Received 27 September 2022, accepted 9 October 2022, date of publication 13 October 2022, date of current version 21 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3214536

APPLIED RESEARCH

A Cross-Project Defect Prediction Model Based on Deep Learning With Self-Attention

WANZHI WEN[1,2], (Member, IEEE), RUINIAN ZHANG[1], CHUYUE WANG[1], CHENQIANG SHEN[1], MENG YU[1], SUCHUAN ZHANG[1], AND XINXIN GAO[1]

[1]School of Information Science and Technology, Nantong University, Nantong 226019, China
[2]Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology, Nanjing 211106, China

Corresponding author: Wanzhi Wen (wenwanzhi@126.com)

This work was supported in part by the College Students' Innovation and Entrepreneurship Training Program under Grant No. 202210304110Y; in part by the Open Funds of Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology, under Grant No. NJ2018014; and in part by the Nantong Science and Technology Research Project under Grant No. JC2021125.

ABSTRACT Cross-project defect prediction technique is a hot topic in the field of software defect research because of the huge difference in data distribution between source project and target project. Software defect prediction technique usually first extracts software project features and then trains prediction models based on various classifiers. However, traditional features lack sufficient semantic information of source code resulting in poor performance of the prediction models. To construct more accurate prediction models based on the semantic information, we propose a cross-project defect prediction framework named BSLDP, which extracts semantic information of source code files through a bidirectional long and short-term memory network with self-attention mechanism. In particular, we provide semantic extractor named ALC to extract source code semantics based on source code files, and propose classification algorithm based on the semantic information of source project and target project, namely BSL, to build a prediction model. Furthermore, we propose an equal meshing mechanism that ALC generates semantic information on small fragments by dividing the numerical token vector to further improve the performance of the proposed model. We evaluated the performance of the proposed model on a publicly available PROMISE dataset. Compared with the four state-of-the-art methods, the experimental results indicate that on average BSLDP improves the performance of cross-project defect prediction in terms of F1 by 14.2%, 34.6%, 32.2% and 23.6%, respectively.

INDEX TERMS Defect prediction, deep learning, long and short-term memory, self-attention mechanism.

I. INTRODUCTION

Under the background of exponential development of Internet scale, software brings great convenience to the life of people. With the development of society, the needs of people in all aspects make software design more and more complicated. Therefore, software developers use software defect prediction techniques to minimize software defects [3], [4], [5]. Software defect prediction technology extracts information from software and trains models used to predict whether new code instances are defective.

In software defect prediction [21], [25], [26], researchers usually extract characterization information from a source project release, and the model is trained to predict code defects in another target project version. The researchers divided the software defect prediction research into two directions based on whether the two project versions came from the same project: they are within-project defect prediction and cross-project defect prediction [50], [51].

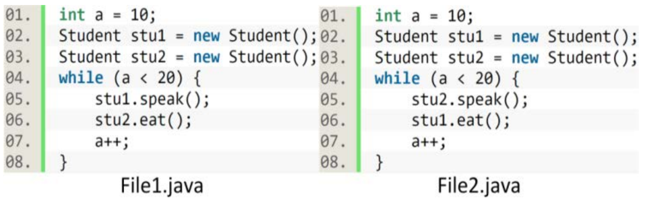
Various previous studies had focused on feature representation of code and built more accurate classification models.

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

Researchers have manually defined a variety of features representing source code to distinguish different code snippets. For instance, six metrics of Mood [1] were proposed based on encapsulation factor and inheritance factor, etc., features based on the change of elements including commit frequencies, experience levels [2], and characteristics based on other coding characteristic of the developers. Predicting software module defects are a classification problem, many classification algorithms are adopted in software defect prediction, such as Naive Bayes [3], Random Forest [4], and Dictionary Learning [5].

However, because artificial static values can only reflect a rough aspect of a code segment, so we used these static features and processing features [6] to train the network model. This method will affect the classification effect to a certain extent. Some static values need to be calculated with special tools, which is very time-consuming. Static values include code modification, the log of deletion, complexity calculation after code change and the degree of code structured change. Relevant studies show that a lot of information cannot be reflected by these metrics, semantic information is the most important among them. Some researchers have noticed that the syntax structure can be represented by using abstract syntax trees [7], [48]. It provides a new direction for their study of defect prediction techniques. Their experiments [8], [9] provided strong evidence that syntactic constructs can provide better code representations than traditional features. However, there are some limitations of the AST-based experiments, where AST is an acronym for abstract syntax tree. For example, their method [8] converts the syntax structure into a complete binary tree or directly as a complete binary tree. This method will change the original syntax structure of the code and even lead to the construction of a larger syntax tree. Then the changed syntax information will affect the precise model created [10]. In addition, the semantic information in the abstract syntax tree cannot be effectively represented by the existing traditional features. However, the abstract syntax tree is an indirect representation of the code, which may lose the semantic information on the source code. Until now, the syntax structure still cannot fully represent the code information.

Compared with syntactic information, semantic information can better distinguish different code fragments [11]. The semantic information of code plays an important role in the research related to code completion and code defect location [7], [12], [13], [14], [15]. Two snippets of code with the same metric may contain different code semantic information. For example, Figure 1 shows a motivating example, there are two Java files, *File1.java* and *File2.java*, both of them contain a while statement, a *speak* function and an *eat* function. The main difference between the two files are the order where the two method calls are made, using the traditional metrics, the feature vectors of the two files are identical in terms of code lines and method calls, etc. However, the semantic information on the two code files is quite different.



```
int a = 10;
Student stu1 = new Student();
Student stu2 = new Student();
while (a < 20) {
    stu1.speak();
    stu2.eat();
    a++;
}
```
File1.java

```
int a = 10;
Student stu1 = new Student();
Student stu2 = new Student();
while (a < 20) {
    stu2.speak();
    stu1.eat();
    a++;
}
```
File2.java

FIGURE 1. Motivating example.

In this paper, we are inspired by the possibility of improving defect prediction and look for ways to avoid intermediate representations of source code that can overcome the limitations of the above AST-based methods. Therefore we propose using leverage a powerful semantic representation algorithm, namely ALC, to represent the semantic information of programs effectively. The above algorithm is used to train software defect prediction model. Specifically, we propose a new classifier for the features extracted by ALC, namely BSL, to improve the results of classification.

More specifically, firstly, we split the source code into token vectors, these vectors retain its own structure and context information, we conduct mapping to convert token vectors into numerical vectors. Secondly, based on the sequence of numerical vectors, we use bidirectional Long Short Term Memory (Bi-LSTM) [16] combined with a self-attention mechanism. It is a type of recurrent neural network, which is used to obtain the semantic vector representation of an instance. Thirdly, we input all semantic vectors from source project into the proposed classifier, in this step can construct a defect prediction model. Finally, we propose an end-to-end framework for cross-project defect prediction (BSLDP), to predict whether a source code file contains defects or not.

In summary, our proposed source code semantic representation algorithm aims to learn the semantic information on the code, which is more effectively than other semantic extraction models. We do experiment on well-known open datasets from the PROMISE repository, and there are ten projects with known defects. For example, the proposed model outperforms four baseline methods by 14.2%, 34.6%, 32.2% and 23.6% in terms of F1 in defect prediction, respectively.

Our main contributions are highlighted as follows:

1) We propose to leverage a powerful semantic representation algorithm, namely ALC, composed of a type of recurrent neural network (Bi-LSTM) combined with a self-attention mechanism, to represent the semantic information of programs effectively and use the representation to train software defect prediction model.
2) We propose a new classifier for the semantic vectors extracted by ALC to improve the results of classification, namely BSL, composed of a type of recurrent neural network (Bi-LSTM) combined with a self-attention mechanism.
3) We propose an end-to-end framework for cross-project defect prediction (BSLDP), to predict whether a source code file contains defects or not. The experimental

results show that our approach can improve the performance of cross-project defect prediction.

The rest of the article is organized as follows: Section II provides the introduce backgrounds on defect prediction, long and short-term memory, and self-attention mechanism. Section III describes the details of our approach. We detail the experimental setup and the results in Section IV and Section V. Section VI and Section VII present the possible threats to validity and related work respectively. Finally, Section VIII concludes our work.

II. PRELIMINARY

In this chapter, we briefly introduce file level defect prediction techniques, long short term memory neural network and self-attention mechanism. The so-called file level defect prediction technique means that the basic unit of semantic extraction and model prediction is a source code file.

A. DEFECT PREDICTION

Figure 2 shows the implementation process of software defect prediction technology based on file level, it has been implemented in numerous existing studies [20], [29], [30], [31], [47].

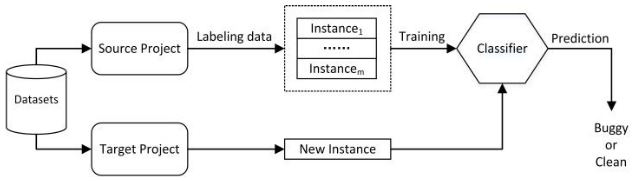


FIGURE 2. Defect prediction process.

The first step is to mark the files in the project based on the defects of the previous release of the project. If the file contains defects in the previous project, it will be marked as defective, otherwise it will be marked as non-defect. The second step is to extract features of files, and these features are mainly divided into traditional features and semantic features. Traditional characteristics such as Halstead features [32] based on operator and operand counts, McCabe features [33] based on dependencies, and CK features [34] based on function and inheritance counts, etc., semantic features are extracted from source code by deep learning correlation algorithm [11], [35]. Thirdly, we used the extracted feature instances and corresponding labels to build prediction models with various machine learning algorithms [49] such as Logistic Regression, Naive Bayes, and Random Forest. Finally, the researchers used trained models to predict whether new instances were defective.

The researchers referred to the dataset that builds the model as the training set, and the data predicted by the model as the test set. When researchers predict software defects within a project, training set and test set come from the same project. When researchers predict software defects across projects, training set and test set come from different projects.

In this study, we focus on cross-project defect prediction due to the large data difference between the two sets in cross-project.

B. LONG SHORT TERM MEMORY

Long and short-term memory neural network is a neural network of the ability to memorize long and short-term information, it can solve long-term problems. The architecture of LSTM is shown in Figure 3. The core element of LSTM is the cell state, which is transmitted along the time line like a conveyor belt of information. As it passes through each cell unit, the information is slightly changed. LSTM also realizes the addition and deletion of information. In other words, LSTM can solve the long-term dependence problem of RNN in that LSTM introduces the gate mechanism to control the flow and loss of features. LSTM is composed of a series of LSTM units, and each of them contains an input gate, a forgetting gate and an output gate.

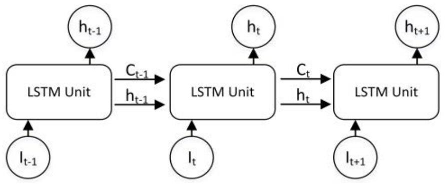


FIGURE 3. Architecture of LSTM. The architecture consists of three LSTM units. The input to each cell contains the current input data, the hidden layer output of the previous cell, and the cell state of the previous cell. The output of each cell contains the hidden layer output as well as the cell state.

The cell state C_t acts as follows:

$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t \tag{1}$$

where f_t refers to the forgetting gate, C_{t-1} refers to the cell states of the upper unit, i_t refers to the data of the input gate of the current unit, $\tilde{C}_t$ refers to the updated values of the cell states at the time of t.

Forget gate f_t acts as follows:

$$f_t = \sigma\ (W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2}$$

where w_f represents the weight of the forgetting gate, b_f represents the bias of the forgetting gate, $[\cdot]$ represents the concatenation of vectors by row, σ represents sigmoid activation function, h_{t-1} represents the state of hidden layer at the time of t-1, x_t refers to the input vector at the time of t.

Input gate i_t acts as follows:

$$i_t = \sigma\ (w_i \cdot [h_{t-1}, x_t] + b_i) \tag{3}$$

where w_i represents the weight of the input gate, b_i represents the bias of the input gate, $[\cdot]$ represents the concatenation of vectors by row, σ represents sigmoid activation function, h_{t-1} represents the state of hidden layer at the time of t-1, x_t refers to the input vector at the time of t.

The updated value of the cell state $\tilde{C}_t$ acts as follows:

$$\tilde{C}_t = \tanh\,(w_c \cdot [h_{t-1}, x_t] + b_c) \tag{4}$$

where w_c represents the weight of updated cell status, b_i represents the bias of updated cell status, $[\cdot]$ represents the concatenation of vectors by row, tanh represents tanh activation function, h_{t-1} represents the state of hidden layer at the time of t-1, x_t refers to the input vector at the time of t.

Hidden layer output acts h_t as follows:

$$o_t = \sigma\,(w_o\cdot[h_{t-1}, x_t]+b_o) \tag{5}$$

$$h_t = o_t \times \tanh\,(c_t) \tag{6}$$

where o_t represents the output of the output gate, w_o represents the weight of the output gate, b_o represents the bias of the output gate, tanh represents tanh activation function, h_{t-1} represents the state of hidden layer at the time of t-1, x_t refers to the input vector at the time of t.

LSTM has demonstrated its success in numerous research areas. For example, Zhang and Lu [36] used LSTM combined with other algorithm to build acoustic models, and they reduced the loss of the entire sequence and predicted the sequence label correctly in the prediction probability of the LSTM output. In the field of image description, Li and Shen [37] used bidirectional LSTM to generate sentences in both forward and backward direction with richer information. In text classification research, Zhang [38] constructed a LSTM neural network classification model to classify text information, and the model had obvious improvement in performance and classification accuracy compared with traditional methods.

In this paper, we use LSTM to construct semantic extractor and classifier. Semantic extractor captures semantic information from source code files, and classifier trains models based on instance semantic information and corresponding labels to predict defects of new instances.

C. SELF-ATTENTION MECHANISM

Attention mechanism in nature is consistent with the human eye, human vision is a focus and scope, and vision subject to change with the change of the focal point. When people find themselves interested in things in a vision, people will learn the characteristics of the scene at this point, and will focus on the things they are interested in when they meet in similar scenes.

The calculation process of attention mechanism is divided into three steps. The first step is to calculate the similarity between query and each key to get the weight of each key; the second step is to normalize all the weights; the third step is to sum the weight and corresponding value, and finally to get the attention value of the query. In the general attention mechanism, key and value are equal. Self-attention mechanism is a special case of general attention where query, key, and value are equal, i.e. query = key = value. In other words, each unit is evaluated for attention with all units in the sequence.

The basic calculation process of self-attention mechanism is shown in Figure 4. Input tokens are a1, a2, a3, a4, respectively. We take a1 as an example to calculate attention, let a1 times the query weight to get q1, a1 times the key weight

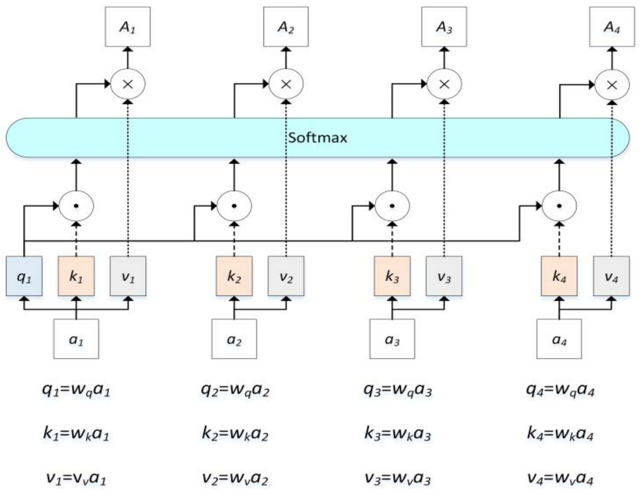


FIGURE 4. Architecture of self-attention mechanism.

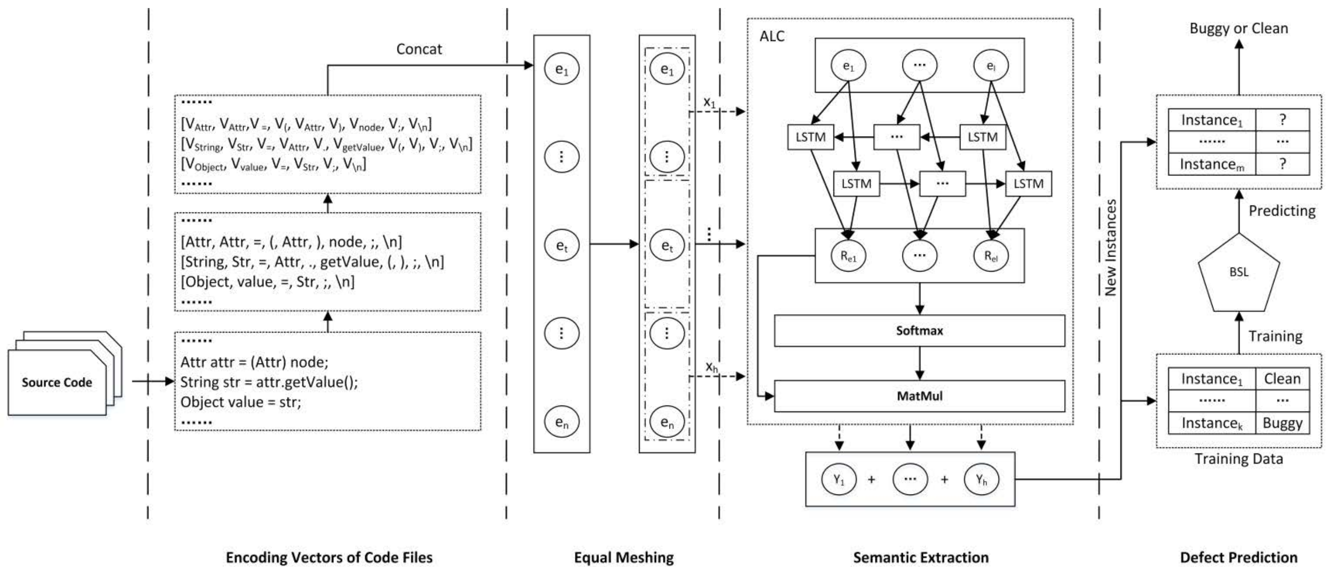
to get k1, and a1 times the value weight to get v1, the corresponding other tokens through the same calculation process to get the corresponding query, key, value, the query vector of a1 dot with the key vector of all tokens to get the similarity value. Then normalize all the similarity obtained in the sequence, and perform weighted average operation between the normalized similarity and the corresponding value vector to obtain the attention value of a1. Likewise, other tokens obtain the corresponding attention value according to the above steps.

For example, Zhang et al. [39] proposed a model based on ensemble learning techniques and attention mechanisms to offer more comprehensive prediction information. This method helped developers by locating suspect lines of code when making method-level defect predictions. Yu et al. [40] constructed a deep learning model called Defect Prediction via Self-Attention mechanism to extract semantic features and predict defects automatically.

In this paper, we use the self-attention mechanism to learn word dependencies within source code files and capture the internal structure of source code, and it can fully capture and enhance the correlation between the LSTM outputs and compensate for the loss of information over long distances.

III. APPROACH

In this section, we elaborate on our proposed BSLDP approach, which can generate features from source code files to further improve the performance of cross-project defect prediction. The overall architecture of BSLDP is shown in Figure 5. Our approach takes tokens from source project and target project. Then we use ALC to generate semantic features from the tokens. Lastly, we leverage BSL, and it is constructed to build a defect prediction model based on the semantic features. Specifically, we obtain tokens from the source code of the project, and each code file is mapped to a token vector. Since the semantic extractor requires digital data, we build a mapping rule to convert each token into a digital representation, and eventually each token vector into



```
......
Attr attr = (Attr) node;
String str = attr.getValue();
Object value = str;
......
```

```
......
[Attr, Attr, =, (, Attr, ), node, ;, \n]
[String, Str, =, Attr, ., getValue, (, ), ;, \n]
[Object, value, =, Str, ;, \n]
......
```

```
......
[V_Attr, V_Attr, V_=, V_(, V_Attr, V_), V_node, V_;, V_\n]
[V_String, V_Str, V_=, V_Attr, V_., V_getValue, V_(, V_), V_;, V_\n]
[V_Object, V_value, V_=, V_Str, V_;, V_\n]
......
```

FIGURE 5. The framework of our proposed BSLDP.

a digital vector. Then we use a semantic extractor to extract the semantics of the number vectors generated by code files in the source project and the target project respectively. Finally, we use classifiers to build cross-project defect prediction models based on semantic information of source projects and use semantic information of target projects to evaluate model performance.

To sum up, our approach consists of four major steps: 1) converting the source code to a numerical representation vector, it is the expected inputs of ALC, 2) carrying out semantic extraction from digital representation vector, 3) building defect prediction model based on the semantic vector of the source project, 4) using models to predict defects based on the semantics of the target project.

Algorithm 1 Token Extraction Algorithm

Input: Source code file F, it is constructed by the Java source code of the instance.

Output: Token vector for instance T, where the length of vector is the number of tokens obtained by the instance after splitting rules.

```
1. var txt_F, T := [];
2. begin;
3.     txt_F := read(F) // Java files -> TXT files
4.     T := partition(txt_F) //'\n', '(', '[', '-', ',', '.', '(', ')', '{', '}', ';', '"', '+', ']', ')'
5.     return T
6. end;
```

A. ENCODING TOKEN VECTORS

To generate semantic features by using ALC, we first build a token extraction algorithm, which converts each source code file into a token vector. We convert the source code file into TXT file, and divide the source code according to the defined basic partition identifier to get token vectors.

According to Algorithm 1, we convert all code files in the source and target projects into token vectors, but each element in the token vectors is a string, and it cannot be computed mathematically, so we need to encode token vectors to integer vectors. Each token has a unique integer identifier while different variable names and class names will be treated as different tokens, and we specifically add newline identifiers.

Second, we build a mapping algorithm between integers and tokens. Taking the string "student" as an example, the string "student" is split into seven separate character list, it is ['s', 't', 'u', 'd', 'e', 'n', 't']. Then we get the ASCII value of the single character, the converted list is [115, 116, 117, 100, 101, 110, 116], we sum the list up to get the digital representation, it is 775. Through this mapping process, we converted all the tokens to digital representation.

The specific steps of token mapping algorithm are shown in Algorithm 2.

However, we find in the experiment that each source code file is converted into a token vector of different lengths, so, we need to align the token vector lengths and get the vector of the same length. So, we build a semantic vector alignment algorithm, the source and target projects are calculated respectively in the median length of the file. In order to make full use of the data information on the source project, the median of the combined file length of the two projects is simultaneously calculated. If the median of the target project is greater than the median of the source project, the median of the two project sets is used, otherwise, the median of the source project is used. We use 0 to expand all vector lengths of the two projects to the maximum length, after that, we intercept the expanded vector using the median obtained above.

Algorithm 2 Token Mapping Algorithm

Input: Token vector for instance T, where the length of vector is the number of tokens obtained by the instance after splitting rules.

Output: Numerical vector for instance NT, where the length of vector is the number of tokens obtained by the instance after splitting rules.

```
1. var $NT$ := [], $v$:integer;
2. begin
3.   for $t$ in T do //traverse the token
4.   begin
5.     v := 0 //the ASCII value of token
6.     for $ec$ in t do // traverse single character
7.       ec_ascii := ASCII($ec$)
8.       v := ec_ascii+v
9.   end;
10. end;
```

The specific steps of semantic vector alignment algorithm are shown in Algorithm 3.

Algorithm 3 Semantic Vector Alignment Algorithm

Input: Collection of all numeric vectors for the source project D_S, where the length of collection is the number of instances, each vector represents a code file in the source project, and each vector length varies by code files.
Collection of all numeric vectors for the target project T_S, where the length of collection is the number of instances, each vector represents a code file in the target project, and each vector length varies by code files.

Output: Semantic Vector Alignment vectors of the source project DA_S. Semantic Vector Alignment vectors of the target project DA_T

```
1. var LS := [], LT := [],$S_m$:integer, $T_m$:integer
2. begin
3.   for i in $D_S$ do
4.   begin
5.     inst_len := len(i) //length of each instance
6.     LS.append(inst_len)
7.   end;
8.  $S_m$ := int(median(LS));
9.  begin
10.    for j in $D_T$ do
11.      begin
12.        inst_len := len(j) //length of each instance
13.        LT.append(inst_len)
14.      end;
15.    $T_m$ := int(median(DT_LEN))
16.  end;
17. E := $S_m$//effective length
18. CST := LS.append(LT)
19. $C_m$ := int(median(DT_LEN))
20. if $S_m < T_m$, then $E := C_m$
21. $l_m$ := max(CST) //the maximum length of instance
22. len($D_S$)$-> l_m$, len($D_T$)$-> l_m$// expand the length of each vector in $D_S$ and $D_T$ to $l_m$ using 0
23. len($D_S$)$-> E$, len($D_T$)$-> E$ // cut each of the vectors $D_S$ and $D_T$ to $E$ lengths
24. return $DA_S$, $DA_T$;
25. end;
```

B. SEMANTIC EXTRACTION

The code files of source project and target project are digitized and encoded to get digitized vectors, they are fed into the proposed semantic extractor for semantic extraction. The semantic extractor is composed of Bi-LSTM and self-attention mechanism. Bi-LSTM adopts bidirectional data training mode, the number of basic LSTM units is determined by the length of semantic vector. After a model training, we add the output of forward hiding layer and reverse hiding layer of the same basic LSTM unit to obtain the final output of the unit. Then we use the self-attention mechanism for the outputs of LSTM to improve the internal correlation between outputs. The implementation of the model is based on PyTorch, by PyTorch we can build the neural network model easily and quickly. The specific steps of semantic extractor algorithm are shown in Algorithm 4.

C. BUILDING MODEL

In order to achieve a better classification effect and fit the obtained semantic vector, we propose a classification algorithm. The classification algorithm is composed of Bi-LSTM and self-attention mechanism, it is the same as semantic extractor. We take the final value processed by the self-attention mechanism as the probability output of the code file. We obtain the corresponding probability output of the source project, and the target project under the same processing steps. Then we use the two groups of probability to draw PR curves, and obtain the F1 corresponding to different points through PR curves. We determine the index corresponding to the maximum F1. P in PR curve stands for precision, R stands for recall, and it represents the relationship between precision and recall. In general, recall is set as the abscissa and precision as the ordinate. The index is used to select the corresponding value of threshold vector of PR curve and take it as the threshold value of classification. Finally, we get the predicted classification label through calculation. The specific steps of classification algorithm are shown in Algorithm 5.

We observe that the token vector length of the code file is too long in the experiment, it may affect the performance of the model. Under the incentive of further improving the cross-project defect prediction performance, we propose an equal-spacing token partition method called equal meshing mechanism. Specifically, the method obtains the token vector length of the code file firstly, sets the starting value of the pane

Algorithm 4 Semantic Extractor Algorithm

```
Input: Numerical vectors $NT$ for project after the alignment
       and its labels $NT_L$.
Output: Semantic vectors $SV$ for project.
1. begin
2.   for $i$ in epoch do
3.   begin
4.     use Adam //optimizer
5.     set and initialize $h0$, $c$ 0// hidden layer parameters
6.     $NT$ , $h$ 0, $c0$ > Bi-LSTM // model training
7.     add the hidden layer bidirectional values
8.     $out_{\mathrm{lstm}}$ = output of Bi-LSTM // ($b$, $l$, $h\_ dim$)
9.     $out_{lstm}$− > 2D matrix // ($b$, $l_1h\_ dim$)
10.      $PC$ := nn.Sequential 0// process containers
11.      Linear($h\_dim$, 24), ReLU(True), Linear( 24,1) →> PC
12.      PC (matrix) − > matrix. view (b, −1); // ($batch$, $l$)
13.      $w_{\mathrm{nor}}$ = normalize(2D matrix) // unsqueeze (2), ($b$, $l$, 1)
14.      Attn := $\left(w_{\mathrm{nor}} {}^* \ out_{\mathrm{stm}}\right) \cdot$ sum(dim = 1); //($b$, $l$, $h$) → ($b$, h)
15.      $FCL_2$ = Linear($h\_dim$, $c\_num$);
16.      $out_{\mathrm{prob}}$ = $FCL_2$(Attn)//($\bar{b}$, 1)
17.      $FCL_3$ := Linear ($c\_$num, feature$_{\mathrm{n}}$)
18.      $out_{feature}$ := $FCL_3$ ( $out_{\mathrm{prob}}$ )
19.      loss := Loss $\left(out_{\mathrm{prob}}, N\ T_L\right)$
20.      Loss.backward(), optimizer.step0
21.      SV.append($out_{feature}$)
22.    end;
23.    return $SV$
24. end;
```

Algorithm 5 Classification Algorithm

```
Input: Semantic vectors $D_S$ for source project and its labels
       $S_L$. Semantic vectors $D_T$ for target project.
Output: Prediction label $P_L$ for the target project.
1. begin
2.   use Adam//optimizer
3.   set model mode // training
4.   initialize $h0$, $c0$ of Bi-LSTM
5.   $D_S$ > Bi-LSTM // model training
6.   add the hidden layer bidirectional values;
7.   $out_{lstm}$ := output of Bi-LSTM //( b, l, $h\_dim$ )
8.   $out_{lstm}$− > 2D matrix // ($b$, $l$, $h\_ dim$)
9.   $PC$ := nn. Sequential (// process containers
10.    Linear($h\_dim$, 24), ReLU(True), Linear (24, 1) → $PPC$
11.    $PC$ (matrix) > matrix view (b, −1); // ($batch$, $l$)
12.    $w_{\mathrm{nor}}$ = normalize(2D matrix) // unsqueeze (2), ($b$, $l$, 1)
13.    Attn := $\left(w_{\mathrm{nor}} {}^* \ out_{\mathrm{lstm}}\right) \cdot$ sum(dim = 1); //($b$, $l$, $h$) − ($b$, $h$)
14.    $FCL_2$ := Linear($h\_dim$, $c\_num$);
15.    $out_{prob}$ = $FCL_2$(Attn)//($b$, 1)
16.    loss := Loss ( $out_{prob}$, $S_L$)
17.    Loss.backward(), optimizer.step();
18.    set model mode // test
19.    Do from Step3 to Step15, change $D_S$ to $D_T$ in Step5;
20.    Get $out_{trpob}$ from Step15;
21.    Draw the PR curve using $out_{sprob}$ and $out_{tprob}$;
22.    Computer $F1$ using the point of $PR$ curve and find the index of maximum F1;
23.    thre := thresholds[index], $P_L$ := [x > thre for x in $out_{\mathrm{prob}}$ ]
24.    return $P_L$
25. end;
```

size as 100, step size as 50, and gradually accumulates until it is greater than the vector length, records all values obtained in the process of accumulation in the list, and removes the values exceeding the vector length. Finally, we get the equal-spacing partition list and label the equal-spacing value as γ.

To sum up, we use algorithm one and algorithm two to convert the code files in the source and target projects into digital vectors, algorithm three is used to make them same dimension. Then, the equal-spacing token partition method is used to segment the digital vector, in this step the equal space segmentation listed is obtained, we use semantic extractor to semantic extraction of digital vector of the project under every spacing. Finally, the semantic vectors obtained at all spacing are superimposed as the final semantic information on the file. Under each partition step, the proposed classification algorithm is used to build a prediction model for the final semantic information on the source project and to predict the final semantic information on the target project. The maximum F1 obtained at all spacing is taken as our final prediction index.

D. PREDICTING DEFECTS

We use BSL as our classifier, it is more suitable for the semantic features we obtain, and we use the above semantic extraction steps to process all the code files in the source and target projects. Then we get the semantic information for each code file. Finally, we use the semantic vectors of the source project and the corresponding label to train model, input all the code files in the target project into the model to obtain the prediction label, and evaluate the performance of the model by F1.

IV. EXPERIMENT SETUP

In this section, we compare our proposed method with four baseline methods. And our experiments are based on the following questions:

- RQ1: Is the ALC-based semantic information more effective than traditional metrics?
- RQ2: Does the proposed classifier outperform other classification algorithms?
- RQ3: Does the equal meshing mechanism improve the performance of our proposed model?
- RQ4: Does the proposed model outperform the four baseline methods in CPDP?

We conduct several experiments to study the performance of the proposed model, BSLDP, and run experiments on a 2.9GHz i7-10700F machine with 8GB GPU.

A. DATASET DESCRIPTION

To directly compare our model with other approaches, we obtain available data from PROMISE [41], and it is widely used in other defect prediction studies [44]. These projects cover a wide range of applications such as enterprise integration framework, XML parser and data transport adapters, etc. Table 1 lists project name, the description of project, the versions, the average number of files, and the average buggy rate of instances.

TABLE 1. Dataset description.

Project	Description	Releases	Avg Files	%Avg Defective
ant	Java based build tool	1.6	351	26.0%
camel	Enterprise integration framework	1.4	848	17.0%
jEdit	Text editor designed for programmers	4.0,4.1	296	26.0%
log4j	Logging library for Java	1.1	104	36.0%
lucene	Text search engine library	2.2	247	58.0%
xalan	A library for transforming XML files	2.5	772	50.0%
xerces	XML parser	1.3	452	15.0%
ivy	Dependency management library	2.0	352	11.0%
synapse	Data transport adapters	1.1,1.2	239	30.5%
poi	Java library to access Microsoft format files	3.0	442	64.0%

B. BASELINE METHODS

To evaluate the performance of our proposed framework BSLDP for cross-project defect prediction, we compare it with the following baseline methods:

- DBN-CP [26], [42]: the state-of-the-art method, it uses Deep Belief Network to learn semantic features from token vectors extracted from ASTs of programs, and leverages the learned semantic features to build machine learning models for predicting defects.
- TCA+ [18]: the state-of-the-art technique for CPDP that improve the performance of cross-project defect prediction by regularized transfer learning method.
- AST-LSTM [43]: A model based on the tree-structured Long Short Term Memory network, and it directly matches with the Abstract Syntax Tree representation of source code.
- DP-CNN [44]: A Convolutional Neural Network based model, and it leverages deep learning for effective feature generation. They train the classifier by combining semantic features with traditional features.

We choose the same 20 test pairs as in paper [26]. When using the four baseline approaches, we use the same network architecture and model parameters as described in the paper.

C. EVALUATION

It is necessary to use metrics that are used widely when we evaluate the performance of the model. The basic indicators predicted by the model are *TP*, *FN*, *FP* and *TN*, respectively. *TP* means that a file that is actually defective is predicted to be defective. *FN* means that a file that is actually defective is predicted to be non-defective. *FP* means that a file that is actually non-defective is predicted to be defective. *TN* means that a file that is actually non-defective is predicted to be non-defective. In previous studies [26], [42], [43], a variety of performance evaluation indicators have been proposed, such as F1, Recall, Precision.

The metrics of model evaluation are defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

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

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{9}$$

V. RESULTS

To evaluate the effectiveness of BSLDP, we focus on the performance of our proposed semantic information and answer the following research questions:

RQ1: Is the ALC-based semantic information more effective than traditional metrics?

To answer this question, we conducted twelve cross-project defect prediction experiments to evaluate the performance of the trained model, in the model the source project is used to train the network model, and the target project is used as the test set. We illustrate the validity of semantic information compared to traditional metrics from two aspects.

On one hand, we will use only Bi-LSTM to obtain semantic information compared to F1 obtained by traditional metrics. Table 2 shows the F1 of cross-project defect prediction experiments that use the same classification algorithm, namely Logistic Regression [45]. We used the same number of training epoches and parameters. This method usually has good performance. For example, in the first pair of project comparisons, the training set is camel-1.4 and the test set is xerces-1.3, the F1 of using NALC is 0.262, while the F1 is only 0.210 with Tra-metrics (traditional features from PROMISE), thus the semantic information extracted by NALC outperforms Tra-metrics by 24.7%. Among the 12 project pairs, the lowest improvement on F1 is 0.1% where the training set is synapse-1.2 and the test set is poi-3.0 in all positive improvement project pairs, the highest improvement of F1 is 168.9% where the training set is ant-1.6 and the test set is poi-3.0 in all positive improvement project pairs. On average, Tra-metrics achieve a F1 of 0.356, while NALC achieved a F1 of 0.409. The experimental results show NALC improved F1 of defect prediction by 23.7% compared with Tra-metrics on average on 12 project pairs. On the other hand, we compare the semantic information obtained by combining Bi-LSTM and self-attention with F1 obtained by traditional metric elements. For example, in the eleventh pair of project comparisons, the training set is ant-1.6 and the test set is poi-3.0, the F1 of using ALC is 0.777, while the F1 is only 0.290 with Tra-metrics (traditional features from PROMISE),

TABLE 2. **Comparison between deep learning-based semantic information (NALC and ALC) and traditional metrics (Tra-Metrics). NALC denotes the semantic information, and it is extracted by Bi-LSTM. ALC denotes the semantic information, and it is extracted by Bi-LSTM and Self-Attention. Source denotes the training version and Target denotes the test set version. F1 denotes F1 score. Improved-1 denotes the improvement of NALC with respect to Tra-Metrics and is measured in percentage. Improved-2 denotes the improvement of ALC with respect to Tra-Metrics and is measured in percentage. Positive improvements are highlighted in bold.**

Source	Target	Tra-metrics	NALC	ALC	Improved-1	Improved-2
		F1				
camel-1.4	xerces-1.3	0.210	0.262	0.308	**24.7%**	**47.0%**
xalan-2.5	xerces-1.3	0.274	0.266	0.292	-2.6%	**6.8%**
xalan-2.5	ivy-2.0	0.341	0.204	0.346	-40.1%	**1.5%**
lucene-2.2	ivy-2.0	0.266	0.277	0.286	**4.4%**	**7.6%**
lucene-2.2	xerces-1.3	0.371	0.309	0.318	-16.9%	-14.3%
jedit-4.0	xerces-1.3	0.198	0.286	0.332	**44.4%**	**68.1%**
jedit-4.0	ant-1.6	0.457	0.482	0.517	**5.4%**	**13.1%**
ivy-2.0	ant-1.6	0.204	0.417	0.416	**104.9%**	**104.4%**
poi-2.5	synapse-1.1	0.450	0.426	0.526	-5.5%	**16.8%**
synapse-1.2	poi-3.0	0.780	0.781	0.777	**0.1%**	-0.3%
ant-1.6	poi-3.0	0.290	0.780	0.777	**168.9%**	**168.1%**
camel-1.4	jedit-4.1	0.435	0.422	0.417	-3.0%	-4.3%
Avg		0.356	0.409	0.443	23.7%	**34.5%**

thus the semantic information extracted by ALC outperforms Tra-metrics by 168.1%. Among the 12 project pairs, the lowest improvement on F1 is 1.5% where the training set is xalan-2.5 and the test set is ivy-2.0 in all positive improvement project pairs. On average, Tra-metrics achieve a F1 of 0.356, while ALC achieved a F1 of 0.443. The experimental results demonstrate ALC improved F1 of defect prediction by 34.5% compared with Tra-metrics on average on 12 project pairs.

Having said all of these above, semantic information is more effective than traditional metrics. Both NALC and ALC outperform Tra-metrics in terms of F1 on average, in detail, NALC gets the maximum improvement of 23.7%, ALC gets the maximum improvement of 34.5%. In these two semantic extraction methods, ALC is 45.6% better than NALC in the aspect of F1 on average improvement, so ALC is superior to NALC, thus we use ALC as the semantic extractor in the subsequent experiments.

RQ2: Does the proposed classifier outperform other classification algorithms?

In this section, we explore whether the classification algorithm proposed is superior to other classification algorithms. We use logistic regression and naive Bayes as comparative classification models because previous studies [46] had shown that they are superior to other classification algorithms. We conduct a total of 20 groups of experiments in cross-project defect prediction, where the source project and the target project are exactly different.

The semantic extractor extracts a total of 100 features of each source code file, and a total of 100 project features of the same feature combination of all source code files for a project. Under the same project features, the classification algorithm is used to build a classification model for the project features of the source project. At last, the model is used to predict the defects of the project features of the target project.

Figure 6 illustrates the distribution of F1 scores of using three different classification algorithms on semantic features. From the figure, we can compare the performance of the three classification algorithms in the same group of experiments. For the median indicator values, BSL achieves favorable F1 scores in 20 groups of experiments. In addition, BSL gets better median F1 in terms of LR where the index of the experiment is 8, 11, 12, 19 and 20, respectively, compared with other project pairs. BSL gets better median F1 in terms of NB where the index of the experiment is 7, 8, 9, 11, 12 and 20, respectively, compared with other project pairs. Besides, we observe that BSL has improved more with respect to LR than with respect to NB where the index of the experiment is 3, 5, 10, 19 and 20, respectively.

Due to the different projects of the 100 kinds of characteristics of three kinds of classifiers prediction performance difference is bigger, in order to further reduce three classifier classification results and better predictive results of the said project for us to do the following definition, under a classifier, the project of the 100 species of the predicted value of the project to get maximum as the project of F1.

Because the prediction performance of the three classifiers varies greatly from the 100 features of different project pairs, so in order to further narrow the differences between the classification results of the three classifiers and better represent the predicted results of item pairs. We make the following definition that the maximum predicted value of 100 project feature pairs of project pair is regarded as F1 of the project pair under a classifier. Table 3 shows the F1 scores of running three different classification algorithms on semantic

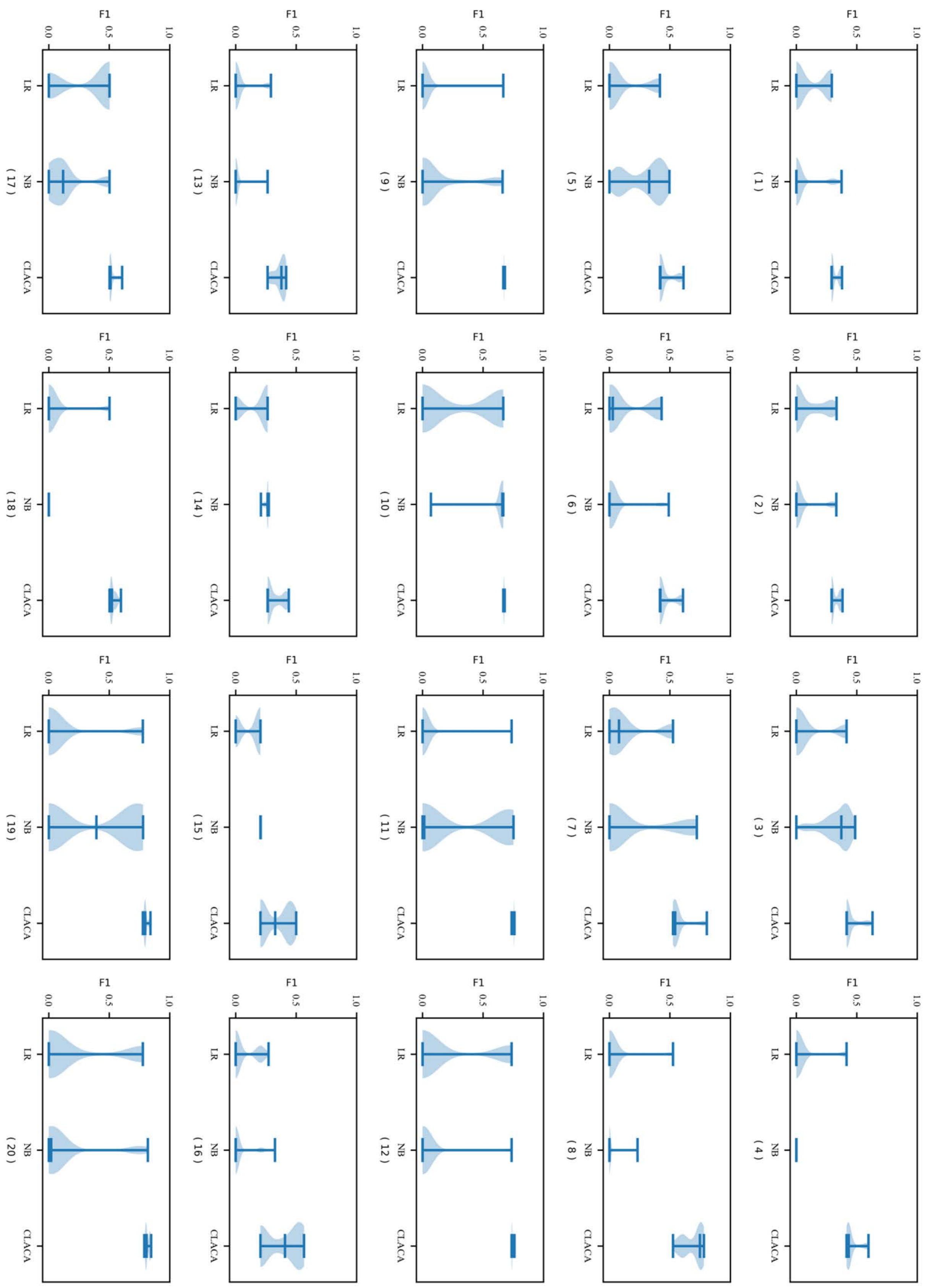


FIGURE 6. Violin plots of defect prediction performance of one hundred different features with three classification algorithm.

TABLE 3. Comparison of prediction results of three classification algorithms. LR denotes Logistic Regression classification algorithm. NB denotes Naïve Bayes classification algorithm. BSL denotes the proposed classification algorithm. F1 denotes F1 score. Improved-1 and Improved-2 denote the improvement of BSL with respect to LR and NB respectively, it is measured in percentage. The larger ones are highlighted in bold.

Index	Source	Target	LR	NB	BSL	Improved-1	Improved-2
			F1				
1	ant-1.6	camel-1.4	0.293	0.375	0.378	**29.1%**	0.8%
2	jedit-4.1	camel-1.4	0.332	0.331	0.382	15.0%	**15.4%**
3	camel-1.4	ant-1.6	0.415	0.487	0.629	**51.5%**	29.3%
4	poi-3.0	ant-1.6	0.415	0.000	0.596	**43.5%**	-
5	camel-1.4	jedit-4.1	0.417	0.497	0.612	**46.9%**	23.3%
6	log4j-1.1	jedit-4.1	0.431	0.490	0.607	**41.1%**	23.9%
7	jedit-4.1	log4j-1.1	0.525	0.722	0.806	**53.6%**	11.6%
8	lucene-2.2	log4j-1.1	0.525	0.233	0.781	48.9%	**235.9%**
9	lucene-2.2	xalan-2.5	0.668	0.661	0.681	1.9%	**3.0%**
10	xerces-1.3	xalan-2.5	0.668	0.668	0.680	**1.9%**	1.8%
11	xalan-2.5	lucene-2.2	0.737	0.752	0.761	**3.3%**	1.2%
12	log4j-1.1	lucene-2.2	0.737	0.737	0.761	3.3%	3.3%
13	xalan-2.5	xerces-1.3	0.292	0.265	0.418	43.2%	**58.0%**
14	ivy-2.0	xerces-1.3	0.265	0.273	0.440	**65.9%**	60.9%
15	xerces-1.3	ivy-2.0	0.204	0.208	0.500	**145.0%**	140.6%
16	synapse-1.2	ivy-2.0	0.272	0.324	0.566	**108.3%**	74.5%
17	ivy-2.0	synapse-1.2	0.503	0.503	0.607	20.7%	20.7%
18	poi-3.0	synapse-1.2	0.503	0.000	0.596	**18.6%**	-
19	synapse-1.2	poi-3.0	0.777	0.780	0.840	**8.1%**	7.7%
20	ant-1.6	poi-3.0	0.777	0.820	0.847	**8.9%**	3.3%

features extracted by ALC. We compare the performance of semantic features of three classification algorithms. In the improvement on BSL relative to the other two classification algorithms, better improvements are highlighted in bold. Take the first project pair as an example, the source project is ant-1.6 and the target project is camel-1.4, the feature produces a F1 of 0.293 when the model is built on LR, while BSL gets a F1 of 0.378 and it is 29.1% higher than using LR. For the same project pair, the feature using NB produces a F1 of 0.375, it is 0.8% lower than using BSL only. Among the twenty groups of experiments, the maximum improvement of BSL relative to LR is 145.0% where the source project is xerces-1.3 and the target project is ivy-2.0. Also, the maximum improvement of BSL relative to NB is 235.9% where the source project is lucene-2.2 and the target project is log4j-1.1. Similarly, the improvement on BSL relative to LA outperforms that of BSL relative to NB in 14 out of the 20 times. In conclusion, we use BSL as the classification algorithm in the subsequent experiments. We present the data onto Table 3 in the form of the line chart to make the presentation of the data more intuitive, as is shown in Figure 7.

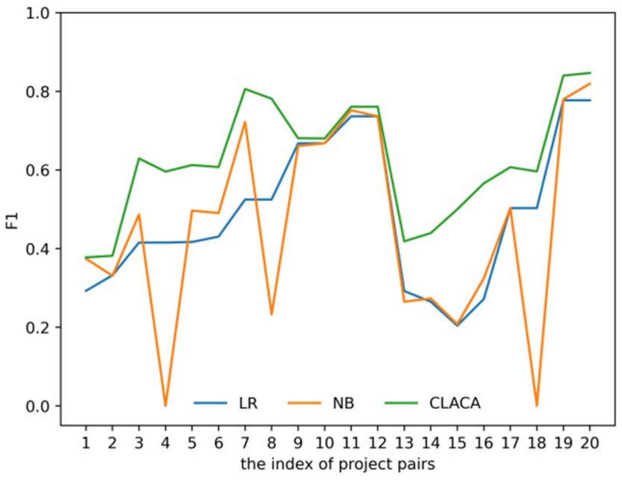


FIGURE 7. Defect prediction performance of twenty project pairs with three classification algorithms.

RQ3: Does the equal meshing mechanism improve the performance of our proposed model?

Parameter γ is used to segment the numerical token vectors. It can avoid that numerical token vectors are too long to better capture semantics between codes. In each group of experiments, the parameter γ is initially set as 100, increased by 50 as a step until γ exceeds the length of the instance token vector and stop. The previous γ is taken as the last experimental parameter of this group of experiments. Based on RQ2,

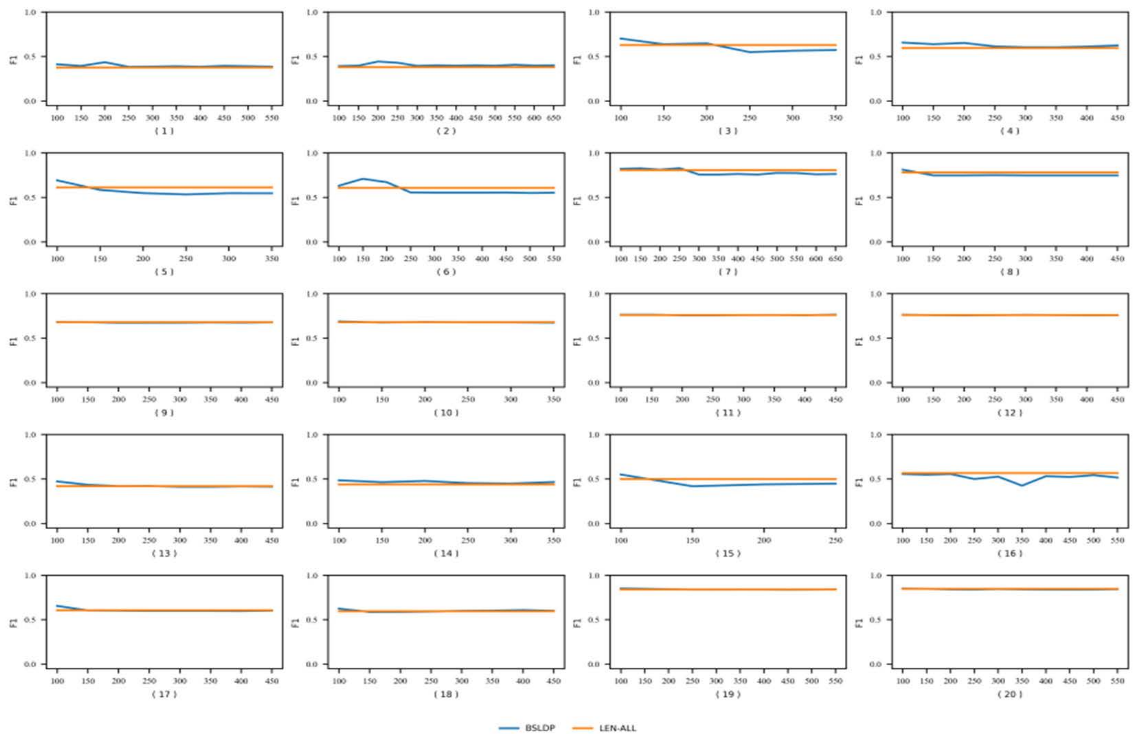


FIGURE 8. Line charts of the F1 score of BSLDP with different γ values on all cross-project pairs.

because of the better prediction performance of BSL, we use BSL as the classifier of our experiment. Here we investigate whether different γ values impact the performances of the proposed model in a project pair. Guided by prior work, we conducted twenty groups of CPDP experiments.

Figure 8 depicts the line charts of F1 score BSLDP with different γ values on all cross-project pairs. From the figure, we observe that BSLDP with different γ values achieves higher F1 score compared with using all numerical token vectors of the instance in 19 out of the 20 times. In the experiment of the first project pair, the maximum value is achieved where the γ is 200, while the maximum value is achieved where the γ is 150 in the experiment of the sixth project pair. In addition, BSLDP improves the F1 slightly where the index of experiments are 9, 10, 11 and 12. However, BSLDP with different γ values do not improve the F1 compared with using all numerical token vectors of the instance where the index of experiment is 16.

In order to better analyze the dispersion degree and data distribution of F1 obtained by different γ, we present the data onto the form of violin diagram. As shown in Figure 9, we can find that the influence of BSLDP with different γ values varies in different project pair. The range of F1 obtained by BSLDP is large in the experimental index which is 3, 5, 6, 15, and 16 respectively. However, the F1 obtained by BSLDP has only minor changes in the experimental index which is 9, 10, 11, 12 and 19, respectively.

Since different γ has different effects on the performance of model in different projects. In order to objectively express the predicted performance results of project pairs, we take the maximum predicted results of different γ as the F1 of this project pair under the same project pair. Table 4 presents the F1 results of BSLDP and ATV. From the table, we observe that nine pairs of experiments improved by more than 10%, in the nine pairs, the biggest improvement was 16.8% where the source project is log4j-1.1 and the target project is jedit-4.1, the minimum improvement was 10.1% where the source project is xerces-1.3 and the target project is ivy-2.0.

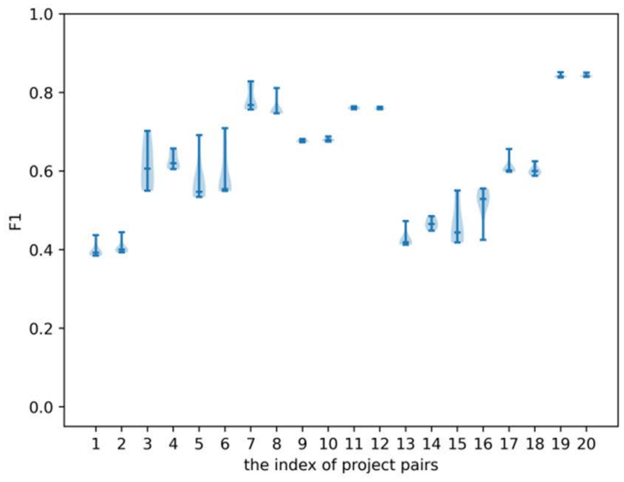


FIGURE 9. Violin plots of the F1 score of BSLDP with different γ values on all cross-project pairs.

TABLE 4. Comparison of F1 between ATV and BSLDP. ATV denotes the experiment uses all numerical token vectors of the instance. BSLDP denotes the proposed model. F1 denotes F1 score. Improved denotes the improvement of BSLDP with respect to ATV and is measured in percentage. Over 10% of improvements are highlighted in bold.

Index	Source	Target	ATV	BSLDP	Improved
			F1		
1	ant-1.6	camel-1.4	0.308	0.437	**15.6%**
2	jedit-4.1	camel-1.4	0.292	0.444	**16.4%**
3	camel-1.4	ant-1.6	0.346	0.702	**11.5%**
4	poi-3.0	ant-1.6	0.286	0.658	**10.4%**
5	camel-1.4	jedit-4.1	0.318	0.691	**12.9%**
6	log4j-1.1	jedit-4.1	0.332	0.709	**16.8%**
7	jedit-4.1	log4j-1.1	0.517	0.829	2.8%
8	lucene-2.2	log4j-1.1	0.416	0.812	3.9%
9	lucene-2.2	xalan-2.5	0.526	0.682	0.2%
10	xerces-1.3	xalan-2.5	0.777	0.688	1.1%
11	xalan-2.5	lucene-2.2	0.777	0.765	0.5%
12	log4j-1.1	lucene-2.2	0.417	0.763	0.3%
13	xalan-2.5	xerces-1.3	0.308	0.473	**13.0%**
14	ivy-2.0	xerces-1.3	0.292	0.485	**10.4%**
15	xerces-1.3	ivy-2.0	0.346	0.550	**10.1%**
16	synapse-1.2	ivy-2.0	0.286	0.556	-1.9%
17	ivy-2.0	synapse-1.2	0.318	0.656	8.1%
18	poi-3.0	synapse-1.2	0.332	0.625	4.8%
19	synapse-1.2	poi-3.0	0.517	0.852	1.4%
20	ant-1.6	poi-3.0	0.416	0.851	0.5%

However, in the experiments that improved by less than 10%, there are four pairs where the improvement is positive and less than 1%, we also observe that there is one pairs where the improvement is negative.

In conclusion, the equal meshing mechanism can improve the performance of the proposed model, which also improves the model differently depending on target project and source project.

RQ4: Does the proposed model outperform the four baseline methods in CPDP?

In order to answer this question, we compare the proposed model with four baseline methods. DBN-CP runs on the semantic features generated by DBN automatically. TCA+ is a state-of-the-art method of CPCP based on the original twenty features of the PROMISE dataset. AST-LSTM is a deep learning method that the tree-structured LSTM network naturally matches the AST representation to capture the syntax and different levels of semantics in source code. DP-CNN is also a deep learning method, and it uses convolutional Neural Network to automatically learn semantic and structural features of programs.

Guided by prior work, we conduct twenty groups of CPDP experiments. Each experiment selects two versions separately from different projects, where we use one project as the training set and the other project as the test set. Table 5 presents the F1 results of BSLDP and the four baseline approaches. The highest F1 values are in bold. Take an example of the experiments where the source project is xerces-1.3 and the target project is xalan-2.5, we train model with the source project to predict defects in the target project. Our proposed model achieves a F1 of 0.688, while running DBN-CP, TCA+, AST-LSTM and DP-CNN achieve 0.572, 0.581, 0.676 and 0.562, respectively, so we find that BSLDP outperforms four baseline methods by 20.3%, 18.4%, 1.8% and 22.4%, respectively. On average, the F1 of BSLDP is 0.661, and DBN-CP, TCA+, AST-LSTM and DP-CNN achieve 0.579, 0.491, 0.500 and 0.535, respectively, thus, the proposed model outperforms them by 14.2%, 34.6%, 32.2% and 23.6%, respectively. By comparison, BSLDP achieves the biggest improvement on F1 that is 34.6% where the method of comparison is TCA+, while BSLDP achieves the smallest improvement on F1 that is 14.2% where the method of comparison is DBN-CP.

In conclusion, our proposed BSLDP improves the performance of cross-project defect prediction. The semantic features learned by ALC are effective and able to capture the common characteristics of defects across projects.

VI. THREATS TO VALIDITY

We identify the following threats to validity.

TABLE 5. Comparison of F1 among five methods including the proposed method. BSLDP denotes the proposed model. F1 denotes F1 score. The best F1 scores among five methods are highlighted in bold.

Index	Source	Target	DBN-CP	TCA+	AST-LSTM	DP-CNN	BSLDP
			F1				
1	ant-1.6	camel-1.4	0.316	0.292	0.321	0.323	**0.437**
2	jedit-4.1	camel-1.4	**0.693**	0.330	0.318	0.651	0.444
3	camel-1.4	ant-1.6	**0.979**	0.616	0.448	0.607	0.702
4	poi-3.0	ant-1.6	0.478	0.598	0.386	0.532	**0.658**
5	camel-1.4	jedit-4.1	**0.615**	0.537	0.394	0.547	0.691
6	log4j-1.1	jedit-4.1	0.503	0.419	0.389	0.423	**0.709**
7	jedit-4.1	log4j-1.1	0.645	0.574	0.574	0.656	**0.829**
8	lucene-2.2	log4j-1.1	0.618	0.571	0.578	0.632	**0.812**
9	lucene-2.2	xalan-2.5	0.550	0.530	0.680	0.540	**0.682**
10	xerces-1.3	xalan-2.5	0.572	0.581	0.676	0.562	**0.688**
11	xalan-2.5	lucene-2.2	0.594	0.561	0.750	0.621	**0.765**
12	log4j-1.1	lucene-2.2	0.692	0.524	0.750	0.663	**0.763**
13	xalan-2.5	xerces-1.3	0.386	0.394	0.340	0.391	**0.473**
14	ivy-2.0	xerces-1.3	0.426	0.398	0.261	0.421	**0.485**
15	xerces-1.3	ivy-2.0	0.453	0.409	0.264	0.467	**0.550**
16	synapse-1.2	ivy-2.0	**0.824**	0.383	0.261	0.371	0.556
17	ivy-2.0	synapse-1.2	0.433	0.570	0.530	0.456	**0.656**
18	poi-3.0	synapse-1.2	0.514	0.542	0.503	0.532	**0.625**
19	synapse-1.2	poi-3.0	0.661	0.651	0.785	0.671	**0.852**
20	ant-1.6	poi-3.0	0.619	0.343	0.785	0.627	**0.851**
Avg			0.579	0.491	0.500	0.535	**0.661**

A. THREATS TO EXTERNAL VALIDITY

In terms of external validity, the most important potential threat is that we simply test the performance of the proposed model on projects written in Java. Our model may not perform well on projects written in other languages or other projects written in Java languages. To improve the adaptability of our model to other projects, we will test our model on projects written in the Java language and in other programming languages in future.

B. THREATS TO INTERNAL VALIDITY

In this section, classification algorithms may be the threat to the internal validity. The used classifier may influence the experiment results, so we will use more classification algorithms in future. Another internal threat is the performance of semantic extractor, so we will use other semantic extractors to extract semantic information on the code in order to obtain semantic information closer to the source codes in future research.

C. CONSTRUCT VALIDITY

In this section, we only use F1 as the evaluation indicators of our model, which enables us to have a comprehensive evaluation of the proposed model. In future, we will use other evaluation indicators for a more comprehensive evaluation of the proposed model.

VII. RELATED WORK

A. CROSS-PROJECT DEFECT PREDICTION

When there is not enough labeled data to build defect prediction models, we often build models using labeled data from other projects. Therefore, cross-project defect prediction technology has aroused the discussion of many researchers where the training data and test data come from different projects.

To solve the problems described above, the researchers used other projects to build cross-project defect prediction models [50], [51], [52], [53], [54], [55]. Briand et al. [17] assessed whether fault-proneness models are applicable and can be viable decision making tools when applied from one object-oriented system to the other in a given environment. Nam et al. [18] proposed TCA+ to improve CPDP, it combined Transfer Component Analysis (TCA) with optimized TCA's normalization process, TCA+ maps source and target domain data into a feature space where data distributions between them are similar. Watanabe et al. [19] proposed an approach for CPDP that researchers changed the distribution of data for the target project by multiplying the metric of each instance on the target project by a weight. The weight is the ratio of the mean of the corresponding metric in the target project to the mean of the corresponding metric in the source data project. Nam et al. [20] and Jing et al. [21] addressed the heterogeneous data problem of cross-project defect prediction from different data processing perspectives.

The main differences between our proposed BSLDP and above approaches for cross-project prediction are as follows. First, we use the proposed semantic extractor, namely ALC, to generate semantic features from source code files, while models on the above existing approaches are based on manually designing features. Second, we use the proposed semantic extractor, namely BSL, to improve the results of classification for the features extracted by ALC.

B. DEEP LEARNING AND SEMANTIC FEATURE GENERATION IN SOFTWARE ENGINEERING

In recent years, deep learning algorithm is widely used in the field of software engineering because of its outstanding performance in feature extraction and significant achievements, such as speech recognition [22], image classification [23] and text classification [24], etc. Specifically, Lam et al. [28] proposed to combine deep learning with information retrieval technique to improve the performance in localizing buggy files for bug reports. White et al. [8] introduced learning-based detection techniques where everything for representing terms and fragments in source code is mined from the repository, and they proposed a framework, it relies on deep learning, for automatically linking patterns mined at the lexical level with patterns mined at the syntactic level. Mou et al. [27] trained a model based on syntax tree with tree-based CNN, and it preserved the structural information. Yang et al. [25] proposed an approach Deeper, and it leveraged deep learning techniques to predict defect-prone changes. They used the features to train model, and features were generated from existing features by using DBN. Wang et al. [26] further applied DBN on token vectors. These vectors are extracted from ASTs of programs for file level defect prediction, and then leveraged the learned semantic features to build machine learning models for predicting defects.

Our work differs from the above study mainly in three aspects. First, we use ALC to learn semantic information on source code, while features generated from their approach are relations between existing features. Since traditional metrics that are set manually are not effective against distinguishing snippets of code, features are combined or extracted from them and these are still ineffective in distinguishing snippets of code. Second, we use different classification algorithms to evaluate the validity of our semantic features of cross-project defect prediction. Third, our semantic extractor works directly on the source code without using an abstract syntax tree. This method is an intermediate representation of the source code.

VIII. CONCLUSION AND FUTURE WORK

Cross-project defect prediction is one of the hot topics in the field of software defect research due to the difference and insufficiency of data. To improve the performance of cross-project defect prediction, we propose a defect prediction framework called BSLDP. Specifically, we deploy ALC. This method is built by LSTM with a self-attention mechanism to learn semantic features of token vectors extracted from source code files. With the semantic and structural information preserved, this method leverages the learned semantic features to build models for predicting defects. Besides, we propose a new classifier for the semantic vectors extracted by ALC to further improve the results of classification, namely BSL.

Our experiments show that the learned semantic features could improve cross-project defect prediction compared to traditional features by logistic regression on average by 34.5% in F1. The learned semantic features by the proposed classifier improve F1 in twenty pairs of experiments compared with other two classification algorithms. On average, BSLDP outperforms four baseline methods by 14.2%, 34.6%, 32.2% and 23.6% in terms of F1 in defect prediction, respectively.

In the future, we will apply the proposed method to projects written in other programming languages. Meanwhile, this paper shows the effectiveness of applying deep learning in the field of software defect prediction, and we will try to use other deep learning-based models in the area of cross-project defect prediction. In addition, since this model only extracts semantic information on the level of source code files, we will explore the possibility of extracting semantics on the other levels, such as change level, method level, and class level.

REFERENCES

[1] 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.

[2] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 279–289.

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

[4] Y. N. Soe, P. I. Santosa, and R. Hartanto, "Software defect prediction using random forest algorithm," in *Proc. 12th South East Asian Tech. Univ. Consortium (SEATUC)*, Mar. 2018, pp. 1–5.

[5] Z. Xu, P. Yuan, T. Zhang, Y. Tang, S. Li, and Z. Xia, "HDA: Cross-project defect prediction via heterogeneous domain adaptation with dictionary learning," *IEEE Access*, vol. 6, pp. 57597–57613, 2018.

[6] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Quality J.*, vol. 23, no. 3, pp. 393–422, 2015.

[7] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 858–868.

[8] 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.

[9] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, Aug. 2017, pp. 3034–3040.

[10] X. Zhu, P. Sobhani, and H. Guo, "Long short-term memory over recursive structures," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 1604–1612.

[11] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 334–345.
[12] P. Devanbu, "On the naturalness of software," in *Proc. 6th India Softw. Eng. Conf. (ISEC)*, 2013, pp. 837–847.
[13] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," 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. 383–392.
[14] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. FSE*, 2005, pp. 306–315.
[15] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 269–280.
[16] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur, "Recurrent neural network based language model," in *Proc. 11th Annu. Conf. Int. Speech Commun. Assoc.*, vol. 2, 2010, pp. 1045–1048.
[17] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 706–720, Jul. 2002.
[18] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 382–391.
[19] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proc. 4th Int. Workshop Predictor models Softw. Eng. (PROMISE)*, 2008, pp. 19–24.
[20] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.
[21] 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.
[22] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2012, pp. 4277–4280.
[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2012, pp. 1106–1114.
[24] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2015, pp. 649–657.
[25] 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.
[26] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 297–308.
[27] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, Feb. 2016, pp. 1287–1293.
[28] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 476–481.
[29] 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.
[30] 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.
[31] 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.
[32] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Amsterdam, The Netherlands: Elsevier, 1977.
[33] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 2, pp. 308–320, Dec. 1976.
[34] 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.
[35] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.
[36] Y. Zhang and X. Lu, "A speech recognition acoustic model based on LSTM -CTC," in *Proc. IEEE 18th Int. Conf. Commun. Technol. (ICCT)*, Oct. 2018, pp. 1052–1055.
[37] J. Li and Y. Shen, "Image describing based on bidirectional LSTM and improved sequence sampling," in *Proc. IEEE 2nd Int. Conf. Big Data Anal. (ICBDA)*, Mar. 2017, pp. 735–739.
[38] Y. Zhang, "Research on text classification method based on LSTM neural network model," in *Proc. IEEE Asia–Pacific Conf. Image Process., Electron. Comput. (IPEC)*, Apr. 2021, pp. 1019–1022.
[39] T. Zhang, Q. Du, J. Xu, J. Li, and X. Li, "Software defect prediction and localization with attention-based models and ensemble learning," in *Proc. 27th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2020, pp. 81–90.
[40] T.-Y. Yu, C.-Y. Huang, and N. C. Fang, "Use of deep learning model with attention mechanism for software fault prediction," in *Proc. 8th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Aug. 2021, pp. 161–171.
[41] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (Jun. 2012). *The PROMISE Repository of Empirical Software Engineering Data*. [Online]. Available: http://promisedata.googlecode.com
[42] 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.
[43] 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)*, May 2019, pp. 46–57.
[44] 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.
[45] C. L. Prabha and N. Shivakumar, "Software defect prediction using machine learning techniques," in *Proc. 4th Int. Conf. Trends Electron. Informat. (ICOEI)*, Jun. 2020, pp. 728–733.
[46] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.
[47] H. Tong, B. Liu, and S. Wang, "Kernel spectral embedding transfer ensemble for heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1886–1906, Sep. 2021.
[48] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project defect prediction," *IEEE Access*, vol. 7, pp. 170844–170853, 2019.
[49] M. Assim, Q. Obeidat, and M. Hammad, "Software defects prediction using machine learning algorithms," in *Proc. Int. Conf. Data Analytics Bus. Ind., Way Towards Sustain. Economy (ICDABI)*, Oct. 2020, pp. 1–6.
[50] D. Chen, X. Chen, H. Li, J. Xie, and Y. Mu, "DeepCPDP: Deep learning based cross-project defect prediction," *IEEE Access*, vol. 7, pp. 184832–184848, 2019.
[51] S. Tang, S. Huang, C. Zheng, E. Liu, C. Zong, and Y. Ding, "A novel cross-project software defect prediction algorithm based on transfer learning," *Tsinghua Sci. Technol.*, vol. 27, no. 1, pp. 41–57, Feb. 2022.
[52] M. F. Sohan, M. I. Jabiullah, S. S. M. M. Rahman, and S. M. H. Mahmud, "Assessing the effect of imbalanced learning on cross-project software defect prediction," in *Proc. 10th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2019, pp. 1–6.
[53] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, May 2018, p. 1063.
[54] Z. Yuan, X. Chen, Z. Cui, and Y. Mu, "ALTRA: Cross-project software defect prediction via active learning and tradaboost," *IEEE Access*, vol. 8, pp. 30037–30049, 2020.
[55] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, "Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 786–802, Mar. 2022.

WANZHI WEN (Member, IEEE) received the B.S. degree from Anhui Normal University, Anhui, China, in 2004, and the Ph.D. degree in computer software and theory from Southeast University, Nanjing, Jiangsu, China, in 2013. He is currently an Associate Professor with Nantong University, Nantong, Jiangsu. His research interests include software fault localization, change impact analysis, and software defect prediction.

RUINIAN ZHANG received the B.S. degree in computer science and technology from the Nanjing University of Information Science and Technology, in 2020. He is currently pursuing the master's degree with the School of Information Science and Technology, Nantong University. His research interests include software defect prediction, machine learning, and software security.

CHUYUE WANG was born in Lianyungang, Jiangsu, China, in 2002. She is currently pursuing the degree with the School of Information Science and Technology, Nantong University. Her research interests include intelligent software engineering and program analysis.

CHENQIANG SHEN was born in Suzhou, Jiangsu, China, in 1999. He received the B.S. degree from the School of Software Engineering, Xinglin College of Nantong University, Nantong, China, in 2021, where he is currently pursuing the master's degree with the School of Electronic information. His research interests include software defect prediction and software fault localization.

MENG YU was born in Huaian, Jiangsu, China, in 2002. She is currently pursuing the degree in computer science and technology with Nantong University. Her research interests include software engineering and computer networks.

SUCHUAN ZHANG was born in Yancheng, Jiangsu, China, in 2002. He is currently pursuing the bachelor's degree with the School of Information Science and Technology, Nantong University, China. His research interests include artificial intelligence and software engineering.

XINXIN GAO was born in Bozhou, Anhui, China, in 2002. He is currently pursuing the bachelor's degree with the School of Information Science and Technology, Nantong University, China. His research interests include code recommendation and software defect prediction.