(X\,max - X\,min) \tag{8}$$

Where max(x) and min(x) represent the maximum and minimum value of the attribute x respectively.

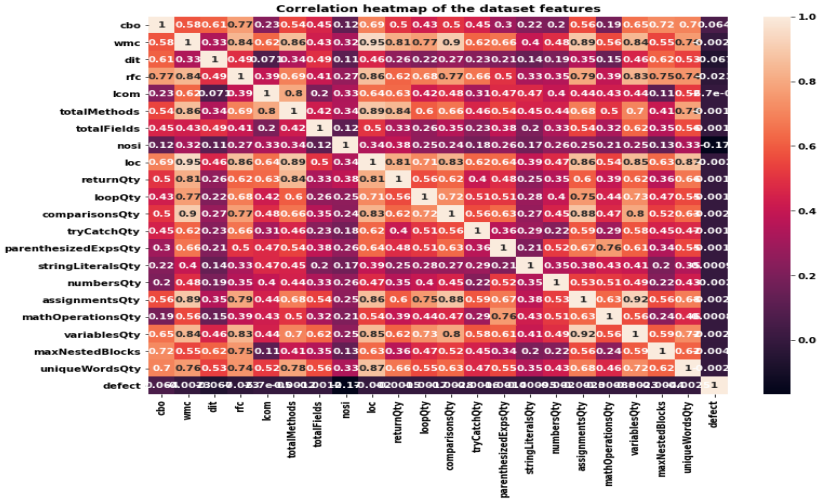

Figure 4. Heatmap representing correlation-based feature selection

4.3. Models building and evaluation

Different ML algorithms are used to build defect prediction models and each algorithm has its own benefits. Most studies of SDP divide the data into two sets: a training set and a test set. The training set is used to train the model, whereas the testing set is used to evaluate the performance of the defect's prediction model. Once a defects prediction model is built, its performance needs to be evaluated [29, 30]. Implementation framework of our models: We use Keras as a high-level API based on TensorFlow to build our models for simplicity and correctness, training is performed with 80% of the dataset (random selection of features), while the remaining 20% is used for validation. We evaluate the performance of our proposed models based on a set of common performance measures such as confusion matrices, MCC, AUC, and MSE as a Loos Function. MCC is a measure used for model evaluation by measure the difference between the predicted values and actual values. It takes into account true and false positives and negatives. AUC, which plots the false positive rate on the x-axis and true positive rate on the y-axis over all possible classification thresholds. MSE is a metric which measures the amount of error the model. It assesses the average squared difference between the actual and predicted values. A confusion matrix is a specific table used to measure the performance of a model. A confusion matrix summarizes the results of the testing algorithm and presents a report of True Positive (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). The subsections below describe the confusion matrix and performance measures applied as it is shown in Table 2 and equations.

Table 2. The Confusion matrix

Predicted	Actual	
	Class X	Class Y
Class X	TN	FP
Class Y	FN	TP

$$\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN}). \quad (9)$$

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP}). \quad (10)$$

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN}). \quad (11)$$

$$\text{F} - \text{measure} = (2 * \text{Recall} * \text{Precision}) / (\text{Recall} + \text{Precision}). \quad (12)$$

$$\text{MCC} = \text{TP} * \text{TN} - \text{FP} * \text{FN} / \sqrt{(\text{TP} + \text{FP}) * (\text{TP} + \text{FN}) * (\text{TN} + \text{FP}) * (\text{TN} + \text{FN})} \quad (13)$$

$$\text{AUC} = \frac{\Sigma_{ins_i \in Positive\ Class} \text{rank}(ins_i) - \frac{\text{M}(\text{M}+1)}{2}}{\text{M} . \text{N}} \quad (14)$$

Where $\sum_{ins_i \in Positive\ Class} \text{rank}(ins_i)$ is the sum of ranks of all positive samples, M and N are the number of positive samples and negative samples, respectively.

$$\text{MSE} = \frac{1}{\text{n}} \sum_{\text{i}=1}^{\text{n}} (\text{x(i)} - \text{y(i)})^2 \tag{15}$$

Where n is the number of the observations, x(i) is the actual value, y(i) is the observed or predicted value for the i^{th} observation.

5. Experimental results and discussion

The experiments have been performed on open-source Java projects to predict software defects based on class-level and metric-level code metrics. The proposed models are developed based on classification patterns and compared together for classifying the software defects. The comparison is performed on the basis of seven standard classification parameters.

Results: Table 3 presents the training results of our CNN and BI-LSTM Models on benchmark dataset in terms of accuracy, precision, recall, F-Measure, MCC, AUC, and MSE. We notice that the highest accuracy was achieved by CNN model which is 81%, the lowest accuracy was achieved by BI-LSTM model which is 80%. The highest precision was achieved by CNN model which is 79%, the lowest precision was achieved by BI-LSTM model which is 77%. The highest recall was achieved by CNN model which is 85%, the lowest recall was achieved by BI-LSTM model which is 84%. The highest f-measure was achieved by CNN model which is 82%, the lowest f-measure was achieved by BI-LSTM model which is 80%. The highest MCC was achieved by CNN model which is 61%, the lowest MCC was achieved by BI-LSTM model which is 59%. The highest AUC was achieved by CNN model which is 88%, the lowest AUC was achieved by BI-LSTM model which is 84%. The highest MSE was achieved by BI-LSTM model which is 0.152, the lowest MSE was achieved by CNN model which is 0.140.

Table 4 presents the validation results of our CNN and BI-LSTM Models on benchmark dataset in terms of accuracy, precision, recall, F-Measure, MCC, AUC, and MSE. We notice that the highest and lowest accuracy was achieved by the both models (CNN and BI-LSTM) which is 80%. The highest and lowest precision was achieved by the both models (CNN and BI-LSTM) which is 77%. The highest recall was achieved by BI-LSTM model which is 85%, the lowest recall was achieved by CNN model which is 84%. The highest f-measure was achieved by BI-LSTM model which is 81%, the lowest f-measure was achieved by CNN model which is 80%. The highest MCC was achieved by BI-LSTM model which is 60%, the lowest MCC was achieved by CNN model which is 59%. The highest AUC was achieved by BI-LSTM model which is 84%, The lowest AUC was achieved by CNN model which is 83%. The highest MSE was achieved by CNN model which is 0.158, the lowest MSE was achieved by BI-LSTM model which is 0.151.

Figures 5 and 6 below show the training and validation accuracy and training and validation loss of the models on dataset. Figures 7 and 8 below show the AUC obtained by the models on dataset. After comparing the results obtained by the proposed models, we noticed that both models got the best scores on the both training and validation datasets, which indicated that the proposed models performed well in SDP.

We also compared our results with the results obtained in previous studies based on the accuracy and AUC. Table 5 compare the values of accuracy and AUC obtained by our models and the values of accuracy and AUC in previous studies. According to compression results, some of the results in the previous studies are better than ours, but in the most cases, our approach is outperforming the other state-of-the-art approaches on the benchmark datasets in terms of evaluation measures such as accuracy and AUC.

Table 3. Performance measures for the proposed models over dataset –Training Results

proposed models	Performance measures						
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**MSE**
CNN	0.81	0.79	0.85	0.82	0.61	0.88	0.140
BI-LSTM	0.80	0.77	0.84	0.80	0.59	0.84	0.152

Table 4. Performance measures for the proposed models over dataset – Validation Results

proposed models	Performance measures						
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**MSE**
CNN	0.80	0.77	0.84	0.80	0.59	0.83	0.158
BI-LSTM	0.80	0.77	0.85	0.81	0.60	0.84	0.151

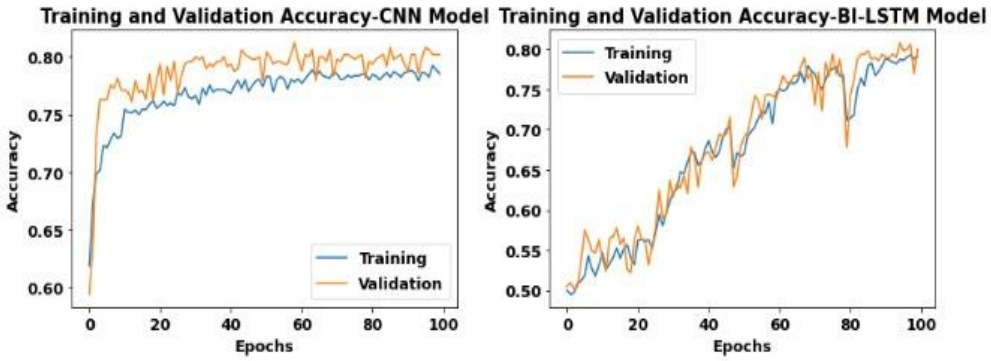

Figure 5. Training and Validation Accuracy for the models over dataset

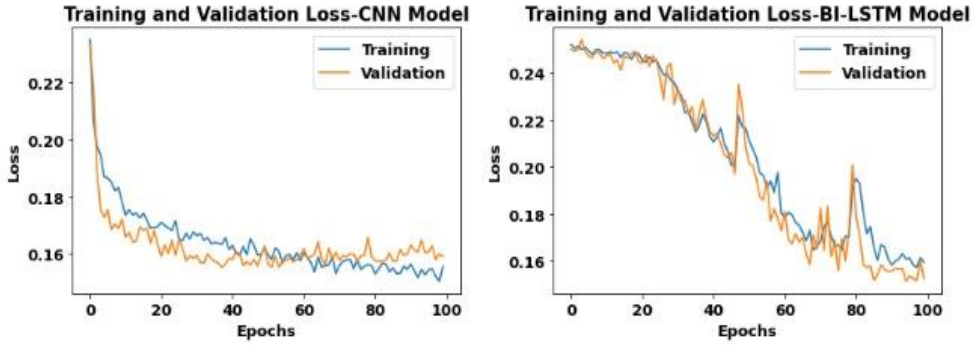

Figure 6. Training and Validation Loss for the models over dataset

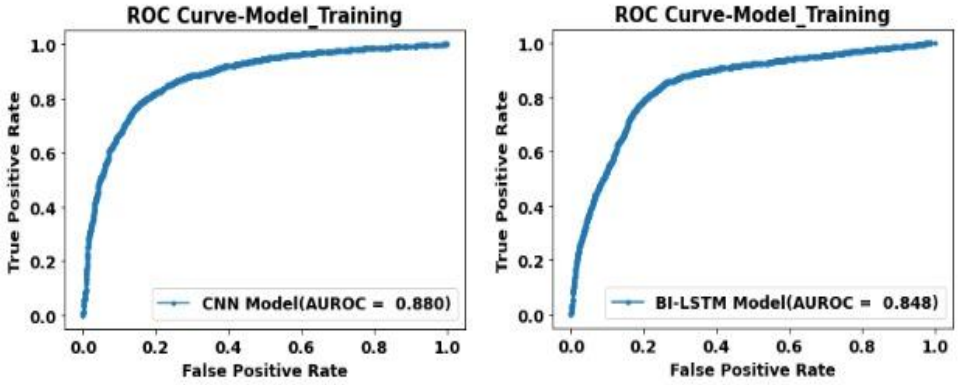

Figure 7. ROC curves for models training over dataset

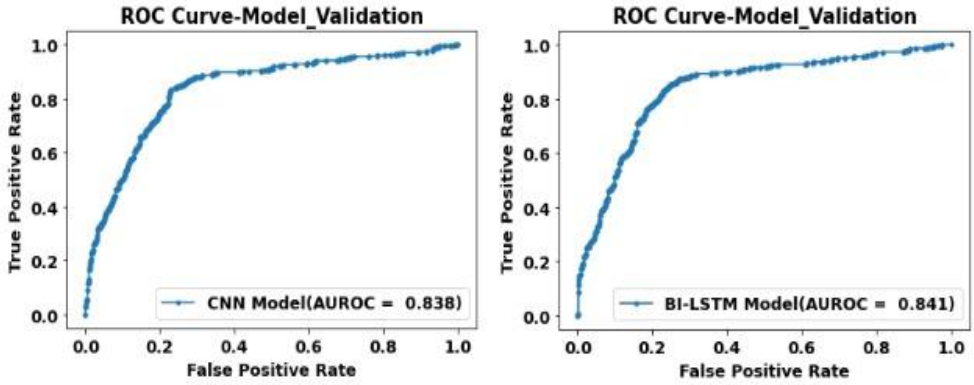

Figure 8. ROC curves for models validation over dataset

Table 5. Comparison of the proposed approach with other existing approaches based on the accuracy and AUC

Approaches	Datasets	Accuracy	AUC
deep neural networks [5]	Unified Bug Dataset (PROMISE Dataset, Bug Prediction Dataset, GitHub Bug Dataset)	–	0.81
LSTM [7]	Code4Bench for C/C++ code	0.70	–
Random Forest [7]	Code4Bench for C/C++ code	0.62	–
deep neural network [8]	PROMISE datasets (CM1, JM1, KC1)	0.87, 0.79, 0.75	–
Linear Twin Support Vector Machine [11]	PROMISE datasets (CM1, PC1, KC1, KC2)	0.90, 0.94, 0.86, 0.86	–
LSTM [13]	Bug report datasets (Eclipse Platform UI, JDT)	0.67, 0.76	–
LSTM [14]	JIRA dataset	0.89	–
CBIL model [16]	PROMISE datasets (Camel, Jedit, Lucene, Poi, Synapse, Xalan, Xerces)	–	0.96, 0.91, 0.83, 0.95, 0.95, 0.76, 0.98
Convolutional Neural Network and Random Forest with Boosting [21]	Bug report datasets (Mozilla, Eclipse, JBoss, Open FOAM, Firefox)	0.94, 0.95, 0.94, 0.98, 0.97	–
Our models – training (CNN, BI-LST	GHPR Dataset	0.81, 0.80	0.88, 0.84
Our models – validation (CNN, BI-LSTM)	GHPR Dataset	0.80, 0.80	0.83, 0.84

6. Conclusion

Software defects have a major impact of software development life cycle and defect prevention plays an important role in the software quality assurance. In previous studies there are many papers discussed different types of datasets, presented different software metrics, and examine the applicability of different ML algorithms and evaluation criteria. In this study, we proposed approach based on two DL models to predict software defects. We conducted the experiments based on the dataset were obtained from open-source java projects GitHub repository. In order to validate the proposed approach, different performance measures were used. The evaluation results showed that the proposed approach have a good performing and can significantly improve upon the state-of-the-art approaches. In our future work, our approach will be evaluated on various datasets to validate the robustness and we would like to combine more DL techniques with data balancing methods to improve the accuracy of SDP.

Acknowledgement

The authors gratefully acknowledge the financial assistance from the Institute of Information Science, Faculty of Mechanical Engineering and Informatics, University of Miskolc.

References

[1] Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P. and Misilov, V.E.: A survey on software defect prediction using deep learning. *Mathematics*, vol. 9, no.11, p. 1180, 2021. https://doi.org/10.3390/math9111180

[2] Omri, S. and Sinz, C.: Deep learning for software defect prediction: a survey. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 209–214, Association for Computing Machinery, New York, NY, USA, 2020, https://doi.org/10.1145/3387940.3391463.

[3] Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T. and Kim, C.J.: *A deep tree-based model for software defect prediction*. arXiv preprint arXiv:1802.00921, 2018, https://doi.org/10.48550/arXiv.1802.00921.

[4] Miholca, D.L., Czibula, G. and Tomescu, V.: COMET: A conceptual coupling based metrics suite for software defect prediction. *Procedia Computer Science*, vol. 176, pp. 31–40, 2020, https://doi.org/10.1016/j.procs.2020.08.004.

[5] Ferenc, R., Bán, D., Grósz, T. and Gyimóthy, T.: Deep learning in static, metric-based bug prediction. *Array*, vol. 6, p. 100021, 2020. https://doi.org/10.1016/j.array.2020.100021

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

[7] Majd, A., Vahidi-Asl, M., Khalilian, A., Poorsarvi-Tehrani, P. and Haghighi, H.: SLDeep: Statement-level software defect prediction using deep-learning model on static code features. *Expert Systems with Applications*, vol. 147, p. 113156, 2020. https://doi.org/10.1016/j.eswa.2019.113156

[8] Samir, M., El-Ramly, M. and Kamel, A.: Investigating the use of deep neural networks for software defect prediction. In *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–6, Abu Dhabi, United Arab Emirates, 2019, https://doi.org/10.1109/AICCSA47632.2019.9035240.

[9] Xu, J., Wang, F. and Ai, J.: Defect prediction with semantics and context features of codes based on graph representation learning. *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 613–625, 2020, https://doi.org/10.1109/TR.2020.3040191.

[10] Al-Ahmad, B.: Using Code Coverage Metrics for Improving Software Defect Prediction. *J. Softw.*, vol. 13, no. 12, pp. 654–674, 2018. https://doi.org/10.17706/jsw.13.12.654-674

[11] Agarwal, S. and Tomar, D.: A feature selection based model for software defect prediction. *International Journal of Advanced Science and Technology*, vol. 65, pp. 39–58, 2014, http://dx.doi.org/10.14257/ijast.2014.65.04.

[12] Deng, J., Lu, L. and Qiu, S.: Software defect prediction via LSTM. *IET Software*, vol. 14, no. 4, pp. 443–450, 2020, https://doi.org/10.1049/iet-sen.2019.0149.

[13] Ye, X., Fang, F., Wu, J., Bunescu, R. and Liu, C.: Bug Report Classification using LSTM architecture for more accurate software defect locating. In *International Conference on Machine Learning and Applications (ICMLA)*, IEEE, pp. 1438–1445, Orlando, FL, USA, 2018, https://doi.org/10.1109/ICMLA.2018.00234.

[14] Bani-Salameh, H. and Sallam, M.: A deep-learning-based bug priority prediction using RNN-LSTM neural networks. *e-Informatica Software Engineering Journal*, vol. 15, no. 1, pp. 29–45, 2021, https://doi.org/10.37190/e-Inf210102.

[15] Liang, H., Yu, Y., Jiang, L. and Xie, Z.: Seml: A semantic LSTM model for software defect prediction. *IEEE Access*, vol. 7, pp. 83812–83824, 2019. https://doi.org/10.1109/ACCESS.2019.2925313

[16] Farid, A.B., Fathy, E.M., Eldin, A.S. and Abd-Elmegid, L.A.: 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, https://doi.org/10.7717/peerj-cs.739.

[17] Zhou, X. and Lu, L.: Defect prediction via LSTM based on sequence and tree structure. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 366–373, IEEE, Macau, China, 2020. https://doi.org/10.1109/QRS51102.2020.00055

[18] Pan, C., Lu, M., Xu, B. and Gao, H.: An improved CNN model for within-project software defect prediction. *Applied Sciences*, vol. 9, no. 10, p. 2138, 2019. https://doi.org/10.3390/app9102138

[19] Zhu, K., Zhang, N., Ying, S. and Zhu, D.: Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software*, vol. 14, no. 3, pp. 185–195, 2020. https://doi.org/10.1049/iet-sen.2019.0278

[20] Li, J., He, P., Zhu, J. and Lyu, M.R.: Software defect prediction via convolutional neural network. In *international conference on software quality, reliability and security (QRS)*, pp. 318–328, IEEE, Prague, Czech Republic, 2017. https://doi.org/10.1109/QRS.2017.42

[21] Kukkar, A., Mohana, R., Nayyar, A., Kim, J., Kang, B.G. and Chilamkurti, N.: A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting. *Sensors*, vol. 19, no. 13, p. 2964, 2019, https://doi.org/10.3390/s19132964.

[22] Alsawalqah, H., Hijazi, N., Eshtay, M., Faris, H., Radaideh, A.A., Aljarah, I. and Alshamaileh, Y.: Software defect prediction using heterogeneous ensemble classification based on segmented patterns. *Applied Sciences*, vol. 10, no. 5, p. 1745, 2020, https://doi.org/10.3390/app10051745.

[23] Sethi, T.: Improved approach for software defect prediction using artificial neural networks. In 2016 *5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pp. 480–485, IEEE, Noida, India, 2016, https://doi.org/10.1109/ICRITO.2016.7785003.

[24] Fan, G., Diao, X., Yu, H., Yang, K. and Chen, L.: Software defect prediction via attention-based recurrent neural network. *Scientific Programming*, p. 6230953, 2019. https://doi.org/10.1155/2019/6230953

[25] Sirshar, M., Mir, H., Amir, K. and Zainab, L.: Comparative Analysis of Software Defect Prediction Techniques. automotive engineering, *Preprints*, 2019, 2019120075

[26] Zhao, L., Shang, Z., Zhao, L., Zhang, T. and Tang, Y.Y.: Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks. *Neurocomputing*, vol. 352, pp. 64–74, 2019, https://doi.org/10.1016/j.neucom.2019.03.076.

[27] Miholca, D.L., Czibula, G. and Czibula, I.G.: A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks. *Information Sciences*, vol. 441, pp. 152–170, 2018. https://doi.org/10.1016/j.ins.2018.02.027

[28] Khan, M.Z.: Hybrid Ensemble Learning Technique for Software Defect Prediction. *International Journal of Modern Education & Computer Science*, vol. 12, no. 1, pp. 1–10, 2020, https://doi.org/10.5815/ijmecs.2020.01.01.

[29] Kamei, Y. and Shihab, E.*:* Defect prediction: Accomplishments and future challenges. In 2016 *IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5, pp. 33–45, IEEE, Osaka, Japan, 2016. https://doi.org/10.1109/SANER.2016.56

[30] Yang, Z. and Qian, H.: Automated Parameter Tuning of Artificial Neural Networks for Software Defect Prediction. In *Proceedings of the 2nd International Conference on Advances in Image Processing*, pp. 16–18, Association for Computing Machinery, Chengdu, China, 2018, https://doi.org/10.1145/3239576.3239622.