IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 16, 2020, accepted January 7, 2021, date of publication January 18, 2021, date of current version January 25, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3051957

Semantic Feature Learning via Dual Sequences for Defect Prediction

JUNHAO LIN[1] **AND LU LU**[1,2]

[1]School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[2]Modern Industrial Technology Research Institute, South China University of Technology, Meizhou 528400, China

Corresponding author: Lu Lu (lul@scut.edu.cn)

This work was supported in part by the National Nature Science Foundation of China under Grant 61370103, in part by the Meizhou Produce and Research Fund under Grant 2019A0101019, and in part by the Guangzhou Produce and Research Fund under Grant 201902020004.

ABSTRACT Software defect prediction (SDP) can help developers reasonably allocate limited resources for locating bugs and prioritizing their testing efforts. Existing methods often serialize an Abstract Syntax Tree (AST) obtained from the program source code into a token sequence, which is then inputted into the deep learning model to learn the semantic features. However, there are different ASTs with the same token sequence, and it is impossible to distinguish the tree structure of the ASTs only by a token sequence. To solve this problem, this paper proposes a framework called Semantic Feature Learning via Dual Sequences (SFLDS), which can capture the semantic and structural information in the AST for feature generation. Specifically, based on the AST, we select the representative nodes in the AST and convert the program source code into a simplified AST (S-AST). Our method introduces two sequences to represent the semantic and structural information of the S-AST, one is the result of traversing the S-AST node in pre-order, and another is composed of parent nodes. Then each token in the dual sequences is encoded as a numerical vector via mapping and word embedding. Finally, we use a bi-directional long short-term memory (BiLSTM) based neural network to automatically generate semantic features from the dual sequences for SDP. In addition, to leverage the statistical characteristics contained in the handcrafted metrics, we also propose a framework called Defect Prediction via SFLDS (DP-SFLDS) which combines the semantic features generated from SFLDS with handcrafted metrics to perform SDP. In our empirical studies, eight open-source Java projects from the PROMISE repository are chosen as our empirical subjects. Experimental results show that our proposed approach can perform better than several state-of-the-art baseline SDP methods.

INDEX TERMS Software defect prediction, abstract syntax tree, deep learning, bi-directional long short-term memory network.

I. INTRODUCTION

With the increasing scale and complexity of software, software testing has become one of the most critical phases in the software life cycle [1], [2]. Software defect prediction (SDP) techniques have been proposed to detect defects and help the software quality assurance team allocate resources more efficiently. The idea behind SDP is to use the historical versions of the software as a data set to train a machine learning model and predict whether new instances of code regions (e.g., files, changes, and functions) contain defects. It is a challenging task of how to extract suitable features from programs to improve the learning performance.

Handcrafted software metrics such as lines of code, number of methods, and cyclomatic complexity play important roles in the development of SDP. Many existing studies, such as [3]–[6], use handcrafted software metrics to describe the features of software and take them as input to train various machine learning models. These handcrafted software metrics are manually designed by researchers (e.g., McCabe metrics [7] based on dependencies, MOOD metrics [8] built on polymorphic factors and coupling factors, Halstead metrics [9] based on operation and operand counts, and CK metrics [10] developed from function and inheritance counts). However, manually designed metrics ignore the

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

```
1  public void myFunc(Stack<Integer> s) {
2    int i = 0;
3    for(i = 0; i < 10; i++) {
4      s.push(i);
5      if(i > 5) {
6        …
7          foo();
8      }
9      s.pop();
10   }
11 }
```

File1.java

```
1  public void myFunc(Stack<Integer> s) {
2    int i = 0;
3    for(i = 0; i < 10; i++) {
4      s.pop();
5      if(i > 5) {
6        …
7          foo();
8      }
9      s.push(i);
10   }
11 }
```

File2.java

```
1  public void myFunc(Stack<Integer> s) {
2    int i = 0;
3    for(i = 0; i < 10; i++) {
4      s.push(i);
5      if(i > 5) {
6        …
7      }
8      foo();
9      s.pop();
10   }
11 }
```

File3.java

FIGURE 1. Example java files.

programs' semantic and structural information because they mainly focus on the statistical characteristics of programs and assume that these characteristics can be used to distinguish between programs. Take *File1.java* and *File2.java* in Figure 1 as examples. Both contain a `for` statement, a `push` function, and a `pop` function. The only difference between the two files is the order of `push` function and `pop` function. In *File2.java*, a `NoSuchElementException` will occur if the `pop` function is called at the beginning when the stack is empty. But both share the same handcrafted software metrics. That is to say, handcrafted software metrics cannot tell the difference between them.

The abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the AST denotes a construction occurring in the source code. The AST contains all the semantic and structural information in the program source code. To bridge the gap between the programs' semantic information and features used for defect prediction, the state-of-the-art method [11] extracts token sequences from programs' ASTs to learn semantic features by deep belief network (DBN). Li *et al.* [12] leverages convolutional neural network (CNN) to generate semantic features from ASTs for defect prediction. Their experiments show that models based on semantic features outperform those traditional handcrafted metric-based approaches. However, the original tree structure is destroyed after converting AST into a token sequence, which loses a lot of tree structural information. For example, *File1.java* and *File3.java* in Figure 1 share the same pre-order token sequence (i.e., `[for, push, if, foo, pop]`). The difference between these two files is the position of the `foo` function. We cannot distinguish these two files by the token sequence. Therefore, we argue that a token sequence cannot fully capture the AST's tree structural information.

To solve the problem, this paper proposes a framework called Semantic Feature Learning via Dual Sequences (SFLDS) to capture both the semantic and tree structural information of ASTs for semantic feature generation. Specifically, based on the AST, we select the representative nodes in the AST and convert the program source code into a simplified AST (S-AST). Throwing away unneeded AST nodes will turn the AST into a forest and degrade the performance of the defect prediction model. Thus, we splice the children of these nodes onto their parents to build S-AST and maintain the tree structure of the S-AST. Our method introduces two sequences to represent the semantic and tree structure information of S-AST. One is the result of traversing the S-AST node in pre-order, and another is composed of parent nodes. Second, to input token sequences into the model, mapping and word embedding are performed to convert the token sequences into numerical vectors. Finally, the two numerical vectors are input into a bi-directional long short-term memory (BiLSTM) based neural network to automatically generate semantic features from the S-AST. In addition, to leverage the statistical characteristics contained in the handcrafted metrics, we also propose a framework called Defect Prediction via SFLDS (DP-SFLDS) which combines the semantic features generated from SFLDS with handcrafted metrics to perform SDP.

In summary, this paper makes the following contributions:

- We propose S-AST to exclude AST nodes that do not contribute to the entire project and maintain the tree structure of AST.
- We propose to learn contextual semantic features from the pre-order and parent token sequences extracted from S-AST by a BiLSTM-based neural network.
- We combine the semantic features generated from SFLDS with handcrafted software metrics for defect prediction, taking advantage of nonlinear features and statistical characteristics.

The rest of this paper is organized as follows. Section II introduces the background of defect prediction, word embedding, and BiLSTM network. Section III describes our approach. Section IV shows the experimental setup and results. Section V discusses why do our approach work and threats to validity. Section VI presents related work. Last, we summarize this paper and discuss plans for future work in Section VII.

II. BACKGROUND

In this section, we briefly introduce defect prediction, word embedding, and BiLSTM network.

A. DEFECT PREDICTION

Defect prediction techniques use software historical data to build machine learning models and predict whether new code regions contain defects, which can help developers

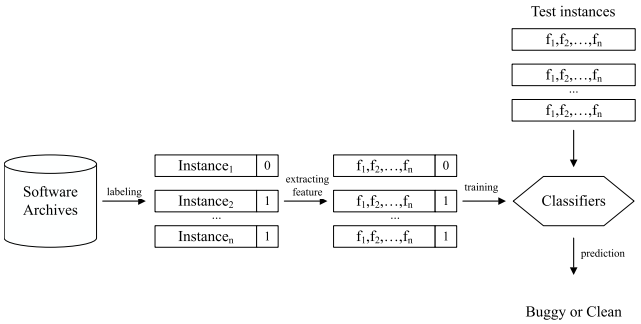


FIGURE 2. Software defect prediction process. $\{f_1, f_2, \ldots, f_n\}$ refers to the software metrics of the instance. 0 and 1 indicate whether the instance contains defect.

focus their attention on buggy code [5]. Figure 2 presents the typical defect prediction procedure, widely used in research [11]–[14]. As the process shows, the first step is to label the instance (i.e., source code files) collected from software archives as buggy or clean. This step is based on the bug tracking system (BTS). One could collect the post-release defect of each instance from the BTS by linking bug reports to its bug-fixing changes. A instance is labeled as buggy if it contains at least one bug-fixing change. Otherwise, the instance is labeled as clean. The second step is to collect the corresponding handcrafted software metrics of each instance. In past studies, the most commonly used handcrafted metrics could be categorized into code metrics (e.g., MOOD metrics and CK metrics) and process metrics (e.g., change histories). Instances with features and labels are employed to train the machine learning models such as naive bayesian (NB) [15], decision tree (DT) [16] and support vector machine (SVM) [17]. Finally, the trained model is used to predict whether the new instance contains defects.

In this work, we focused on within-project defect prediction, which means the training and test set are different versions of the same project. Moreover, instances from older versions of the project were used as a training set, and instances from newer versions were used as a test set.

B. WORD EMBEDDING

Word embedding techniques are widely used in natural language processing (NLP) [18]–[20]. In NLP, words in a sentence are processed by a machine learning model for sentiment analysis, text classification, etc. But machines cannot directly understand human language. Word embedding is proposed to map each word in a sentence to a fixed-length numerical vector so that words can be input into the machine learning model. Word mapping can be learned before or during machine learning model training. Furthermore, a NLP task can benefit from word embedding, which does not treat individual words as unique symbols. After learning the word mapping, all words in the data set have corresponding numerical vector representation, and the semantic distance between two words can be measured by various distance calculation methods. This kind of distributed representation can improve a model's performance on NLP tasks.

Each node in the S-AST is similar to the words in NLP. Specifically, we recorded the node name in the S-AST as a token and map the token onto a positive integer. A fully connected layer was used to perform word embedding to determine the semantic distance between different tokens.

C. BI-DIRECTIONAL LONG SHORT TERM MEMORY NETWORK

Long short term memory (LSTM) network is a variant of a standard recurrent neural network (RNN), which consists of an input layer, a hidden layer, and an output layer [21]. The form of an input sequence is $x = (x_1, x_2, x_3, \ldots, x_t)$. After receiving input x_t at time t, the hidden layer state of the RNN is h_t and the output value is o_t. The calculation method is shown in Equations (1) and (2):

$$h_t = f(Ux_t + Wh_{t-1} + b) \tag{1}$$
$$z_t = \sigma(Vh_t) \tag{2}$$

where U is the weight matrix of the input x, W is the weight matrix of the hidden layer state h_{t-1} at time $t-1$ as the input at time t, b is the bias matrix, $f(z)$ is the hidden layer activation function, V is the weight of the output layer, and $\sigma(z)$ is the activation function of the output layer.

Because of the gradient dissipation and explosion problems [22], the traditional RNN model is inefficient for long-sequence modeling. Thus, LSTM network was proposed to preserve long-term data dependencies by gate mechanism (e.g., the forget gate, the input gate, and the output gate). The typical LSTM cell is shown in Figure 3. The input of each gate is determined by the input of the current time step x_t and the output of the last time step h_{t-1}:

$$f_t = \sigma(W^f x_t + U^f h_{t-1} + b^f) \tag{3}$$
$$i_t = \sigma(W^i x_t + U^i h_{t-1} + b^i) \tag{4}$$
$$o_t = \sigma(W^o x_t + U^o h_{t-1} + b^o) \tag{5}$$

where f_t, i_t, and o_t are the states of the forget gate, the input gate, and the output gate, respectively; W and U are the weight matrices of the gates, and b is the bias matrix. The cell state is updated by the information transmitted by the forget gate, the input gate, and a candidate value $\widehat{c}_t$ calculated by Equations (6) and (7):

$$\widehat{c}_t = \tanh(W^c x_t + U^c h_{t-1} + b^c) \tag{6}$$
$$c_t = i_t \odot \widehat{c}_t + f_t \odot c_{t-1} \tag{7}$$

Finally, the output h_t is calculated by the cell state and the output gate:

$$h_t = o_t \odot \tanh(c_t) \tag{8}$$

In RNNs and LSTM networks, information can only be propagated forward, the result being that the state of time t only depends on the information before time t. To solve that problem, BiLSTM network uses two independent LSTM cells to process the sequence in two directions separately and combine the two output vectors from both directions.

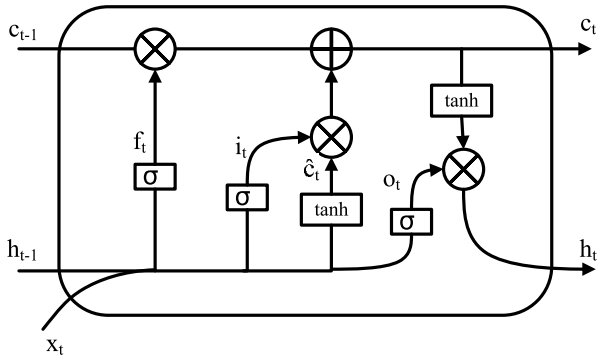


FIGURE 3. An LSTM cell.

BiLSTM network has been empirically proven effective because it makes use of the context in both directions [23], [24].

III. APPROACH

Figure 4 shows the overall framework of DP-SFLDS. Based on the AST, the representative nodes are selected to form the S-AST. Then two token sequences are extracted from S-AST. One is the result of traversing the S-AST node in pre-order, and another is composed of parent nodes. We build a mapping between integers and tokens, and employ word embedding to encode token sequences as numerical vectors which are input to subsequent BiLSTM-based neural network called dual sequences LSTM (DS-LSTM). DS-LSTM automatically generates semantic and structural features of S-AST from the dual input vectors, which are then combined with several handcrafted software metrics. Finally, the joint features are fed into a Logistic Regression (LR) classifier. After building our classifier model, we can produce a probability for each source code file, indicating whether it is buggy or clean.

Our approach consisted of four major steps: 1) extracting dual token sequences from S-AST, 2) encoding tokens into numerical vectors and handling data imbalance, 3) employing a BiLSTM-baesd neural network to generate semantic features and combining handcrafted metrics, and 4) building a Logistic Regression classifier to predict whether the source code files are buggy or clean.

A. EXTRACTING DUAL TOKEN SEQUENCES FROM S-AST

We used the open-source Python project javalang[1] to parse the source code in the software repository into AST. Following the state-of-the-art method [11], we chose only three types of nodes on the AST as tokens: 1) nodes of function invocation and class instance creation, 2) declaration nodes, and 3) control-flow nodes. The first type of node was recorded as its function name or class name (e.g., in Figure 1, functions `myFunc` and `foo` were recorded as their function names). The second type of node (e.g., method declaration, type declaration, and enumeration declaration) and the third type of node (e.g., `WhileStatement`, `IfStatement`, `ForStatement`, and `ReturnStatement`) were recorded as its node name. For example, the `ForStatement` node is simply recorded as `for`. In summary, we excluded AST nodes that do not fall into these three categories, such as intrinsic type declaration, because they are often based on specific function or class, which may not apply to the entire project. The AST nodes that we used to form S-AST are shown in Table 1.

[1]https://github.com/c2nes/javalang

TABLE 1. The selected AST nodes.

Node Category	Node Type
Nodes of function invocations and instance creations	MethodInvocation, SuperMethodInvocation, ClassCreator
Declaration-related nodes	PackageDeclaration, InterfaceDeclaration, ClassDeclaration, ConstructorDeclaration, MethodDeclaration, VariableDeclarator, VariableDeclaration, FormalParameter
Control-flow-related nodes	IfStatement, ForStatement, WhileStatement, DoStatement, AssertStatement, BreakStatement, ContiuneStatement, ReturnStatement, ThrowStatement, TryStatement, SynchronizedStatement, SwitchStatement, BlockStatement, CatchClauseParameter, TryResource, CatchClaus, SwitchStatementCase, ForControl, EnhancedForControl
Other nodes	BasicType, MemberReference, ReferenceType, SuperMemberReference, StatementExpression

Throwing away unneeded AST nodes will turn the AST into a forest and destroy the original tree structure of the AST, losing a lot of tree structural information. To preserve the tree structure of AST, we propose the S-AST in which the children of unwanted nodes are spliced to the parents of unwanted nodes. Moreover, the largest difference between the S-AST and the sentence in NLP is that the S-AST is a tree structure. The context of a sentence can be constructed from a word sequence, but the original tree structure cannot be recovered from a token sequence. For example, in Figure 1, *File1.java* and *File3.java* have the same pre-order token sequence (i.e., `[for, push, if, foo, pop]`). However, the context of the two files is different. To fully capture the semantic and structural information in the S-AST, we organized two token sequences. One is the result of traversing the S-AST node in pre-order, and another is composed of parent nodes. With these two token sequences, we were able to restore the original tree structure of the S-AST. For example, *File1.java* can be represented as `[for, push, if, foo, pop]` and `[myFunc, for, for, if, for]`.

B. ENCODING TOKENS AND HANDLING IMBALANCE

In this work, we leveraged DS-LSTM to generate the semantic features. Given that the neural network requires input as numerical vectors, we needed to convert the tokens to numerical vectors. We first collected all the tokens that appear in the S-AST and built a unified mapping dictionary between the

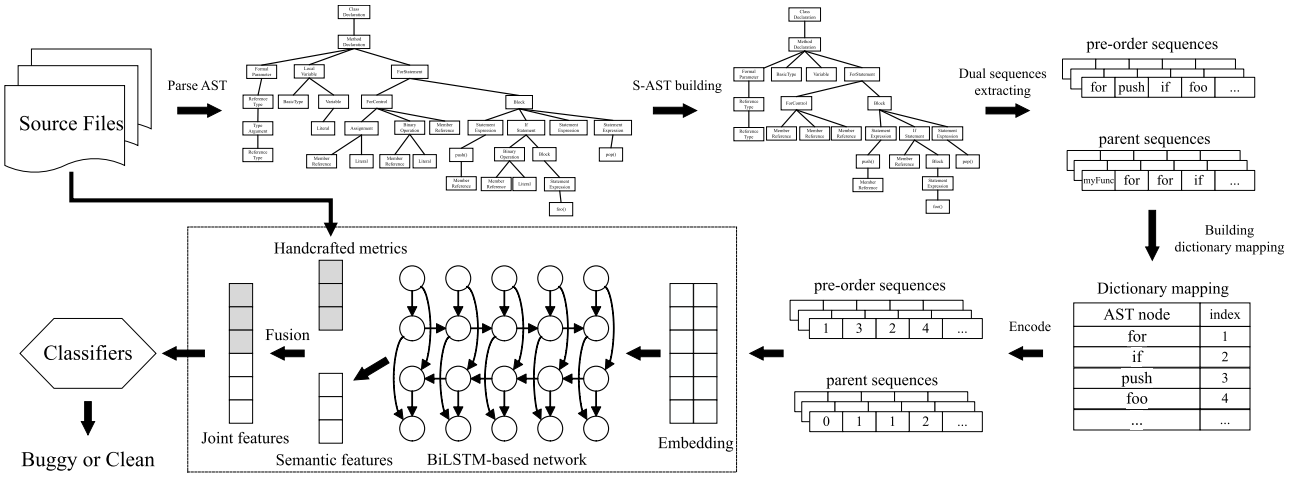


FIGURE 4. The overall framework of DP-SFLDS.

integers and the tokens. According to the mapping dictionary, tokens are mapped to integers ranging from 1 to the total number of tokens. In addition, the LSTM layer requires that the input sequences have a fixed length. However, the token sequences generated from different S-ASTs varied in length. To address this problem, we appended 0 to each sequence, making its length equal to the longest sequence. `Null` was mapped to 0 because 0 does not have any meaning. We also employ word embedding in the encoding phase. However, our word embedding is built and trained at the same time as DS-LSTM. So we wrap word embedding as a part of our model and discuss it in the following model building part.

Software defect data always comes with data imbalance problem because the number of buggy instances is much smaller than the number of clean instances. Imbalanced data will affect the model's tendency to predict instances as clean and will degrade the performance of the model [25]. To address the imbalance problem, there are two feasible approaches. One approach is to randomly duplicate instances from minority class until a balance is reached. Another approach is to remove instances from major class randomly. Because the second method will eliminate some of the information in the major class, we used the first approach to deal with the imbalance problem in the training set.

C. BUILDING BiLSTM-BASED NEURAL NETWORK AND COMBINING HANDCRAFTED METRICS

Our DS-LSTM network is illustrated in Figure 5. In particular, DS-LSTM consisted of an embedding layer (i.e., word embedding), two LSTM layers, a fully connected layer, and finally a ReLU layer as activation layer. In this step, the pre-order sequence and parent sequence are input into the DS-LSTM network to learn semantic features that preserves the semantic and tree structural information of the S-AST. We chose LSTM as a feature generation neural network because it is good at extracting semantic information from sequence input. Considering that this work engaged

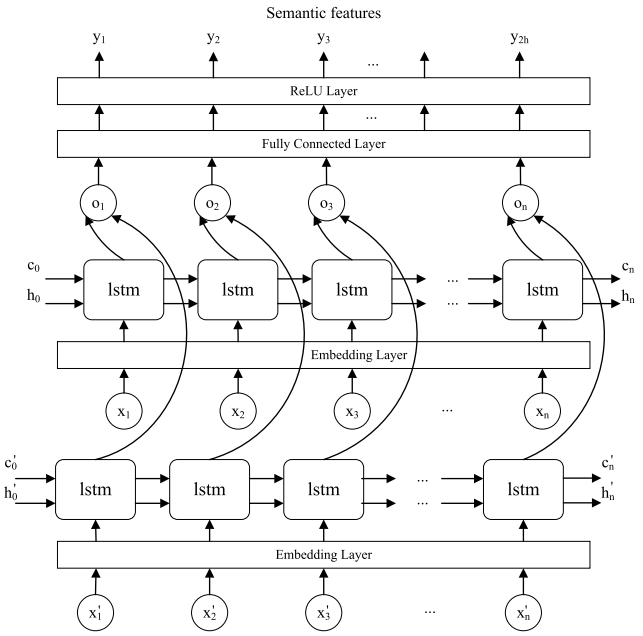


FIGURE 5. Our proposed DS-LSTM network. $\{x_1, x_2, \ldots, x_n\}$ refers to the pre-order sequence and $\{x'_1, x'_2, \ldots, x'_n\}$ refers to the parent sequence.

LSTM only as an application to generate semantic features, we adopted a standard architecture of LSTM rather than a complex architecture. Our implementation was based on PyTorch,[2] an open-source deep learning framework that provides efficient tools for neural network construction and flexible development. The sequences input into LSTM network must have a fixed length, and so we appended 0 to each sequence in Section III-B. However, padding too many 0 can degrade the performance of LSTM network. We solved this problem by wrapping the input vector sequence into PyTorch's PackedSequence, which will ignore the padded 0 in the input sequence when training the LSTM network.

[2]https://pytorch.org/

TABLE 2. Data set description.

Project Name	Description	Version	Average Files	Buggy Rate(%)
camel	An enterprise integration framework	1.2, 1.4	740	24.4
jedit	A mature programmer's text editor	4.0, 4.1	309	25.0
log4j	Log management tool for Java	1.1, 1.2	157	73.5
lucene	Text search engine library	2.0, 2.2	221	53.2
poi	The Java API for Microsoft documents	2.5.1, 3.0	413	63.9
synapse	A lightweight and high-performance Enterprise Service Bus	1.1, 1.2	239	30.5
velocity	A Java-based template engine	1.5, 1.6.1	221	49.7
xalan	A XSLT processor for transforming XML documents into other document types	2.5, 2.6	844	47.3

Word embedding is defined as $f : M \rightarrow \mathbb{R}^n$, where f is a function of mapping tokens to n-dimensional vectors and M represents the set of tokens. The parameter of embedding layer is initialized randomly and learned with other parameters in the DS-LSTM network. Thereafter, the LSTM layer gets a 2-D vector $x \in \mathbb{R}^{l \times n}$ as input, where l and n denote the length of input sequences and the embedding dimension, respectively. As discussed in Section II-C, BiLSTM network constructs the context of a sentence through a forward sequence and a reverse sequence, which has been proven effective. Inspired by the BiLSTM network, we constructed the context of an S-AST by a pre-order sequence and a parent sequence. In this way, we could further mine the semantic and tree structural information of the S-AST. Then, the two output vectors from the two LSTM layers were concatenated into a 2-D vector $o \in \mathbb{R}^{l \times 2h}$, where h was the hidden size of the LSTM layer. The concatenated vector was input into a fully connected layer and a ReLU layer to learn the semantic features. Moreover, to leverage the statistical information of handcrafted metrics, the semantic features were combined with handcrafted metrics to form the joint features. Finally, the joint features were input into the logistic regression classifier. To demonstrate the effectiveness of combining handcrafted metrics, we design a framework called SFLDS which directly feeds the semantic features to final classifier without concatenation. In the experiments part, we will compare this framework with DP-SFLDS, as well as other state-of-the-art-methods.

D. PREDICTING DEFECTS

Logistic regression classifier is employed as the final classifier because it is widely used in studies [11], [12] and we mainly focus on feature generation in this paper. Our model is trained using the mini-batch stochastic gradient descent (SGD) algorithm [26] with the Adam optimizer [27]. Further, we use the cross-entropy cost as the loss function and set the number of training epochs to 100. We will discuss the details of parameter tuning, such as embedding dimension and hidden dimension, in Section IV. After we train our model using the training files with their corresponding labels, both the weights and the biases in our DS-LSTM network and logistic regression classifier are fixed. Then each file in the test set is fed into our defect prediction model and the final classifier will predict the probability of the instance being defective.

IV. EVALUATION

In this section, we compare the performance of DP-SFLDS with the state-of-the-art methods to verify the validity of DP-SFLDS. In particular, we focus on the following questions:

- RQ1: Does our proposed SFLDS perform better than the state-of-the-art methods only based on handcrafted metrics or semantic features?
- RQ2: How is the performance of DP-SFLDS, which combines handcrafted metrics?
- RQ3: How is the performance of SFLDS under different parameter settings?

All our experiments were run on a Linux server with one GTX-2070s GPU, one Intel E3 1230-v3 CPU, and 16GB memory.

A. DATA SET

To evaluate the performance of DP-SFLDS, we selected 8 Java open-source projects from PROMISE Repository[3] as our evaluation data set, which has been widely used in SDP studies [11]–[13]. Table 2 shows the details of these projects, including project name, description, version for model training and testing, the average number of files, and the average buggy rate. In this repository, we are provided with 20 handcrafted software metrics for each project file, which are listed in Table 3, and a label indicating whether the file is defective or not. With the project names and concrete version numbers, we collected the source files of each project from Github.[4] On average, the number of files in each project was 393, and the buggy rate of each project was 45.9%.

B. PERFORMANCE MEASURES

Table 4 shows the confusion matrix, in which the values stored are used to figure out widely used evaluation measures in binary classification tasks. For better validating the performance of the proposed approach, and following the recommendations in [25], Recall, F-measure, Area Under

[3]http://openscience.us/repo/defect/
[4]https://github.com/

TABLE 3. Description of the 20 handcrafted software metric in PROMISE data set.

Metric Symbol	Description
WMC	The number of methods in a given class.
DIT	The maximum distance from a given class to the root of an inheritance tree.
NOC	The number of immediate descendants of a given class.
CBO	The number of classes coupled to a given class.
RFC	The number of distinct methods invoked by code in a given class.
LCOM	The number of method pairs in a given class that do not share access to any class attributes.
CA	Afferent couplings, measuring the number of classes that depend on a given class.
CE	Efferent couplings, measuring the number of classes on which a given class depends.
NPM	The number of public methods in a given class.
LCOM3	Another type of LCOM metric proposed by Henderson–Sellers.
LOC	The number of lines of code in a given class.
DAM	The ratio of the number of private or protected attributes to the total number of attributes in a given class.
MOA	The number of attributes belonging to user-defined types in a given class.
MFA	The number of methods a given class inherits divided by the total number of methods that can be accessed by member methods of a given class.
CAM	The ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class.
IC	The number of parent classes to which a given is coupled.
CBM	The total number of new methods or override methods to which all inherited methods in a given class are coupled.
AMC	The average size of a method in a given class.
MAX_CC	The maximun McCabe's cyclomatic complexity score of a method in a given class.
AVG_CC	The arithmetic average of the McCabe's cyclomatic complexity scores of methods in a given class.

TABLE 4. Confusion matrix.

	Truly Positive	Truly Negative
Predictive Positive	TP	FP
Predictive Negative	FN	TN

Curve (AUC), and Matthews Correlation Coefficient (MCC) are adopted as performance measures in our experiments. We repeated each experiment ten times and took the average of the performance measure as a result.

In SDP, developers pay more attention to finding all defect instances than to predicting accuracy. Recall (R) refers to the proportion of the number of instances correctly predicted among all the results that are truly positive. Recall is calculated by Equation (9):

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

F-measure is a composite measure of precision and recall, ranging from 0 to 1. The higher the F-measure, the better the prediction model's performance. The F-measure can be calculated by Equation (11):

$$P = \frac{TP}{TP + FP} \tag{10}$$

$$F\text{-}measure = \frac{2PR}{P + R} \tag{11}$$

AUC that can measure the discrimination power of constructed models is defined as the area under the receiver operating characteristic curve (ROC). ROC is a curve of the false positive rate (FPR) against the true positive rate (TPR). The FPR and TPR can be calculated by Equation (12) and (13):

$$FPR = \frac{FP}{FP + TN} \tag{12}$$

$$TPR = \frac{TP}{TP + FN} \tag{13}$$

MCC, a correlation coefficient between the actual classification and predicted classification, has comprehensive consideration of various indicators. Its value ranges from 0 to 1, higher values mean better results. MCC is calculated by Equation (14):

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{14}$$

C. BASELINES

We compared DP-SFLDS with the following state-of-the-art approaches:

- Methods only using handcrafted software metrics in Table 3.
  - **LR**: A traditional method for SDP.
  - **SVM**: Support vector machine, a classification method commonly used in machine learning.
  - **DT**: Decision Tree, a popular method in machine learning.
  - **RF**: Random forest, an ensemble learning method for classification.
- Methods using features generated from deep learning models. We adopted the same parameter setting as in [11], [12]. For fairness, we used a consistent LR implementation as a final classification.
  - **DBN**: The state-of-the-art method [11], which employs a standard DBN model to extract semantic features from AST.
  - **CNN**: The state-of-the-art method [12], which captures semantic information through standard CNN based on sequence convolution.
  - **LSTM**: An SDP method that leverages standard LSTM network to generate semantic features from AST.
  - **DP-DBN**: An enhanced DBN by combining generated features and handcrafted metrics.

- **DP-CNN**: An variant of CNN that combines the generated features and handcrafted metrics.
- **DP-LSTM**: An variant of LSTM.

D. PERFORMANCE OF SFLDS (RQ1)

We first compare the handcrafted metric-based method with semantic feature-based methods which have not combined with handcrafted metrics (i.e., DBN, CNN, LSTM, and SFLDS). We conducted eight sets of experiments on those projects listed in Table 2 for each method. To clearly show the difference in performance between different methods, we used the Scott-Knott ESD test to compare the performance of different SDP methods. The Scott-Knott ESD test divides performance measure results into statistically different groups through hierarchical clustering analysis. A detailed introduction to the Scott-Knott ESD test is available [28].

Figure 6 shows the corresponding Scott-Knott ESD test results. The methods in the experiment could be generally divided into two categories: deep learning methods (i.e., SFLDS, CNN, and LSTM) and the other methods. The deep learning methods are superior to the other methods in the average performance measures. Particularly, the average Recall of SFLDS is 0.65, which outperforms LSTM by 3.83%. The average F-measure of SFLDS is 0.593, which outperforms CNN by 1.72%. The average AUC of SFLDS is 0.667, which outperforms CNN by 0.90%. The average MCC of SFLDS is 0.353, which outperforms CNN by 3.52%. The following two points can be drawn from the above experiment results:

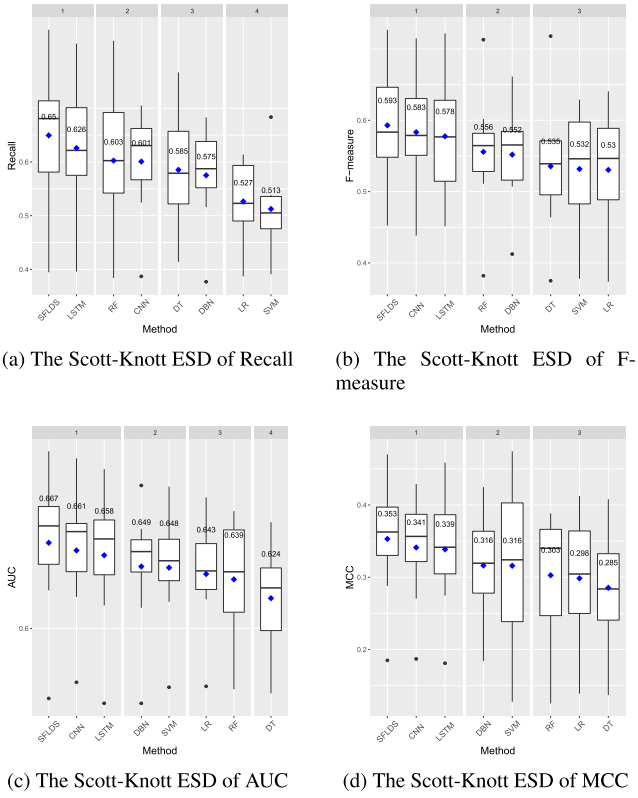
1) Deep learning methods could learn the semantic features from the S-AST token sequence and improve the performance of the SDP.

2) The SFLDS, which considers the semantic and tree structural information of S-AST, could perform better than the other SDP methods only based on semantic features or handcrafted metrics.

E. PERFORMANCE OF DP-SFLDS (RQ2)

In this part, we combined the semantic features generated from SFLDS with handcrafted software metrics to form the joint features. By taking advantage of nonlinear features and statistical characteristics, we can expect to achieve a better performance of SDP. For a fair comparison, we also combined the semantic features generated from DBN, CNN, and LSTM with handcrafted metrics, called DP-DBN, DP-CNN, and DP-LSTM. As before, we conducted eight sets of experiments for each method.

Table 5, 6, 7, and 8 respectively presents the Recall, F-measure, AUC, and MCC results for each method. By comprehensive observation, DP-SFLDS performed better than other methods in most projects, except for the project *log4j*, where DP-CNN outperforms DP-SFLDS in terms of the four performance measures. The reason may be that the training set of *log4j* was relatively small and the buggy rate was fairly high, could make DP-SFLDS overfit and



(a) The Scott-Knott ESD of Recall

(b) The Scott-Knott ESD of F-measure

(c) The Scott-Knott ESD of AUC

(d) The Scott-Knott ESD of MCC

FIGURE 6. The Scott-Knott ESD ranking of different SDP methods. The blue diamond indicates the average performance measure of the corresponding method.

degrade the performance. However, by introducing handcrafted metrics, DP-SFLDS can achieve better performance than SFLDS. More specifically, on the average value, our proposed DP-SFLDS achieves Recall as 0.679, F-measure as 0.612, AUC as 0.684, and MCC as 0.372 across eight projects and obtains the best average value on the four performance measures as compared to the other baseline methods. Compared with the best performance measures among the baseline methods, DP-SFLDS achieves improvements of, 2.87%, 2.00%, 1.03%, and 2.19% in terms of Recall, F-measure, AUC, and MCC, respectively. In summary, by combining handcrafted metrics with semantic features, we can achieve better performance than by using only semantic features or handcrafted metrics.

F. PARAMETER TUNING (RQ3)

Two hyper-parameters significantly affect the performance of SFLDS: 1) the embedding dimension of tokens, and 2) the hidden dimension of the LSTM layer. In this part, we discuss how to adjust these two parameters to achieve the best SFLDS performance. Based on F-measure, we tuned these two hyperparameters by conducting experiments on project *jedit*, *lucene*, *xalan*, and *velocity* with seven values (i.e., 10, 20, 30, 50, 70, 90, and 120). Given that other training parameters do not directly affect the performance of SFLDS, we set these parameters to a fixed value. More specifically, the learning rate, the step size, the decay rate, the batch size,

TABLE 5. Comparison results of different methods in terms of Recall. The best performance results are shown in bold and W/T/L represents the number of wins, ties, and losses of DP-SFLDS compared with other methods.

Project name	LR	SVM	DT	RF	DBN	CNN	LSTM	SFLDS	DP-DBN	DP-CNN	DP-LSTM	DP-SFLDS
camel	0.614	0.517	0.641	0.606	0.683	0.696	0.701	0.703	0.699	0.710	0.706	**0.710**
jedit	0.595	0.684	0.600	0.689	0.633	0.643	0.620	0.658	0.738	0.734	0.713	**0.747**
log4j	0.387	0.391	0.414	0.384	0.377	0.387	0.396	0.394	0.419	**0.457**	0.440	0.449
lucene	0.486	0.493	0.453	0.438	0.573	0.524	0.566	0.580	**0.601**	0.559	0.580	0.587
poi	0.495	0.480	0.705	0.702	0.601	0.651	0.701	0.711	0.619	0.637	0.708	**0.722**
synapse	0.593	0.535	0.558	0.577	0.564	0.581	0.578	0.581	0.606	0.614	0.625	**0.627**
velocity	0.551	0.461	0.767	0.826	0.654	0.705	0.820	0.846	0.679	0.769	0.833	**0.858**
xalan	0.491	0.538	0.545	0.599	0.516	0.618	0.623	0.720	0.569	0.620	0.672	**0.729**
DP-SFLDS: W/T/L	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**7**/0/1	**6**/1/1	**8**/0/0	-
Average	0.527	0.513	0.585	0.602	0.575	0.600	0.625	0.649	0.616	0.638	0.660	**0.679**

TABLE 6. Comparison results of different methods in terms of F-measure. The best performance results are shown in bold and W/T/L represents the number of wins, ties, and losses of DP-SFLDS compared with other methods.

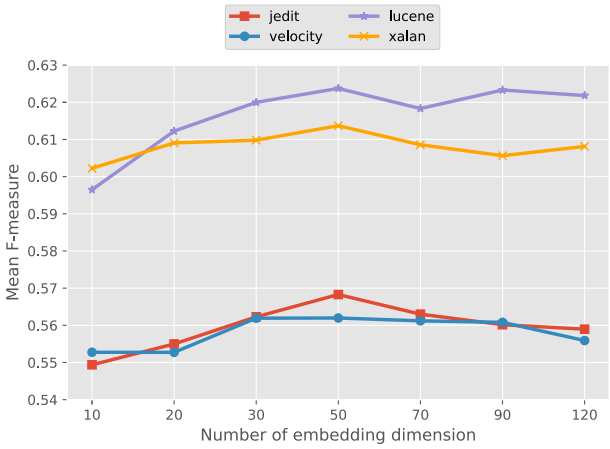
Project name	LR	SVM	DT	RF	DBN	CNN	LSTM	SFLDS	DP-DBN	DP-CNN	DP-LSTM	DP-SFLDS
camel	0.374	0.378	0.375	0.382	0.412	0.438	0.451	0.452	0.420	0.443	0.457	**0.467**
jedit	0.568	0.579	0.571	0.575	0.576	0.580	0.578	0.583	0.585	0.597	0.593	**0.601**
log4j	0.452	0.473	0.464	0.511	0.556	0.577	0.507	0.548	0.586	**0.595**	0.552	0.570
lucene	0.598	0.599	0.572	0.560	0.575	0.647	0.648	0.659	0.613	0.640	0.656	**0.664**
poi	0.641	0.629	0.718	0.713	0.661	0.715	0.722	0.727	0.692	0.723	0.725	**0.730**
synapse	0.501	0.513	0.524	0.534	0.507	0.524	0.517	0.548	0.529	0.561	0.554	**0.569**
velocity	0.524	0.486	0.554	0.569	0.519	0.560	0.576	0.583	0.578	0.591	0.592	**0.607**
xalan	0.585	0.597	0.506	0.602	0.607	0.625	0.621	0.642	0.625	0.649	0.650	**0.686**
DP-SFLDS: W/T/L	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**7**/0/1	**8**/0/0	**8**/0/0	**7**/0/1	**7**/0/1	**8**/0/0	-
Average	0.530	0.532	0.535	0.556	0.552	0.583	0.578	0.593	0.579	0.600	0.597	**0.612**

TABLE 7. Comparison results of different methods in terms of AUC. The best performance results are shown in bold and W/T/L represents the number of wins, ties, and losses of DP-SFLDS compared with other methods.

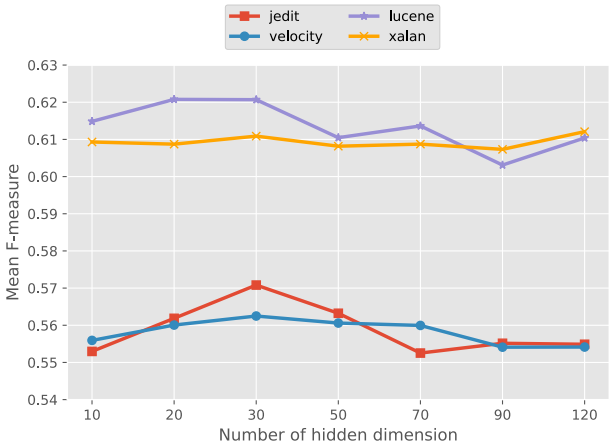
Project name	LR	SVM	DT	RF	DBN	CNN	LSTM	SFLDS	DP-DBN	DP-CNN	DP-LSTM	DP-SFLDS
camel	0.645	0.657	0.579	0.601	0.667	0.692	0.707	0.718	0.680	0.715	0.714	**0.727**
jedit	0.703	0.712	0.684	0.692	0.713	0.734	0.725	0.740	0.721	0.741	0.738	**0.744**
log4j	0.554	0.553	0.549	0.552	0.541	0.557	0.541	0.545	0.585	**0.586**	0.579	0.585
lucene	0.646	0.650	0.638	0.643	0.654	0.651	0.650	0.657	0.658	0.660	0.667	**0.673**
poi	0.673	0.680	0.670	0.681	0.678	0.680	0.678	0.684	0.682	0.711	0.703	**0.713**
synapse	0.623	0.621	0.605	0.617	0.616	0.625	0.618	0.630	0.629	0.638	0.635	**0.653**
velocity	0.633	0.643	0.625	0.646	0.665	0.673	0.670	0.678	0.677	0.684	0.681	**0.685**
xalan	0.664	0.667	0.640	0.676	0.656	0.679	0.671	0.688	0.673	0.682	0.672	**0.689**
DP-SFLDS: W/T/L	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**7**/1/0	**7**/0/1	**8**/0/0	-
Average	0.643	0.648	0.624	0.639	0.649	0.661	0.658	0.667	0.663	0.677	0.673	**0.684**

TABLE 8. Comparison results of different methods in terms of MCC. The best performance results are shown in bold and W/T/L represents the number of wins, ties, and losses of DP-SFLDS compared with other methods.

Project name	LR	SVM	DT	RF	DBN	CNN	LSTM	SFLDS	DP-DBN	DP-CNN	DP-LSTM	DP-SFLDS
camel	0.217	0.225	0.235	0.251	0.299	0.340	0.324	0.344	0.303	0.341	0.328	**0.344**
jedit	0.396	0.474	0.408	0.388	0.371	0.412	0.459	0.470	0.374	0.428	0.476	**0.491**
log4j	0.139	0.128	0.137	0.126	0.184	0.187	0.181	0.185	0.208	**0.212**	0.205	0.207
lucene	0.269	0.284	0.261	0.235	0.268	0.270	0.275	0.288	0.289	0.277	0.294	**0.308**
poi	0.412	0.408	0.374	0.362	0.425	0.429	0.421	0.424	0.428	0.432	0.432	**0.432**
synapse	0.340	0.401	0.319	0.322	0.340	0.373	0.375	0.388	0.394	0.413	0.407	**0.416**
velocity	0.261	0.243	0.242	0.358	0.281	0.339	0.315	0.348	0.325	**0.408**	0.336	0.379
xalan	0.353	0.364	0.307	0.380	0.361	0.379	0.359	0.377	0.375	0.398	0.387	**0.403**
DP-SFLDS: W/T/L	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**8**/0/0	**7**/1/0	**7**/0/1	**5**/1/2	**7**/1/0	-
Average	0.298	0.316	0.285	0.303	0.316	0.341	0.338	0.353	0.337	0.364	0.358	**0.372**



(a) The performance of SFLDS under different embedding dimensions



(b) The performance of SFLDS under different hidden dimensions

FIGURE 7. (a), (b) show the performance of SFLDS under different hyperparameters settings.

and the training epoch were set as 0.01, 3, 0.5, 15, 100, respectively.

Figure 7 shows the average F-measure of SFLDS under different parameter settings. In Figure 7a, we set the embedding dimension at a fixed value and took the average of F-measures under different hidden dimensions as the final result. Figure 7b was drawn similarly. Figure 7a shows that most of the result curves were convex and reached their maximum value when the embedding dimension was set to 50. From the results shown in Figure 7b, considering time cost, we finally chose 30 as the hidden dimension in our experiment for better prediction performance, although *xalan* achieved the best performance in 120.

V. DISCUSSION

A. WHY SFLDS AND DP-SFLDS WORKS?

The experiment results show that our proposed approach performs better than other approaches based on handcrafted software metrics or semantic features. The probable reasons are listed as following:

1) Traditional machine learning methods based on handcrafted software metrics focus on statistical information of the source code (e.g., LOC, the number of function calls), which cannot capture the semantic information. Compared with these approaches (i.e., LR, SVM, RF, DT), SFLDS directly generates features from source code with more complex neural network connections. These neural network connections enable the SFLDS to generate features with multiple levels of abstraction and high-level semantics. Moreover, DP-SFLDS combines the semantic features generated from SFLDS with handcrafted software metrics, taking advantage of nonlinear features and statistical characteristics, which could further improve the performance of SDP.

2) Compared with the previous semantic feature-based methods (i.e., DBN, CNN, LSTM, DP-DBN, DP-CNN, DP-LSTM), SFLDS and DP-SFLDS simultaneously consider semantic and tree structural information in the S-AST. Instead of directly serializing an AST into a token sequence and removing useless nodes from the token sequence, we construct the S-AST to maintain the semantic and tree structural information of AST. First, based on the AST, the representative nodes are selected to form the S-AST while the AST nodes that do not contribute to the entire project are removed. Second, we can get the S-AST by splicing the children of removed nodes to its parent. The largest difference between the S-AST and the sentence in NLP is that the S-AST is a tree structure. Therefore, we organized two token sequences (i.e., pre-order sequence, parent sequence) where each pair of the token is time-related to construct the context of the S-AST. Last, we use BiLSTM-based network to generate semantic features which simultaneously considers semantic and tree structural information in the S-AST. Thus, we believe that SFLDS-generated features can contribute to the performance of SDP.

B. THREATS TO VALIDITY

We discuss threats to the validity of our study in this part.

- **Implementation of Compared Methods:** In our experiments, we compared our proposed approach with ten referential methods. Because the original implementations of DBN, LSTM, and CNN have not yet been publicly released, we implemented our version via PyTorch. Although we strictly followed the procedures and parameter settings of that approach, our implementation may not reflect all the details of the comparison approach. To make a fair comparison, we performed the same pre-processing process on the input of each model and employed a logistic regression layer as the final classifier.
- **Data selection:** Our approach is only validated in open-source projects. It is difficult to evaluate DP-SFLDS in closed-source project because the source code of the project is needed to generate AST. However, the structure of AST in open-source projects is the same as in closed-source projects. We believe that the process

of generating semantic features in open-source projects can be generalized to closed-source projects. Moreover, all the data in our experiment came from the PROMISE repository, so they might not be representative of all projects. In particular, the projects in the PROMISE repository were written in Java. Therefore, our proposed method might produce better or worse results in those projects based on other programming languages (e.g., Python, C++).

VI. RELATED WORK

A. SOFTWARE DEFECT PREDICTION

Software defect prediction is an active research area in Software Engineering [29]–[36]. In the literature, most defect prediction techniques focus on manually designing new discriminative features or new combined features from labeled software repository, which are fed into machine learning-based classifiers to identify code defects [29]. Commonly used metrics can be generally divided into three categories: static code metrics, network metrics, and process metrics. Static code metrics measure the complexity of source code and assume that the more complex the source code is, the more likely defects are to appear [37]. Network metrics [38], which are effective for SDP, are social network analysis (SNA) metrics calculated based on the dependency graph of a software system and quantify the topological structure of each node of the dependency graph in a certain sense. Process metrics represent development changes on software projects, such as the number of revisions, authors, past bug fixes, and ages of files [39]. Based on the above metrics, SDP tasks can be categorized into within-project defect prediction (WPDP) and cross-project defect prediction (CPDP).

In WPDP, machine learning models are trained using the old version of the program modules, after which they predict whether the new version of the program module contains defects. This problem has been well explored in the past. For example, Turhan *et al.* [40] evaluated the feasibility of Naïve Bayes (NB) classification in predicting software defects, and they compared NB with the other three statistical learning methods on eight NASA datasets. Tua et.al [31] improved the performance of the software metric-based method by adding the process of selecting features using ARM to the NB method. Their experiment results indicated that NB-ARM could perform better than the other three machine learning methods on five PROMISE datasets. Wei et.al [32] introduced the local tangent space alignment (LTSA) algorithm to support vector machine (SVM) for defect prediction. Compared with the single SVM and the LLE-SVM, Their LTSA-SVM improved the F-measure by 1-4% in 13 NASA datasets. To better cope with noise and imprecise information, Marian et.al [33] proposed to employ the fuzzy decision trees (FuzzyDT) algorithm in detecting defective entities. Their evaluation on PROMISE repository showed that their DT based approach can improve defect prediction results compared to the other machine learning-based classifiers.

Recently, to address the problem of insufficient training data for new projects, more and more papers studied the CPDP task, where the training data and test data come from different projects. Zimmermann *et al.* [41] ran 622 pairs of CPDP tasks on 12 software projects to verify whether CPDP is feasible, and they found that only 3.4% of prediction tasks could achieve acceptable performance, thus concluding that CPDP task will fail in most cases unless appropriate training data are selected. In order to find suitable source projects, Herbold *et al.* [42] proposed a training data selection (TDS) by measuring the Euclidean distance between the source and target projects. They evaluated TDS in 44 versions of 14 different open source projects from the PROMISE repository and the experiment results showed that TDS improves the success rate of CPDP significantly, though the quality of the results still cannot compete with WPDP. Nam *et al.* [43] adopted a transfer learning technique called transfer component analysis (TCA) which made feature distributions in source and target projects similar and further improved TCA as TCA+ by adding customized normalization rules to preprocess data. TCA and TCA+, however, only took marginal distribution into account. To simultaneously consider both the marginal distribution and conditional distribution, Qiu *et al.* [44] proposed joint distribution matching (JDM) to construct new feature representation that is effective and robust for substantial distribution difference for CPDP. Different from JDM, balanced distribution adaption (BDA) methods proposed by Xu *et al.* [45] not only take the marginal distribution and conditional distribution into account but also adaptively assign different weights to them.

Our approach differs from the aforementioned SDP approaches in that we utilize deep learning technique to automatically generate discriminative features from the program source code, rather than handcrafted software metrics, which can capture the semantic and tree structural information of S-AST and improve the performance of SDP.

B. DEEP LEARNING AND SOFTWARE ENGINEERING

With the rapid development of deep learning, more and more researchers are committed to combining deep learning and SDP. Yang *et al.* [46] proposed a deep belief network (DBN) algorithm to build a set of expressive features from a set of initial change features for just-in-time defect prediction. Their experiment result showed that on average across the 6 projects, their approach could discover 32.22% more bugs than the traditional approach. To mine source-level semantic and syntactic features, Phan *et al.* [47] proposed to leverage precise graphs representing program execution flows, and convolutional neural networks (CNN) for automatically learning defect features. In addition, Wang *et al.* [11] proposed the state-of-the-art approach that leveraged DBN to automatically learn the semantic features of programs from abstract syntax trees (ASTs). Based on Wang *et al.*'s approach, Li *et al.* [12] leveraged the token sequence extracted from the AST and CNN to perform defect

prediction. Li *et al.* [48] employed a BiLSTM model to generate semantic features from programs' ASTs and perform defect prediction. The experiment results of the aforementioned deep learning methods showed that deep learning-based method could perform better than traditional methods in defect prediction.

The differences between our work and previous work are as follows. First, we construct S-AST to exclude AST nodes that do not contribute to the entire project and maintain the tree structure of AST rather than directly delete the unneeded nodes from the token sequence. Second, we construct the context of AST by pre-order and parent sequences extracted from S-AST and a BiLSTM-based model, compared to learn semantic features from a token sequence.

There are also other studies that leverage deep learning techniques to solve other problems in software engineering. White *et al.* [49] used deep learning to link the patterns mined at the lexical level with patterns mined at the syntactic level for code clone detection. Gu *et al.* [50] proposed an RNN encoder-decoder model to generate API usage sequences for a given natural language query. Gupta *et al.* [51] introduced a multilayered sequence-to-sequence neural network model with attention to predicting erroneous C program locations along with the required correct statements.

VII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel SDP approach to generate semantic features that simultaneously considered semantic and tree structural information in the AST. We also combined the generated semantic features with handcrafted software metrics to form the joint features, taking advantage of nonlinear features and statistical characteristics. The experimental results showed that our approach outperformed the referential methods on eight open-source projects from PROMISE repository.

In the future, to make our approach more generalized, we will continue to conduct experiments in a variety of programming languages (e.g., Python, C++) and projects to further improve our approach.

REFERENCES

[1] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. 2nd Int. Conf. Softw. Eng.*, Piscataway, NJ, USA: IEEE Computer Society Press, 1976, pp. 592–605.

[2] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Reading, MA, USA: Addison-Wesley, 2002.

[3] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software defect prediction with static code metrics," in *Proc. Int. Conf. Eng. Appl. Neural Netw.* Berlin, Germany: Springer, 2009, pp. 223–234. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-03969-0_21

[4] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug. 2015, pp. 496–507.

[5] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Softw. Eng.*, vol. 17, no. 4, pp. 375–407, Dec. 2010.

[6] J. Nam and S. Kim, "CLAMI: Defect prediction on unlabeled datasets (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 452–463.

[7] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[8] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[9] M. Halsted, *Elements of Software Science* (Operating and Programming Systems Series). New York, NY, USA: Elsevier, 1977. [Online]. Available: https://dl.acm.org/doi/book/10.5555/540137

[10] 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.

[11] 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.

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

[13] H. Khanh Dam, T. Pham, S. Wee Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*. [Online]. Available: http://arxiv.org/abs/1802.00921

[14] 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.

[15] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A Bayesian belief network for assessing the likelihood of fault content," in *Proc. 14th Int. Symp. Softw. Rel. Eng., ISSRE*, Nov. 2003, pp. 215–226.

[16] J. Wang, B. Shen, and Y. Chen, "Compressed C4.5 models for software defect prediction," in *Proc. 12th Int. Conf. Qual. Softw.*, Aug. 2012, pp. 13–16.

[17] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.

[18] O. Levy and Y. Goldberg, "Linguistic regularities in sparse and explicit word representations," in *Proc. 18th Conf. Comput. Natural Lang. Learn.*, 2014, pp. 171–180.

[19] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin, "Learning sentiment-specific word embedding for Twitter sentiment classification," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, 2014, pp. 1555–1565.

[20] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 2177–2185.

[21] D. P. Mandic and J. Chambers, *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. Hoboken, NJ, USA: Wiley, 2001.

[22] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[23] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.

[24] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Netw.*, vol. 18, nos. 5–6, pp. 602–610, Jul. 2005.

[25] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1253–1269, Dec. 2019.

[26] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*. Heidelberg, Germany: Springer, 2010, pp. 177–186. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-7908-2604-3_16

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980

[28] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[29] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: Do different classifiers find the same defects?" *Softw. Qual. J.*, vol. 26, no. 2, pp. 525–552, Jun. 2018.

[30] R. H. Chang, X. D. Mu, and L. Zhang, "Software defect prediction using non-negative matrix factorization," *J. Softw.*, vol. 6, no. 11, pp. 2114–2120, Nov. 2011.

[31] F. M. Tua and W. D. Sunindyo, "Software defect prediction using software metrics with Naive Bayes and rule mining association methods," in *Proc. 5th Int. Conf. Sci. Technol. (ICST)*, vol. 1, Jul. 2019, pp. 1–5.

[32] H. Wei, C. Hu, S. Chen, Y. Xue, and Q. Zhang, "Establishing a software defect prediction model via effective dimension reduction," *Inf. Sci.*, vol. 477, pp. 399–409, Mar. 2019.
[33] Z. Marian, I.-G. Mircea, I.-G. Czibula, and G. Czibula, "A novel approach for software defect prediction using fuzzy decision trees," in *Proc. 18th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2016, pp. 240–247.
[34] R. Özakıncı and A. Tarhan, "Early software defect prediction: A systematic map and review," *J. Syst. Softw.*, vol. 144, pp. 216–239, Oct. 2018.
[35] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Softw.*, vol. 12, no. 3, pp. 161–175, Jun. 2018.
[36] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.
[37] Y. A. Alshehri, K. Goseva-Popstojanova, D. G. Dzielski, and T. Devine, "Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them," in *Proc. SoutheastCon*, Apr. 2018, pp. 1–7.
[38] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 13th Int. Conf. Softw. Eng. ICSE*, 2008, pp. 531–540.
[39] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 13th Int. Conf. Softw. Eng. - ICSE*, 2008, pp. 181–190.
[40] B. Turhan and A. Bener, "Analysis of naive Byes' assumptions on software fault data: An empirical study," *Data Knowl. Eng.*, vol. 68, no. 2, pp. 278–290, Feb. 2009.
[41] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. Domain vs. Process," in *Proc. 7th joint meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. - ESEC/FSE*, 2009, pp. 91–100.
[42] S. Herbold, "Training data selection for cross-project defect prediction," in *Proc. 9th Int. Conf. Predictive Models Softw. Eng.*, Oct. 2013, pp. 1–10.
[43] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 382–391.
[44] S. Qiu, L. Lu, and S. Jiang, "Joint distribution matching model for distribution–adaptation-based cross-project defect prediction," *IET Softw.*, vol. 13, no. 5, pp. 393–402, 2019.
[45] Z. Xu, S. Pang, T. Zhang, X.-P. Luo, J. Liu, Y.-T. Tang, X. Yu, and L. Xue, "Cross project defect prediction via balanced distribution adaptation based transfer learning," *J. Comput. Sci. Technol.*, vol. 34, no. 5, pp. 1039–1062, Sep. 2019.
[46] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.
[47] A. Viet Phan, M. Le Nguyen, and L. Thu 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.
[48] H. Li, X. Li, X. Chen, X. Xie, Y. Mu, and Z. Feng, "Cross-project defect prediction via ASTToken2 Vec and BLSTM-based neural network," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2019, pp. 1–8.
[49] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Aug. 2016, pp. 87–98.
[50] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 631–642.
[51] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351.

JUNHAO LIN is currently pursuing the master's degree with the School of Computer Science and Engineering, South China University of Technology. His research interests include software defect prediction and transfer learning.

LU LU received the Ph.D. degree from Xi'an Jiaotong University, in 1999. He is currently a Professor with the School of Computer Science and Engineering, South China University of Technology, China. His main research interests include software engineering, software testing, and software architecture design.