IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 20 December 2023, accepted 28 January 2024, date of publication 6 February 2024, date of current version 15 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3362896

RESEARCH ARTICLE

DP-CCL: A Supervised Contrastive Learning Approach Using CodeBERT Model in Software Defect Prediction

SADIA SAHAR[1,3], MUHAMMAD YOUNAS[2], MUHAMMAD MURAD KHAN[1], AND MUHAMMAD UMER SARWAR[1]

[1]Department of Computer Science, Government College University Faisalabad, Faisalabad 38000, Pakistan
[2]Department of Information Technology, Government College University Faisalabad, Faisalabad 38000, Pakistan
[3]Department of Computer Science, Government College Women University Faisalabad, Faisalabad 38000, Pakistan

Corresponding author: Muhammad Younas (younas.76@gmail.com)

ABSTRACT Software Defect Prediction (SDP) reduces the overall cost of software development by identifying the code at a higher risk of defects at the initial phase of software development. SDP helps the test engineers to optimize the allocation of testing resources more effectively. Traditional SDP models are built using handcrafted software metrics that ignore the structural, semantic, and contextual information of the code. Consequently, many researchers have employed deep learning models to capture contextual, semantic, and structural information from the code. In this article, we propose the DP-CCL (Defect Prediction using CodeBERT with Contrastive Learning) model to predict the defective code. The proposed model employs supervised contrastive learning using this CodeBERT Language model to capture semantic features from the source code. Contrastive learning extracts valuable information from the data by maximizing the similarity between similar data pairs (positive pair) and meanwhile minimizing the similarity between dissimilar data pairs (negative pair). Moreover, The model combines the semantic features with software metrics to obtain the benefits of both semantic and handcrafted features. The combined features are input to the logistic regression model for code classification as either buggy or clean. In this study, ten PROMISE projects were utilized to conduct the experiments. Results show that the DP-CCL model achieved significant improvement i.e., 4.9% to 14.9% increase in F-Score as compared to existing approaches.

INDEX TERMS Software bug prediction, software fault prediction, software defect prediction, BERT, CodeBERT, language model, pre-trained model, deep learning, contrastive learning, contrastive loss.

I. INTRODUCTION

Complex and large-scale software are significant challenges for developers in terms of debugging and sustaining the software quality [1]. Software defects that need to be effectively addressed in the initial development phase can escalate into more intricate issues with the progression of software development. The propagated defects are difficult to rectify because they are more hidden, challenging, and resource-intensive at the last stages. This drawback leads to potential disruptions in functionality, delayed timelines, necessitating extensive rework, increased cost, and ultimately the system failure [2], [3].

To address the issues mentioned above, various researchers have introduced software defect prediction (SDP) [1], [4], [8], [12], [13], [14], which empowers developers to identify the code in advance that are at higher risk of defects [3], [23]. SDP helps the test engineers to optimize the testing resource allocation more effectively [2]. Hence, it reduces the cost of software development [5], [6]. Traditional SDP models are trained on software metrics (features) extracted from historical defective data. The most popular software metrics used in the research are Line of Code (LOC) [15], Halstead code metrics [16], McCabe Cyclomatic Complexity

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

feature [17] object-oriented features like Chidamber and Kemerer(CK) metrics [20], MOOD [19], QMOOD) [18], code churn metrics [21], Developers micro-interaction metrics [8], [22], and change software metrics [9], [10], [11].

The traditional software metrics ignore the structural, semantic, and contextual information of the code [7], [13], [25], [26]. Most of the traditional metrics cannot distinguish code module that has different semantics because the metrics may hold the same information for both code snippets [13], [14], [24], [27], [28].

Recent research work [12], [13], [14], [29], [30] have used different deep learning algorithms like deep belief network (DBN) [13], [32], convolutional neural network (CNN) [29], recurrent neural network (RNN) [31], Long Short-Term Memory (LSTM) [30] to extract semantics and contextual information from the raw code. DBN model generates features by processing tokens one by one, so there is no semantic dependency between the preceding and subsequent tokens. CNN models generate features that have local dependency and do not guarantee global dependency [3]. Recurrent Neural Networks [31] have been employed to capture global dependencies between the tokens of sequence vectors. However, RNN loses essential information while processing longer token sequences due to the vanishing gradient problem [3]. Therefore, the feature representation of these methods is not optimal.

In this study, we introduced a Defect Prediction CodeBERT model with Contrastive Learning (DP-CCL). The model utilizes a pre-trained CodeBERT [33] model to extract code features and apply contrastive learning (CL) to enhance overall performance. Contrastive learning learns the useful information of data [36], [37] by maximizing the similarity between similar data pairs (positive pair) and minimizing the similarity between dissimilar data pairs (negative pair) [34], [35]. The CodeBERT has been used in various software engineering domains [38], [39], [40] to enhance feature representations. In most of the studies, the combination of CL with the pre-trained language model has been proven effective and achieved satisfying results [34], [35], [36], [37], [41]. DP-CCL model was evaluated on the Promise datasets using F1 and MCC metrics to measure the effectiveness of the model. Results show that the DP-CCL approach outperformed the existing approaches.

A. CONTRIBUTION

This work has the following main contributions:

- Utilization of a pretrained CodeBERT language model to get feature representation of source code. As it handles programming-related contexts effectively, it generates semantic information of source code more effectively than previous approaches.
- Designed a contrastive learning approach to capture the similarity among code instances with the same labels and dissimilarity among those with different labels.
- Fusion of semantic and handcrafted features to get a better final feature representation.
- Evaluated ten projects of the PROMISE dataset and improved the performance as compared to the state-of-the-art studies.

B. PAPER ORGANIZATION

The structure of the remaining article is as follows: Section II deals with background information and reviews existing literature on defect prediction; Section III introduces the proposed DP-CCL model; Section IV shows the result of DP-CCL approach; Section V highlights some validity threats; and Section VI concludes the work and discusses the future work.

II. RELATED LITERATURE

A. SOFTWARE DEFECT PREDICTION

The study [13] employed Deep Belief Network, a deep learning algorithm, to capture semantic and structural information of the code. The author converted the code into an Abstract Syntax Tree (AST). The AST is then parsed to obtain the token sequence. Each node has some information. The information of the node is considered as the token. A number uniquely identifies each token in the token vector. This token sequence is then fed to the DBN to get the semantic feature of the given source code. These features are then used to train LR or Random Forest model to classify the given code examples as defective or clean. Results of the proposed study showed that the semantic features enhanced the model's performance compared to the traditional model trained on traditional handcrafted features.

The author of [29] developed the DP-CNN model to capture the semantic and structural features by parsing AST nodes of the source code. Subsequently, the model combined the captured semantic features with traditional handcrafted features. The combined features are then input into a convolutional neural network to make the final defect predictions. Comments of code with semantic features generated by CNN merged in a study [25] to predict the defect proneness of code.

Some other work captured semantic features using RNN [60], [61], and semantic and syntactic features from the code using LSTM [14], [30], [62]. The studies [3], [12], [65] also used the LSTM model to capture semantic features from code as well as from handcrafted features and then combined both features using a gated mechanism. The author of [63] and [64] trained the TreeLSTM model to capture a better representation of code.

The proposed approach incorporates contrastive learning, which has been acknowledged as a potent tool for capturing enhanced feature representations in various domains of software engineering.

B. SOFTWARE DEFECT PREDICTION AND PRE-TRAINED LANGUAGE MODEL

Pre-trained language models are machine learning models that have been trained on large text data and can subsequently be fine-tuned for downstream NLP (Natural Language

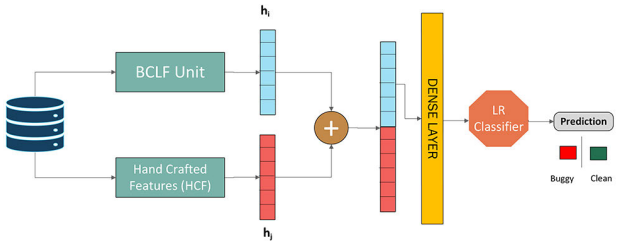

FIGURE 1. Workflow of DP-CCL model.

Processing) tasks. These pretrained models have shown remarkable achievement in different NLP tasks [42], [44] as well as software engineering research like code clone [45], code search [46], [47], code summarization [48], and code vulnerability [49], [50] etc.

Researchers [51], [52], and [53] have investigated the effectiveness of pretrained language models in the field of defect prediction. The author of [52] proposed a defect prediction model that mines the semantic feature using BiLSTM and BERT algorithms. In the study [51], a feature representation of the source program was generated by utilizing the UniXcoder (a pretrained Language model) in combination with a multi-channel convolutional neural network. This approach extracted semantic information by incorporating both the program text and the associated code comments.

The proposed method is different from the existing approaches because the suggested model incorporates the CodeBERT model with Contrastive Loss to get a better representation of the code at the file level.

C. CONTRASTIVE LEARNING

Contrastive learning has been quite successfully used in various applications, including computer vision [34], [55], [56], natural language processing [35], [37], [41], [54] and software engineering tasks [38], [39], [40], [41], [57]. Contrastive learning allows models to learn meaningful representations of the instance from large datasets. Nowadays, the integration of contrastive learning with pretrained models like BERT is pervasive to fine-tune the model to enhance the learning ability of the language model [35], [58], [59].

The proposed model applied contrastive learning using pretrained language models in order to explore the effectiveness of the combination in defect prediction tasks.

III. PROPOSED APPROACH

Figure 1 provides a comprehensive overview of the proposed model's (DP-CCL) workflow. Initially, The model parses the source files using Abstract Syntax Tree to gather the sequence of tokens extracted from the AST. This sequence retains the semantic information of the code [12]. Subsequently, the model provides this sequence to the BCLF (CodeBERT Contrastive Learning Feature) unit to obtain vector representations for each sequence. These word embedding vectors are then combined with Handcrafted Features (HCF) to exploit the benefits of both semantic and HC features. The combined features are input into the logistic regression model for code classification as either buggy or clean.

A. DATA CONSTRUCTION

The data construction phase starts with the conversion of source code into an Abstract Syntax Tree. AST can generate better syntactic representation as compared to the traditional approaches [13], [70]. AST is a tree-based structure that contains rich semantic and structural information of the program [14], [27]. Each node on the tree represents a structure in the source code. In the proposed method, three types of AST nodes are considered: (1) The nodes retain information about method invocations or the creation of class instances; (2) Declaration nodes hold information regarding different objects like method declarations, enum declarations, or type declarations; (3) Nodes belong to the Control flow structure like loop structure (statements belong to for, while, etc.), conditional structure such as 'if statement', 'catch statement' or 'throw statement', etc.

All three types are recorded using the type and name of the node, separated with "_". For example, if the node in AST is represented as "MethodDeclaration_setWebxml", then MethodDeclaration will be the type of node and setWebxml will be the name of the node.

Table 1 enlists different types of nodes that are used in the study. Except for the enlisted nodes, all other nodes are excluded. The exclusion of unnecessary nodes is because they are often specific to particular methods or classes and may not be relevant to the entire project. Including them could dilute the significance of other nodes [13]. Some other nodes, like curly braces or keywords like "args" or "main", are also excluded.

The model splits each token into sub-tokens. For example, The model splits the "MethodDeclaration_setWebxml" as ["MethodDeclaration", "setWebxml"]. The reason to split the tokens is to standardize the naming styles [12] and enhance the expressive ability of the token across the project [66], [67], [68], [69]. These identical subtokens often share a common logical meaning across different software projects. Splitting of token reduces the chance of occurrence of unknown tokens during embedding [66]. Special tokens [CLS] and [SEP] are added at the beginning and the end of each sequence.

The final look of the token sequence is like this:

$$Token_Seq = \{[CLS], tok_1, tok_2, \ldots\ldots, tok_n, [SEP]\} \quad (1)$$

The sequence is input to the CodeBERT model to generate an embedding vector for each token in the token vector. The CodeBERT language model generates similar embedding vectors for nodes of the same type when they have similar contexts. One token sequence is created for one code file.

B. CLASS IMBALANCE

Oversampling and undersampling are two common class imbalance handling techniques used in previous studies. Both

TABLE 1. Selected AST nodes.

Type of Node	Desription
Nodes For method invocation/instance creation	(1): SuperMethodInvocation, (2): MethodInvocation, (3): ClassCreator
Declaration Nodes	Package Declaration, Interface Declaration, ClassDeclaration, Constructor Declaration, MethodDeclaration, Variable Declarator, FormalParameter
Nodes For Control-Flow Structure	IfStatement, For Statement, While Statement, DoStatement, AssertStatement, Break Statement, Continue Statement, ReturnStatement, ThrowStatement, TryStatement, SynchronizedStatement, Switch Statement, BlockStatement, Catch ClauseParameter, TryResource, Catch Clause, SwitchStatementCase, For Control, EnhancedForControl
Additional Nodes	BasicType, MemberReference, Reference Type, SuperMember Reference, Statement Expression

techniques have their limitations. Oversampling replicates the samples, which leads to overfitting in a model, while undersampling may lose important information associated with majority class samples. Here, we applied the ''weighted Random Sampling'' technique to balance the samples belonging to different classes. weighted Random Sampling technique applies different weights to different samples according to the class they belong to.

C. CONSTRUCTION OF BCLF UNIT

CodeBERT Contrastive Learning Feature (BCLF) Unit comprises two parts, i.e., (1) Extraction of features from the CodeBERT model and (2) Application of Contrastive Learning.

1) EXTRACTION OF CODEBERT FEATURES

The proposed technique employs the CodeBERT language model to extract the semantic and contextual information of the code. CodeBERT [33] is a pretrained model, the architecture of which is optimized to handle programming-related contexts effectively. It is trained on a large corpus of programming languages. The CodeBERT is designed to understand the semantics of source code and has been proven to be an efficient model in various software engineering applications like code clone [45], code summarization [72], vulnerability detection [71], and defect prediction [53] etc. Figure 2 shows the extraction of features from the CodeBERT Model.

To get the embedding vector of the token sequence, the model converts the token sequence into three encoding vectors i.e., (1) Token Embeddings $E[tk]$ (2) Segment Embeddings $[EA]$ and (3) Position embedding $[EP]$. Position embedding helps the encoder to embed the token according to its position. If a token has two different positions, the encoder generates the embedding vector for that token corresponding to its position. CodeBERT has 12 layers of a transformer having 768 hidden units for each layer. The multi-head and feed-forward mechanism of the encoder extracts the semantic information of the source code. The encoder uses three encoding vectors to map a token into an m-dimensional integer vector of a fixed length. CodeBERT maps each token into 768 dimensions. Each source code has a ''d_n x d_t'' dimensional vector, where d_n is the total number of

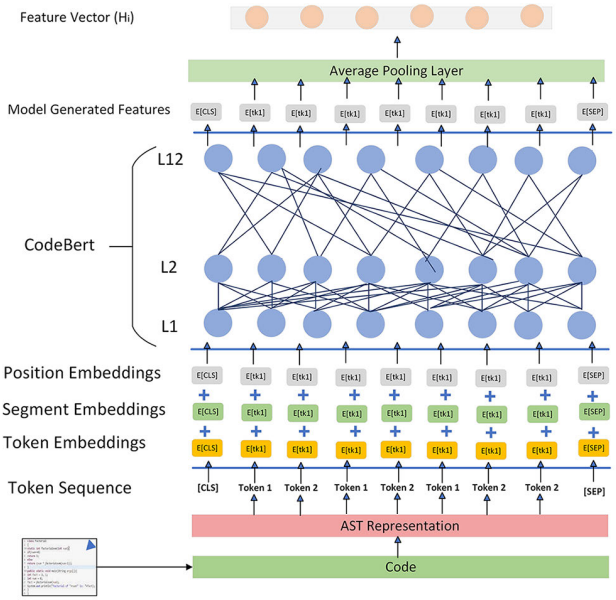

FIGURE 2. Feature extraction via CodeBERT model.

tokens in the token sequence of a code file and d_t is the dimension of each vector. Here, we selected 512 tokens. The CodeBert-generated features are as follows:

$$E(TS) = \{E[CLS], E[T_1], E[T_2], \ldots\ldots, E[T_n], E[SEP]\} \tag{2}$$

Here, $E[CLS]$ is the vector that stores the information to classify the particular source code. Finally, the model merges all the vectors using the average pooling layer to get the final feature representation $\mathcal{H}$.

2) APPLICATION OF CONTRASTIVE LEARNING

The model applies Supervised Contrastive learning [73] on the CodeBERT-generated Features $\mathcal{H}$ to obtain a similar representation of source code to the positive and dissimilar to the negative instance. Figure 3 shows the application of supervised contrastive learning.

The dataset used in the study contains N training examples. Each training example x_i can be categorized into two classes, i.e., y = [0, 1]. In the defect prediction problem, ''0'' is for the clean class, and ''1'' is for the defect class. x_i is the input code

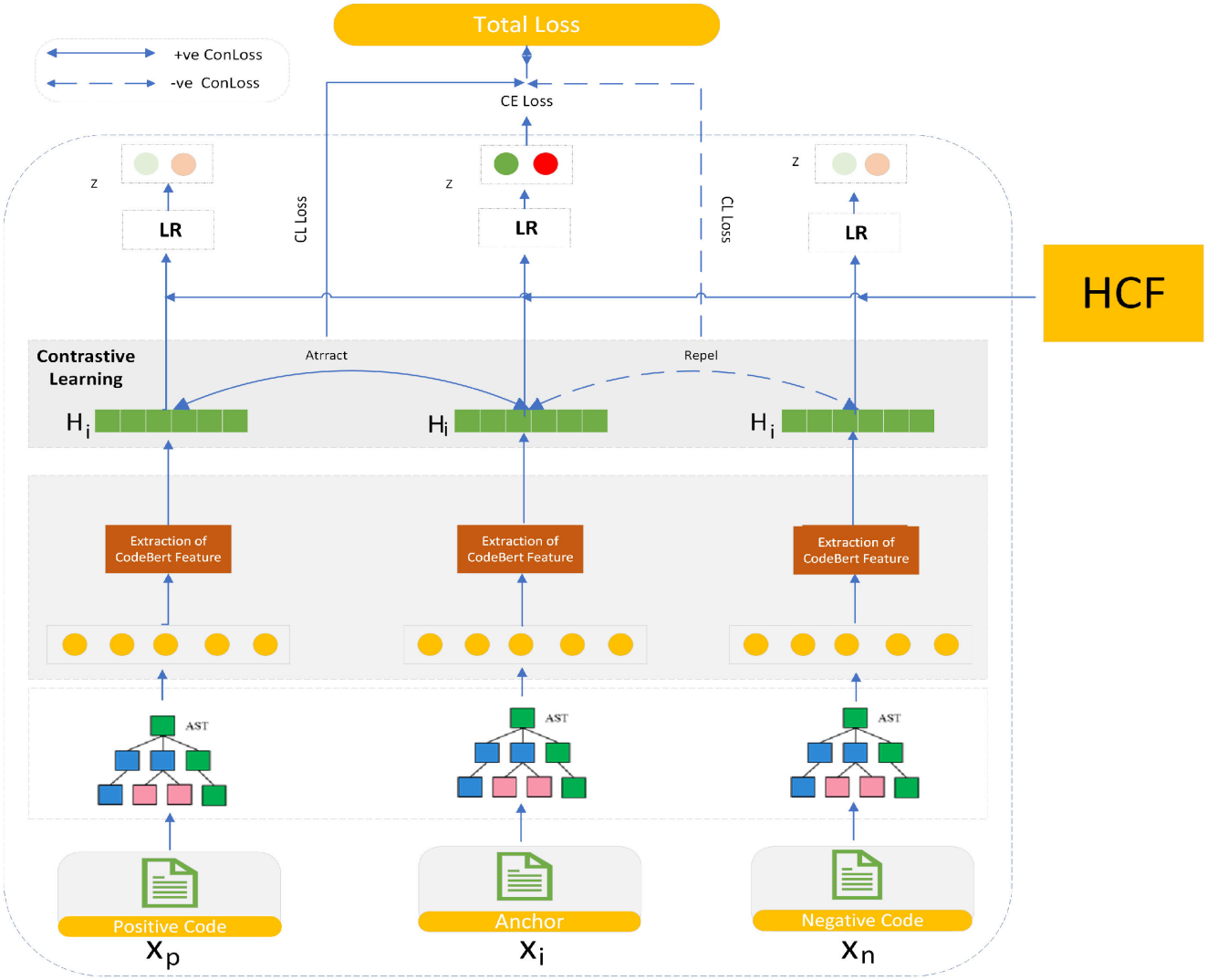

FIGURE 3. Application of contrastive learning.

file with L tokens considered as an anchor having the label y_i. Here, at least one example behaves as a positive sample x_p with the same label y_i, and the remaining examples are from the negative class x_n. The positive class has the same label as y_i, and the label of the negative class is different from y_i. The Supervised contrastive loss applied in the study is:

$$\mathcal{L}_{sup} = \frac{1}{N} \sum_{i \epsilon \mathcal{I}} \frac{1}{|\mathcal{P}_i|} \sum_{p \epsilon \mathcal{P}_i} -log \frac{exp(z_i \bullet z_p / \tau)}{\sum_{a \epsilon \mathcal{A}_i} exp(z_i \bullet z_a / \tau)} \quad (3)$$

where z_i and z_p are the normalized representations of anchor xi and positive example x_p, $\mathcal{I}$ is the set of training examples, $\mathcal{A}_i$ is the set of the contrastive samples of x_i, $\mathcal{P}_i$ is the set of positive samples having the same label as x_i, $|\mathcal{P}_i|$ is the cardinality of $\mathcal{P}_i$, the "$\bullet$" symbol denotes the dot product and $\tau \epsilon \mathbb{R}^+$ is the temperature factor.

This contrastive loss learns generic representations $\mathcal{H}_i$ for the input examples.

D. FUSION OF BCLF WITH HCF (HANDCRAFTED FEATURES

Previous approaches have proven the effectiveness of fusing semantic features with handcrafted features. To obtain the benefit of both features, the CodeBERT features are combined with handcrafted features that are listed in Table 2. These eighteen features are used in this work because these features have been widely selected by earlier literature [3], [12], [29] to measure the effectiveness of handcrafted with deep features. Second, they are publicly available. The standard-scaling approach is used to normalize the handcrafted features.

E. PREDICTION OF BUGGY CODE

The feature $\mathcal{Z}$ generated by the DP-CCL model is input to the Dropout layer to avoid overfitting. The output of the Dropout layer is then given to the dense layer to get the final representation $\mathcal{H}$. The final feature vector is given to the

TABLE 2. Handcrafted features used in the study.

List of Hand-Crafted Features	Metric Symbol
Data Access Metric	(DAM)
Lines Of Code	(LOC)
Inheritance Coupling	(IC)
Coupling Between Object Class	(CBO)
Afferent Couplings	(CA)
Weighted Methods per Class	(WMC)
Number of Public Methods	(NPM)
Cohesion Among Methods of Class	(CAM)
Response For a Class	(RFC)
Number Of Children	(NOC)
Average Method Complexity	(AMC)
Lack of Cohesion in Methods	(LCOM3)
Coupling Between Methods	(CBM)
Measure Of Aggregation	(MOA)
Efferent Couplings	(CE)
Lack of Cohesion in Methods	(LCOM)
Depth of Inheritance Tree	(DIT)
Measure of Functional Abstraction	(MFA)

classifier to learn the respective class of input code example. The prediction of the classifier is as follows:

$$\hat{y} = Sigmoid(\mathcal{H}) \tag{4}$$

F. OBJECTIVE FUNCTION

The Cross-Entropy (CE) loss has been calculated to measure the accuracy of classification as follows:

$$\mathcal{L}_{CE} = \sum_{i \epsilon \mathcal{I}} \sum_{i \epsilon \mathcal{Y}} \hat{y}_i^j log y_i^j + \lambda ||\theta||^2 \tag{5}$$

where $\mathcal{I}$ is the set of all training examples, $\mathcal{Y}$ is the set of class labels (buggy or clean), y_i is the correct label, $\hat{y}$ is the predicted label of instance i, and $\lambda||\theta||^2$ is the l2 regularization.

The total loss has been calculated by merging the CE and CL as follows:

$$Total_loss = \mathcal{L}_{sup} + \mathcal{L}_{CE} \tag{6}$$

The total loss enhances the feature representation and the predicted output of the classifier.

IV. EVALUATION

A. EXPERIMENT SETUP

1) EVALUATION DATASET

The study utilized ten Java projects to analyze the performance of the proposed work. The data of these projects were collected from the PROMISE[1] repository, which is a publicly accessible code repository. The corresponding source code was collected from GitHub[2] and Apache[3] websites. Fifteen experiments were conducted to evaluate the model. For each experiment, the model is trained on an old version of a project and validated/tested on the latest version of the project. For example, Ant version 1.5 was used to train the model, and Ant version 1.6 to test the effectiveness of the model. The repository was chosen because it is publicly available and has been extensively utilized by many studies to evaluate the performance of their proposed work. Table 3 outlines the details of the dataset i.e., the project name, versions of projects employed in various experiments, the project description, the average No. of Java files in a project, and the corresponding percentage of defects. The average Java files across all projects span from 122 to 815, while the average rate of bugs ranges from 20 percent to 63 percent.

[1]http://openscience.us/repo/defect/
[2]https://github.com/
[3]https://www.apache.org/

2) EVALUATION METRICS

F1 and MCC Scores were calculated to measure the performance of the DP-CCL model. These evaluation metrics have been widely used in previous studies [13], [26], [27], [66], [74]. F1 is the harmonic mean of precision and recall and assigns equal weight to both precision and recall. The better the F1 score, the higher the performance of the prediction model.

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \tag{7}$$

Recall and Precision can be calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

where TP stands for True Positive, TN for True Negative, FP for False Positive, and FN for False Negative.

Matthews Correlation Coefficient (MCC) is the evaluation metric that is less affected by the class imbalance [9], [75]. It provides more informative and accurate scores when applied to binary classifications [9], [76]. It is a correlation coefficient between the predicted and true class labels, ranging from -1 to 1. The upper value, 1, represents that the prediction is perfect, and the lower value,-1, indicates that the model is worst [9] in its performance. MCC can be calculated as:

$$MC = \frac{(TP \times TN - FP \times FN)}{(TP + FP)(TP + FN)(TN + FP)(TN + FN)} \tag{10}$$

where TP represents True Positive; TN represents True Negative; FP represents False Positive; FN represents False Negative;

3) STATISTICAL METHODS

To prove the statistical efficacy of the DP-CCL model, Scott-Knott ESD (effect size difference) was applied [3], [74]. Scott-Knott ESD is a hierarchical clustering method that arranges different data into ranks, facilitating a deeper understanding of the data distribution.

The win\tie\loss test on the F1 and MCC scores was also applied to compare the model with existing approaches. In case the DP-CCL model achieved the best result on the same project and metric (such as MCC or F1), it was counted as a win for the DP-CCL model. Conversely, if the model performed worse, it was considered a loss for the

TABLE 3. Detail of PROMISE dataset.

Project Name	Overview	Project Release	No. of files (Average)	No. of Defects (Average)
ant	A Java-based build tool.	1.5, 1.6, 1.7	463	19.8
Jedit	A text editor.	3.2, 4.0, 4.1	296.7	27.6 %
poi	A library providing access to Microsoft Office files in Java.	1.5, 2.5, 3.0	354.7	62.5 %
log4j	A logging library for Java.	1.0, 1.1	122	29.6 %
xerces	An open-source XML parser.	1.3, 1.4.4	520	48.6 %
camel	An open-source integration framework.	1.2, 1.4, 1.6	815	23.9 %
Lucene	An open-source search software.	2.0, 2.2, 2.4	260.7	54.9 %
xalan	Java tool to convert XML documents.	2.4, 2.5	763	31.7%
Velocity	A generic template engine in Java	1.5, 1.6	221	49.7 %
synapse	Java library used to transport high speed data	1.1, 1.2	239	30.5 %

proposed model. If both approaches had the same result, it was considered a tie. Moreover, the delta effect size was also applied to measure the effectiveness of each method.

4) PARAMETER SELECTION

During the training phase, a dropout rate of 0.2 was applied to prevent overfitting. A Dense layer with 300 hidden units, a training batch size of 16, the Adam optimizer, and 25 epochs were used to fine-tune the model.

5) BASELINE METHODS

Following defect prediction approaches were adopted as Baseline methods:

1) HC-RF: Random forests Model with Traditional handcrafted features.
2) DBN-LR: The model generates semantic features using Deep-Belief Network followed by Logistic Regression [13].
3) CNN: The model generates semantic features using a Convolutional Neural Network followed by a Logistic Regression Classifier.
4) CNN-HC: A modified version of the CNN model. It combines semantic features with traditional handcrafted features [29] to classify the code.
5) LSTM: LSTM-based deep features with LR Classifier [30].
6) LSTM-HC: LSTM with traditional metrics to classify the bugs.
7) CodeBERT-LR: CodeBERT generated features with LR Classifier.
8) CodeBERT-CNN: CodeBERT incorporated with the CNN model to predict the defects.
9) CodeBERT-LSTM: CodeBERT incorporated with the LSTM model to predict the defects.

B. EXPERIMENTAL RESULTS

This section answers the following four research questions.

RQ1: How proficient is the proposed method in performance as compared to the existing algorithms?

RQ2: How does the performance of features extracted by DP-CCL using contrastive learning compare to existing approaches?

RQ3. How do different variants of the Bert model behave in the context of evaluation measures?

RQ4. What is the impact of the number of hidden units in the Dense layer on the performance of the proposed model?

RQ1 involves assessing the performance of the proposed model against existing algorithms using F1 and MCC evaluation metrics. RQ2 aims to examine the impact of contrastive learning on factors like the F-score and provides visual representations of features extracted by both the proposed and existing approaches. RQ3 is dedicated to evaluating the performance of various Bert variants. RQ4 delves into the influence of Hidden units of the Dense Layer on the defect prediction model's performance.

1) RQ.1: HOW PROFICIENT IS THE PROPOSED METHOD IN PERFORMANCE AS COMPARED TO THE EXISTING ALGORITHMS?

a: MOTIVATION

Contrastive learning with the pretrained language model has been proven effective and achieved satisfying results [34], [35], [36], [37], [41]. That motivates us to incorporate contrastive learning to predict defective files. In this question, the study has examined whether the DP-CCL can achieve superior performance compared to the existing approaches or not.

b: APPROACH

To address this question, Fifteen experiments were conducted to evaluate the model. For each experiment, the model was trained on an old version of a project and validated/tested on the latest version of the project. F1 and MCC scores were used to compare the performance of DP-CCL with baseline approaches.

c: RESULTS

Table 4 displays the F-Score of DP-CCL and the baseline studies. The proposed approach achieved an average F-score of 0.66, compared to 0.0.51 for CNN, 0.53 for CNN-HC, 0.51 for DBN, 0.51 for LSTM, 0.53 for LSTM-HC models, 0.56 for CodeBERT-LR, 0.58 for CodeBERT-LSTM, and 0.61 for CodeBERT-CNN. It is worth noticing that methods that generated features using the pretrained CodeBERT model outperformed the other baseline methods, which

TABLE 4. F1-score of existing and proposed approaches.

Train Project	Test Project	DP-CCL	HC-RF	DBN-LR	LSTM	LSTM-HC	CNN	CNN-HC	CodeBERT-LR	CodeBERT-LSTM	CodeBERT-CNN
ant_1.5	ant_1.6	**0.633**	0.286	0.417	0.256	0.300	0.216	0.230	0.494	0.534	0.606
ant_1.6	ant_1.7	**0.588**	0.556	0.544	0.442	0.543	0.496	0.521	0.455	0.414	0.536
camel_1.2	camel_1.4	**0.494**	0.371	0.319	0.474	0.355	0.410	0.466	0.385	0.325	0.363
camel_1.4	camel_1.6	0.507	0.439	0.320	0.447	0.338	0.326	0.350	**0.510**	0.424	0.384
jedit_3.2	jedit_4.0	**0.632**	0.600	0.591	0.549	0.571	0.613	0.616	0.415	0.463	0.472
jedit_4.0	jedit_4.1	0.593	**0.617**	0.586	0.573	0.548	0.559	0.559	0.393	0.304	0.560
log4j_1.0	log4j_1.1	**0.730**	0.688	0.507	0.600	0.687	0.688	0.688	0.061	0.615	0.579
lucene_2.0	lucene_2.2	0.691	0.634	0.759	0.662	0.630	0.611	0.597	**0.760**	0.759	0.736
lucene_2.2	lucene_2.4	0.774	0.673	0.363	0.633	0.690	0.648	0.636	**0.780**	0.763	0.770
poi_1.5	poi_2.5	**0.862**	0.844	0.709	0.791	0.557	0.784	0.803	0.794	0.794	0.796
poi_2.5	poi_3.0	**0.819**	0.745	0.781	0.715	0.699	0.727	0.717	0.816	0.813	0.816
synapse_1.1	synapse_1.2	0.502	0.406	0.526	0.395	0.558	0.434	0.476	0.528	0.519	**0.568**
xalan_2.4	xalan_2.5	**0.644**	0.399	0.386	0.314	0.398	0.333	0.346	0.638	0.619	0.607
xerces_1.3	xerces_1.4.4	0.792	0.370	0.299	0.309	0.488	0.227	0.263	**0.799**	0.745	0.773
Average		**0.656**	0.547	0.507	0.510	0.530	0.511	0.525	0.559	0.578	0.607
Win		-	14	13	15	13	14	14	10	13	13
Tie		-	00	00	00	00	00	00	00	00	00
Loss		-	01	02	00	02	01	01	05	02	02
Effect Size		-	large	large	large	large	large	large	large	large	large

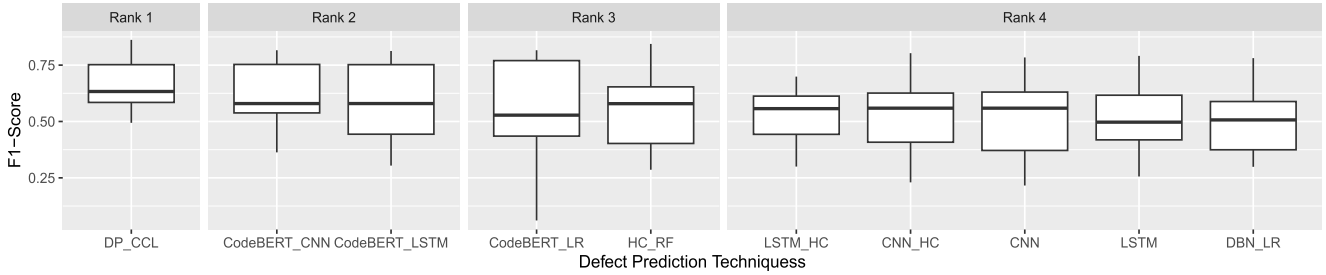

FIGURE 4. Ranking of existing and proposed methods using F1-Score with Scott-Knott.

shows that features generated by CodeBERT are more effective. The DP-CCL model incorporated contrastive learning with CodeBERT and achieved a higher F-Score compared to all baseline methods.

Table 5 shows the average MCC-Scores of DP-CCL and existing models. The average scores of the proposed technique, HC-RF, DBN-LR, CNN, CNN-HC, LSTM, LSTM-HC, CodeBERT-LR, CodeBERT-LSTM, and CodeBERT-CNN models are 0.37, 0.32, 0.21, 0.24, 0.26, 0.29, 0.31,0.22,0.21 and 0.28 respectively. The mean value of the DP-CCL model is 4.7% to 16.3% higher than the baseline methods.

The Scott-Knott ESD test further validates that the DP-CCL approach consistently occupies the highest rank in F1 and MCC Score. Figure 4 and Figure 5 illustrate the results obtained from the Scott-Knott ESD analysis based on the F1 and MCC scores, respectively. The middle horizontal lines of the graphs represent the median value of the F-measure and MCC Score.

Win\Tie\Loss test also supports the suggested approach over the baseline approaches. The result of the delta effect size shows that the performance difference between the proposed and baseline is statistically significant, with a noticeable effect size.

In conclusion, DP-CCL significantly improves accuracy, which means it can detect more defective software instances than baseline methods. This may be because our study used CodeBERT, which extracts the embedding of the specific token according to its position in the statement. Previous studies exploited the word2vec. Word2vec generates only one embedding vector regardless of its position. Second, our study applied contrastive learning. Contrastive learning generates embedding vectors of positive classes that are the same as other positives and apart from negative classes. Previous studies did not apply Contrastive learning at the file level.

The proposed method exhibits a higher level of accuracy, improving the state-of-the-art by 4.7% to 16% in terms of MCC and 10.9% to 14.9% in terms of F1-Score.

2) RQ2. HOW DOES THE PERFORMANCE OF FEATURES EXTRACTED BY DP-CCL USING CONTRASTIVE LEARNING COMPARE TO EXISTING APPROACHES?

To assess the impact of features derived from DP-CCL, a comparative analysis of the proposed model was conducted under two scenarios. Initially, we compared the DP-CCL, which incorporates contrastive learning, with a model that lacks contrastive learning. The results of this comparison are presented in Figure 6 and Table 6. The comparison demonstrates that the DP-CCL showed 3% higher results as compared to the model without contrastive learning.

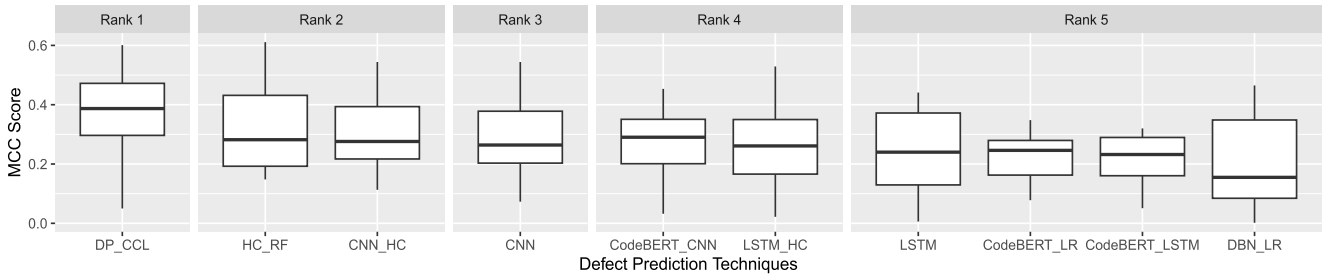

FIGURE 5. Ranking of existing and proposed methods using MCC-score with Scott-Knott.

TABLE 5. MCC-score of existing and proposed approaches.

Train Project	Test Project	DP-CCL	HC-RF	DBN-LR	LSTM	LSTM-HC	CNN	CNN-HC	CodeBERT-LR	CodeBERT-LSTM	CodeBERT-CNN
ant_1.5	ant_1.6	**0.489**	0.176	0.010	0.162	0.022	0.200	0.204	0.263	0.314	0.443
ant_1.6	ant_1.7	**0.454**	0.407	0.391	0.258	0.390	0.341	0.372	0.272	0.212	0.400
camel_1.2	camel_1.4	**0.382**	0.194	0.155	0.360	0.202	0.264	0.338	0.246	0.126	0.184
camel_1.4	camel_1.6	**0.387**	0.261	0.130	0.343	0.130	0.209	0.236	0.143	0.219	0.140
jedit_3.2	jedit_4.0	**0.524**	0.456	0.465	0.384	0.416	0.476	0.482	0.188	0.269	0.283
jedit_4.0	jedit_4.1	0.455	**0.482**	0.453	0.403	0.409	0.415	0.415	0.213	0.051	0.453
log4j_1.0	log4j_1.1	0.601	**0.611**	0.064	0.441	0.529	0.544	0.544	0.138	0.255	0.105
lucene_2.0	lucene_2.2	0.269	**0.282**	0.005	0.240	0.116	0.183	0.173	0.167	0.192	0.270
lucene_2.2	lucene_2.4	0.050	0.191	0.208	0.006	0.297	0.104	0.113	0.301	0.232	**0.304**
poi_1.5	poi_2.5	**0.525**	0.519	0.300	0.390	0.220	0.481	0.495	0.287	0.320	0.297
poi_2.5	poi_3.0	0.393	0.395	0.359	0.236	0.278	0.307	0.276	0.311	0.313	**0.397**
synapse_1.1	synapse_1.2	0.106	0.185	**0.338**	0.150	0.261	0.207	0.255	0.158	0.129	0.290
xalan_2.4	xalan_2.5	**0.180**	0.148	0.105	0.109	0.057	0.073	0.151	0.078	0.073	0.032
xerces_1.3	xerces_1.4.4	**0.356**	0.200	0.110	0.041	0.310	0.206	0.230	0.348	0.270	0.291
velocity_1.5	velocity_1.6	0.324	0.289	0.002	0.030	0.224	0.336	**0.365**	0.248	0.309	0.218
Average		**0.369**	0.322	0.206	0.237	0.260	0.286	0.306	0.222	0.213	0.278
Win		-	9	13	14	13	12	12	13	13	11
Tie		-	0	0	0	0	0	0	0	0	0
Loss		-	6	2	1	2	3	3	2	2	4
Effect Size		-	medium	large	large	large	large	medium	large	large	large

TABLE 6. F1-Score of DP-CCL model and CodeBERT-CE model.

Train Project	Test Project	DP-CCL	CodeBert-CE
ant vers. 1.5	ant vers. 1.6	0.633	0.611
ant vers. 1.6	ant vers. 1.7	0.588	0.551
camel vers. 1.2	camel vers. 1.4	0.494	0.506
camel vers. 1.4	camel vers. 1.6	0.507	0.542
jedit vers. 3.2	jedit vers. 4.0	0.632	0.533
jedit vers. 4.0	jedit vers. 4.1	0.593	0.584
log4j vers. 1.0	log4j vers. 1.1	0.730	0.824
lucene vers. 2.0	lucene vers. 2.2	0.691	0.721
lucene vers. 2.2	Lucene vers. 2.4	0.774	0.726
poi vers. 1.5	poi vers. 2.5	0.862	0.825
poi vers. 2.5	poi vers. 3.0	0.819	0.828
synapse vers. 1.1	synapse vers. 1.2	0.502	0.404
xalan vers. 2.4	xalan vers. 2.5	0.644	0.556
Xerces vers. 1.3	Xerces vers. 1.4.4	0.792	0.602
velocity vers. 1.5	velocity vers. 1.6	0.581	0.573
Average		0.656	0.626

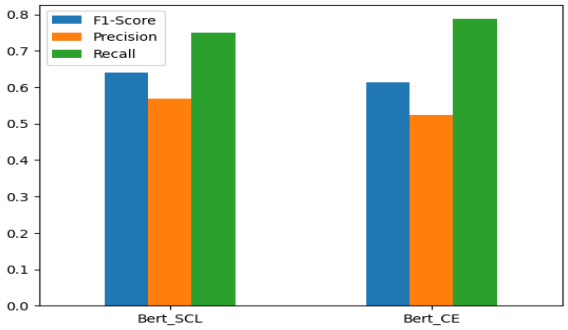

FIGURE 6. Impact of contrastive learning on model using F1-score.

In the second scenario, the study validates the enhanced discriminative capabilities of the proposed model. The study compares the proposed and baseline techniques by visualizing and analyzing the features extracted from the different datasets using t-distributed stochastic neighbor embedding (t-SNE) [43]. Figure 7 illustrates the t-SNE distribution of each model. In this visualization, red denotes instances with defects, while blue represents instances without defects.

Upon close inspection of Figure 7, it becomes evident that there is an increased overlap of red and blue regions within the feature distributions of DBN, CNN, CNN-HC, LSTM, LSTM-HC, CodeBERT-LR, CodeBERT-LSTM, and CodeBERT-CNN algorithms. DP-CCL exhibits a clearer separation compared to Figure 7. This indicates that different types of instances, both with and without defects, maintain consistent intra-class distinctions in DP-CCL.

Consequently, these results strongly suggest that the implementation of contrastive learning through DP-CCL significantly enhances the discriminative power of the model for both defective and non-defective data.

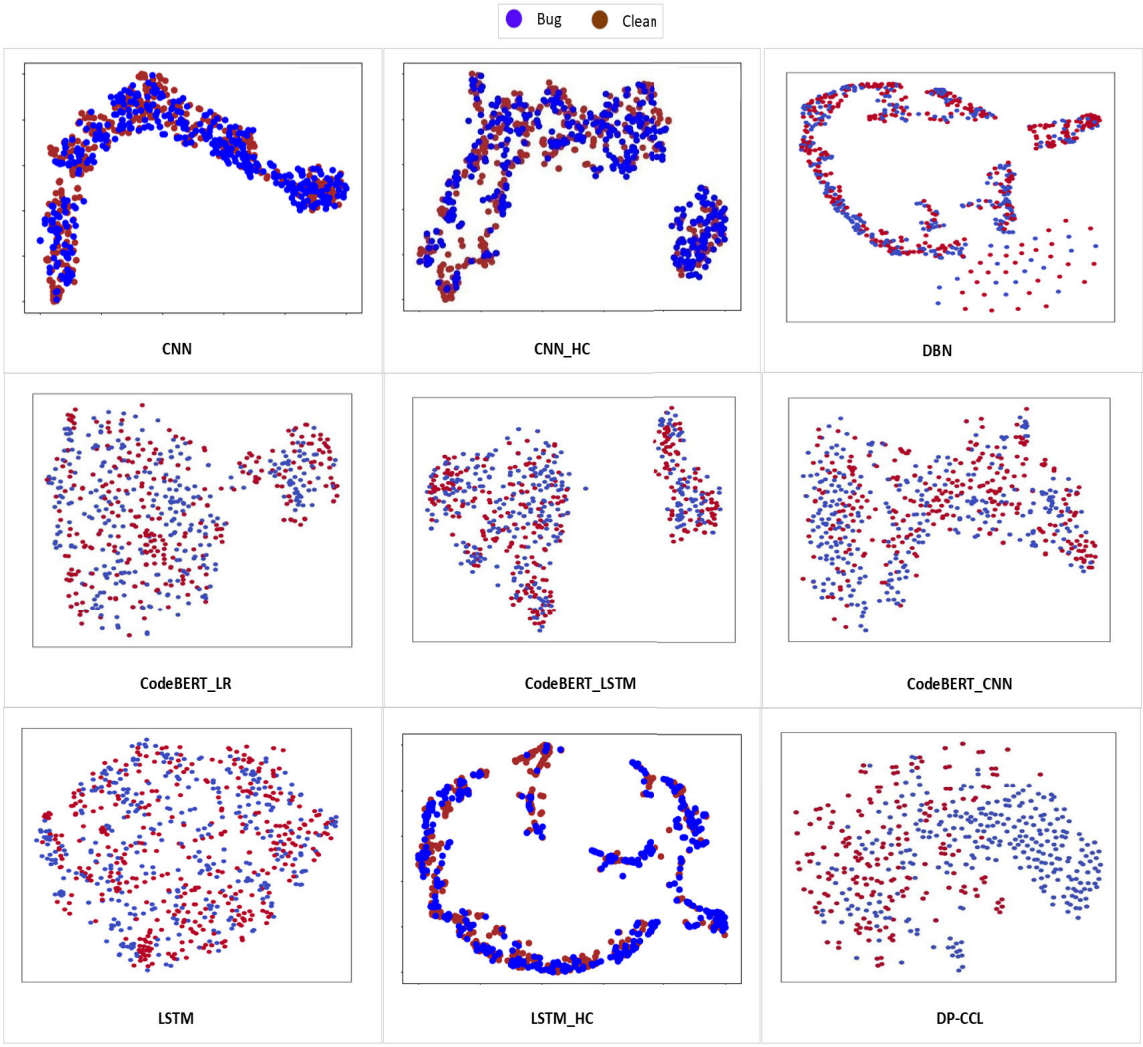

FIGURE 7. Tsne visualization of DP-CCL and exisiting apporaches.

3) RQ3. HOW DO DIFFERENT VARIANTS OF THE BERT MODEL BEHAVE IN THE CONTEXT OF EVALUATION MEASURES?

The study performed a comparative evaluation of various BERT model variants to gauge their respective performance. Table 7 illustrates the F-Score comparison among the BERT-case model, Roberta model, and CodeBERT model.

The outcome demonstrated that across all scenarios, CodeBERT surpassed both BERT-Case and Roberta encoders, achieving the highest classification performance. CodeBERT exhibited an average improvement of 4%, and in comparison to Roberta, it showed a 2% improvement.

TABLE 7. F1-Score of different variants of BERT.

Train Project	Test Project	DP-CCL	Bert-SCL	Roberta-SCL
ant vers.1.5	ant vers.1.6	0.633	0.512	0.580
ant vers. 1.6	ant vers. 1.7	0.588	0.373	0.556
camel vers. 1.2	camel vers. 1.4	0.494	0.520	0.535
camel vers. 1.4	camel vers. 1.6	0.507	0.529	0.517
jedit vers. 3.2	jedit vers. 4.0	0.632	0.489	0.544
jedit vers. 4.0	jedit vers. 4.1	0.593	0.571	0.569
log4j vers. 1.0	log4j vers. 1.1	0.730	0.531	0.707
lucene vers. 2.0	lucene vers. 2.2	0.691	0.761	0.760
lucene vers. 2.2	Lucene vers. 2.4	0.774	0.768	0.745
poi vers. 1.5	poi vers. 2.5	0.862	0.854	0.841
poi vers. 2.5	poi vers. 3.0	0.819	0.777	0.718
synapse vers. 1.1	synapse vers. 1.2	0.502	0.508	0.486
xalan vers. 2.4	xalan vers. 2.5	0.644	0.559	0.601
Xerces vers. 1.3	Xerces vers. 1.4.4	0.792	0.621	0.597
velocity vers. 1.5	velocity vers. 1.6	0.581	0.815	0.754
Average		0.656	0.613	0.634

4) Q4. WHAT IS THE IMPACT OF THE NUMBER OF HIDDEN UNITS IN THE DENSE LAYER ON THE PERFORMANCE OF THE PROPOSED MODEL?

Within this segment, the primary focus is to assess the impact of the number of hidden units of the Dense Layer on the performance of the DP-CCL model. To address this question, specific options for the hidden units within the dense layer and the lambda value of L2 regularization have been chosen. The selected hidden values are 500, 300, and 100. For the lambda value of L2 regularization, the values considered

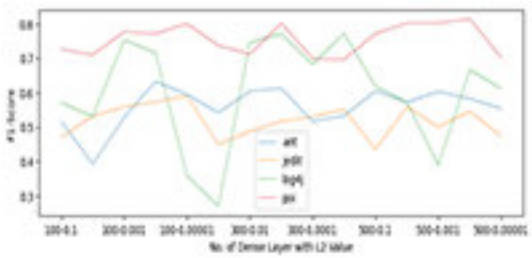

FIGURE 8. Selection of hidden nodes with L2-value in dense layer.

are 0.1, 0.01, 0.001, 0.0001, 0.00001, and 0.000001. The performance of the model on these parameters was evaluated using Ant, Log4j, Jedit, and Poi projects. Figure 8 depicts four distinct trend lines, each corresponding to one of these projects.

V. VALIDITY THREATS

The following threats may affect the validity of the DP-CCL model.

A. INTERNAL VALIDITY THREATS

1) The Abstract Syntax Tree was employed to generate the token sequences in the proposed model. Other parsing tools, such as Control Flow Graphs (CFG), can produce different sequences, potentially affecting the overall results.
2) Furthermore, the selection of parameters to fine-tune the model may not be the optimal one. There could be alternative parameter combinations that result in better model performance.
3) The CodeBERT model has a maximum token length limit of 512, while the code may have up to 2000 token features. This limit might result in the loss of crucial information and degrade the performance of the model. In the future, We plan to explore a solution to hold more relevant features to save more important information.

B. EXTERNAL VALIDITY THREATS

1) In the experiments, the study used ten different projects written in the Java language. Projects written in other programming languages or different Java projects may yield distinct results. We plan to include various language versions for the experiment in the Future.
2) The study implemented the baseline technique from scratch because the code of the original paper is not publicly available. This may loss some important information. However, we try our best to adjust the parameters of the baseline model according to the original paper.

VI. CONCLUSION AND FUTURE WORK

In this study, we designed a model DP-CCL that employs a pretrained language model to extract the semantic features of the source code. Results show that features generated by CodeBERT are more effective. We integrated supervised contrastive learning with pretrained CodeBERT and found that it generated similarity among code instances with the same labels and dissimilarity among those with different labels. CodeBERT-generated Features were combined with handcrafted software metrics to obtain the benefits of both semantic and HC features. The combined features are input to the logistic regression model for code classification as either buggy or clean. The approach was evaluated on Promise datasets. Results show that the DP-CCL model achieved significant improvement, specifically a 4.9 to 14.9% increase in accuracy compared to existing approaches in terms of F-Score. In our opinion, our model would be helpful for the industry in identifying defective code more accurately. Moreover, researchers can use contrastive learning to improve their methodology.

In the future, we will experiment with Cross-Project Defect Prediction (CPDP). Additionally, we will assess the performance of DP-CCL on projects written in different languages. Furthermore, other pretrained models, such as Unixcoder, will also be employed to evaluate the efficacy of the DP-CCL model. The supplementary material of research work is available at https://github.com/saharsadia/DP-CCL

ACKNOWLEDGMENT

The authors would like to thank the esteemed reviewer and insightful feedback, which significantly contributed to the refinement of their manuscript and also would like to thank the Chief Editor and his team for their valuable guidance throughout the review process. Their expertise and support have been instrumental in enhancing the overall quality of their article.

REFERENCES

[1] M. S. Hasan, F. Alvares, T. Ledoux, and J.-L. Pazat, "Investigating energy consumption and performance trade-off for interactive cloud application," *IEEE Trans. Sustain. Comput.*, vol. 2, no. 2, pp. 113–126, Apr. 2017.
[2] S. Qiu, H. Huang, W. Jiang, F. Zhang, and W. Zhou, "Defect prediction via tree-based encoding with hybrid granularity for software sustainability," *IEEE Trans. Sustain. Comput.*, 2023, doi: 10.1109/TSUSC.2023.3248965.
[3] D. Fang, S. Liu, and A. Liu, "Gated homogeneous fusion networks with jointed feature extraction for defect prediction," *IEEE Trans. Rel.*, vol. 71, no. 2, pp. 512–526, Jun. 2022, doi: 10.1109/TR.2022.3165115.
[4] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 2086–2104, Jun. 2022, doi: 10.1109/TSE.2021.3051492.
[5] S. Qiu, L. Lu, S. Jiang, and Y. Guo, "An investigation of imbalanced ensemble learning methods for cross-project defect prediction," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 33, no. 12, Nov. 2019, Art. no. 1959037.
[6] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–37, Jul. 2021.
[7] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020, doi: 10.1109/JPROC.2020.2993293.
[8] T. Lee, J. Nam, D. Han, S. Kim, and H. Peter In, "Developer micro interaction metrics for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 11, pp. 1015–1035, Nov. 2016, doi: 10.1109/TSE.2016.2550458.
[9] L. Šikic, P. Afric, A. S. Kurdija, and M. ŠIlic, "Improving software defect prediction by aggregated change metrics," *IEEE Access*, vol. 9, pp. 19391–19411, 2021, doi: 10.1109/ACCESS.2021.3054948.

[10] M. Wen, R. Wu, and S.-C. Cheung, "How well do change sequences predict defects? Sequence learning from software changes," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1155–1175, Nov. 2020, doi: 10.1109/TSE.2018.2876256.

[11] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, Vancouver, BC, Canada, May 2009, pp. 78–88, doi: 10.1109/ICSE.2009.5070510.

[12] H. Wang, W. Zhuang, and X. Zhang, "Software defect prediction based on gated hierarchical LSTMs," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 711–727, Jun. 2021, doi: 10.1109/TR.2020.3047396.

[13] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020, doi: 10.1109/TSE.2018.2877612.

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

[15] M. Caulo and G. Scanniello, "A taxonomy of metrics for software fault prediction," in *Proc. 46th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Portoroz, Slovenia, Aug. 2020, pp. 429–436, doi: 10.1109/SEAA51224.2020.00075.

[16] M. H. Halstead, *Elements Of Software Science*. Amsterdam, The Netherlands: Elsevier, 1977.

[17] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[18] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[19] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996, doi: 10.1109/32.544352.

[20] 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, Mar. 1994, doi: 10.1109/32.295895.

[21] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proc. Int. Conf. Softw. Maintenance*, Bethesda, MD, USA, 1998, pp. 24–31, doi: 10.1109/icsm.1998.738486.

[22] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Sep. 2011, Art. no. 311321.

[23] F. Yang, H. Xu, P. Xiao, F. Zhong, and G. Zeng, "A method-level defect prediction approach based on structural features of method-calling network," *IEEE Access*, vol. 11, pp. 7933–7946, 2023, doi: 10.1109/ACCESS.2023.3239266.

[24] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, Montreal, QC, Canada, May 2019, pp. 34–45, doi: 10.1109/MSR.2019.00016.

[25] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Singapore, Nov. 2018, pp. 1049–1054, doi: 10.1109/ICDM.2018.00133.

[26] J. Lin and L. Lu, "Semantic feature learning via dual sequences for defect prediction," *IEEE Access*, vol. 9, pp. 13112–13124, 2021, doi: 10.1109/ACCESS.2021.3051957.

[27] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 613–625, Jun. 2021, doi: 10.1109/TR.2020.3040191.

[28] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Deep semantic feature learning with embedded static metrics for software defect prediction," in *Proc. 26th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Putrajaya, Malaysia, Dec. 2019, pp. 244–251, doi: 10.1109/APSEC48747.2019.00041.

[29] 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)*, Jul. 2017, pp. 318–328.

[30] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via LSTM," *IET Softw.*, vol. 14, no. 4, pp. 443–450, Aug. 2020.

[31] Q. P. Hu, M. Xie, S. H. Ng, and G. Levitin, "Robust recurrent neural network modeling for software fault detection and correction prediction," *Rel. Eng. Syst. Saf.*, vol. 92, no. 3, pp. 332–340, Mar. 2007.

[32] G. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics*, 2020, pp. 1536–1547.

[34] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," 2021, *arXiv:2104.08821*.

[35] V. Suresh and D. C. Ong, "Not all negatives are equal: Label-aware contrastive loss for fine-grained text classification," 2021, *arXiv:2109.05427*.

[36] S. Mai, Y. Zeng, and H. Hu, "Learning from the global view: Supervised contrastive learning of multimodal representation," *Inf. Fusion*, vol. 100, Dec. 2023, Art. no. 101920.

[37] S. Mai, Y. Zeng, S. Zheng, and H. Hu, "Hybrid contrastive learning of tri-modal representation for multimodal sentiment analysis," *IEEE Trans. Affect. Comput.*, vol. 14, no. 3, pp. 2276–2289, Jul. 2022, doi: 10.1109/TAFFC.2022.3172360.

[38] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, Pittsburgh, PA, USA, May 2022, pp. 413–424, doi: 10.1145/3524610.3527911.

[39] X. Wang, Q. Wu, H. Zhang, C. Lyu, X. Jiang, Z. Zheng, L. Lyu, and S. Hu, "HELoC: Hierarchical contrastive learning of source code representation," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, Pittsburgh, PA, USA, May 2022, pp. 354–365, doi: 10.1145/3524610.3527896.

[40] M. A. Fokam and R. Ajoodha, "Influence of contrastive learning on source code plagiarism detection through recursive neural networks," in *Proc. 3rd Int. Multidisciplinary Inf. Technol. Eng. Conf. (IMITEC)*, Windhoek, Namibia, Nov. 2021, pp. 1–6, doi: 10.1109/IMITEC52926.2021.9714688.

[41] Z. Liu, C. Wen, Z. Su, S. Liu, J. Sun, W. Kong, and Z. Yang, "Emotion-semantic-aware dual contrastive learning for epistemic emotion identification of learner-generated reviews in MOOCs," *IEEE Trans. Neural Netw. Learn. Syst.*, 2023.

[42] Q. Meng, Y. Song, J. Mu, Y. Lv, J. Yang, L. Xu, J. Zhao, J. Ma, W. Yao, R. Wang, M. Xiao, and Q. Meng, "Electric power audit text classification with multi-grained pre-trained language model," *IEEE Access*, vol. 11, pp. 13510–13518, 2023, doi: 10.1109/ACCESS.2023.3240162.

[43] R. Sharma, F. Chen, F. Fard, and D. Lo, "An exploratory study on code attention in BERT," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, Pittsburgh, PA, USA, May 2022, pp. 437–448, doi: 10.1145/3524610.3527921.

[44] S. Yu, J. Su, and D. Luo, "Improving BERT-based text classification with auxiliary sentence and domain knowledge," *IEEE Access*, vol. 7, pp. 176600–176612, 2019, doi: 10.1109/ACCESS.2019.2953990.

[45] S. Arshad, S. Abid, and S. Shamail, "CodeBERT for code clone detection: A replication study," in *Proc. IEEE 16th Int. Workshop Softw. Clones (IWSC)*, Limassol, Cyprus, Oct. 2022, pp. 39–45, doi: 10.1109/IWSC55060.2022.00015.

[46] A. A. Ishtiaq, M. Hasan, M. M. A. Haque, K. S. Mehrab, T. Muttaqueen, T. Hasan, A. Iqbal, and R. Shahriyar, "BERT2Code: Can pretrained language models be leveraged for code search?" 2021, *arXiv:2104.08017*.

[47] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1804–1822, Apr. 2023, doi: 10.1109/TSE.2022.3192755.

[48] R. Wang, H. Zhang, G. Lu, L. Lyu, and C. Lyu, "FRET: Functional reinforced transformer with BERT for code summarization," *IEEE Access*, vol. 8, pp. 135591–135604, 2020, doi: 10.1109/ACCESS.2020.3011744.

[49] X. Sun, Z. Ye, L. Bo, X. Wu, Y. Wei, T. Zhang, and B. Li, "Automatic software vulnerability assessment by extracting vulnerability elements," *J. Syst. Softw.*, vol. 204, Oct. 2023, Art. no. 111790.

[50] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Nepal, Dec. 2022, pp. 481–496.

[51] J. Liu, J. Ai, M. Lu, J. Wang, and H. Shi, "Semantic feature learning for software defect prediction from source code and external knowledge," *J. Syst. Softw.*, vol. 204, Oct. 2023, Art. no. 111753.

[52] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, "Software defect prediction employing BiLSTM and BERT-based semantic feature," *Soft Comput.*, vol. 26, no. 16, pp. 7877–7891, Aug. 2022.

[53] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using CodeBERT model," *Appl. Sci.*, vol. 11, no. 11, p. 4793, May 2021.

[54] Q. Chen, R. Zhang, Y. Zheng, and Y. Mao, "Dual contrastive learning: Text classification via label-aware data augmentation," 2022, *arXiv:2201.08702*.
[55] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 1597–1607.
[56] T. Pan, Y. Song, T. Yang, W. Jiang, and W. Liu, "VideoMoCo: Contrastive video representation learning with temporally adversarial examples," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2021, pp. 11205–11214.
[57] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 519–531.
[58] H. Fang, S. Wang, M. Zhou, J. Ding, and P. Xie, "CERT: Contrastive self-supervised learning for language understanding," 2020, *arXiv:2005.12766*.
[59] Y. Meng, C. Xiong, P. Bajaj, P. Bennett, J. Han, and X. Song, "COCO-LM: Correcting and contrasting text sequences for language model pertaining," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 23102–23114.
[60] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021.
[61] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Sci. Program.*, vol. 2019, pp. 1–14, Apr. 2019.
[62] R. B. Bahaweres, D. Jumral, I. Hermadi, A. I. Suroso, and Y. Arkeman, "Hybrid software defect prediction based on lstm (long short term memory) and word embedding," in *Proc. 2nd Int. Conf. Smart Cities, Autom. Intell. Comput. Syst. (ICON-SONICS)*, Tangerang, Indonesia, Oct. 2021, pp. 70–75, doi: 10.1109/ICON-SONICS53103.2021.9617182.
[63] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, Montreal, QC, Canada, May 2019, pp. 46–57, doi: 10.1109/MSR.2019.00017.
[64] X. Zhou and L. Lu, "Defect prediction via LSTM based on sequence and tree structure," in *Proc. IEEE 20th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Macau, Dec. 2020, pp. 366–373, doi: 10.1109/QRS51102.2020.00055.
[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[66] K. Shi, Y. Lu, J. Chang, and Z. Wei, "PathPair2Vec: An AST path pair-based code representation method for defect prediction," *J. Comput. Lang.*, vol. 59, Aug. 2020, Art. no. 100979.
[67] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, vol. 2091, 2016, p. 2100.
[68] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," in *Proc. Int. Conf. Learn. Represent.*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX
[69] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, May 2018, pp. 200–210, doi: 10.1145/3196321.3196334.
[70] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 783–794.
[71] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Secur. Commun. Netw.*, vol. 2022, pp. 1–12, Jan. 2022, doi: 10.1155/2022/5203217.
[72] Y. Wang, Y. Dong, X. Lu, and A. Zhou, "GypSum: Learning hybrid representations for code summarization," in *Proc. IEEE/ACM 30th Int. Conf. Program Comprehension (ICPC)*, Pittsburgh, PA, USA, May 2022, pp. 12–23, doi: 10.1145/3524610.3527903.
[73] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," in *Advances in Neural Information Processing Systems*, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2020, pp. 18661–18673.
[74] C. Pornprasit and C. K. Tantithamthavorn, "DeepLineDP: Towards a deep learning approach for line-level defect prediction," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 84–98, Jan. 2023, doi: 10.1109/TSE.2022.3144348.
[75] R. M. O'Brien, "A caution regarding rules of thumb for variance inflation factors," *Qual. Quantity*, vol. 41, no. 5, pp. 673–690, Sep. 2007.
[76] J. Akosa, "Predictive accuracy: A misleading performance measure for highly imbalanced data," in *Proc. SAS Global Forum*, vol. 12, 2017, pp. 1–4.

SADIA SAHAR is currently pursuing the Ph.D. degree with Government College University Faisalabad (GCUF), Pakistan. Also, she is a Lecturer with the Computer Science Department, Government College Women's University Faisalabad (GCWUF), Pakistan. Her research interests include software engineering, software defect prediction, and natural language processing.

MUHAMMAD YOUNAS received the Ph.D. degree from the Faculty of Engineering, School of Computing, Universiti Teknologi Malaysia (UTM). He is currently an Assistant Professor with the Computer Science Department, Government College University Faisalabad, Pakistan. His research interests include software engineering, agile software development, cloud computing, and code clone detection.

MUHAMMAD MURAD KHAN received the Ph.D. degree from the Faculty of Engineering, School of Computing, UTM. He is currently an Assistant Professor with the Computer Science Department, Government College University Faisalabad, Pakistan. His research interests include recommender systems, android security, software security, and data mining.

MUHAMMAD UMER SARWAR received the Ph.D. degree from the Department of Computer Science, GCUF, Pakistan. He is currently an Assistant Professor with the Computer Science Department, Government College University Faisalabad, Pakistan. His research interests include software engineering and code clone detection.