Defect Prediction via LSTM Based on Sequence and Tree Structure

Xuan Zhou†, Lu Lu*
Computer Science and Engineering
South China University of Technology
GuangZhou, China
Email: †emmyzhou@foxmail.com, *lul@scut.edu.cn

***Abstract*—With the ever-expanding spread of contemporary software, software defect prediction (SDP) is attracting more and more attention. However, sequential networks used in previous studies, weaken syntactic information and fail to capture long-distance dependencies. To solve these problems, we develop a long short-term memory network based on bidirectional and tree structure (LSTM-BT). Specifically, LSTM-BT combines bidirectional long short-term memory networks (BI-LSTM) and tree long short-term memory networks (Tree-LSTM) to capture semantic and syntactic features from source codes. First, token vectors are captured from the abstract syntax tree (AST). Second, an embedding layer is used to extract semantic information hidden inside the AST nodes. Last, features are fed to the LSTM-BT, which is used to conduct predictions of defect-proneness. To validate our method, we carried out experiments on 8 pairs of Java open-source projects and the results show that LSTM-BT performs better compared to several state-of-the-art defect prediction models.**

***Keywords*—Software defect prediction, Long short-term memory networks, Deep learning, Two-channel framework**

I. Introduction

Software defect prediction (SDP) is a procedure for predicting whether software entities are defective. Over the past few years, SDP has received much attention, and it has become one of the most important research fields in software engineering. It is typically believed that fixing bugs after deployment is more expensive than dealing with them in the development process [1]. Due to the huge cost, improving the ability to predict defects is very important [2]. Thus, an excellent SDP model is necessary. It can be utilized to support a better configuration of limited test resources [3] and assist in locating possible faults. Figure 1 demonstrates the process of SDP. The process is divided into four steps: obtaining data, representing features, training models, and applying models.

In traditional SDP methods, many metrics have been raised, including traditional metrics, object-oriented metrics, process metrics, and resource metrics [4]. However, most of them are hand-crafted and often ignore the structure of codes. Furthermore, collecting hand-crafted metrics is a time-consuming and error-prone task.

Recently, using deep learning (DL) to extract semantic and syntactic features has become popular [5], [6]. These methods, which are based deep belief network (DBN) or convolutional neural network (CNN), are novel but show three drawbacks: (1) they encode abstract syntax nodes by artificially assigning real numbers to abstract syntax nodes, which leads to the loss of information hidden in AST nodes because the distance between two different nodes become unmeasurable; (2) although they capture features from the abstract syntax tree (AST), they feed the features to a sequential network, which weakens structural information; (3) since both DBN and CNN require input vectors with a fixed length, they append 0 to make their lengths are the same as the longest vector, which will dilute the importance of effective information.

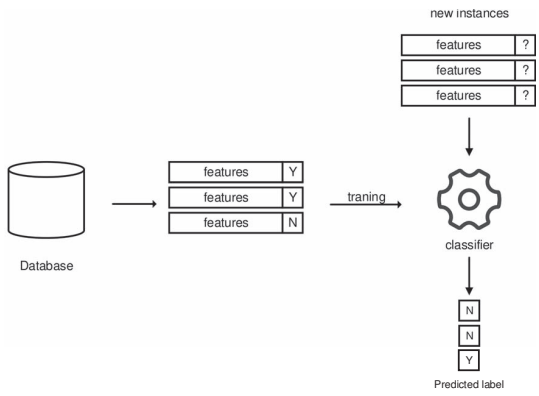


Fig. 1. The schematic diagram of software defect prediction process

Moreover, as reported by DL researchers, long short-term memory network (LSTM) is more capable of handling indefinite sequences tasks and detecting long-distance dependencies such as code snippet 1. In code snippet 1, it will encounter an exception when calls start() of the same thread again. However, due to lots of code lines between the two start() calls, other models may not be able to detect the former call and therefore cannot detect defects.

```
1 static void myFunc(Thread thread){
2   thread.start();
3   //Lines of unrelated code
4   //Call the start() of the same thread again
5   thread.start();
6 }
```

Code Snippet 1. A motivating example

In this paper, we propose our approach, called a long short-term memory network based on bidirectional and tree structure (LSTM-BT). To make the difference among nodes

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00055

quantifiable, a word-embedding layer is added to LSTM-BT. Besides, a bidirectional long short-term memory network (BI-LSTM) shows good performance because it combines past and future information, whereas a tree long short-term memory network (Tree-LSTM) can capture syntactic information well by preserving the structure of the AST. To fully combine the two aspects of information, we propose a two-channel framework to automatically capture features by BI-LSTM and Tree-LSTM. Then a softmax output layer is used to classify software entities respectively. After that, we get the final prediction result of classification through logistic regression (LR).

Furthermore, according to prediction granularity, each software entity can represent a method, a class, a file, a package or a change [7]. In this paper, we select files as the representation granularity. In fact, the file-level is a widely used representation in SDP tasks [8]–[12].

The main contributions of our study are as follows:

- Although different topologies of LSTM have been proposed, to our best knowledge, this is the first time to combine BI-LSTM and Tree-LSTM to obtain multiple aspects of information to improve the performance of SDP.
- We validate the effectiveness of LSTM-BT by conducting experiments on 8 pairs of Java open-source projects with well-established labels. Experiments show that LSTM-BT performs better in MCC and AUC compared to other defect prediction models.

The remainder of this paper is structured as follows. Section II presents the related work. In Section III, we introduce our proposed method and explicate its steps in detail. Section IV shows the experimental particulars, including datasets, baseline methods, evaluation measures, settings and results. Next, we have a discussion about why LSTM-BT works and threats of effectiveness in Section V. Conclusion and prospect for future work are reported in Section VI.

II. Related work

In early SDP studies, machine learning is most commonly used on elaborately designed features. [13] presented structural scattering and semantic scattering to capture human characteristics and project characteristics as resource metrics. [14] leveraged cross-entropy, which carries information about the difference between two probability distributions as a new feature. However, these previous studies focused only on certain pre-selected features [15], and all these features are designed by humans and often ignore the structure of codes. Moreover, static hand-crafted features, which may be redundant or irrelevant, can reduce the performance [16]. Different from the aforementioned approaches, LSTM-BT, using DL, automatically captures generalized features from AST nodes.

Recently, DL has been applied in SDP, [17] used DBN on 14 pre-selected features. To mine source-level semantic and syntactic features, AST and control flow graph (CFG) [18] can be used to represent the source-level information. [5] leveraged DBN to capture features automatically from AST while [6] used CNN, and the promising results showed that semantic and syntactic features can contribute to the performance of SDP.

LSTM, proposed by [19] as a special recurrent neural network (RNN), can capture long-distance dependencies. Many different variants [20] and topologies have appeared after LSTM was proposed. In recent years, LSTM has received more and more attention in different fields. The emergence of LSTM has made many DL problems solvable [21]. [22] used LSTM for document classification, [23] designed a lexicon-enhanced LSTM to solve the inaccuracy of word representation in sentiment analysis tasks. Most recently, researchers tried to employ LSTM for SDP. [24] employed BI-LSTM to get a transfer representation whereas [25], [26] employed it to get features in within-project defect prediction. [27] constructed a deep tree-based model for SDP.

BI-LSTM takes both past and future information at the same time, while Tree-LSTM values syntactic information well. Therefore, we present a two-channel framework for SDP.

III. Methods

In this section, we demonstrate LSTM-BT in detail. Specifically, LSTM-BT contains four steps: (1) parsing source code; (2) embedding AST nodes; (3) training with improved LSTM approach; and (4) trade-off between BI-LSTM and Tree-LSTM.

A. Parsing source code

We parse each source file into an AST by an open-source tool[1]. Figure 2 shows an AST generated from a sample snippet of Java code.

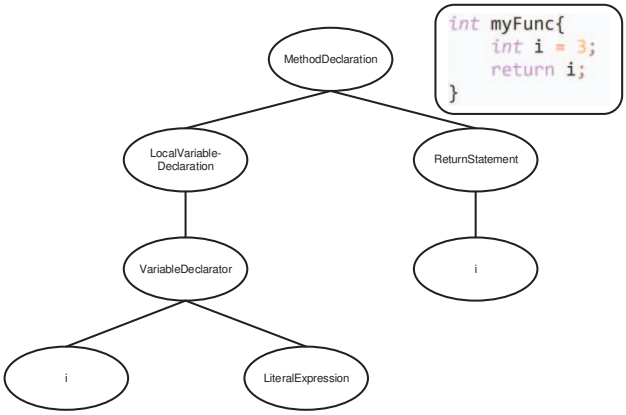


Fig. 2. An example of AST generated from a snippet of Java code

All of our selected nodes are shown in Table I. Similar to previous research [12], we select 35 different types of AST nodes. For clarity, they are divided into four categories: 1) nodes of method invocations and class instance creations. On the one hand, in a project, the same name usually indicates the same class or method. On the other hand, different classes

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

or methods tend to have dissimilar defect-proneness. Hence, nodes of this type are identified by their method name or class name; 2) declaration-related nodes, which are also identified by their names; 3) control-flow-related nodes, which represent the main part of the entire flow. As enough information can be obtained from their node types, they are only identified by their node types; and 4) other nodes, including various references and expressions.

The AST nodes which not in these four categories, such as assignment and intrinsic type declaration, are excluded, because they are often too method-specific or class-specific to be generalized to the whole project. Adding them will weaken the importance of other nodes.

TABLE I
THE SELECTED TYPES OF AST NODES

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

B. Embedding AST nodes

Since LSTM can only accept real vectors as input, we have to convert AST nodes into real vectors. Most of the existing work assigned AST nodes to meaningless real numbers, which makes the distance between two different nodes unmeasurable. So, we add an embedding layer to parse AST nodes to a high dimension vector through an embedding matrix. The process can be expressed using the following mapping:

$$f(M) : nodes \rightarrow R^n \tag{1}$$

where *M* is the embedding matrix. For every specific node in the node-set, the mapping $f(M)$ is a parameterized function that maps each token into an n-dimensional vector, where n is a super-parameter representing the converted vector length.

The embedding matrix is initialized randomly and will be updated through training. A trained matrix can capture semantic features hidden AST nodes well. That is, the Euclidean distance between two vectors represents the semantic difference between the two nodes.

C. Training with improved LSTM approach

To enhance the performance of LSTM and adapt it to SDP, we improve the LSTM in several ways:

1) We combine BI-LSTM with Tree-LSTM for SDP. Figure 3 illustrates the network architecture of LSTM-BT. [28] first proposed the naturalness hypothesis, i.e., programs are rather repetitive and mostly simple, which gives them helpful predictable statistical properties that can be extracted from statistical language models and used for software engineering tasks. We can learn that natural language and programs have certain inherent similarities and that two-way information in programs is also meaningful. Thus, BI-LSTM is used as a sequential network to combine information from both past and future contexts. Additionally, in existing DL-generated-feature-based methods for SDP [5], [6], [25], although features are captured from AST, they are only fed to a sequential network, which might weaken the original syntactic information. Therefore, we introduce Tree-LSTM to extract the original syntactic information.

2) Plenty of experiments were conducted by [29] and concluded that forget-gates hold the greatest importance in LSTM. To fully exploit the function of forget-gates, according to biologically plausible to a certain extent in LSTM [30], we borrow from the biological mechanism by which neurons determine whether to trigger the threshold for action potentials. We replace forget-gates with excite-gates and inhibit-gates. When the cell receives a new input, the input will through both of these gates. Gate is a way to optically let information through, which means that the information through excite-gates represents the excitatory signal, and the information through inhibit-gates represents the inhibitory signal. The outputs from these two gates work together to determine whether to forget or not.

Figure 4 displays the structure of the LSTM cell we used. The cell state and the hidden state will be updated according to the following formulas:

$$\begin{aligned}
e_t &= \sigma(W_e \cdot [h_{t-1}, x_t] + b_e) \\
inh_t &= \sigma(W_{inh} \cdot [h_{t-1}, x_t] + b_{inh}) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
f_t &= e_t * inh_t \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t * tanh(C_t)
\end{aligned} \tag{2}$$

In the above equations, $W_e, W_{inh}, W_i, W_c, W_o$ represent the weight matrices of excite-gates, inhibit-gates, input-gates, cell-gates, and output-gates respectively. Similarly, b_* is the bias of each gate. C_* and h_* are the cell state and hidden state of the corresponding moment.

BI-LSTM runs two LSTMs: one handles the sequential data from start to end, and the other one handles it from end to start. A fully connected layer is added to combine

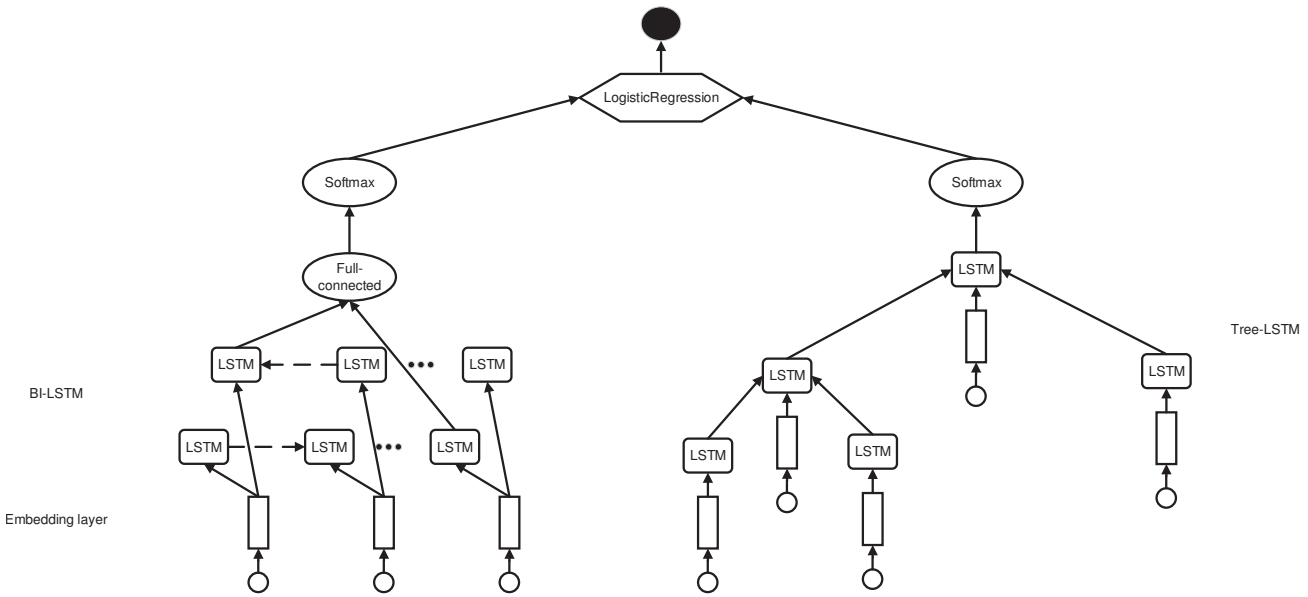


Fig. 3. The network architecture of LSTM-BT

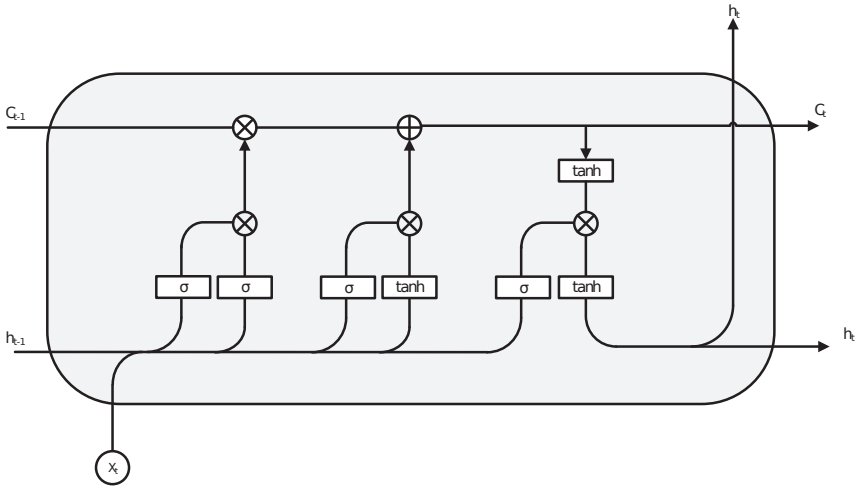


Fig. 4. The internal structure of proposed LSTM cell

$P(W_i = w|W_1, ..., W_{i-1})$ and $P(W_i = w|W_n, ..., W_{i+1})$ simultaneously.

Because the Tree-LSTM cell accepts input from its children, the input hidden state takes the mean of the output hidden state of all its child nodes. For a specific node, the input hidden state can be shown as:

$$h_t = \frac{\sum h_{ci}}{|children|} \tag{3}$$

where h_{ci} denotes the i_{th} child's output hidden state, $|children|$ represents the number of the specific node's children.

Specifically, we can predict the label of the parent node by all its children's hidden states through the following equation:

$$P(W_i = w|W_{c1}, ..., W_{cn}) = \frac{exp(U_i h_t)}{\sum_{w'} exp(U_{w'} h_t)} \tag{4}$$

where U_* is a free parameter.

For the training process, we select binary cross-entropy (BCE) loss, which is defined as:

$$l(x, y) = L = \{l_1, l_2, ..., l_n\} \tag{5}$$

where n represents the batch size, and l_i denotes the i_{th} instance's cost:

$$l_i = -w_i[y_i log\hat{y}_i + (1 - y_i) log(1 - \hat{y}_i)] \tag{6}$$

For a specific sample, w_i indicators the weight, y_i and $\hat{y}_i$ represent the real value and the predicted value respectively. The training process tries to minimize the loss function.

D. Trade-off between BI-LSTM and Tree-LSTM

Considering the difference in importance between the two kinds of information, we add a logistic regression layer after two different LSTMs, same as the baseline methods. Parameters in the logistic regression layer will be updated during training to automatically determine the weight.

In addition, the class imbalance is an intrinsic problem in SDP. Researchers [31] have performed experiments and concluded that the methods without processing the class imbalance problem resulted in comparatively unsatisfactory performance. Given the above situation, we adopt cost-sensitive learning in our framework, the penalty of the i_{th} class can be expressed as follows:

$$C_i = \frac{Num_{entity}}{Num_{(entity\_with\_i\_label)} * Num_{class}} \tag{7}$$

where Num_{entity} means the total number of entities, $Num_{(entity\_with\_i\_label)}$ and Num_{class} represent the number of entities with i label and the number of classes respectively.

From the above equation, since the Num_{entity} and Num_{class} are fixed values in any specific dataset, we can know that the penalty of the minority class is higher than the majority class. Although this method does not directly perform operations at the data level, such as over-sampling and under-sampling, it indirectly affects the direction of learning due to different penalties.

IV. Experiments

This section demonstrates the particulars of experiments, which mainly consist of experiment datasets, baseline methods, measures, and results. Our implementations, including LSTM-BT and baseline methods, are based on Pytorch[2].

A. Datasets

To evaluate the performance of LSTM-BT, we selected 8 pairs of Java open-source projects, which shown in Table II and widely used in SDP [8] [14] [31]. The older version is for training and the newer one for testing. The corresponding PROMISE data [32], including defect labels and static features, and source code are public. The 20 traditional static features are shown in Table III.

B. Baseline

We compared LSTM-BT with six models.

- Methods only using hand-crafted features in Table III.
  - **LR:** A traditional method for SDP.
  - **RF:** Random forest, a classification method commonly used in machine learning.
- Methods using DL-generated features. For fairness, we adopted the super-parameter settings described in the corresponding papers.

[2]https://pytorch.org/

TABLE II
DATASET CHARACTERISTICS

Project	Versions	Avg. Files	Avg. Buggy Rate(%)
jedit	4.0 4.1	309	25.0
log4j	1.0 1.2	170	58.7
lucene	2.0 2.2	221	53.2
poi	1.5 2.0	276	35.6
synapse	1.0 1.2	202	23.0
velocity	1.5 1.6.1	443	49.7
xalan	2.5 2.6	844	47.3
xerces	1.2 1.3	447	15.7

  - **DBN:** A model using DBN to capture semantic and syntactic information for SDP [5].
  - **CNN:** A method automatically captures semantic and syntactic information through standard CNN based on text sequence convolution.
  - **DBN+:** An enhanced DBN by combining pre-selected features and DL-generated features.
  - **CNN+:** An enhanced CNN for SDP [6].

C. Measures

Evaluation measures play crucial roles in predictive performance. Tabel IV shows the confusion matrix, in which the values stored are used to figure out widely-used evaluation measures in dichotomous classifiers. However, F1-score, as the commonly used measure in SDP, excludes the true negatives (TN), which may be problematic [33] because SDP is definitely concerned about whether instances are truly non-defective. Thus, we selected the Matthews Correlation Coefficient (MCC) as a measure. MCC, a correlation coefficient between the actual classification and predicted classification, has comprehensive consideration of various indicators. Its value ranges in [-1,1], higher values mean better results. MCC is calculated by the following formula:

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

Moreover, when computing MCC, a critical value is required to determine whether the instance is defective. The critical value can affect the performance and the default value (i.e. 0.5) may not be the best [34]. Thus, we also adopted the Area Under Curve (AUC), which is unrelated to critical value and has good tolerance for imbalanced datasets, to evaluate the performance of approaches.

AUC is defined as the area under the receiver operating characteristic curve (ROC). ROC is a curve of the false positive rate (FPR) against the true positive rate (TPR). The FPR and TPR can be represented in Equation 8 and 9. From the above definition, it can be seen the AUC value ranges in [0, 1], the higher AUC implies a better result, and 0.5 in AUC means that the effect of the classifier is the same as the random guessing.

TABLE III
20 STATIC METRICS, THE DESCRIPTIONS OF ATTRIBUTES ARE REFERRED TO [25]

Metrics	Description
WMC	The number of methods in the class
DIT	The position of the class in the inheritance tree
NOC	The number of immediate descendants of the class
CBO	The value increases when the methods of one class access services of another
RFC	Number of methods invoked in response to a message to the object
LCOM	Number of pairs of methods that cannot share a reference to an instance variable
LCOM3	another type of lcom metric
NPM	The number of all the methods in a class that are declared as public
DAM	Ratio of the number of private (protected) attributes to the total number of attributes
MOA	The number of data declarations (class fields) whose types are user-defined classes
MFA	Number of methods inherited by a class plus number of methods accessible by member methods of the class
CAM	Summation of the number of different types of method parameters in every method divided by the multiplication of the number of different method parameter types in whole class and number of methods
IC	The number of parent classes to which a given class is coupled
CBM	Total number of new/redefined methods to which all the inherited methods are coupled
AMC	The number of JAVA byte codes
Ca	How many other classes use the specific class
Ce	How many other classes are used by the specific class
Max(CC)	Maximum McCabe's cyclomatic complexity values of methods in the same class
Avg(CC)	Average McCabe's cyclomatic complexity values of methods in the same class
LOC	Measures the volume of the code

TABLE IV
THE CONFUSION MATRIX

	Predicted defective	Predicted non-defective
Actual defective	True Positive (TP)	False Negative (FN)
Actual non-defective	False Positive (FP)	True Negative (TN)

$$FPR = \frac{FP}{FP + TN} \tag{8}$$

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

D. Settings and results

To evaluate the performance of LSTM-BT, we perform the experiments on the 8 pairs of Java open-source projects. In terms of super-parameter settings, we directly show values we used: the number of hidden nodes is 40, the token length is 30, the training epoch is 15, and the learning rate is assigned to 0.01.

As introduced in Section IV-C, we selected MCC and AUC as measures when validating the effectiveness of LSTM-BT. Since simple value comparisons cannot detect whether there are significant differences, the Scott-Knott test, which divides different methods into groups with significant statistical differences by hierarchical clustering analysis, is used for comparing performance between methods. In our study, we employed the Scott-Knott effect size difference (ESD) [35], the normality and effect size aware variant of traditional Scott-Knott test. Compared with the traditional Scott-Knott test, the variant would (1) redress non-normal distributions of inputs, which are assumed as normally distributed in the traditional Scott-Knott test; and (2) merge groups with a negligible effect size into one group.

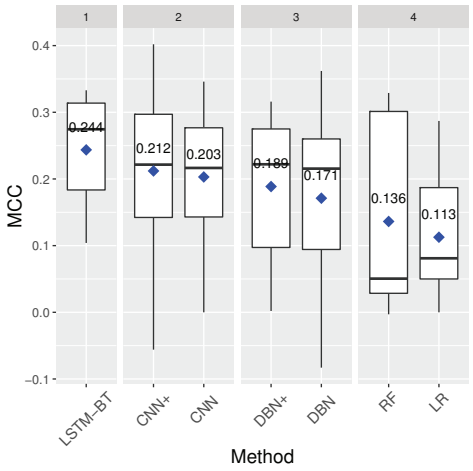
The result of the Scott-Knott ESD test is shown in Figure 5. From Figure 5(a), the average MCC of LSTM-BT is 0.244, which respectively outperforms CNN+, CNN, DBN+, DBN, RF, LR by 15.09%, 20.20%, 29.10%, 42.69%, 79.41%, 115.93% . From Figure 5(b), the average AUC of LSTM-BT is 0.637, which respectively outperforms CNN+, CNN, DBN+, DBN, RF, LR by 3.41%, 4.85%, 7.60%, 13.75%, 16.67%, 17.10%.

V. DISCUSSION

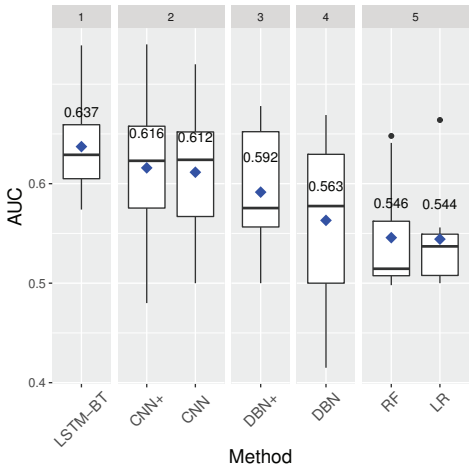
A. Why LSTM-BT works?

The experiment results show that LSTM-BT performs better than other approaches. The probable reasons are listed as following:

1) Traditional machine learning methods based on hand-crafted features focus on pre-selected features, other features



(a) The Scott-Knott ESD of MCC



(b) The Scott-Knott ESD of AUC

Fig. 5. The Scott-Knott ESD ranking of 7 different methods across the 8 pairs of SDP tasks. The blue diamond indicates the average value of methods.

which not be considered may be defect-prone. Compared with these approaches (e.g. LR, RF), LSTM-BT automatically captures semantic and syntactic information from the source code for finer-grained and more comprehensive features.

2) Compared with the previous DL-generated-feature-based methods (e.g. DBN, DBN+, CNN, CNN+). First, since sometimes it is difficult to determine whether a software entity is defective simply based on local patterns, we use LSTM to capture long-term dependencies. Second, by adding a word embedding layer, the semantic information is enhanced because the distance between different nodes is expressed. Last, instead of feeding features to a sequential model, the implementation of LSTM-BT, merging the bidirectional model and tree structure model, is conducive to capture the information of different aspects. Thus, we believe that LSTM-BT-generated features can contribute to the performance of SDP.

In traditional LSTM, forget-gates make information in cell states selective forget, input-gates record new information selectively into the cell states, output-gates determines what information is output. Since forget-gates in LSTM turn out to be of the greatest importance, LSTM-BT realizes the function of forget-gates by excite-gates and inhibit-gates, more parameters enhance its learning ability. Moreover, more parameters make it better adapted to the situation of a large number of samples and more suitable for SDP of which the samples increase with time.

In our experiments, considering LSTM-BT may bring greater time costs due to a complex network and parameter tuning, we speed it up according to the super parameter analysis in [36]. The analysis showed that the learning rate and the hidden layer size are the most important parameters in LSTM, and there is no relationship between them, so the parameters can be tuned independently. Besides, the learning rate can be calibrated with a small network, which can save a lot of time.

B. Threats to validity

We discuss a few threats to the validity of our study in this section.

- **Implementation of compared methods:** Since we did not have the source code of Wang's [5] and Li's [6] experiment, we replicated their experiments according to the main algorithm they reported, and may not achieve all the details in comparison methods.
- **Acquisition of datasets:** The defect labels we used were collected by [32], which were identified in the comments of the version control system. Given that the comment guide varies among different projects and comments are not always well reported [37], it is a possible threat of our data.
- **Results may not be universal:** Experiments have been performed on 8 pairs of Java open-source projects. We believe that these instances, which were extracted from the real world, have trusted quantity and credible quality. However, since various projects, especially projects with different programming languages, have their own properties, we can not guarantee that our findings apply to all other projects such as those based on other programming languages.

VI. Conclusion and Future Work

With the ever-expanding spread of software, software reliability has gained more and more attention. However, we found that most previous studies did not express semantic and syntactic information well. To enhance the feature learning ability and the performance of SDP, we constructed LSTM-BT. Using our approach, vectors are captured from an AST. Then, an embedding layer is used to extract features, which are fed to LSTM-BT and get prediction results. We conducted experiments on 8 pairs of projects to prove the effectiveness of LSTM-BT, and results showed that LSTM-BT shows better performance in MCC and AUC compared with several defect prediction models.

In the future, to make results more universal, more experiments should be conducted on more datasets from different sources and with different programming languages to evaluate LSTM-BT. Furthermore, we will extend our approach to the semi-supervised version by introducing reinforcement learning.

ACKNOWLEDGMENTS

This work was supported in part by Guangzhou Produce & Research Fund under grant no. 201902020004 and MeiZhou Produce & Research Fund under grant no. 2019A0101019.

REFERENCES

[1] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction," in *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*. IEEE, 2007, pp. 69–72.

[2] 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.

[3] R. S. Wahono, "A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks," *Journal of Software Engineering*, vol. 1, no. 1, pp. 1–16, 2015.

[4] R. Özakıncı and A. Tarhan, "Early software defect prediction: a systematic map and review," *Journal of Systems and Software*, vol. 144, pp. 216–239, 2018.

[5] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[6] 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)*. IEEE, 2017, pp. 318–328.

[7] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.

[8] S. Qiu, L. Lu, and S. Jiang, "Multiple-components weights model for cross-project software defect prediction," *IET Software*, vol. 12, no. 4, pp. 345–355, 2018.

[9] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[10] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 481–490.

[11] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 118–127.

[12] S. Qiu, H. Xu, J. Deng, S. Jiang, and L. Lu, "Transfer convolutional neural network for cross-project defect prediction," *Applied Sciences*, vol. 9, p. 2660, 06 2019.

[13] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2017.

[14] X. Zhang, K. Ben, and J. Zeng, "Cross-entropy: A new metric for software defect prediction," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 111–122.

[15] T. Shippey, D. Bowes, and T. Hall, "Automatically identifying code features for software defect prediction: Using ast n-grams," *Information and Software Technology*, vol. 106, pp. 142–160, 2019.

[16] W. Liu, S. Liu, Q. Gu, X. Chen, and D. Chen, "Fecs: A cluster based feature selection method for software fault prediction with noises," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 276–281.

[17] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[18] 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)*. IEEE, 2017, pp. 45–52.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber, "Evolving memory cell structures for sequence learning," in *International Conference on Artificial Neural Networks*. Springer, 2009, pp. 755–764.

[21] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[22] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, 2016, pp. 1480–1489.

[23] X. Fu, J. Yang, J. Li, M. Fang, and H. Wang, "Lexicon-enhanced lstm with attention for general sentiment analysis," *IEEE Access*, vol. 6, pp. 71 884–71 891, 2018.

[24] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[25] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Scientific Programming*, vol. 2019, 2019.

[26] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic lstm model for software defect prediction," *IEEE Access*, vol. 7, pp. 83 812–83 824, 2019.

[27] 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 *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 46–57.

[28] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[29] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *International Conference on Machine Learning*, 2015, pp. 2342–2350.

[30] R. C. O'Reilly and M. J. Frank, "Making working memory work: a computational model of learning in the prefrontal cortex and basal ganglia," *Neural computation*, vol. 18, no. 2, pp. 283–328, 2006.

[31] X.-Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 321–339, 2016.

[32] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 9.

[33] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1253–1269, 2018.

[34] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2107–2145, 2016.

[35] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[36] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[37] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.