IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 29 August 2022, accepted 2 October 2022, date of publication 12 October 2022, date of current version 18 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3213844

RESEARCH ARTICLE

Cognitive Complexity and Graph Convolutional Approach Over Control Flow Graph for Software Defect Prediction

MANSI GUPTA, KUMAR RAJNISH, AND VANDANA BHATTACHARJEE
Department of Computer Science and Engineering, Birla Institute of Technology Mesra, Ranchi 835215, India
Corresponding author: Mansi Gupta (phdcs10003.19@bitmesra.ac.in)

ABSTRACT The software engineering community is working to develop reliable metrics to improve software quality. It is estimated that understanding the source code accounts for 60% of the software maintenance effort. Cognitive informatics is important in quantifying the degree of difficulty or the efforts made by developers to understand the source code. Several empirical studies were conducted in 2003 to assign cognitive weights to each possible basic control structure of software, and these cognitive weights are used by several researchers to evaluate the cognitive complexity of software systems. In this paper, an effort has been made to categorize the Control Flow Graphs (CFGs) nodes according to their node features. In our case, we extracted seven unique features from the program, and each unique feature was assigned an integer value that we evaluated through Cognitive Complexity Measures (CCMs). We then incorporated CCMs' results as a node feature value in CFGs and generated the same based on the node connectivity for a graph. In order to obtain the feature representation of the graph, a node vector matrix is then created for the graph and passed to the Graph Convolutional Network (GCN). We prepared our data sets using GCN output and then built Deep Neural Network Defect Prediction (DNN-DP) and Convolutional Neural Network Defect Prediction (CNN-DP) models to predict software defects. The Python programming language is used, along with Keras and TensorFlow. Three hundred twenty Python programs were written by our talented UG and PG students, and all experiments were carried out during laboratory classes. Together with three skilled lab programmers, they compiled and ran each individual program and detected defect/no-defect programs before categorizing them into three different classes, namely Simple, Medium, and Complex programs. Accuracy, Receiver Operating Characteristics (ROC), Area Under Curve (AUC), F-measure, Precision and hyper-parameter tuning procedures are used to evaluate the approaches. The experimental results show that the proposed models outperformed state-of-the-art methods such as Nave Bayes (NB), Decision Tree (DT), Support Vector Machine (SVM), and Random Forest (RF) in all evaluation criteria.

INDEX TERMS Basic control structures, cognitive complexity measures, control flow graphs, graph convolutional network, neural network, software defect prediction, cognitive informatics.

I. INTRODUCTION

Software defect is miscalculation within the code or incorrect behavior in software execution, conjointly outlined as failure to satisfy meant or such that requirements. code responsibility is thought to be one in all the crucial issues in software engineering. Thus, the models accustomed guarantee software quality is required, and therefore the software defect prediction model is one of them. Defect prediction will estimate the foremost defect-prone software elements exactly and facilitate developers allot restricted resources to those bits of the systems that are possibly to contain defects in testing and maintenance phases [1].

The traditional approaches concentrate on creating and fusing program features. The majority of product metrics are based on statistics on source code. For instance, Halstead

The associate editor coordinating the review of this manuscript and approving it for publication was Yudong Zhang.

metrics are determined by counting the number of operators and operands [2], Chidamber and Kemerer (CK) metrics are determined by counting the number of functions and inheritance [3], and McCabe's metric calculates a program's complexity by examining its control flow graph [4].

Unfortunately, ancient ways of software package defect prediction primarily specialize in making static code metrics that are fed into machine learning classifiers to predict code defects. Meanwhile, intensive machine learning algorithms Decision Tree (DT) [5], Random Forest (RF) [6], Support Vector Machine (SVM) [7], and Naive Bayes (NB) [8] are utilized by several researchers to classify software modules and verify whether or not every module is defect prone or not, so train their model accordingly. Deep neural network (DNN) and Convolutional Neural Network (CNN) models made victimization acceptable sophistication selections are essential for achieving the specified classifier performance. This can be particularly vital once predicting the fault nature of software package modules. When properly known, this might facilitate to scale back testing prices by guiding a lot of attention to the modules identified as fault prone. Since deep learning design will with efficiency capture very advanced nonlinear characteristics, deep learning has become a superb technique for automatic feature generation. Some researchers [9], [10] have already used deep learning algorithms, suggestive of Deep Belief Network (DBN) and Convolutional Neural Network (CNN), in learning linguistics options from programs Abstract Syntax Trees (ASTs), and verified that it outperforms typical camp-made features in defect prediction. This is often as a result of deep learning algorithms have the power to come up with powerful features. Furthermore, ancient handcrafted features reminiscent of modularity, centrality, and node degree are still utilized in defect prediction modelling victimization network analysis technologies. Network illustration learning, a rising deep learning technology, becomes a completely unique approach for mechanically learning latent options of nodes in a very network [11] and receives heaps of attention. As a result, using representation learning to extract structural information from code files then applying the learned features to defect prediction might improve the performance of existing prediction models significantly. Early neural network deviants may solely be enforced using regular data; however, they need found tremendous success as GCN are helpful because several information sets within the space have non-regular underlying graph structures.

Large-scale software systems are acknowledged as being extremely complex and singular artefacts that defy all known physical laws. Software engineering must establish its own framework and laws, which are thought to be primarily based on cognitive informatics, in order to become a mature engineering discipline. Understanding the core traits of software systems is the focus of Cognitive Informatics (CI). Cognitive complexity describes the effort it takes a person to complete a task or the difficulty encountered when trying to understand the source code. According to CI, the input, output and other information that the software contains, as well as its internal flow of control, determine how functionally complex the software is [12]. Numerous cognitive metrics have been developed to assess the complexity of software components. Section II covers a comprehensive review of his work (Related Studies). In contrast to previous studies, we summarize our contributions to the current state of research as follows:

1) Three Hundred twenty Python programs has been prepared by our skilled UG and PG students and all experiments were conducted during laboratory classes. Along with three skilled lab programmers have compiled and run each individual program and have detected defect/no-defect programs and finally they have categorized in into three different categories of classes i.e., Simple, Medium and Complex programs.
2) An effort has been made to categorize the Control Flow Graphs (CFGs) nodes according to their node features (in our case, we extracted seven unique features from the program) and each unique feature was assigned an integer value that we evaluated through Cognitive Complexity Measures CCMs) and incorporated CCMs' results as a node feature value in CFGs and generated the same based on the node connectivity for a graph.
3) In order to obtain the feature representation of the graph, a node vector matrix is then created for the graph and passed to the Graph Convolutional Network (GCN) which aggregates information and generates useful representation for nodes in a graph. We prepared our data sets (Simple, Medium and Complex) using GCN output.
4) Finally, we formulate Deep Neural Network Defect Prediction (DNN-DP) and Convolutional Neural Network Defect Prediction (CNN-DP) models to predict software defects.

The rest of this paper is organized as follows: Section II is an overview of the work related to this topic. Sections III describe related theories and propose approaches to program level defect prediction, cognitive complexity measures, GCN, CNN and Deep Neural Network (DNN). Section IV describe the overall architecture of propose DNN and CNN approaches to defect prediction and parameter tuning procedures. Section V contains detailed experimental settings and key results. Some threats to validity that may affect our research are presented in Section VI. Finally, we finish work in Section VII and present the agenda for future work.

II. BACKGROUND OF THE RELATED WORK

In this section, we present the review of research papers on cognitive complexity measurement approach, deep learning-based software error prediction, deep learning-based GCN, and deep learning in software development.

A. COGNITIVE COMPLEXITY MEASUREMENT APPROACH

Various cognitive techniques were advanced to degree the complexity of conventional object-orientated software

program layout structures. Some of them are Cognitive Function Size (CFS), Cognitive Information Complexity Measurement (CICM), Modified Cognitive Complexity Measurement (MCCM), Cognitive Program Complexity Measurement (CPCM), Adamo Measurement, Shehab Measurement, Weighted Class Complexity (WCC), and Cognitive Code Complexity (CCC) [13], [20]. Adamo supplied a layout of experiments and equipment to discover the cognitive weight of BCS for a selected programming language. The test become attended via way of means of 14 skilled college students with enjoy in Java programming [5]. Some code snippets are furnished to participants, and the time it takes every player to recognize the snippet is recorded for each accurate and wrong answers. Each size of cognitive complexity has its very own blessings and limitations, and locating the maximum suitable size for all varieties of software program structures is not a clean task. All of the above complexity measurements are from Wang et al. We used the cognitive weights (W_c) of the Basic Control Structure (BCS) proposed via way of means of [12]. This is a continuous process that can accurately capture all aspects of the software system. Pantsar et al. [21] presented human cognitive function using heuristic and didactic tools in the cognitive process. They accurately portray human cognitive tasks involved in solving math problems, including aspects related to the algorithm level.

B. DEEP LEARNING BASED SOFTWARE DEFECT PREDICTION

Software defect prediction technology is wide utilized in software system quality assurance and has the potential to considerably cut back software development costs. Produce a predictor mistreatment the recent defect knowledge and use the established model to predict whether or not or not the new code contains a defect. Ancient software defect predictions may be imprecise today. It consists of 2 steps. The primary step is to extract options. This will increase the potency of defect illustration by manually planning some features or combining existing features. The second technique is classification using machine learning. It specifically employs learning algorithms to construct accurate models and make higher predictions.

For defect prediction, several machine learning algorithms are used, together with SVM [7], Bayesian Belief Network [22], NB [8], DT [5,] Neural Network [23], and Ensemble Learning [24]. For example, Kumar and Singh [26] evaluated the capabilities of SVMs in predicting defective code modules employing a combination of various feature choice and extraction techniques and tested them on 5 National Aeronautics and Space Administration (NASA) datasets. The author of [22] used the Thomas Bayesian belief network to predict software quality. Arar and Ayan [25] planned and tested a Feature-Dependent Naive Bayes (FDNB) classification for predicting software failures at the PROMISE repository. He et al [1] investigated the performance of tree-based machine learning algorithms in terms of metric simplification in error prediction. Li et al. [24] planned a brand-new Two Step Ensemble Learning (TSEL) technique for predicting failure in heterogeneous data. They tested the proposed TSEL approach on thirty public comes and located it to be superior to several competitive methods. Mansi et al. [27] demonstrate the importance of hyper-parameter standardization within the development of effective deep neural network models for predicting code module error condition and examination the results to alternative machine learning algorithms. Within the majority of cases, their proposed model outperforms other algorithms.

Furthermore, several studies have projected the Cross-Project Defect Prediction (CPDP) model to beat the insufficiency of training data. Turhan et al. [28] proposed mistreatment the closest neighbor filter for the target project to pick out the training data to enhance the performance of CPDP. TCA+ was proposed by Nam et al. [29], that employs a contemporary technique referred to as Transfer Component Analysis (TCA) and an optimized standardized procedure. They tested TCA+ on eight ASCII text file comes and discovered that it considerably improved CPDP. Nam et al. [30] conjointly conferred incapacity prediction ways cherish different measures in numerous projects so as to address the matter of heterogeneous knowledge in CPDP. Software defect prediction using a hybrid model (CBIL), as proposed by Ahmed et al. [57], was tested using a sample of seven open-source Java projects from the PROMISE dataset. Applying the following evaluation metrics, CBIL is assessed: Area under the curve and F-measure (AUC). Their results shows that the CBIL model outperforms CNN by 25% in terms of average F-measure. In terms of average AUC, the CBIL model outperforms the Recurrent Neural Network (RNN), which achieves the best performance among the baseline models chosen for their experiments. With the help of recent research on deep learning fault prediction algorithms, Safa et al [58] were able to bridge the gap between program's semantics and fault prediction features and produce accurate predictions.

C. DEEP LEARNING BASED GRAPH CONVOLUTIONAL NETWORK

Thomas et al. [31] presents an evolutionary approach for semi-supervised learning of graph-structured data based on the efficient transformation of convolutional neural networks that act directly on the graph, and convolutions of spectral data by local first-order approximation and use the architecture of complex graph. The model scales in proportion to the number of edges in the graph and learns a hidden class representation that encodes both the local graph structure and the node properties. Anhetal. [32] Demonstrated in their work that they automatically learn error properties using accurate graphs that represent program execution flows and deep neural networks. In their article, they first create a control flow graph from assembler instructions obtained by compiling the source code, and then apply a multi-layer

multi-view convolutional neural network (DGCNN) to learn semantic features. They tested on four real-world datasets in a way that transcends baseline, including several other deep learning approaches.

Meilong et al. [33] presented a representational learning lever for generating semantic and structural features. It extracts the symbol vector from the AST-based code file and feeds the symbol vector to the built-in neural network to automatically learn the semantic features. It also builds complex network models based on the dependencies between code files, the software network (SN). They applied the network integration method to learn the structural features. Finally, they built a new model for predicting software errors based on the learned semantic and structural features (SDP-S2S). They evaluated the method for six projects collected in the PROMISE public repository. Their results show that the contribution of structural features extracted from the software network is important, and the results appear to be better when combined with semantic features. Comparing their contribution to traditional manual functionality, the SDP-S2S F readings have increased overall, with a maximum growth rate of 99.5%. It also examines the sensitivity of parameters in learning semantic and structural features and provides guidance on predictive optimization. Sharma et al. [34] used the PHEME Rumored Tweet dataset, which includes five incidents: Charlie Hebdo, German Wing Drop, Ottawa Shooting, the Siege of Sydney, and rumored and non-rumored tweets about Ferguson. They converted the rumored tweet dataset to a suspicious user dataset before using a graph neural network (GNN) -based approach to identify suspicious users. They examine GCN, a type of GNN, to identify suspicious users, and use GCN results and compared with basic approaches such as SVM, RF, and deep-based long short-term memory (LSTM). Their experiments were performed on real-world datasets and achieved values of up to 0.864 for F1 scores and 0.720 for Area Under-Curve (AUC) Receiver Operating Characteristics (ROC) for GNN-based effectiveness to identify a user.

Banerjee et al. [35] examined knowledge-based entity-relationship graphs and language-related dependency graphs to calculate richer representations of words and entities. They tested with four datasets: a modified DSTC2 dataset, a mixed code version of the recently released four-language DSTC2 dataset, a Wizard-style Oz CAM676 dataset, and a Wizard-of-Oz-style multi-WOZ. For all four datasets, their methods are in many ways superior to existing methods. In order to process projects of various sizes with the same level of detail, Lucija et al [56] created an end-to-end model for defect prediction based on a convolutional graph neural network (GCNN) whose architecture can be customised to the analysed software. Based on the processing of the nodes and edges from the abstract syntax tree (AST) of the source code of a software module, their model determines whether the module is defective or not. Their proposed model outperforms conventional defect prediction models in terms of AUC and F-score, according to experiments on open-source Java projects. Their model has demonstrated comparable predictive abilities for the studied projects when compared to existing state-of-the-art models based on their F-scores.

D. DEEP LEARNING IN SOFTWARE DEVELOPMENT

In addition to software defect prediction, deep learning models are used in areas such as software maintenance [36], code clone detection [37], and error detection [38]. Guo et al. [36] used a recurrent neural network (RNN) model to link software maintenance requirements, designs, source code, test cases, and other deliverables. Li et al. [37] proposed a clone detection method based on deep learning. In their treatise, AST tokens were used to represent method-level code clones, and non-clones were used to train classifiers that could recognize code clones. Their method produced similar results in a shorter amount of time. Nguyen et al. [38] found an error in defect prediction using a deep neural network. The goal of their model was to solve the lexical mismatch problem, and they found that the terms used in bug reports differed from the terms and code tokens used in the source files. Their model achieved 70% accuracy with 5 recommended files.

Deep learning has also influenced source code organization [39], runtime behavior analysis [40], feature placement [41], vulnerability analysis [42], and coder identification [43].

III. THEORIES AND APPROACHES

A. SOFTWARE DEFECT PREDICTION

Figure 1 represents the propose architecture for program level defect prediction process based on deep learning concepts. The first four steps (*step-1, step-2, step-3 and step-4*) of the procedure is to gather programs and labelled them as clean (no-defect) and buggy (defect), extract useful features (in our study we extract seven features which will discuss next in Section *B*) from the program and get the values of the feature through cognitive complexity measures, built CGF and assigned features value to the individual node in a graph and same will be generated based on the node connectivity in a graph through feature relationship and finally we built node vector matrix for the entire graph. In the next three steps (*step-5, step-6 and step-7*), a node vector matrix is passed to GCN which generates fixed size features representation for an individual node in a graph and finally we applied one-way max polling approach to get fixed size vector of the graph regardless its shape and size. We prepared our data sets repository for simple, medium and complex programs from the output of the GCN (shown in *step-8*). *Step-11* is to form training instances by extracting features from the categorized program modules (shown in *step-9* and *step-10*). *Step 12* is to create a classification model and train using a training instance.

In our case, we chose four machine learning models (RF, DT, NB, and SVM) to compare with the proposed DNN-DP

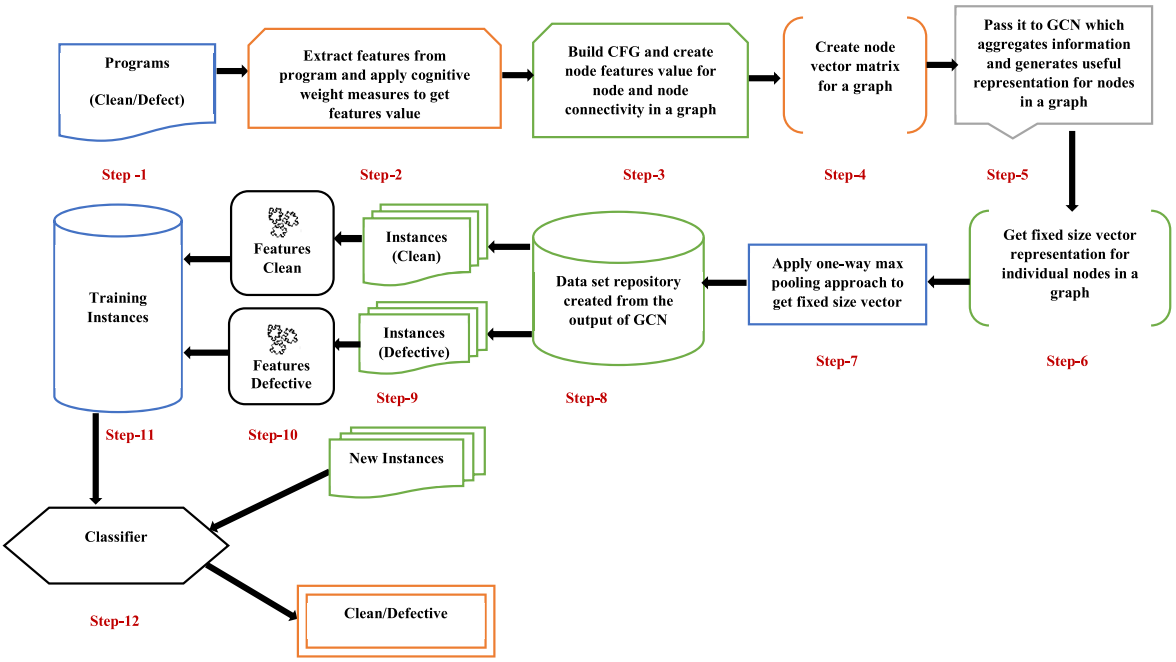

FIGURE 1. Proposed architecture of program level defect prediction process.

and CNN-DP. Finally, new program function instances are sent to a trained classifier to predict whether the source file has flaws.

B. COGNITIVE APPROACH FOR EVALUATING NODE FEATURES

The cognitive approach focuses on both the internal and control structures of the software while calculating complexity. Sequential, branched and iterative control structures are usually identified in the software. Cognitive weights were used by Wang et al. [12] assigned to all of the above Basic Control Structures (BCS) depending on the difficulty or relative amount of time and effort required to understand a given software structure. Cognitive weights (W_c) are assigned by many cognitive psychology experiments in cognitive informatics, and the assigned weights are shown in Table 1.

Our general approach for evaluating the node features through cognitive complexity measures as (a) Extract features from the source code (in our study we use seven node features i.e., IN (input), OUT (output), IF-THEN-ELSE (branch), WHILE (iteration), FOR (iteration), EXP (expression) and FC (function call) and represented as in the given order [IN, OUT, IF-THEN-ELSE, WHILE, FOR, EXP, FC] for a node in CFG. (b) Built CFGs for a program and generates node features value. (c) Finally, a node vector matrix is created for the entire CFG and pass it to the GCN for automatic learn features. The process for evaluation for cognitive complexity measure values as well as node features value are mentioned below:

TABLE 1. Cognitive weights (W_C) of each BCS.

Sl. No	CATEGORY	BCS	Cognitive Weight
1	Sequence	Sequence (SEQ)	1
2	Branch	If-Then-Else (ITE)	2
3	Branch	Case (CASE)	3
4	Iteration	For-do (R_i)	3
5	Iteration	Repeat-Until (R_i)	3
6	Iteration	While-Do (R_o)	3
7	Embedded Component	Function Call (FC)	2
8	Embedded Component	Recursion (REC)	3
9	Concurrency	Parallel (PAR)	4
10	Concurrency	Interrupt (INT)	4

1. *Evaluation of CCMs is composed of following steps: -*

a) Require: Program

b) Find out the BCS, its weights and existence of layers.

c) If there are no nested If-Then-Else conditions, for or while loops, the cognitive weight is calculated by taking the sum of all BCSs within the block.

d) In case of nesting, the cognitive weight is calculated by multiplying the cognitive weight of inner BCS with cognitive weight of outer BCS i.e., for a component with q-linear blocks which consists of m-layers of nesting BCS with each layer having n-linear BCS, the total cognitive weight may

be define in [12] as

$$Wc = \sum_{j=1}^{q} [\prod_{k=1}^{m} \sum_{i=1}^{n} Wc(j, k, i)] \quad (1)$$

e) If q blocks do not contain any embedded BCS then the step (c) formula can be simplified as follows:

$$Wc = \sum_{j=1}^{q} \sum_{k=1}^{m} (j, k) \quad (2)$$

f) Figure 2 shows an example of calculating the cognitive weight (W_c) of a program. In this example, loop iterations are not considered because iterations do not increase the difficulty of the source code and require an understanding of the program. The cognitive weights of BCS were taken from Table 1. The overall cognitive complexity scale of the program is 60.

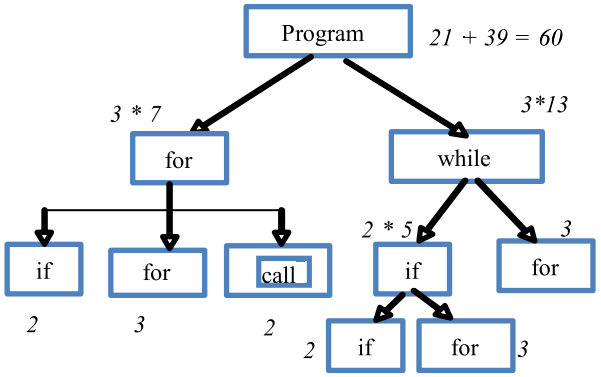

FIGURE 2. Example to evaluate the cognitive weight of a program.

2. *Evaluation of node features value is composed of following steps: -*

a) Require: *CFG.*

b) The example mentioned in the Figure 2, is taken for consideration. Apart from the features mentioned in the example (like *FOR=24, WHILE=39, IF-THEN-ELSE=12, FC=2),* we assume other features value for the program have *i.e., IN=1, OUT=5 and EXP=2.*

c) Fixed sized vector is created for a node in the CFG i.e., [IN, OUT, IF-THEN-ELSE, WHILE, FOR, EXP, FC].

d) For a node connectivity in a graph if there exist a path then we assigned an integer value otherwise we put zero for others.

e) Figure 3 shows how nodes features value is created for a node in a CFG.

C. BUILDING GCN FOR FEATURE REPRESENTATION

Machine learning graphs can be a daunting task due to their very complex yet informative graph structure. This section shows how to perform deep graph training using the Graph Convolutional Network (GCN). GCN is a influential type of neural network designed to manipulate graphs directly to take advantage of structural information. Combining the impressions from [31], it shows how the representation of information moves through the hidden layers of his GCN. We will use coding examples of how GCN previously aggregates

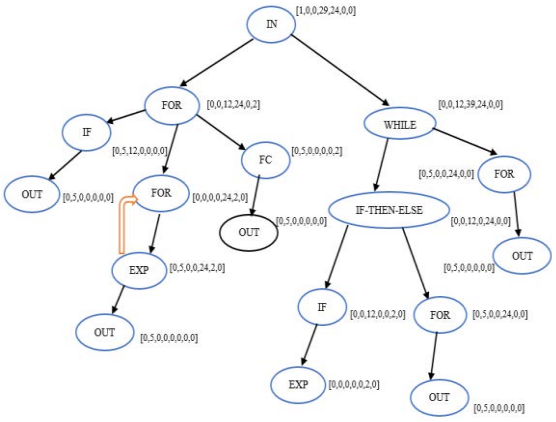

FIGURE 3. Process of evaluating nodes features value for a node in a CFG.

information from layers and how this mechanism creates a useful representation of a node in the generated diagram. Spatial Graph Convolutional Networks and Spectral Graph Convolutional Networks are his two main types of GCN algorithms. The high-speed approximate spectrum-based graph convolutional network of [31] is the focus of our research.

Before elaborating on the calculations performed by GCN, let's briefly explain the concept of forward propagation in neural networks. To transfer the feature representation to the next layer (forward path) of the neural network, use the following equation:

$$H^{[i+1]} = \sigma(W^{[i]}H^{[i]} + b^{[i]}) \quad (3)$$

where *H[i+1]* is feature representation at layer *i+1*, σ is that the activation function, $W[i]H[i]$ is that the weight and representation at layer *i*, $b[i]$ bias at layer *i.* In simple regression, this is often approximately up to $y = mx + b$, where m is the same because the weights. The input features are represented by x. The bias is b. The step forward (3) above from statistical regression is that neural networks apply non-linear activation functions to represent the non-linear features in latent dimension. So (3) for the primary hidden layer i = 0) can be rewritten as follows:

$$H^{[1]} = \sigma(W^{[0]}H^{[0]} + b^{[0]}) \quad (4)$$

where the features represented at layer *0* are essentially the input features (X).

In this method, Adjacency Matrix (A) will include within the forward propagation equation together with the node features (or so-called input features) within the forward propagation equation, 'A' may be a matrix that represents the perimeters or connections between the nodes. By including 'A' within the forward pass equation, the model is in a position to find out feature representations supported node connectivity. The bias b is omitted for clarity's sake. The resulting GCN is considered as a first-order approximation of the spectral graph convolution within the diversity of a message-forwarding network, within which information is propagated along the graphs of neighboring nodes [31]. By including

the adjacency matrix as an additional element, the continuity equation becomes

$$H^{[i+1]} = \sigma(W^{[i]}H^{[i]}A^*) \tag{5}$$

$A*$ signifies the normalized version of A. We can better understand why we need to normalize A and what happens during forward pass in GCNs by building one. The following are the steps for creating GCN for one graph example as mentioned in Figure 2 and Figure 3:

a. Use *NetworkX* to create the graph (G). Node features are assigned to each node in a graph. Define the graph's edges.
b. Insert A into the forward pass equation to get the adjacency matrix (A) and the node feature matrix (X) from graph G. As 'A' is inserted into the forward-pass equation, the model's feature representation becomes richer.
c. Taking the dot product of A and X *(i.e., AX)* that represents the sum of neighbouring node features.
d. In step (c) AX sums up the features of adjacent nodes but does not consider the node's own features. Now, inserting self-loops and normalizing A.
e. In step (d) the elements of AX are not normalized. As with any neural network operation, to prevent numerical instabilities and vanishing/exploding gradients, need to normalize the features in order for the model to converge. For normalizing a data in GCNs by computing the degree matrix (D) and performing the dot product operation of the inverse of D with AX (represented as DAX).
f. As suggested by [31], perform symmetric normalization to obtain more active feature representation. Looking back at (3) $A*$, this is known as the renormalization trick.
g. Create a *2-layer GCN* by incorporating weights and an activation function. As we had chosen *7* features and RELU as the activation function, we placed *32* neurons in the hidden layer and *7* neurons in the output layer for our study.
h. Use one-way Max Pooling approach to generate a fixed-sized vector of *7* features, which serves as the data set for our proposed DNN-DP and CNN-DP models for defect prediction.

Figure 4 depicts the steps for creating a GCN for single graph example, as well as the output for building a GCN for features representation. From Figure 4 following observations have been made which are as follows:

- *Step-a*, provides the number of nodes, the number of edges, and assigns a feature to each individual node in the graph. In the given example, the total number of nodes is *17*, the number of edges is *16*, and the average degree per node is *1.824*.
- *Step-b,* provides A and X from G. In the given example, the shape of A is *(17, 17)* and the shape of X is *(17, 7)*.
- From the result of *Step-c*, we see that the second row of the dot product of A and X is the sum of the characteristics of the nodes connected to node *1*, which are nodes *0, 1, 2, 3, 4, 5* and *6*. This explains how the propagation mechanism works in the GCN and how node connectivity affects the representation of hidden objects seen by the GCN.
- Since *Step-d* does not take into account the characteristics of the node itself, it inserts a self-looping mechanism to connect the node itself. Since each node is connected to itself, all diagonal elements of A are *1*. Therefore, use a self-loop to rename A to *A_Hat* and recalculate AX. This is the product of *A_Hat* and X.
- Because the element of AX is not normalized. So, from *Step-e* the data will be normalized by computing D which is nothing but the number of edges to which a node is connected. When we compare AX with DAX (D^{-1} *dot AX),* the higher the node degree, the lower the weight of the node features in the DAX. In other words, the lower the level of a node, the more likely it is to belong to a particular group.
- In *Step-f*, perform symmetric normalization (using $DADX = D^{-1/2}AD^{-1/2}$) to get a richer/active object representation. The output of *Step-f* shows the representation of the active features.
- From *Step-g*, we find that even without training, the GCN can learn to represent features. The results clearly show two main groups, on the left side with nodes *8, 10, 15, 3, 5* and *11* and six main groups on the right side with nodes *1, 2, 4, 6, 7, 9, 12, 13, 14, 16* and *17*. We can conclude that GCN can learn to represent feature even without training or back propagation.
- The output of *Step-h* represents a fixed-size vector representation when the one-dimensional maximal aggregation approach is applied.

D. CONVOLUTIONAL NEURAL NETWORK

CNN could be a form of neural network that processes data with a lattice-like topology [44], such as 1D statistical data and 2D image data. CNN has been extremely successful in practice with speech recognition [45], image classification [46] and linguistic communication processing [47], [48]. We biased our proposed model for extracting features from the software repository for defect prediction.

A basic CNN architecture [49] is shown in Figure 7. Convolutional layers, pooling layers and a simple fully connected network, also as a dense network, structure the system. The neuron units of each layer are also connected to every other neuron unit. The convolution layer is sometimes placed immediately after the input layer. It is fed from a convolutional layer to a multi-layer feed forward (MLFF) or fully connected network [50]. The convolution layer performs convolution with other operations on the input. This can be thought of as feature extraction, and MLFF is often thought of as a choice block. So, determine the category of the input or predict the value supported by the features extracted from the

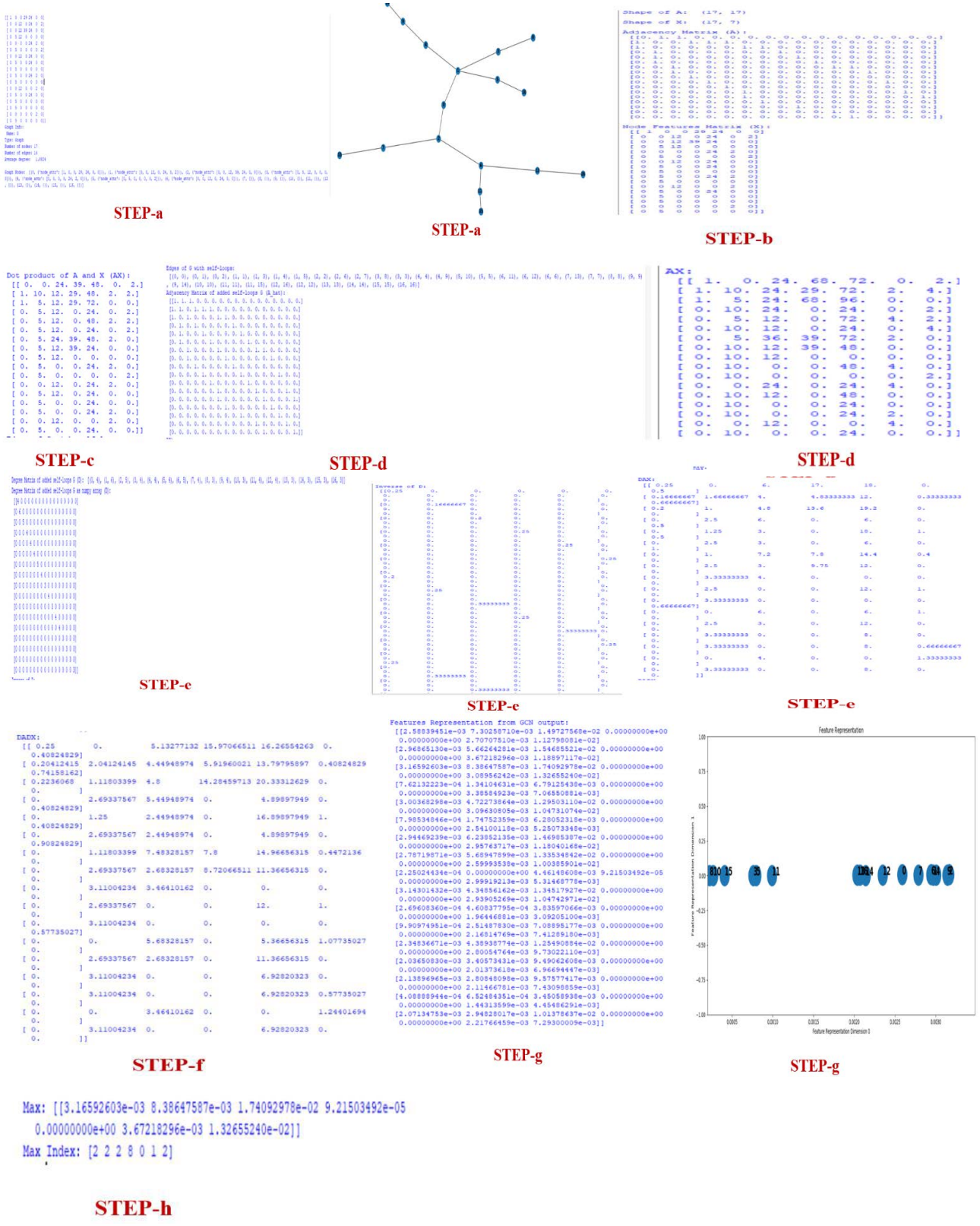

FIGURE 4. Steps (a-h) for creating and building a GCN for feature representation.

previous convolution layer. From Figure 7 it's observed that the i^{th} neuron within the m^{th} layer (convolutional layer), the weights of the i^{th} neuron within the $(m-1)^{th}$ layer as W_i^{m-1}, the bias within the $(m-1)^{th}$ layer as b^{m-1} and in our work,

we use rectified linear units (non-linear activation function) which might be acts as follows:

$$h_i^m = ReLU(W_i^{m-1} * V_i^{m-1} + b^{m-1}) \quad (6)$$

The SoftMax activation function, which can be a variety of sigmoid functions, is used to obtain values from the output layer. SoftMax reduces the output of each neuron to a spread between 1 and 0. It is nonlinear by definition. It is typically used when overcoming an excessive number of classes. The mathematical expression for the SoftMax activation function is:

$$\sigma\,(zi) = e^{zi}/(\sum_{j=1}^{K} e^{zj}) \quad (7)$$

Other operations are as follows:

a) Pooling:

Pooling can be a technique to reduce the dimensionality of a given image while emphasizing the important features. Pooling is typically used after the convolution layer to scale back the dimensions of the convolution output. It also helps reduce over-fitting. Max pooling is the most popular pooling technique. After each pooling operation, the largest value in exactly one window of size f is selected and shifted over the input with a step of length s. In our experiment, we used max-pooling with a pool size of '2' for CNN-DP.

b) Flatten:

The convolution layer output can have a depth greater than one. Flatten concatenates the output of the convolution layers into a flat structure, which is then fed as input to the MLFF network. Figure 5 shows the flatten output.

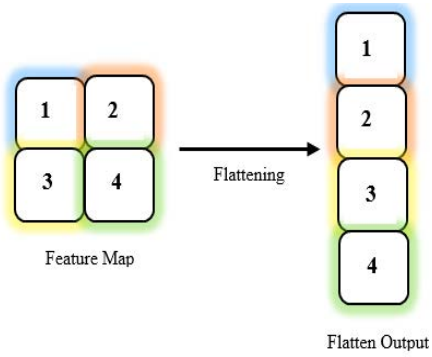

FIGURE 5. Flatten output.

c) Dropouts:

Dropouts are used to preventing over-fitting, which could be loosely defined due to the phenomenon of memorizing inputs rather than learning general properties of inputs. Dropout is the process of removing a neuron's output, resulting in a zero input to the subsequent layer [54]. There are also multiple neurons during a shift and hence the dropout rate P determines whether a neuron drops its output P(drop) or not. If P(drop) = 0.8, each output selects a variance between 0 and 1 in each direction. If the number chosen is less than 0.8, the output is discarded. Figure 6 shows a representation of this idea. For CNN-DP, we used dropouts between the convolution and max pooling layers in our study.

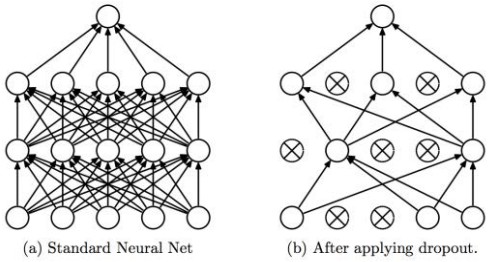

FIGURE 6. Dropout.

d) Multilayer Feed Forward Network:

A fully connected layer, also known as MLFF, is a structure in which neurons of one layer are connected to all neurons of the next layer [50]. MLFF accepts the output of a smoothed convolutional layer.

e) Efficacy:

The effectiveness of CNNs is essentially empirical, with the researcher refining her models that supported the scope and available data. As a result, parameter setting is critical to training a successful CNN. Discuss well how to set these parameters in Section IV of our proposed CNN-DP and DNN-DP models.

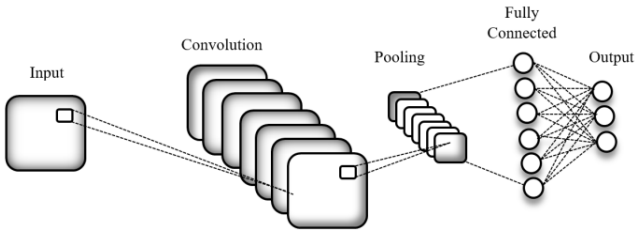

FIGURE 7. Basic Convolutional Neural Network Architecture.

E. DEEP NEURAL NETWORK

The Deep Neural Network is used to identify the most appropriate and acceptable model parameters in order to develop a machine learning model that is structurally effective. As mentioned in [52] and [53], the deep neural network has the power to tell complex relationships. They use the concept of modularity to build a fancy network out of smaller functional units. Some suggestions for combining and modifying these individual entities to model more complex relationships are as follows:

a) Layers:

Neurons in neural networks are usually organized in layers. When linear units with a common set of inputs are grouped together, a dense layer is formed. As shown in Figure 8, The neural network's layers are thought to carry out relatively straightforward transformations, as we might imagine. Neural networks can transform inputs in increasingly complex ways through a deep stack of layers. Each layer of a well-trained

neural network is a transformation that brings us closer to the solution. The DNN-DP study used a high-density 4-layer stack to achieve the optimal solution.

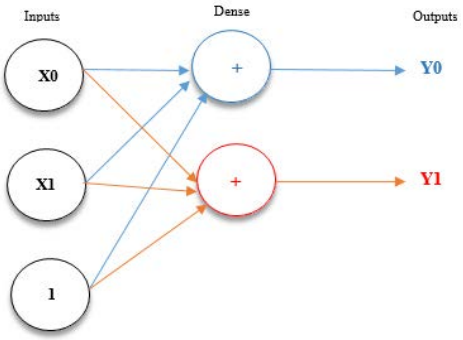

FIGURE 8. A dense layer of two linear units.

b) Activation Functions:

Without the activation function, neural networks can only learn linear relationships. To adjust the curve, we need to use the activation function. Figure 9(a) shows a neural network with a linear relationship. The activation function is a function applied to each output (its activation) of a shift. The most common is the maximum rectification function (0, x). The rectifier function has a graph of a line segment whose negative part is rectified to zero. Applying the function to a neuron's output creates a curve in the data that moves away from a simple line. Figure 9(b) shows the rectification function. If we connect the rectifier to a linear unit, we get a rectified linear unit or RELU. If we apply the RELU trigger to a linear unit, the output peak reaches (0, w $*$ x + b). Figure 9(c) shows a normalized linear unit. The CNN-DP and DNN-DP studies use the RELU activation function.

c) Stacking Dense Layers:

Figure 10 shows a fully connected network with a dense stack of layers. This allows complex data transformations to be performed. It is also known as the hidden layer because the layer's output never appears before the output layer. This study uses binary classification to predict software defects, so we may need an output activation function.

IV. PROPOSED WORK

As covered in Section III (Sub-Title C, Generating and building GCNs for feature Representation), the final fixed-size vector representation when the one-dimensional maximal aggregation approach is applied as shown in Figure 4 (Step-h) for different categories of program, i.e., Simple, Medium and Complex, will become the standard data set for our proposed defect prediction models (DNN-DP and CNN-DP). In the subsections, we first discussed the parameter tuning procedure that helps in building efficient DNN-DP and CNN-DP models because it contains important parameters such as number of hidden layers, number of neurons in each layer and training details such as learning rate, dropout rate, and regularization methods, and then we discuss proposed defect prediction models.

A. PARAMETER TUNING PROCEDURE

We used the theory from previous work for the parameter tuning procedure for DNN-DP and CNN-DP as described in [27]. Machine learning models are primarily empirical, with the researcher customizing their model based on the application domain and available data. However, here we focus on tuning parameters like the number of hidden layers, the number of neurons in each layer, and working on training details like the learning rate and the regularization method.

For each data set, training starts with a small number of hidden layers and a small number of neurons in each layer. If the drawing accuracy is not good, more layers and nodes will be added. The number of epochs will also increase. This larger network strategy and longer training will continue until the data on the train matches well enough, or at least with the accuracy achieved by other classifiers. Then the performance of the validator is tested. If the performance is not good, it is because there is a high variance problem and the network is overloaded with training data but cannot generalize. To remedy this, a process of network regulation is planned.

a) L2 Regularization:

A regularization parameter λ is set which is used as in the Loss (or Cost) function J as follows:

$$J(W, b) = \left[\frac{1}{m}\sum_{n=1}^{m}\left(y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)\right)\right] + \frac{\lambda}{2m}\|W\|_{2^2} \quad (8)$$

Cross-entropy loss function, the first term in (8) right-hand aspect, measures the effectiveness of a class version whose output is a possibility between 0 and 1. in this, y_n denotes the real value, and ŷ_(n) the anticipated value. The L2 Regularization time period, which has the squared norm (additionally referred to as the Frobenius Norm) of the load matrix, is the second time period on the right facet of (8). Here, ''m'' stands for the dataset's sample rely. It is essential to minimize each of the terms on the right-hand side so that you can minimize the loss feature J. The weights are made to end up smaller by using placing the value of to a high price (to minimize J). Smaller weights make a network simpler and prevent it from learning complicated functions. All weights are driven to smaller values by penalizing the squared values of the weights in the cost function since the cost would be high at higher weights. In reality, some neurons either go to sleep or are excluded from the model, simplifying it. Weight decay regularization is another name for L2 regularization. The values in the experiments range from 0.02 to 0.4.

b) Dropout Regularization:

Deep learning-specific regularization methods like dropout are frequently used. In each iteration, some neurons are randomly turned off. It means that randomly chosen neurons are dropped-out at random. Every time we iterate, we actually

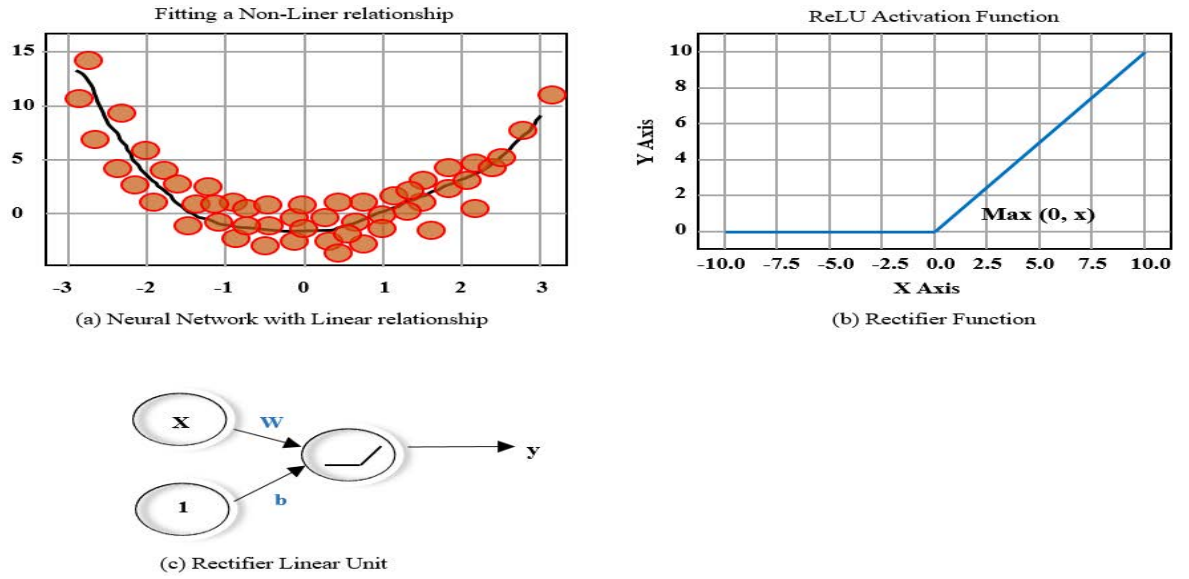

FIGURE 9. Activation Function.

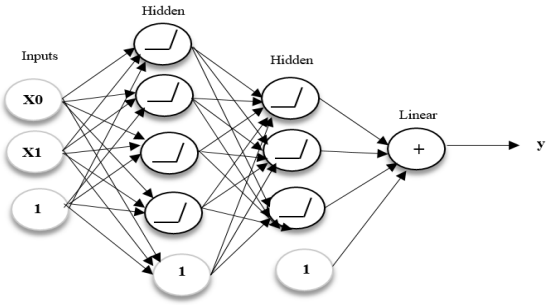

FIGURE 10. A Stack of dense layers makes a fully connected network.

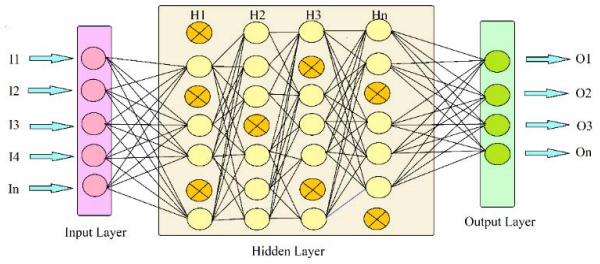

FIGURE 11. Deep Neural Network with some dropout nodes.

train a new model using a different subset of neutrons when some neurons are disabled. As a result, the model's neurons acquire features on their own, without being specifically reliant on another neuron. This means that those neurons that were dropped out were temporarily removed during the forward pass and did not receive any weight updates during the backward pass.

Regularization reduces the network's capacity to overfit the training set, which harms training set performance. The fully connected layers typically experience dropout because they have the most parameters and are therefore more likely to over co-adapt, leading to over-fitting. Figure 11 depicts a DNN with some dropout nodes and Figure 6 depicts a CNN with some dropout nodes.

Consider the layer l node x and the layer l-1 nodes u_1, u_2, u_3, $u_4$that are connected to x. Actually, dropout does nothing more than distribute the weights. It distributes the weight among all the nodes rather than giving any one node more weight. The example that follows will show you how. Let w_{1x}, w_{2x}, w_{3x} and w_{4x} represent the weights of the connections between node x and u_1, u_2, u_3 and u_4.

Squared Norm $||w||^2$ for this layer is

$$||w||^2 = w_{1x}^2 + w_{2x}^2 + w_{3x}^2 + w_{4x}^2 \tag{9}$$

Let the sum of the weights be equal to k, i.e., $\sum w_{ix} = k$

Case 1: If all the weight at is with one connection say $u1$ to x

$$\begin{aligned} w_{1x} &= k, w_{2x} = w_{3x} = w_{4x} = 0 \\ ||w||^2 &= k^2 \end{aligned} \tag{10}$$

Case 2: When the weight is evenly distributed over two links, u_1 to x and u_2 to x.

$$\begin{aligned} w_{1x} &= \frac{k}{2}; w_{2x} = \frac{k}{2}; w_{3x} = 0; w_{4x} = 0 \\ ||w||^2 &= \frac{k^2}{4} + \frac{k^2}{4} = \frac{k^2}{4} \end{aligned} \tag{11}$$

Case 3: When the weight is evenly distributed on all four links.

$$\begin{aligned} w_{1x} &= w_{2x} = w_{3x} = w_{4x} = \frac{k}{4} \\ ||w||^2 &= \frac{k^2}{4} \end{aligned} \tag{12}$$

If the weights are distributed, as in Cases 2 and 3, and not concentrated with a single compound, as in Case 1, the squared norm of the weights decreases in each of the cases. For tiers with many nodes, the probability of failure should be high, and for tiers with few nodes, the probability should be low or even zero. The L2 regularization method reduces over-fitting by changing the cost function, which summarizes the situation. The dropout method, on the other hand, reduces over-fitting by changing the network itself.

B. BUILDING DEEP AND CONVOLUTIONAL NEURAL NETWORKS MODEL FOR DEFECT PREDICTION

The implementation details of the algorithms used and the proposed model is presented in this section. We suggest a deep learning-based architecture for a process that anticipates program-level defects, as discussed in Section-III subtitle A. Using the GCN output to create a data repository for simple, medium, and complex level programs, we then move on to developing our deep and convolutional neural network models for defect prediction.

This section introduces CNN-DP and DNN-DP models. CNN-DP is based on a one-dimensional (1D) CNN model. Figure 12 and Figure 13 show the proposed overall architecture of CNN-DP and DNN-DP for the created dataset repository. We also discuss some of the keywords used in software defect prediction. A test set is a set of examples used to test the learned model, as opposed to a training set, which is a set of examples used to train a model. When using defect prediction, the source file for the training and test datasets is the same. The CNN-DP model has two convolution layers, a max-pooling layer to extract global patterns, a dropout layer to fix overfitting problems, a flattening layer, and a dense layer with L2 regularization to generate deep features and better support simplification, and a convolutional linear sequential model classifier to predict whether a source file was defective or not. The DNN-DP model consists of four dense layers separated by two dropout layers to solve overfitting problems, create deep features, and help simplify. Finally, to determine whether a source file was defective or not, a deep neural linear sequential model classifier was used. For more information on the CNN-DP and DNN-DP architectures, some points are mentioned below:

a) The proposed CNN-DP and DNN-DP models are built using Python 3.5.2. The CNN-DP and DNN-DP related results were generated using Keras-Pre-Processing (version: 1.1.2), a Python-based neural network library that can also be run on TensorFlow (version: 2.3.1). The experiment was performed on a computer equipped with a 64-bit operating system, a x64 processor and 16 GB of RAM.

b) A convolutional linear sequential model classifier is proposed and implemented in Keras to predict software defects. After pre-processing the labeled source files, we split the dataset into training and test sets with a [70:30] split ratio. To get our prediction results, we fed our training data into proposed CNN-DP and DNN-DP models, which have fixed weights and biases, and then we fed each file in the test set into proposed defect prediction model. Based on the result obtained, we predicted whether a source file was defective (buggy) or non-defective (clean). It was considered defective if the result was greater than 0.5, otherwise it was considered non-defective.

c) In CNN-DP, the model is designed with two convolution layers, a max-pooling layer, a dropout layer, a flattening layer, and a dense layer with L2 regularization, and in DNN-DP, the model is designed with four dense layers separated by two dropout layers with L2 regularization.

d) The RELU (Rectified Linear Units) activation function was used in the input and hidden layers of both the CNN-DP and DNN-DP models. The SoftMax function was used in the last layer of CNN-DP for classification and the Sigmoid function was used in the last layer of DNN-DP for classification.

e) We used the Adam optimizer in CNN-DP with a learning rate of 0.001 and DNN-DP with a learning rate of 0.001 as the optimization function to update the network weight after each iteration. We select the learning rate for proposed models following the parameter tuning process discussed in Section V (Results and Analysis). In CNN-DP we used a sparse categorical cross-entropy as the loss function, and in DNN-DP we used a binary cross-entropy.

f) Figure 12 shows that the first two CNN layers should use filter sizes of 512 and 1024 with kernel sizes of 3 and the RELU activation function, while the third CNN layer should use Max-pooling with pool size of 2 and dropouts of 0.3. It employs 200 neurons in the dense layer with RELU activation functions and 0.05 L2 regularization. It flattens the output between convolutional and dense layer.

g) We began with a dense layer of 64 neurons in the input layer, a dropout rate of (0.1), and a RELU activation function from Figure 13. The following three hidden dense layers use 64, 64, and 128 neurons with dropout rates of 0.1 and L2 regularization of 0.03.

V. EXPERIMENTAL SETUP

A. DATA DESCRIPTION

Three hundred and twenty Python programs were written by our talented UG and PG students for defect prediction model analysis, and all experiments were performed during lab classes. Together with three experienced lab programmers, they compiled and ran each individual program and identified buggy/clean programs before dividing them into three different classes, namely simple, medium and complex programs. We used the GCN output to prepare our data sets (simple, medium and complex). Simple level programs have seven attributes or features with 100 instances, medium level programs have seven attributes or features with 100 instances and complex level programs have seven attributes or features

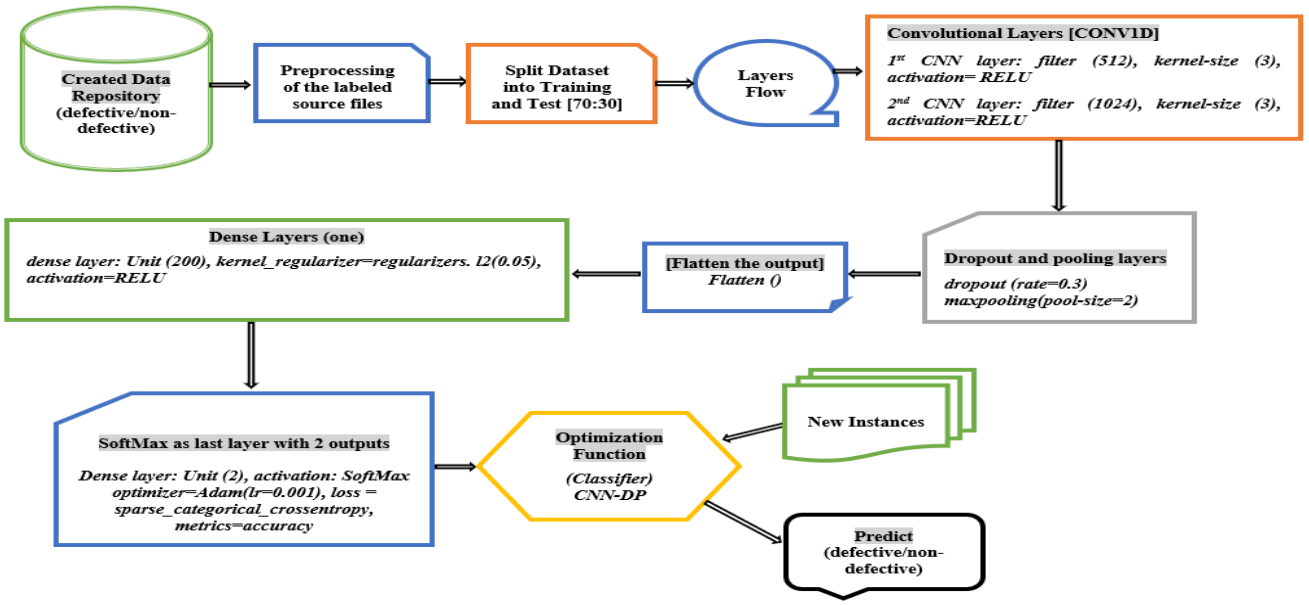

FIGURE 12. Proposed architecture for CNN-DP model for the created data sets repository.

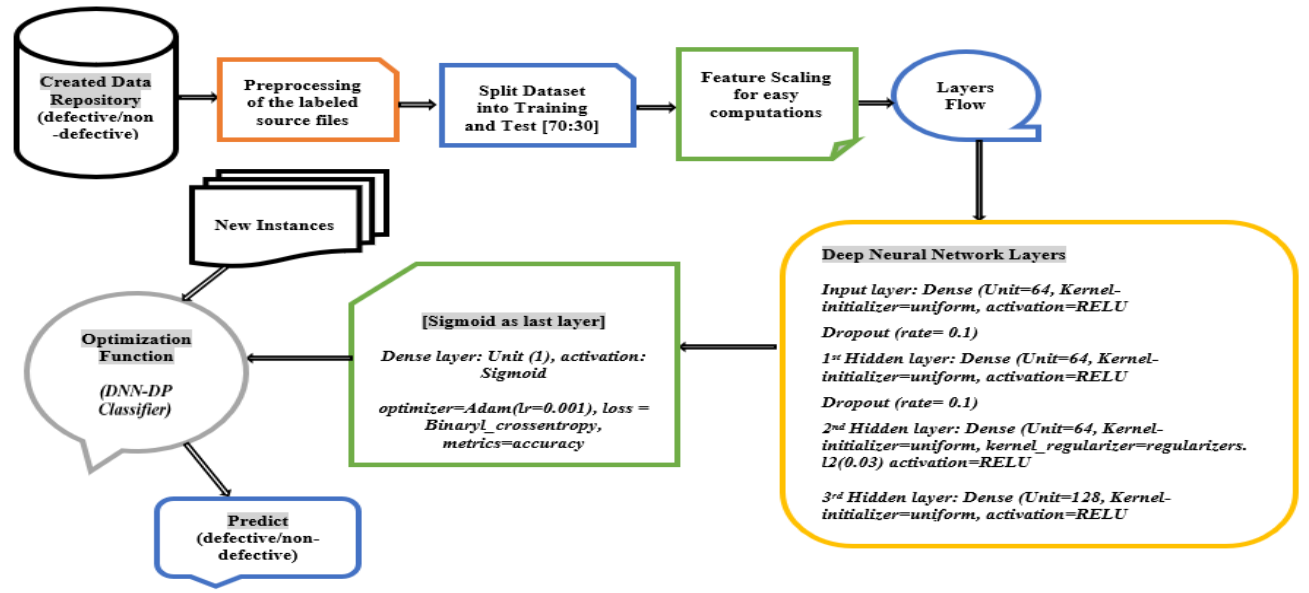

FIGURE 13. Proposed architecture for DNN-DP model for the created data sets repository.

with 120 instances. The dataset properties of programs at the simple, medium, and complex levels are shown in Table 2.

WEKA (Waikato Environment for Knowledge Analysis) was used to process the statistical output of datasets. WEKA is open-source software that allows users to pre-process data, implement well-known machine learning algorithms, and conceptualize their data to develop and apply machine learning techniques to real-world data problems. A variety of classifiers including RF [6], DT [5], NB [8] and SVM [7] have been used to calculate the accuracy of different data sets. The output of these classifiers was compared to the output of the proposed CNN-DP and DNN-DP models.

B. EVALUATION PARAMETERS USED FOR STUDY

The evaluation criteria for the proposed CNN-DP and DNN-DP are briefly summarized in this section. In machine learning, specifically in the statistical classification problem, a confusion matrix, also referred to as an error matrix, is used. A summary of the outcomes of classification problem prediction is a confusion matrix. The number of accurate and

TABLE 2. Characteristics of Data Sets for Simple, Medium and Complex Level programs.

Sl. No	Categories of program	Number of Programs	Defect Ratio	Non-defect ratio	Test Set	Train Set
1	Simple	100	39%	61%	30	70
2	Medium	100	36%	64%	30	70
3	complex	120	40%	60%	36	84

inaccurate predictions is added up and divided by class using count values. A classification model may become confused in a number of different ways when making predictions, as shown by the confusion matrix. It reveals both the quantity and the nature of classification errors. A description of the confusion matrix is shown in Figure 14. Modules that are not defective are labelled ''negative,'' while modules that are defective are labelled ''positive.'' (a) The number of positive cases correctly predicted by the classifier (true positive (TP)), (b) The number of negative cases correctly predicted by the classifier (true negative (TN)), (c) Modules that are not defective but are predicted to be defective are classified as (false positives (FP)) and (d) Modules that are defective but are expected to be defect-free are classified as (false negatives (FN)). This confusion matrix can be used to evaluate performance metrics like accuracy, sensitivity, and specificity. The following paragraphs describe these variables:

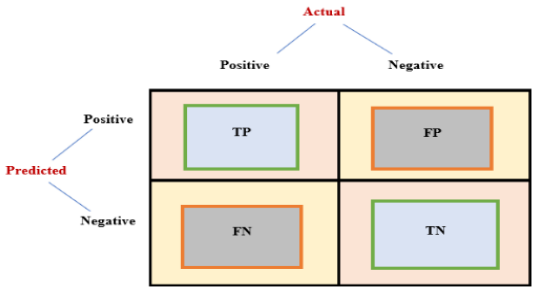

FIGURE 14. Description of confusion matrix.

a) Accuracy (ACC): Modules correctly identified by the classification technique. The correctly predicted confusion matrix values are TP and TN. Accuracy is defined as

$$ACC = (TP + TN)/(TP + TN + FP + FN) \quad (13)$$

b) Sensitivity or Probability of Detection (PD) or True Positive Rate (TPR): It measures how well something can be recognized. This is the percentage of defective modules correctly predicted by the classifier, and it is calculated as

$$Sensitivity = TP/(TP + FN) \quad (14)$$

c) Specificity or True Negative Rate (TNR): It is calculated from the proportion of defect-free modules correctly classified by the classifier

$$Specificity = TN/(TN + FP) \quad (15)$$

d) Probability of false alarm (PF) or False Positive Rate (FPR): The percentage of negative or non-failing cases that are predicted to be positive or defective. This is calculated as

$$FPR = FP/(FP + TN) \quad (16)$$

e) Precision: The percentage of positive or failing cases that are correct is determined as

$$Precision = TP/(TP + FP) \quad (17)$$

f) False Negative Rate (FNR): the percentage of erroneous cases incorrectly classified as non-erroneous or negative, calculated as

$$FNR = FN/(FN + TP) \quad (18)$$

g) F-measure: F-measure is calculated by taking the harmonic mean of the sensitivity and precision.

$$F - Measure = \frac{2 * Precision * Sensitivity}{Precision + Sensitivity} \quad (19)$$

h) Receiver Operating Characteristics (ROC): This is a critical metric for evaluating classifier performance. FPR is plotted on the x-axis while TPR is plotted on the y-axis. The ROC curve accurately reflects the performance of the classifier. Because it is preferable to judge the classifier by value, the Areas under the ROC Curve (AUC) was created. The AUC value indicates how well the classifier outperforms others.

C. RESULTS AND ANALYSIS

This section provides a performance measure for all classification techniques used in the study for the final analysis, namely Precision, Recall, F-measure, Accuracy, ROC, and AUC values. We divided our analysis into four sections: (a) ROC and AUC analysis, (b) error matrix analysis which includes confusion rate analysis and confusion matrix analysis, (c) model performance measures analysis, including precision, recall, F1-measure, and accuracy, (d) Finally, we presented an accuracy comparison of our proposed models for various increasing dropout rates, as well as hyperparameter tuning with various settings to achieve enhanced performance and desired results. Because we used hyperparameter tuning on all data sets, we presented one set of experiments on the complex data set in the paper.

1) ROC AND AUC ANALYSIS

The AUC-ROC curve is a performance measure for classification problems at different thresholds. AUC represents the degree or measure of separability while ROC is a probability curve. It indicates how well the model can distinguish between classes. The larger the AUC, the better the model predicts zero classes as zero and one classes as one. The higher the AUC, the better the model distinguishes between programs with and without the error [55]. The ROC curve is plotted with TPR versus FPR, with TPR on the y-axis and FPR on the x-axis (see Figure 15).

TABLE 3. ROC-AUC Analysis for the created data sets (Simple, Medium and Complex Level).

Algorithms	Simple Level Program Data		Medium Level Program Data		Complex Level Program Data	
	AUC	ROC	AUC	ROC	AUC	ROC
NB [8]	0.67	0.76	0.76	0.75	0.86	0.85
SVM [7]	0.36	0.63	0.67	0.75	0.67	0.80
DT [5]	0.69	0.73	0.83	0.74	0.83	0.73
RF [6]	0.87	0.75	0.84	0.80	0.92	0.86
CNN-DP (with dropout)	0.84	0.73	0.88	0.79	0.85	0.84
CNN-DP (without dropout)	0.86	0.75	0.86	0.81	0.89	0.86
DNN-DP (with dropout)	0.96	0.82	0.86	0.80	0.92	0.85
DNN-DP (without dropout)	0.97	0.85	0.95	0.81	0.91	0.88

An excellent model has an AUC close to one, indicating that it has a high level of separability. A poor model will have an AUC close to zero, indicating it has the worst measure of separability. In fact, it means that the result is reciprocated. It predicts that zeros are ones and ones are zeros. If the AUC is 0.5, the model has no class separation capacity at all. Table 3 shows the ROC-AUC analysis for the datasets constructed. Figure 16, Figure 17, and Figure 18 present an AUC-ROC plot of proposed models and existing classifiers for simple, medium, and complex level datasets.

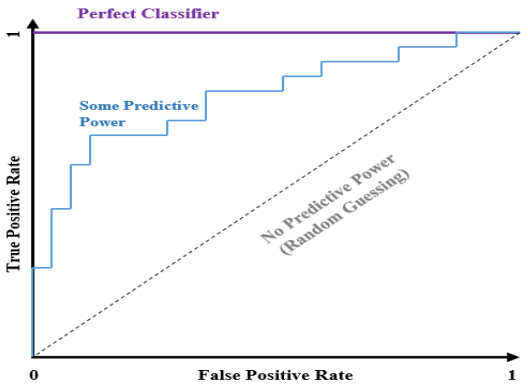

FIGURE 15. ROC-AUC Curve.

The following observations were made for the created data set repositories based on Table 3, Figure 16, Figure 17, and Figure 18:

☐ ROC plots show how different options affect classifier performance. Figures 16, 17, and 18 show how proposed classifiers (DNN-DP (with and without dropouts), CNN-DP (with and without dropouts)), and some existing classifiers (NB. DT, RF, and SVM) plot on a ROC curve. The black dotted line in proposed models and the blue dotted line in existing classifiers represent a classifier that performs no better than random guessing and will plot as a diagonal line. Figure 16, Figure 17, Figure 18, and Table 3 show some observations for proposed and existing classifiers on ROC. Figure 16 and Table 3 show that DNN-DP with dropout (ROC=0.82), without dropout (ROC=0.85) represents a perfect classifier with TPR of 82 percent with dropout and FPR of 18 percent without dropout than CNN-DP with dropout (ROC=0.73), without dropout (ROC=0.75), NB (ROC=0.76), DT (ROC=0.73), RF (ROC=0.75) and SVM (ROC=0.36). In one case, NB outperforms CNN-DP, DT, SVM, and RF by 76 percent TPR and 24 percent FPR. CNN-DP outperforms SVM, RF, and DT for the same task. Figure 17 and Table 3 show that DNN-DP without dropout (ROC=0.81) and CNN-DP without dropout (ROC=0.81) represent a perfect classifier with TPR of 81 percent and FPR of 19 percent in the case of without dropout than CNN-DP with dropout (ROC=0.79), DNN-DP (with dropout (ROC=0.80), NB (ROC=0.75), DT (ROC=0.74), RF (ROC=0.80), and SVM (ROC=0.75). Even in one case, DNN-DP with dropout and RF has the same TPR of 80% and FPR of 20%. It is further observed from Figure 18 and Table 3 for complex level data set, DNN-DP without dropout (ROC=0.88) and CNN-DP without dropout (ROC=0.86) represent perfect classifiers with TPR of 88 percent and FPR of 12 percent for DNN-DP without dropout (ROC=0.88) and TPR of 86 percent and FPR of 14 percent for CNN-DP without dropout (ROC=0.86) than DNN-DP (with dropout (ROC=0.85)), CNN-DP with dropout (ROC=0.84), NB (ROC=0.85), DT (ROC=0.73), RF (ROC=0.86) and SVM (ROC=0.80). RF even outperforms DNN-DP with dropout and CNN-DP with dropout in one case. As a result of the proposed model's high true positive rate and low false positive rate, we conclude from the ROC discussions that the proposed models DNN-DP and CNN-DP provide more predictive power than existing classifiers and outperform existing classifiers i.e., NB, SVM, DT and RF.

☐ Since AUC is a measure of a classifier's ability to distinguish between classes, it is used as a summary of the ROC curve. The greater the AUC, the better the model's performance in distinguishing between positive and negative classes. Table 3 shows that AUC of DNN-DP with dropout=0.96 and without dropout=0.97 and CNN-DP with dropout=0.84 and without dropout=0.97 for simple level data set, AUC of DNN-DP with dropout=0.95 and without dropout=0.86 and CNN-DP with dropout=0.88 and without dropout=0.86 for medium level data set, DNN-DP with dropout= 0.92 and without dropout=0.91 and CNN-DP with drop with dropout=0.85 and without dropout=0.89 for complex level data set, which is between $0.5 < AUC < 1$ indicates

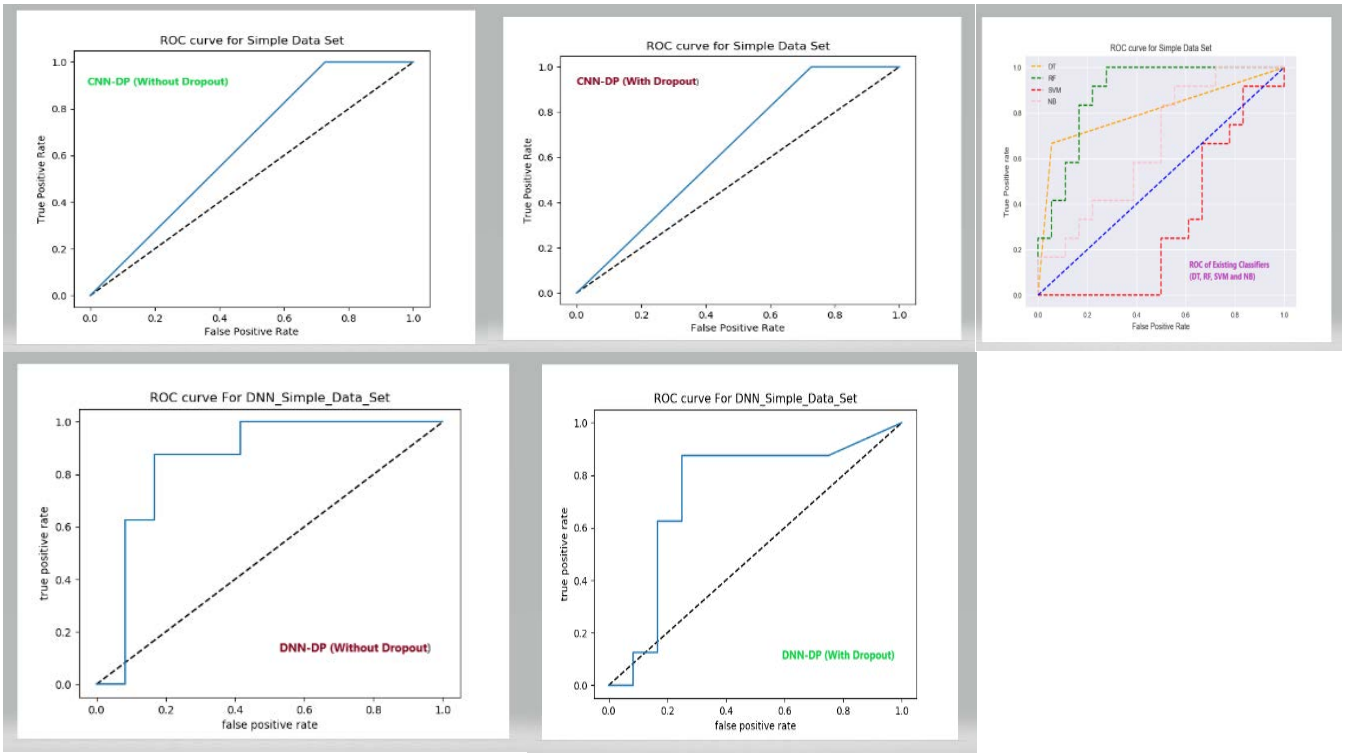

FIGURE 16. AUC-ROC graph of proposed models (CNN-DP, DNN-DP) and existing models (RF, DT, SVM and NB) for Simple Data Set.

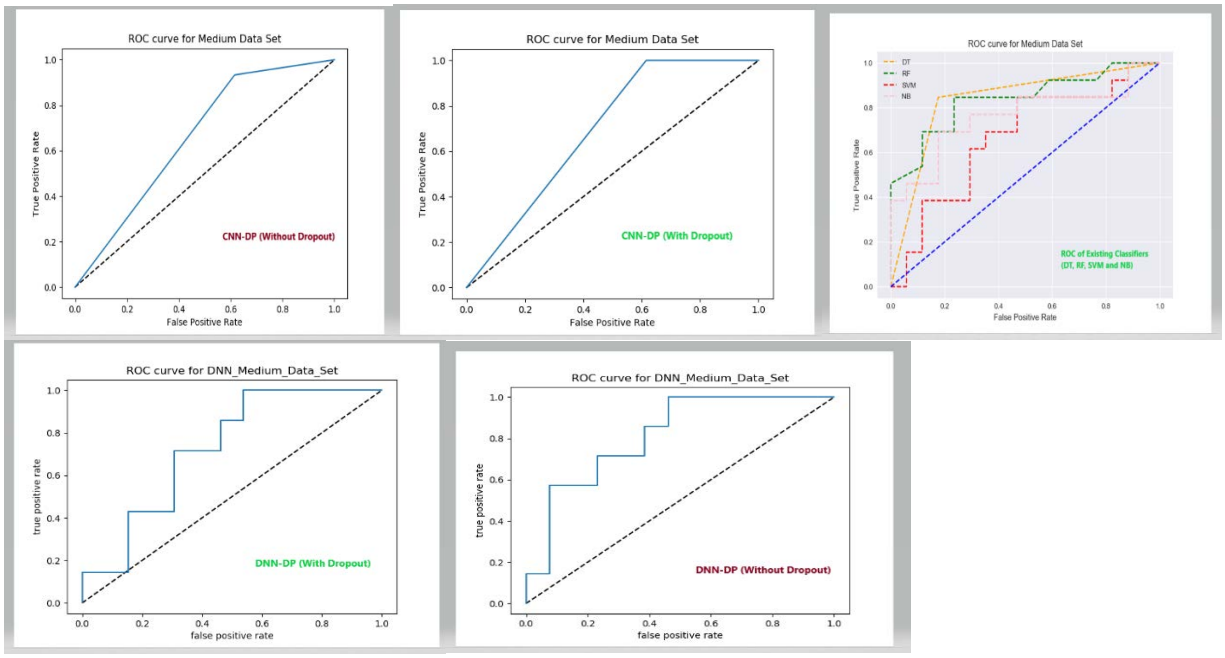

FIGURE 17. AUC-ROC graph of proposed models (CNN-DP, DNN-DP) and existing models (RF, DT, SVM and NB) for Medium Data Set.

that the classifier has a high likelihood of distinguishing between positive and negative class values. This is because the classifier detects more true positives and true negatives than false negatives and false positives than the existing classifiers (NB, SVM, DT, and RF), with the exception of one case where RF (AUC=0.92) distinguishes positive class value from negative class value is the same as DNN-DP classifier with dropout. Since the higher a classifier's AUC value,

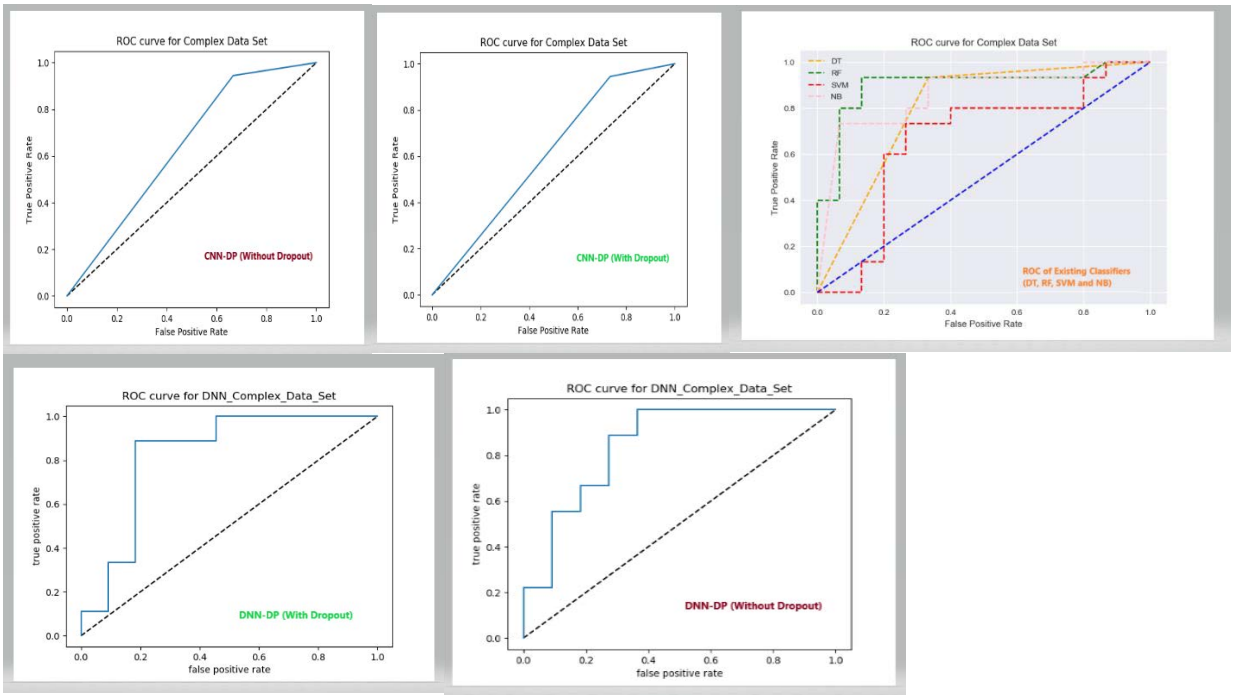

FIGURE 18. AUC-ROC graph of proposed models (CNN-DP, DNN-DP) and existing models (RF, DT, SVM and NB) for Complex Data Set.

the better it is at distinguishing between positive and negative classes. As a result of the discussion, we conclude that the proposed models (DNN-DP and CNN-DP) distinguish more positive classes than negative classes as compared with NB, SVM, RF and DT.

2) ERROR MATRIX ANALYSIS

The confusion matrix or error matrix is an indicator of machine learning classification performance. To improve classifier performance, TPR and TNR should be high and FPR and FNR should be low. Table 4, Table 5, Figure 19 and Figure 20 represents the confusion rate analysis and confusion matrix analysis for the simple, medium and complex level data sets.

Table 4, Table 5, Figure 19 and Figure 20 suggests a few exciting findings for confusion matrix evaluation and confusion rate evaluation. The subsequent observations have been made that is as follows:

☐ Table 4 shows that ({TPR}, {TNR}) of {DNN-DP (with dropout), DNN-DP (without dropout)} and of {CNN-DP (with dropout), CNN-DP (without dropout)} for the set of simple data is ({0.96, 0.98}, {0.97, 0.97}) and ({0.95, 0.95}, {0.54, 0.72} predict a higher frequency true positive and true negative compared to NB, SVM, DT and RF except at a point where SVM is greater than the rate of true negative (0.68) than predicted by CNN-DP (with dropout) is 0.54. Also, for a simple level dataset, Table 4 shows that ({FPR}, {FNR}) from{DNN-DP (with dropout), DNN-DP (without dropout)} and from {CNN-DP (with dropout), CNN-DP (without dropout)} is ({0.03, 0.03}, {0.05, 0.02}) and ({0.46, 0.28}, {0.05, 0.05} predict a lower frequency of false positives and false negatives than NB, SVM, DT and RF except at one point where SVM is less than the false positive rate (0.29) than predicted by CNN-DP (with dropout) is 0.46. For medium level data set, Table 4 shows that that ({TPR}, {TNR}) of {DNN-DP (with dropout), DNN-DP (without dropout)} and of {CNN-DP (with dropout), CNN-DP (without dropout)} is ({0.92, 0.96}, {0.80, 0.95}) and ({0.98, 0.94}, {0.73, 0.84} predict a higher frequency true positive and true negative compared to NB, SVM, DT and RF except at a point where SVM and DNN-DP (with_dropout) predict the same frequency of true positives and the true negative class {0.8, 0.75} of {NB, SVM} than 0.73 of CNN-DP (with dropout). Similarly, for a medium level data set ({FPR}, {FNR}) from{DNN-DP (with dropout), DNN-DP (without dropout)} and from {CNN-DP (with dropout), CNN-DP (without dropout)} is ({0.2, 0.05}, {0.08, 0.04}) and ({0.26, 0.15}, {0.02, 0.06} predict a lower frequency of false positives and false negatives than NB, SVM, DT and RF except at one point where {NB, SVM} predict a lower frequency{0.19, 0.25} of false positives than 0.26 of CNN-DP (with dropout). Table 4 shows that (TPR, TNR) of DNN-DP (with dropout), DNN-DP (without dropout), and CNN-DP (with dropout), CNN-DP (without dropout) are ({0.94, 0.95}, {0.87, 0.86}) and ({0.96, 0.96}, {0.79, 0.81}) predict a higher frequency true positive and true negative compared to NB, SVM, DT, and RF. Furthermore, (FPR, FNR) from DNN-DP (with dropout), DNN-DP (without dropout), and CNN-DP (with dropout), CNN-DP (without dropout) are ({0.12, 0.14}, {0.06, 0.05}) and ({0.19, 0.19},

TABLE 4. Confusion Rate Analysis for the created data sets (Simple, Medium and Complex Level Data Sets).

Algorithms	Simple Level Data Set				Medium Level Data Set				Complex Level Data Set			
	TPR	TNR	FPR	FNR	TPR	TNR	FPR	FNR	TPR	TNR	FPR	FNR
NB [8]	0.59	0.44	0.56	0.41	0.69	0.8	0.19	0.31	0.84	0.24	0.76	0.16
SVM [7]	0.68	0.68	0.29	0.32	0.92	0.75	0.25	0.08	0.87	0.79	0.21	0.13
DT [5]	0.75	0.64	0.36	0.25	0.85	0.67	0.34	0.15	0.93	0.68	0.34	0.09
RF [6]	0.80	0.64	0.36	0.20	1.00	0.71	0.29	0.00	0.93	0.68	0.32	0.07
DNN-DP (with dropout)	0.96	0.97	0.03	0.05	0.92	0.80	0.2	0.08	0.94	0.87	0.12	0.06
DNN-DP (without dropout)	0.98	0.97	0.03	0.02	0.96	0.95	0.05	0.04	0.95	0.86	0.14	0.05
CNN-DP (with dropout)	0.95	0.54	0.46	0.05	0.98	0.73	0.26	0.02	0.96	0.79	0.19	0.04
CNN-DP (without dropout)	0.95	0.72	0.28	0.05	0.94	0.84	0.15	0.06	0.96	0.81	0.19	0.04

TABLE 5. Confusion Matrix Analysis for the created data sets (Simple, Medium and Complex Level Data Sets).

Algorithms	Simple Level Data Set				Medium Level Data Set				Complex Level Data Set			
	TP	FP	FN	TN	TP	FP	FN	TN	TP	FP	FN	TN
NB [8]	22	18	15	14	29	11	13	46	27	13	5	4
SVM [7]	15	25	7	52	21	19	2	57	26	14	4	55
DT [5]	9	31	3	56	11	29	2	57	10	30	1	58
RF [6]	8	32	2	57	15	25	0	59	13	27	1	58
DNN-DP (with dropout)	40	1	2	37	46	6	4	24	30	6	2	42
DNN-DP (without dropout)	44	1	1	34	42	2	2	34	34	6	2	38
CNN-DP (with dropout)	56	18	3	21	54	11	1	30	67	10	3	38
CNN-DP (without dropout)	57	11	3	28	44	8	3	43	68	10	3	42

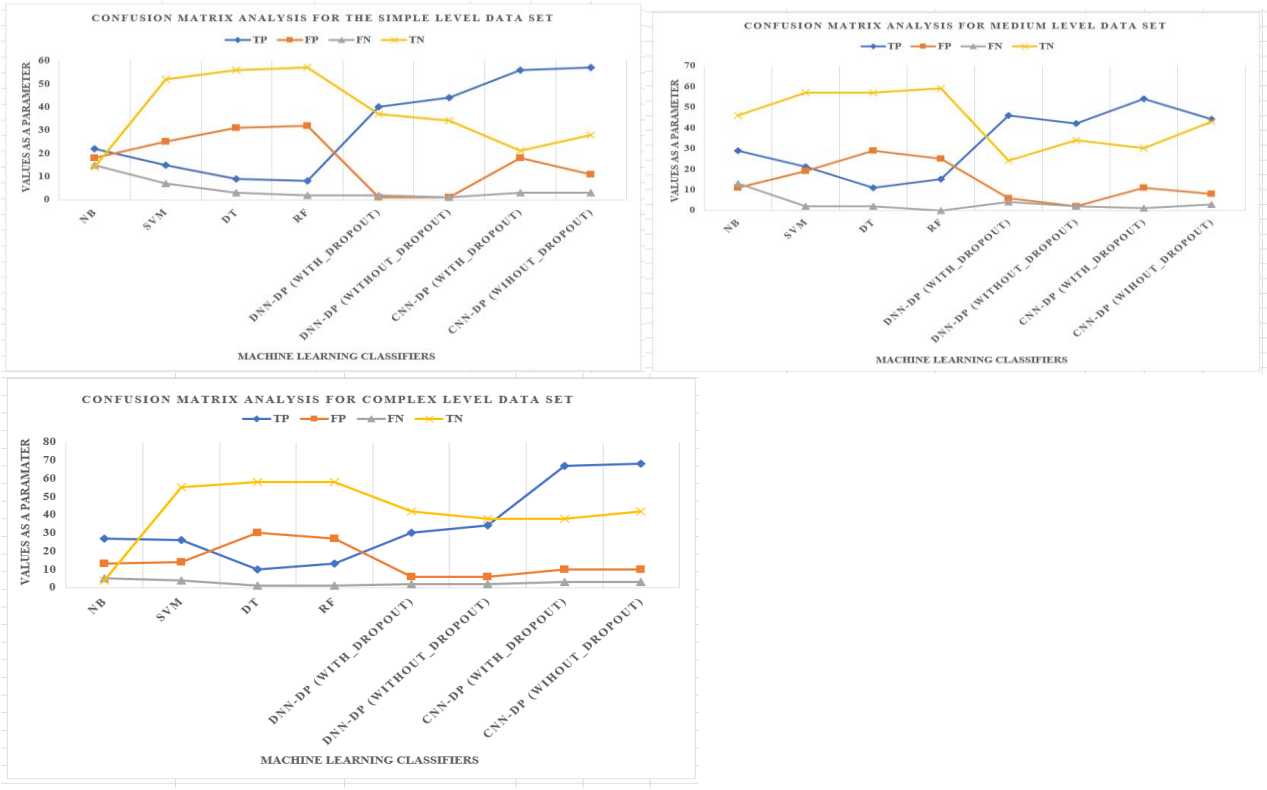

FIGURE 19. Confusion matrix analysis of proposed model with existing classifiers for simple, medium and complex level data sets.

{0.04, 0.04}), respectively, predict a lower frequency of false positives and false negatives than NB, SVM, DT, and RF. Figure 20 depicts the same data for confusion rate analysis, comparing DNN-DP and CNN-DP with NB, SVM, DT, and RF. Based on the discussions, we can conclude that the suggested models for software defect prediction, DNN-DP and

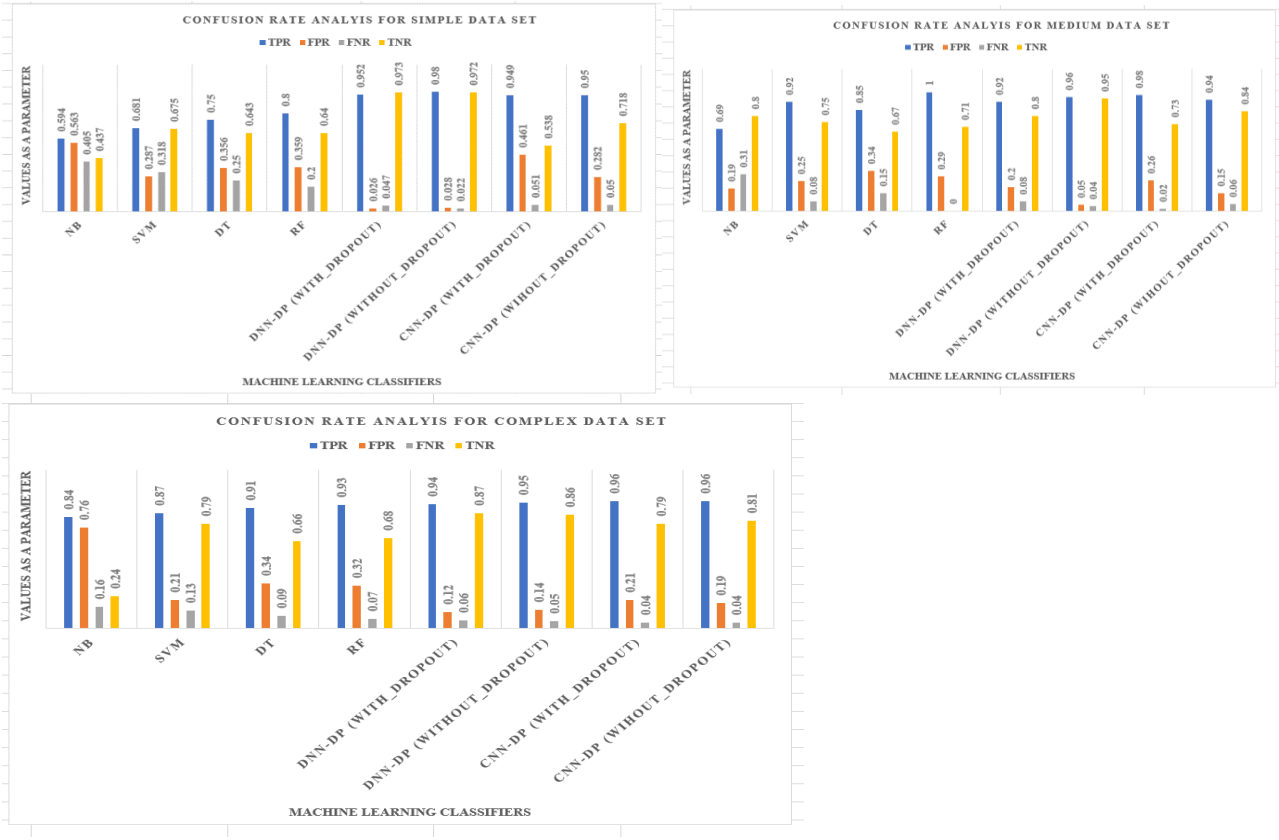

FIGURE 20. Confusion rate analysis of proposed model with existing classifiers for simple, medium and complex level datasets.

CNN-DP, are not underfit or overfit, and that they outperform NB, SVM, DT, and RF in terms of TPR, TNR, FPR, and FNR for the created data set repository.

☐ Table 5 demonstrates that for simple level data set, (DNN-DP with dropout (40), DNN-DP without dropout (44), CNN-DP with dropout (56) and CNN-DP without dropout (57)), (DNN-DP with dropout (46), DNN-DP without dropout (42), CNN-DP with dropout (54) and CNN-DP without dropout (44) for medium level data set), and (DNN-DP with dropout (30), DNN-DP without dropout (34), CNN-DP with dropout (67) and CNN-DP without dropout (68) for complex level data set) predicts more defects than NB, SVM, DT and RF. Table 5 also shows that the proposed models DNN-DP and CNN-DP have good correlations with one another for defect prediction. Figure 19 compares CNN-DP, DNN-DP, NB, SVM, DT, and RF with the same data for confusion matrix or error matrix analysis. Based on the discussion, we draw the conclusion that the proposed models CNN-DP and DNN-DP predict more faults and perform better in predicting software defects than state-of-the-art classifiers (NB, SVM, DT, and RF).

3) MODEL PERFORMANCE MEASURE ANALYSIS

Precision and recall were used to assess the accuracy of the F-measure (or F1 score). In most cases, precision and recall are adjusted. For example, if all test files are predicted to be wrong, this results in a recall value of 1 and a precision of 0. Thus, the best representation of prediction performance is the F-measure, which is a combination of precision and recall. and is between [0,1]. Table 6 contains the information about performance metrics in terms of precision, recall, F1-measure and accuracy for simple, medium and complex level data set. Figure 21 presents the comparison of F1-measure and accuracy of proposed models (DNN-DP, CNN-DP) with existing models (NB, SVM, DT and RF) for the simple, medium and complex level data set.

The following observations were made based on Table 6 and Figure 21:

☐ For simple level data sets, the (precision, recall, F-measure, and accuracy) for DNN-DP (with dropout): (0.98, 0.96, 0.96 0.97), DNN-DP (without dropout): (0.98, 0.98, 0.99, 0.98), CNN-DP (with dropout): (0.90, 0.94, 0.88, 0.87), and CNN-DP (without dropout): (0.90, 0.95, 0.89, 0.89) are higher than NB, SVM, DT, and RF except for one point where RF recall (0.96) measures the same as DNN-DP (with dropout) and slightly higher than CNN-DP (with dropout).

☐ F-measure and accuracy for DNN-DP (with dropout): (0.90, 0.88), DNN-DP (without dropout): (0.96, 0.95), CNN-DP (with dropout): (0.94, 0.88) and CNN-DP (without dropout): (0.88, 0.89) measures higher than existing classifiers NB, SVM, DT, and RF for medium level data set. For the same data set, SVM precision (0.91) outperforms

CNN-DP (with dropout) and CNN-DP (without dropout), as does DT recall (0.97), which outperforms DNN-DP (without dropout) and CNN-DP (without dropout) (without dropout). For the complex level data set, the proposed models DNN-DP (with dropout), DNN-DP (without dropout), CNN-DP (with dropout), and CNN-DP (without dropout) correlate well or outperform the existing classifiers NB, SVM, DT, and RF. It is also observed from complex level data sets that the proposed models, DNN-DP (with and without dropout) and CNN-DP (with and without dropout), outperformed the existing classifiers, NB, SVM, DT, and RF in terms of F-measure and accuracy.

☐ We draw the conclusion that precision quantifies the number of positive class predictions that actually belong to the positive class based on the discussion of the performance measure presented above. Recall measures how many correct class predictions were made using all of the successful examples in the dataset. Precision and recall issues are balanced in a single score by F-Measure, and accuracy offers more true positives and true negatives in prediction.

4) HYPERPARAMETER TUNING PROCESS AND ITS ANALYSIS

Hyperparameter tuning is the process of finding the ideal model construction by adjusting the parameters that define the model building, or hyperparameters. Experiments with multiple parameters tuning process were performed on all three data sets i.e., simple, medium and complex level. Our analysis is broken down into two sections: first, we take dropout into account when evaluating models, and then we present one set of experiments with multiple parameters tuning to suggested models i.e., DNN-DP and CNN-DP on the complex level data set.

☐ Dropout and its evaluation

As is common knowledge, dropout is a technique for preventing model over-fitting. Dropout operates by probabilistically removing, or dropping out, inputs to a layer. These inputs could be input variables in the data sample or activations from a previous layer. Because many different networks with very different network structures are simulated, nodes in the network become generally more resistant to inputs. With dropout, the accuracy will gradually increase while the loss will gradually decrease. After a certain amount of dropout, the model can no longer accurately fit the data It makes sense that a higher dropout rate would lead to an increase in some of the layers' variance, which would make training more difficult. Table 7 compares the proposed models' accuracy with rising dropout rates for the newly created data repository. For our DNN-DP and CNN-DP models, we compare accuracy and loss with rising dropout values in order to obtain the optimal dropout value. To examine the impact of accuracy and loss while implementing dropout regularizations, we start with 0.1 and increase to 0.7. According to Table 7, accuracy increases steadily as dropout rates rise from 0.1 to 0.5 for DNN-DP and CNN-DP, but there is a point for the complex level data set for ACC_CNN-DP where accuracy starts at 0.94 for dropout rates of 0.1 and 0.2 but increases by 1% for dropout rates of 0.3. After certain points, like 0.6, the dropout rate starts to decline.

Based on the results and discussion, we chose a dropout rate of 0.1 for DNN-DP and 0.3 for CNN-DP for our study on DNN-DP and CNN-DP.

☐ *Experiment with several parameter tunings*

By adjusting the parameters, such as learning rate (LR), which is taken as: 0.01, 0.001, 0.0001, and 0.00001, number of epochs (NEP), which is taken as: 100, 200, 300, 400, 500, 1000, along with number of layers and dropout, we performed experiments on DNN-DP and CNN-DP for complex level data set. For DNN-DP, we employ 4 layers with 64, 64, 64, and 128 neurons each and a 0.1 dropout rate between the first and second hidden layers and L2 regularization with 0.03. In the CNN-DP model, we employ two convolutional layers with 512 and 1024 neurons each, one max-pooling layer with a pool size of 2, one dropout layer with a rate of 0.3, one flattening layer, one dense layer with 200 neurons and L2 regularization with 0.05. The main goal of this experiment is to find the optimal learning rate and epoch for our proposed DNN-DP and CNN-DP models for defect prediction. The loss and accuracy of the proposed models (DNN-DP and CNN-DP) with LR = 0.01, 0.001, 0.0001 and 0.00001 and NEP = 100, 200, 300, 400, 500 and 1000 are represented in Table 8, IX, X, and XI on a complex level data set, Figure 22 shows the loss and accuracy analysis of DNN-DP and CNN-DP for LR=0.01, 0.001, 0.0001 and 0.00001. With respect to the learning rates and number of epochs, we reduce the loss function as we iteratively learn the proposed models.

Since we configure LR hyper-parameters into DNN-DP and CNN-DP models to examine the impact on model performance. In order to find the ideal range of learning rates, we tune the LR on DNN-DP and CNN-DP with 0.01 then 0.001 and then go up to 0.00001. Table 8, Table 9, Table 10, and Table 11 show the loss and accuracy for DNN-DP and CNN-DP after each iteration of optimization with respect to learning rate and NEP. we also plot a loss/accuracy graph with respect to various LR and NEP for DNN-DP and CNN-DP, which are depicted in Figure 22. For computing loss or minimizing the value of loss, we use binary cross entropy for DNN-DP and sparse categorical cross entropy for CNN-DP. Only Table 9 shows consistent loss and accuracy behavior for DNN-DP and CNN-DP, according to Figure 22 and Tables 8, 9, 10, and 11. The other tables do not show consistent loss and accuracy behavior, even though the model's prediction can be roughly compared to the actual data, loss values still behave unpredictably after each optimization iteration. Thus, based on Figure 22 and Table 9 for CNN-DP and DNN-DP with various LR=0.01, 0.001, 0.0001 and 0.00001 and NPE=100, 200, 300, 400, 500, and 1000, the following observations were made:

☐ From Table 9 and Figure 22 for DNN-DP for NEP values of 100, 200, 300, 400, 500, and 1000, accuracy is 0.86, 0.86, 0.89, 0.90, 0.92, and 0.97, while loss is 0.380, 0.3387, 0.2561, 0.2345, 0.2292, and 0.1555. The accuracy is typically around 90% (0.90), and the loss is typically around 0.2656. So,

TABLE 6. Performance Comparison in terms of precision, recall, F-measure and accuracy for the created data sets (Simple, Medium and Complex Level Data Sets).

Algorithms	Simple Level Data Set				Medium Level Data Set				Complex Level Data Set			
	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure	Accuracy	Precision	Recall	F-measure	Accuracy
NB [8]	0.71	0.75	0.73	0.53	0.80	0.78	0.80	0.76	0.80	0.96	0.85	0.65
SVM [7]	0.68	0.88	0.76	0.68	0.91	0.96	0.84	0.79	0.86	0.93	0.81	0.82
DT [5]	0.75	0.94	0.77	0.66	0.84	0.97	0.79	0.70	0.90	0.98	0.78	0.70
RF [6]	0.80	0.96	0.77	0.68	0.70	1.00	0.83	0.70	0.92	0.98	0.80	0.72
DNN-DP (with dropout)	0.98	0.96	0.96	0.97	0.88	0.92	0.90	0.88	0.88	0.93	0.90	0.89
DNN-DP (without dropout)	0.98	0.98	0.99	0.98	0.95	0.96	0.96	0.95	0.85	0.95	0.89	0.90
CNN-DP (with dropout)	0.90	0.94	0.88	0.87	0.88	0.98	0.94	0.88	0.88	0.96	0.94	0.93
CNN-DP (without dropout)	0.90	0.95	0.89	0.89	0.89	0.93	0.88	0.89	0.89	0.98	0.94	0.90

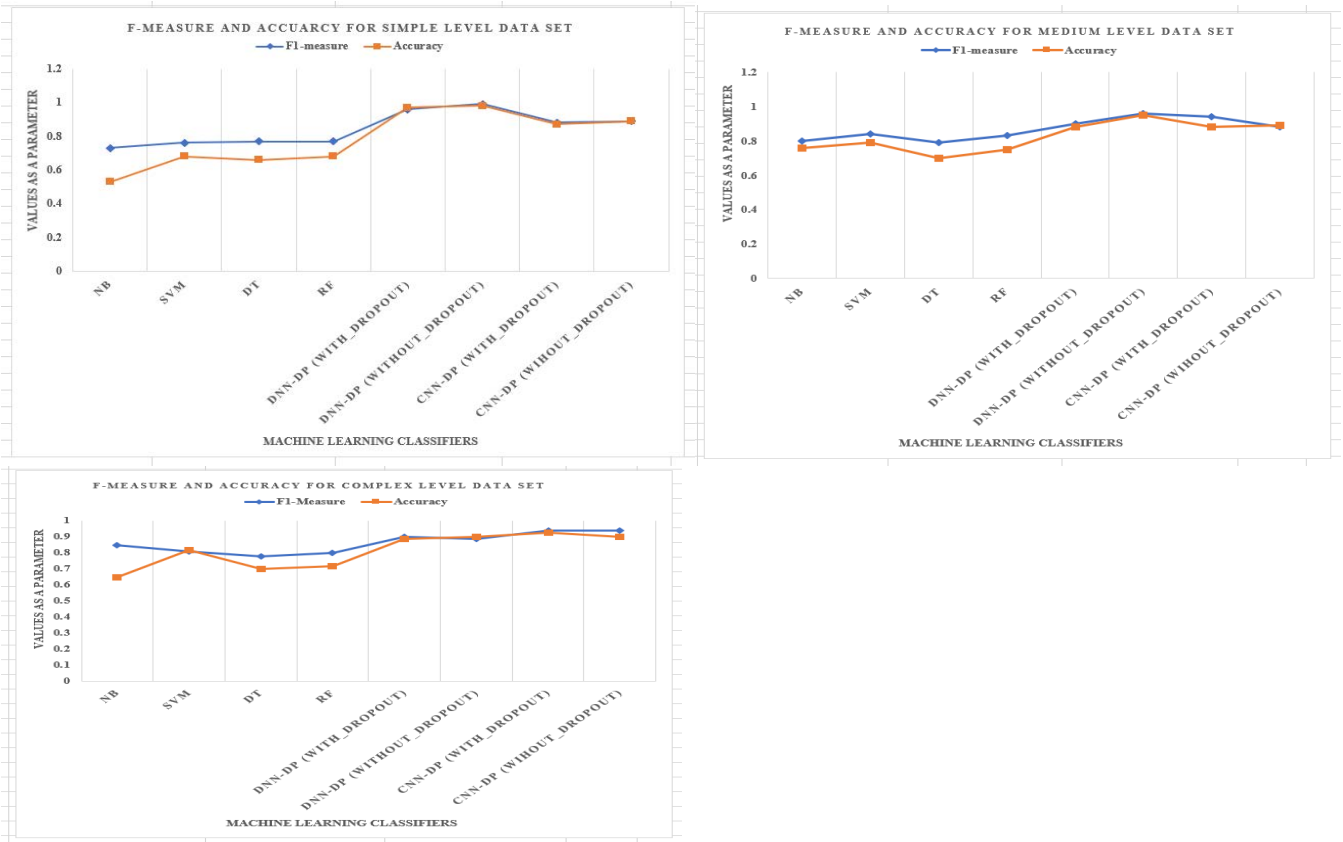

FIGURE 21. Comparison of F-Measure and Accuracy of proposed models with existing classifiers for simple, medium and complex level data sets.

based on the discussion, we determined that LR=0.001 and NEP=1000 were the optimal values for DNN-DP.

□ From Table 9 and Figure 22 for CNN-DP or NEP values of 100, 200, 300, 400, 500, and 1000, accuracy is 0.79, 0.82, 0.89, 0.89, 0.94 and 0.95, while loss is 0.501, 0.412, 0.395, 0.324, 0.307 and 0.2195. The accuracy is typically around 88% (0.88), and the loss is typically around 0.3717. So, based on the discussion, we determined that LR=0.001 and NEP=1000 were the optimal values for CNN-DP.

TABLE 7. Accuracy Comparison of proposed model with increasing dropout rates for Simple, Medium and Complex level data set.

Dropout Rates	Simple Level Program Data		Medium Level Program Data		Complex Level Program Data	
	ACC_DNN-DP	ACC_CNN-DP	ACC_DNN-DP	ACC_CNN-DP	ACC_DNN-DP	ACC_CNN-DP
0.1	0.97	0.87	0.99	0.90	0.96	0.94
0.2	0.97	0.87	0.97	0.90	0.95	0.94
0.3	0.95	0.87	0.97	0.90	0.92	0.95
0.4	0.92	0.82	0.96	0.82	0.91	0.94
0.5	0.90	0.82	0.90	0.77	0.89	0.93
0.6	0.85	0.79	0.90	0.71	0.89	0.89
0.7	0.76	0.71	0.82	0.70	0.84	0.83

TABLE 8. Loss and Accuracy of DNN-DP and CNN-DP for LR=0.01 with NEP= 100, 200, 300, 400, 500 and 1000 on complex level data set.

LR	0.01		0.01		0.01		0.01		0.01		0.01	
NEP	100		200		300		400		500		1000	
Accuracy	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.900	0.81	0.8900	0.79	0.8860	0.86	0.886	0.89	0.92	0.89	0.92	0.95
Loss	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.2987	0.5803	0.3036	0.527	0.3619	0.507	0.2852	0.449	0.1910	0.401	0.1917	0.313

TABLE 9. Loss and Accuracy of DNN-DP and CNN-DP for LR=0.001 with NEP= 100, 200, 300, 400, 500 and 1000 on complex level data set.

LR	0.001		0.001		0.001		0.001		0.001		0.001	
NEP	100		200		300		400		500		1000	
Accuracy	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.86	0.79	0.86	0.82	0.89	0.89	0.90	0.89	0.92	0.94	0.97	0.95
Loss	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.3806	0.501	0.3387	0.412	0.2561	0.395	0.2345	0.324	0.2292	0.307	0.1555	0.2915

TABLE 10. Loss and Accuracy of DNN-DP and CNN-DP for LR=0.0001 with NEP= 100, 200, 300, 400, 500 and 1000 on complex level data set.

LR	0.0001		0.0001		0.0001		0.0001		0.0001		0.0001	
NEP	100		200		300		400		500		1000	
Accuracy	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.71	0.67	0.81	0.69	0.80	0.71	0.81	0.75	0.81	0.80	0.87	0.92
Loss	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.6913	17.49	0.5030	3.961	0.4710	1.07	0.4538	0.593	0.4470	0.522	0.3706	0.3174

VI. THREATS TO VALIDITY

As we will see later, the research described in this article has flaws that could affect the validity of our conclusions. We begin by describing the external risk to potency. Insider threats are then reviewed. Finally, we examine the validity of the conclusion and threats to structural validity.

A. EXTERNAL VALIDITY

Threats to external validity are conditions that limit our ability to generalize the results of our work [51]. In our study, the threat to the external validity of our study was related to the limited number of programs that we analyzed. In addition, all programs are linked to the Python programming language. Therefore, the results may not be generalizable to other platform programs, especially if developed in other programming languages. Furthermore, our results depend on errors in the programming context. Therefore, we cannot draw any conclusions about cross-platform programming errors.

In the next steps of this article, we want to apply the model selection approach to different scenarios. In addition,

TABLE 11. Loss and Accuracy of DNN-DP and CNN-DP for LR=0.00001 with NEP= 100, 200, 300, 400, 500 and 1000 on complex level data set.

LR	0.00001		0.00001		0.00001		0.00001		0.00001		0.00001	
NEP	100		200		300		400		500		1000	
Accuracy	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.64	0.66	0.64	0.66	0.65	0.66	0.65	0.66	0.70	0.66	0.72	0.70
Loss	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP	DNN-DP	CNN-DP
	0.7791	59.68	0.7610	52.03	0.7493	45.65	0.7375	40.05	0.7288	34.99	0.6700	16.75

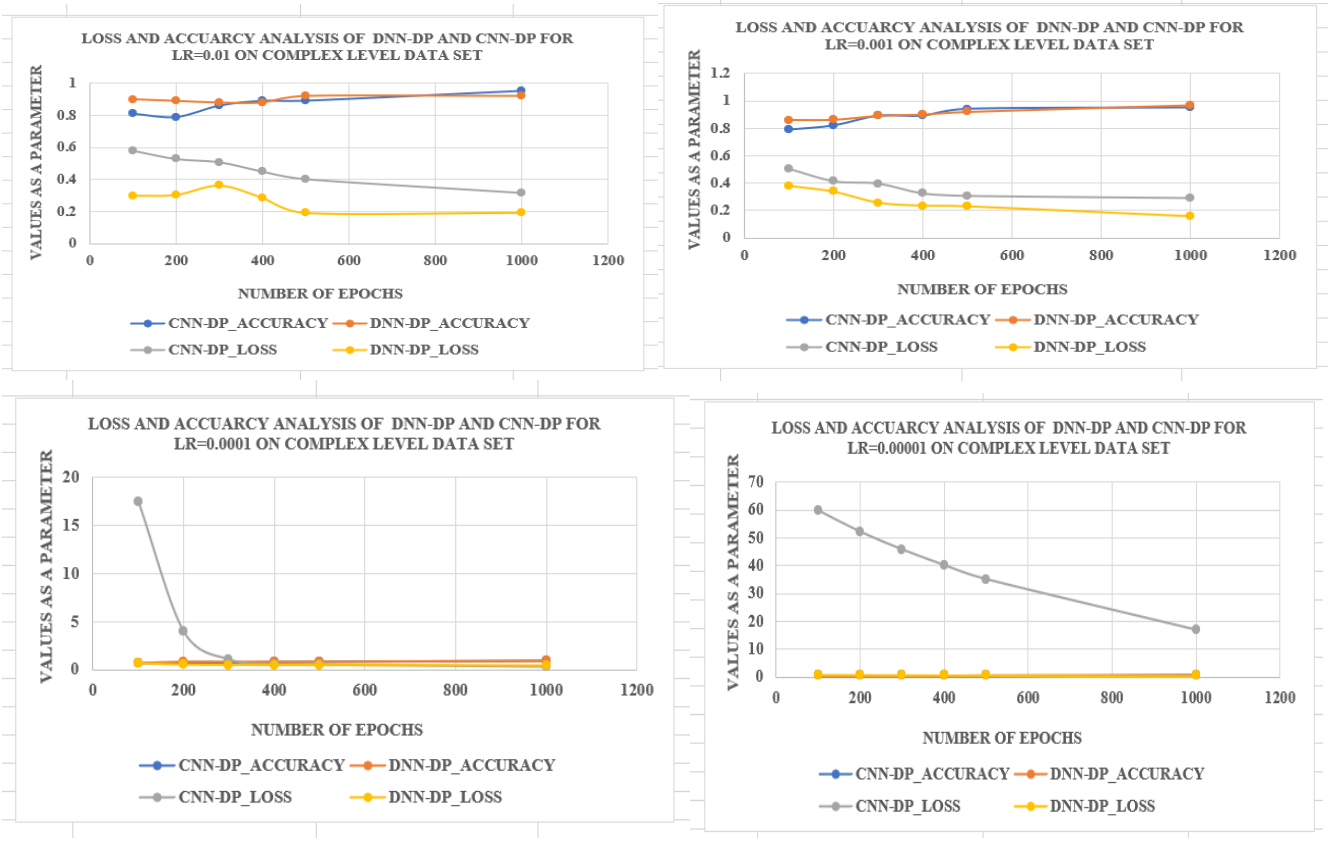

FIGURE 22. Loss and Accuracy analysis of DNN-DP and CNN-DP for LR=0.01, 0.001, 0.0001 and 0.00001 on complex level data set.

we have applied a limited number of core algorithms to classify software modules as bugs. Therefore, we cannot guarantee that the results can be generalized to all currently available classification algorithms. Therefore, we can explore other classification algorithms in later stages of this article.

B. INTERNAL VALIDITY

Since the program was developed by experienced UG / PG students in lab classes, all programs were run and compiled by three experienced lab programmers and divided into three different program categories i.e., simple, medium, complex. We were unable to validate / test the program received from them. Just follow machine learning and graph convolutional network techniques to mitigate the impact of our data. For this reason, future steps in this paper will consider/ apply pre-testing, post-testing, and cross-platform programming approaches.

C. CONSTRUCT VALIDITY

Constructive validity involves inference of empirical results about concepts and theories [51]. This article creates a data warehouse from the output of GCN and proposes two deep learning models CNN-DP and DNN-DP for defect prediction. The results predict that DNN-DP and CNN-DP perform better than modern ML classifiers for defect prediction. However, they are comparable in terms of prediction accuracy and represent variability of defect prediction models. However, we cannot guarantee that the functionality extracted from the

program is sufficient to impact the performance of the defect model. Again, this theory will be used in future work in this paper.

D. CONCLUSION VALIDITY

Threats to outcome problems affect the ability to draw precise associations between course of treatment and outcomes that are related to efficacy [51]. In our study, we revealed that the results and analysis of Section V discussed depended on the defect labels of the generated dataset.

VII. CONCLUSION AND FUTURE SCOPE

In this paper an attempt has been made to extracted seven unique features from the program, and each unique feature was assigned an integer value that we evaluated through Cognitive Complexity Measures (CCMs). We then incorporated CCMs' results as a node feature value in CFGs and generated the same based on the node connectivity for a graph. In order to obtain the feature representation of the graph, a node vector matrix is then created for the graph and passed to the Graph Convolutional Network (GCN). We prepared our data sets using GCN output and then built Deep Neural Network Defect Prediction (DNN-DP) and Convolutional Neural Network Defect Prediction (CNN-DP) models to predict software defects. The proposed models DNN-DP and CNN-DP outperformed state-of-the-art techniques such as NB, SVM, DT, and RF in terms of ROC, AUC, Accuracy, F-measure, Precision, Recall, and experiments with various parameters, according to Section V (Experimental Setup), Subsection-C (Result and Analysis).

We concentrate on a few fundamental issues, which are as follows, with regard to future scope:

✓ Due to the fact that our programs are connected to the Python programming language, and because the outcomes might not apply to other flat-form programs. In order to categorize software modules and programs as bugs or defects, we want to apply the model selection approach to various scenarios and also explore other classification algorithms.

✓ Since the programs were created by skilled UG/PG students during laboratory classes, and those classes did not validate or test the programs we received. So will also take future scope into account for pre-testing, post-testing, and cross flat-form programming approaches.

✓ The program's functionality can be used to determine whether or not the defect model performs well. This theory is also considered for a later time frame.

ACKNOWLEDGMENT

The authors would like to thank the valuable suggestions provided by the anonymous reviewers which greatly helped in preparing the paper in its present form.

REFERENCES

[1] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Inf. Softw. Technol.*, vol. 59, pp. 170–190, Mar. 2015.

[2] M. H. Halstead, *Elements of Software Science*, vol. 7. New York, NY, USA: Elsevier, 1977.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[4] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-4, pp. 308–320, 1976.

[5] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.

[6] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 771–789, Oct. 2006.

[7] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 414–423.

[8] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[9] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. Int. Conf. Softw. Eng.*, Austin, TX, USA, May 2016, pp. 297–308.

[10] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur. (QRS)*, Prague, Czech Republic, Jul. 2017, pp. 318–328.

[11] D. Zhang, J. Yin, X. Zhu, and C. Zhang, "Network representation learning: A survey," *IEEE Trans. Big Data*, vol. 6, no. 1, pp. 3–28, Mar. 2020.

[12] Y. Wang, "On cognitive informatics," in *Proc. 1st IEEE Int. Conf.*, Sep. 2002, pp. 34–42.

[13] Y. Wang and J. Shao, "Measurement of the cognitive functional complexity of software," in *Proc. 2nd IEEE Int. Conf. Cognit. Informat.*, Dec. 2003, pp. 67–74.

[14] D. S. Kushwaha and A. K. Misra, "Robustness analysis of cognitive information complexity measure using Weyuker properties," *ACM SIG Soft. Eng. Notes*, vol. 31, no. 1, pp. 1–6, 2006.

[15] S. Misra, "Modified cognitive complexity measure," in *Proc. Comput. Inf. Sci.*, 2006, pp. 1050–1059.

[16] S. Misra, "Cognitive program complexity measure," in *Proc. 6th IEEE Int. Conf. Cognit. Informat.*, Aug. 2007, pp. 120–125.

[17] D. Adamo, Jr., "An experiment to measure the cognitive weights of code control structures," Tech. Rep., 2014. [Online]. Available: http://www.research.net/publication/284186632, doi: 10.13140/RG.2.1.2877.8960.

[18] A. M. M. Y. W. A. Shehab Tashtoush, M. N. Alandoli, and Y. Jararweh, "An accumulated cognitive approach to measure software complexity," *J. Adv. Inf. Technol.*, vol. 6, no. 1, pp. 145–161, 2015.

[19] S. Misra and I. Akman, "Weighted class complexity: A measure of complexity for object-oriented system," *J. Inf. Sci. Eng.*, vol. 24, pp. 1689–1708, Sep. 2008.

[20] S. Misra, I. Akman, and M. Koyuncu, "An inheritance complexity metric for object-oriented code: A cognitive approach," *Sadhana*, vol. 36, no. 3, pp. 317–337, Jun. 2011.

[21] M. Pansar, "Cognitive and computational complexity: Considerations from mathematical problem solving," *Erkenntnis*, vol. 86, pp. 961–997, Jun. 2019.

[22] N. Fenton, M. Neil, W. Marsh, P. Hearty, Ł. Radliński, and P. Krause, "On the effectiveness of early life cycle defect prediction with Bayesian nets," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 499–537, Oct. 2008.

[23] Q. Cao, Q. Sun, Q. Cao, and H. Tan, "Software defect prediction via transfer learning based neural network," in *Proc. 1st Int. Conf. Rel. Syst. Eng. (ICRSE)*, Oct. 2015, pp. 1–10.

[24] Z. Q. Li, X. Y. Jing, X. K. Zhu, H. Y. Zhang, B. W. Xu, and S. Ying, "Heterogeneous defect prediction with two-stage ensemble learning," *Automated Softw. Eng.*, vol. 26, no. 3, pp. 599–651, Jun. 2019.

[25] Ö. F. Arar and K. Ayan, "A feature dependent naive Bayes approach and its application to the software defect prediction problem," *Appl. Soft Comput.*, vol. 59, pp. 197–209, Oct. 2017.

[26] R. Kumar and K. P. Singh, "SVM with feature selection and extraction techniques for defect-prone software module prediction," in *Proc. 6th Int. Conf. Soft Comput. Problem Solving*. Singapore: Springer, 2017, pp. 279–289.

[27] M. Gupta, K. Rajnish, and V. Bhattacharjee, "Impact of parameter tuning for optimizing deep neural network models for predicting software faults," *Sci. Program.*, vol. 2021, pp. 1–17, Jun. 2021.

[28] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[29] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. IEEE Int. Conf. Softw. Eng.*, San Francisco, CA, USA, May 2013, pp. 382–391.
[30] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.
[31] T. N. Kiff and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. ICLR*, Feb. 2017, pp. 1–14.
[32] A. V. Phan, M. L. Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *Proc. IEEE 29th Int. Conf. Tools with Artif. Intell. (ICTAI)*, Nov. 2017, pp. 45–52.
[33] S. Meilong, P. He, H. Xiao, H. Li, and C. Zeng, "An approach to semantic and structural features learning for software defect prediction," *Math. Problems Eng.*, vol. 2020, pp. 1–13, Apr. 2020.
[34] S. Sharma and R. Sharma, "A graph neural network based approach for detecting suspicious users on online social media," Inst. Comput. Sci., London, U.K., Tech. Rep., Oct. 2020.
[35] S. Banerjee and M. Khapra, "Graph convolutional network with sequential attention for goal-oriented dialogue systems," *Trans. Assoc. Comput. Linguistics*, vol. 7, pp. 485–500, Dec. 2019.
[36] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Buenos Aires, AR, USA, May 2017, pp. 3–14.
[37] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CC learner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Shangai, China, Sep. 2017, pp. 249–260.
[38] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, Buenos Aires, AZ, USA, May 2017, pp. 218–229.
[39] J. Reyes, D. Ramirez, and J. Paciello, "Automatic classification of source code archives by programming language: A deep learning approach," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Las Vegas, NV, USA, Dec. 2016, pp. 514–519.
[40] S. Zekany, D. Rings, N. Harada, M. A. Laurenzano, L. Tang, and J. Mars, "CrystalBall: Statically analyzing runtime behavior via deep sequence learning," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016, pp. 1–12.
[41] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Bremen, Germany, Sep. 2015, pp. 556–560.
[42] Y. Pang, X. Xue, and H. Wang, "Predicting vulnerable software components through deep neural network," in *Proc. Int. Conf. Deep Learn. Technol. (ICDLT)*, Chengdu China, 2017, pp. 6–10.
[43] U. Bandara and G. Wijayarathna, "Deep neural networks for source code author identification," in *Proc. 20th Int. Conf.*, Daegu, South Korea, Nov. 2013, pp. 368–375.
[44] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," *Nature*, vol. 21, pp. 436–444, Dec. 2015.
[45] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Kyoto, Japan, Mar. 2012, pp. 4277–4280.
[46] A. Krizhevsky, I. Sutskever, and E. G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Lake Tahoe, NV, USA, Dec. 2012, pp. 1097–1105.
[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Dec. 1998.
[48] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Proc. Adv. Neural Inf. Process. Syst.*, Montreal, QC, Canada, Dec. 2015, pp. 7–12.
[49] H. Gu, Y. wang, S. Hong, and G. Gui, "Blind channel identification aided generalized automatic modulation recognition based on deep learning," *IEEE Access*, vol. 7, pp. 110722–110729, 2019.
[50] R. J. Schalkoff, *Artificial Neural Networks*, 1st ed. New York, NY, USA: McGraw-Hill, 1997.
[51] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.
[52] K. Rajnish and V. Bhattacharjee, "A cognitive and neural network approach for software defect prediction," *J. Intell. Fuzzy Syst.*, vol. 43, no. 5, pp. 6477–6503, Sep. 2022.
[53] Accessed: Jul. 14, 2022. [Online]. Available: https://www.kaggle.com/code/ryanholbrook/deep-neural-networks
[54] N. Srivastav, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropouts: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Sep. 2014.
[55] Accessed: Aug. 5, 2022. [Online]. Available: https://towardsdatascience.com/understanding-the-roc-curve-and-auc-dd4f9a192ecb
[56] L. Sikic, A. S. Kurdija, K. Vladimir, and M. Silic, "Graph neural network for source code defect prediction," *IEEE Access*, vol. 10, pp. 10402–10415, 2022.
[57] A. B. Farid, E. M. Fathy, A. S. Eldin, and L. A. Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)," *Peer J. Comput. Sci.*, vol. 7, pp. 1–22, Nov. 2021.
[58] S. Omri and C. Sinz, "Deep learning for software defect prediction: A survey," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, Jun. 2020, pp. 209–214, doi: 10.1145/3387940.3391463.

MANSI GUPTA received the B.Tech. degree in computer science and engineering from Jayoti Vidyapeeth Women's University, Jaipur, in 2013, and the M.Tech. degree in computer science from Gautam Buddha University, Greater Noida, Uttar Pradesh, in 2016. She is currently pursuing the Ph.D. degree in computer science with the Birla Institute of Technology, Mesra, Ranchi. Her research interests include software metrics, software fault prediction, machine learning, and neural networks.

KUMAR RAJNISH was born in Ranchi, Jharkhand, India, in 1974. He received the B.Sc. degree in mathematics from the Ranchi College, Ranchi, in 1998, the Master of Computer Application (M.C.A.) degree from the Madan Mohan Malaviya Engineering College (MMMEC), Gorakhpur, India, in 2001, and the Ph.D. degree in computer science and engineering from the Birla Institute of Technology, Mesra, Ranchi, in 2009. From 2002 to 2015, he was an Assistant Professor (Senior Grade) with the Department of CSE, Birla Institute of Technology, where he has been continuing as an Associate Professor with the Department of CSE, since 2016. He is the author of one book and more than 60 articles. His research interests include software metrics, object-oriented software engineering, software quality, and machine learning.

VANDANA BHATTACHARJEE received the B.E. degree in CSE from the Birla Institute of Technology (BIT), Mesra, in 1989, and the M.Tech. and Ph.D. degrees in computer science from Jawaharlal Nehru University, New Delhi, in 1991 and 1995, respectively. She is currently working as a Professor with the Department of Computer Science and Engineering, BIT. She is also the Director In-Charge at BIT Lalpur Campus. She has several national and international publications in journals and conference proceedings. She has coauthored a book on data analysis. She is also working on deep learning techniques applied to the domains of software fault prediction, classification of images, disease prediction, analysis of remote sensing images, and sentiment analysis. Her research interest includes machine learning and its applications. She is a Life Member of Computer Society of India.