Two-Stage AST Encoding for Software Defect Prediction

Yanwu Zhou[a], Lu Lu[a,*], Quanyi Zou[b], Cuixu Li[c]
[a] School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[b] School of Software Engineering, South China University of Technology, Guangzhou, China
[c] Guangdong Meiweixian Flavoring Foods Co.,Ltd., Zhongshan, China
*Corresponding author email: lul@scut.edu.cn

***Abstract*—Software defect prediction (SDP) can find potential containing defect modules, which assists software developers in allocating limited test resources more efficiently. Because traditional software features fail to capture the semantics of source code, various studies have turned to extracting deep learning features. Existing related approaches often parse the program source code into Abstract Syntax Trees (ASTs) for further processing. However, most of these approaches ignore AST nodes' hierarchical and position-sensitive structure. To overcome the aforementioned issues, a two-stage AST encoding (TSE) method is proposed in this paper for software defect prediction. Experiments on eight Java open-source projects showed that our proposed SDP method outperforms several traditional methods and state-of-the-art deep learning methods in terms of F-measure and MCC.**

***Index Terms*—software defect prediction, abstract syntax tree (AST), two-stage encoding, positional encoding**

I. Introduction

As software evolves, the scale and complexity of the system grow dramatically. Under limited time and resource constraints, software becomes more prone to defect [1]. Software defect prediction (SDP) is a promising technology for improving software reliability by detecting program defect modules and prioritizing testing efforts [2]. The goal of SDP is to train a defect predictor that classify code instances as defective or not. It is a requirement for SDP to create defect-distinguishable software representations [3]. Software metrics have been proposed and broadly used in SDP [4]. However, Software metrics are hand-crafted by software specialists and most software metrics focus on the statistical aspects of code, ignoring the semantic characteristics of code.

In recent years, researchers have begun to leverage deep neural networks to exact software features from source code [5]. Related works have revealed that Abstract Syntax Trees (ASTs) are suitably representative of programs' well-defined syntax [6]–[8]. It's common practice to parse source code files into ASTs and then convert into token vectors by traversing the ASTs.

Pre-order traversal is adopted by most studies for AST conversion, treating all nodes as the same level and creating a token corpus for word embedding technology. It has two drawbacks: 1. Not all the nodes are at the same level. The coarse-grained information of node needs to be supplemented by the fine-grained information of its child nodes. 2. If only preorder traversal is employed for AST, the tree structure's positional information will be lost. Furthermore, there is a sequential positional relationship between nodes at the same depth.

In order to tackle the above-mentioned first drawback, it is feasible to decompose the AST into non-overlapping subtrees according to node granularity. For example, Zhang et al. propose ASTNN, which divides the AST into subtrees at the statement level and handles the subtree interior and subtree sequence independently [9]. The second drawback can be addressed by including the additional position information of the tree structure.

In this paper, we mark two types of nodes with different granularities, named ordinary nodes and block nodes, and then execute a two-stage encoding, according to the decomposition strategy. In the first stage, the word embedding and positional embedding of ordinary nodes under the subtree rooted by the block node are aggregated to the block node through the self-attention mechanism to represent the encoding of the block node. In the second stage, the encodings of the block nodes in an AST are collected. The tree structure of block nodes is retained, and the encodings are fed into a Tree-based LSTM network to generate the final AST representation for software defect prediction.

In summary, the main contributions of this paper are listed as follows:

1) We propose an AST decomposition strategy that marks AST nodes as two types of nodes according to the hierarchy of the AST.
2) We apply the strategy on a two-stage bottom-up AST encoding for software defect prediction. Experimental results indicate that our method outperforms other software defect prediction models in terms of F-measure and MCC.

II. Related work

A. Code Representation in Software Defect Prediction

The representation of software code is a critical part of software defect prediction. Wang et al. proposed to parse the source code into ASTs, and then encode it into numerical vectors as code representation [6]. Li et al. concatenate deep

DOI reference number: 10.18293/SEKE2022-039

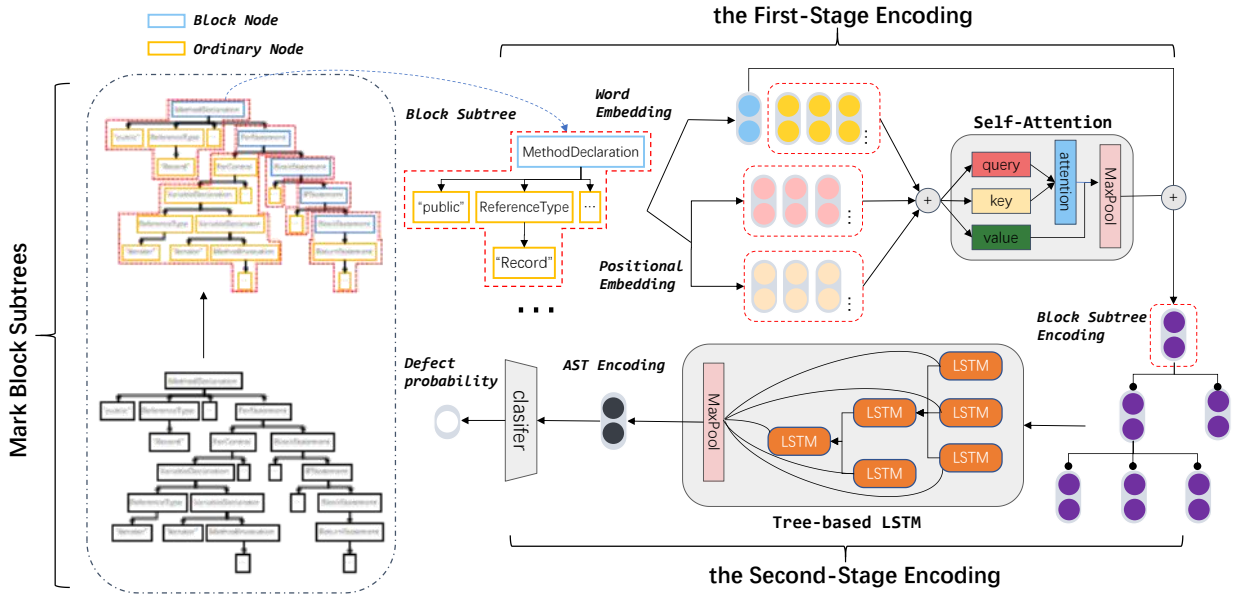

Fig. 1. The overall framework of TSE.

semantic features with code metrics to augment the information of code [7]. In addition to the code representation based on AST, Phan et al. construct the Control Flow Graph (CFG) of the program at the level of assembly instruction [10]. The effectiveness of code visualization is demonstrated by Chen et al., who convert each code file in the program into a two-dimensional image. [11].

B. Deep Learning in Software Defect Prediction

There has been a slew of new deep learning applications for SDP. Wang et al. built the Deep Believe Network (DBN) for extracting features [6]. Li et al. considered that CNN can seize local patterns of sources code more effectively [7]. Xu et al. employed graph neural networks to capture the latent defect information [12]. Wang et al. proposed a modular tree network to dig the semantic differences among different types of AST substructures which shows the advantage of more elaborate structure information extraction [13].

III. Method

In this section, we elaborate on our method in detail. Figure 1 demonstrates the overall framework of our TSE method. Before the encoding stages, the code file is parsed into an AST, and then marked and converted into vectors. In the first stage, the self-attention layer is utilized to aggregate the word embeddings and positional encodings of all ordinary nodes to construct the block node encodings. In the second encoding stage, the block node encodings are fed into a Tree-based LSTM and produce the AST's final encoding by max pooling. Finally, the probability that the code file is defective is output via a fully-connected layer.

A. Marking AST nodes and Converting into Vectors

The open-source code tool, *javalang*[1], is used to parse the code files into ASTs under the token granularity. And then the block nodes and ordinary nodes are marked during the pre-order traversal of the AST. The set of block nodes and ordinary nodes are difined as B and O, separately. Block nodes are listed in the Table I and ordinary nodes are chosen according to Wang's work [6]. Note that the attribution strings of the AST node are also regarded as ordinary nodes. Most of the block nodes chosen create a local scope with specific context. The nodes in the *IfStatement* block, for example, are in the specific context of conditional judgment; hence, the meaning of the nodes of *IfStatement* may differ while in other statement blocks. There is a particular node in the block nodes called *StatementExpression*, which is a statement-level node. This node can further disassemble the statement block and tackle the problem of an excessively large subtree.

An AST T is defined as a collection of its block subtrees:

$$T = [bT_1, bT_2, ..., bT_n] \tag{1}$$

Each block subtree is defined as:

$$bT_i = [b_i, [oT_{i1}, oT_{i2}...]] \tag{2}$$

where $b_i \in B$ is the root node of the block subtree, oT_{ij} is the ordinary subtree under bT_i. An ordinary subtree is denoted as:

$$oT_{ij} = [o_{ij1}, o_{ij2}, ...] \tag{3}$$

where $o_{ijk} \in O$.

After acquiring the marked AST, we utilize word2vec [14] technique to convert block nodes and normal nodes into E-dimensional vectors, denoted as x_i^b and x_{ijk}^o, respectively.

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

B. First Stage: Aggregating Ordinary Node Encodings to The Block Node Encoding

In the first stage, the self-attention mechanism is employed to aggregate their information onto the block nodes as the block node encodings:

$$Atten(X) = Atten(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (4)$$

$$Q = W^Q x_q, K = W^K x_k, V = W^V x_v \quad (5)$$

where W^Q, W^K and W^V are weight matrices for queries, keys and values, and d_k is the dimension of x_k. In the case of self-attention, x_q, x_k and x_v are identical. The self-attention mechanism can flexibly assign the attention weights of the same nodes in different contexts and mask the difference in the number of nodes in a subtree. Self-attention mechanism is not sensitive to the position of nodes. However, the positional relationship of nodes in the subtree could be a significant factor in defect prediction. Therefore, we additionally traverse the block subtree hierarchically and record the positional information of each ordinary node which consists of the depth of the node in the block subtree and the sequence number of the node at the depth. Through an embedding layer, their corresponding positional vectors can be represented separately as x^{depth} and x^{num}.

Given bT_i with N_i nodes, We can obtain the embedding sequence:

$$X_i^{bT} = [x_i^b, x_{i11}^o + x_{i11}^{depth} + x_{i11}^{num}, ..., x_{ijk}^o + x_{ijk}^{depth} + x_{ijk}^{num}] \quad (6)$$

Then the encoding of a subtree $en_i^{bT} \in R^D$ is calculated by:

$$en_i^{bT} = (W^{en})^T x_i^b + Maxpool(Atten(X_i^{bT}|x_i^b)) + b^{en} \quad (7)$$

where D is the encoding dimension, $W^{en} \in R^{E \times D}$ is the weight matrix, b^{en} is the bias term, $Maxpool(.)$ is the max pool layer with the kernel size N_i, $Atten(.)$ is the self-attention layer, $X_i^{bT}|x_i^b$ is the sequence X_i^{bT} excluding x_i^b.

C. Second Stage: Aggregating Block Node Encodings to The AST Encoding

In the first encoding stage, the encodings of all block nodes in an AST are collected:

$$X^T = [en_1^{bT}, ..., en_n^{bT}] \quad (8)$$

Tree-based LSTM is adopted as the encoder to acquire the final AST's encoding, preserving more information about the AST's tree structure. Since all ordinary nodes in an AST have been processed in the first encoding stage, the scale of the AST has been considerably reduced, hence avoiding gradient vanishing induced by long-term dependencies. Specifically, we utilize ChildSum Tree-LSTM [15] to calculate the AST's final encoding by max-pooling the hidden states of block nodes:

$$en^T = Maxpool(TreeLSTM(X^T)) \quad (9)$$

Then we simply use a fully-connected layer as the defect classifier. Cross entropy loss is adopted for optimization.

TABLE I
THE CHOSEN TYPES OF BLOCK NODES

Block Nodes	ClassDeclaration, InnerClassDeclaration, MethodDeclaration, ConstructorDeclaration, BlockStatement, ForStatement, WhileStatement, SwitchStatement, IfStatement, DoStatement, **StatementExpression**

IV. Experimental Settings

A. Evaluated Projects and Datasets

To evaluate the the effectiveness of our TSE approach, we choose publicly available projects from PROMISE repository [16]. Specifically, eight open-source Java projects are exploit in our experiments, which are *ant*, *camel*, *jedit*, *log4j*, *lucene*, *poi*, *xalan* and *synapse*.

B. Evaluation

In this paper, the popular evaluation indicator in SDP, F-measure and MCC, are adopted to evaluate our proposed method. Specifically, F-measure is a harmonic mean of Precision and Recall and MCC is a relatively balanced measure considering diverse indicators, which are calculated by the following equations:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN} \quad (10)$$

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall} \quad (11)$$

$$MCC = \frac{TP * TN - FP * FN}{(TP + FP)(TP + FN)(TN + FP)(TN + FN)} \quad (12)$$

where TP, FN, FP and TN can be derived by the confusion matrix.

C. Baseline Setting

In this paper, our TSE method are compared with the following methods:

1) LR: A traditional method using logistic regression classifier on handcraft defect metrics.
2) SVM: A traditional method using SVM classifier with the Gaussian kernel on handcraft defect metrics.
3) DBN: A deep learning method employing a standard DBN model to extract semantic features from AST for SDP [6].
4) LSTM: A deep learning method using LSTM to capture semantic representations for SDP.
5) CNN+: An enhanced CNN-based method by combining traditional feature and deep feature for SDP [7].

D. Parameters Setting

10-fold cross validation is used to split dataset and each training task is repeated 10 times in the experiment. The AST tokens and the positional vectors are embedded into 100-dimensional vectors; The query, key, and value dimensions are all set to 100 in the self-attention layer; The hidden dimension of the ChildSum Tree-LSTM is 100.

TABLE II
F-MEASURE AND MCC VALUE FOR TSE VERSUS BASELINE METHODS

Task	LR		SVM		DBN		LSTM		CNN+		TSE	
	F-measure	MCC	F-measure	MCC	F-measure	MCC	F-measure	MCC	F-measure	MCC	F-measure	MCC
ant	**0.568**	0.427	0.561	0.417	0.371	0.220	0.533	0.214	0.566	**0.437**	0.505	0.403
camel	0.352	0.167	0.356	0.177	0.359	0.171	0.370	0.199	0.335	0.180	**0.501**	**0.389**
jedit	0.552	0.388	0.534	0.365	0.504	0.174	0.551	0.345	0.561	**0.451**	**0.572**	0.438
log4j	0.409	**0.301**	0.338	0.250	0.611	0.132	**0.633**	0.162	0.593	0.219	0.628	0.255
lucene	0.530	0.161	0.600	0.141	0.574	0.161	0.605	0.171	0.643	0.236	**0.654**	**0.243**
poi	0.589	0.238	0.791	0.479	0.626	0.310	0.752	0.314	0.748	0.328	**0.778**	**0.361**
xalan	0.532	0.101	0.545	0.113	0.592	0.077	0.602	0.124	0.635	0.176	**0.653**	**0.184**
synapse	0.506	**0.294**	0.480	0.292	0.413	0.163	0.344	0.118	0.369	0.257	**0.550**	0.282
Average	0.501	0.249	0.523	0.270	0.506	0.173	0.549	0.205	0.556	0.275	**0.612**	**0.314**

V. EXPERIMENTAL RESULTS

Table II record comparison results of the F-measure and MCC indicators for the TSE method versus other baseline methods. According to the last second line of the two tables, our proposed TSE achieves F-measure as 0.612 and MCC as 0.314 on average value and obtains the best average value on the two indicators. Compared to the traditional machine learning methods, i.e., LR and SVM, which utilize statistical defect metrics, our TSE method has an average improvement of 22.2% and 16.9% in the F-measure indicator, and 26.1% and 16.0% in the MCC indicator. Compared to the deep learning methods, i.e., DBN, LSTM and CNN+, our TSE method has an average improvement of 18.8%, 11.3% and 10.1% in terms of F-measure, and 81.4%, 52.9% and 14.0% in terms of MCC. The above experiments suggest that TSE outperforms traditional machine learning methods and AST-based semantic extracting methods on eight separate projects.

VI. CONCLUSION

In this paper, a two-stage AST encoding method is proposed, which employs a bottom-up encoding to learn the semantic information software defect prediction. The main advantages of TSE are 1. executing the encoding following the natural hierarchy of ASTs. 2. combining the tree positional encoding to augment the structural information of ASTs. The performance of TSE is evaluated by comparing it with traditional methods and state-of-the-art deep learning methods in terms of F-measure and MCC. Experimental results show that TSE achieves better performance versus all baseline methods. In future work, we will verify the performance of our method on other programming languages and repositories.

ACKNOWLEDGMENT

This work was supported in part by the Zhongshan Produce and Research Fund, PR China under grant no. 210602103890051.

REFERENCES

[1] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: do different classifiers find the same defects?," *Software Quality Journal*, vol. 26, no. 2, pp. 525–552, 2018.

[2] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, vol. 47, no. 01, pp. 67–85, 2021.

[3] H. Wang, W. Zhuang, and X. Zhang, "Software defect prediction based on gated hierarchical lstms," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 711–727, 2021.

[4] M. Halstead, "Elements of software science (operating and programming systems series)," 1977.

[5] N. Zhang, S. Ying, K. Zhu, and D. Zhu, "Software defect prediction based on stacked sparse denoising autoencoders and enhanced extreme learning machine," *IET Software*, 2021.

[6] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.

[7] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328, IEEE, 2017.

[8] X. Zhou and L. Lu, "Defect prediction via lstm based on sequence and tree structure," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 366–373, IEEE, 2020.

[9] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, IEEE, 2019.

[10] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 45–52, IEEE, 2017.

[11] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov, "Software visualization and deep transfer learning for effective software defect prediction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 578–589, 2020.

[12] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 613–625, 2020.

[13] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[15] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[16] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM/IEEE international symposium on empirical software engineering and measurement*, pp. 45–54, IEEE, 2013.