Combining Attention-based Gated Bidirectional LSTM and ODCN for Software Defect Prediction

Dingbang Fang[1], Shaoying Liu[1], Ai Liu[1]

Abstract: Software reliability plays an important role in the software lifecycle. Traditional defect prediction adopts static metrics as manual features to predict defects. Although static metrics can measure the complexity of software, they lack semantic and structural information about the code. This paper proposes a novel neural network: GB-ODCN, capable of capturing critical features from the source code to predict defects. GL-ODCN consists of a gated bi-directional long short-term memory network (Bi-LSTM) and a one-dimensional convolutional neural network (ODCN). Bi-LSTM constructs dependency of semantic features from the code. Besides, ODCN captures high-level semantic features for judging whether there are defects in the code. Moreover, attention mechanisms based on both networks enhance the importance of features by assigning weights. Experimental results show that GB-ODCN outperforms current state-of-the-art algorithms on several open-source repositories.

Keywords: Defect prediction, Neural network, Attention mechanism

1. Introduction

As software development continues to grow in the scale, it causes problems for testers and maintainers to find defects through manual debugging. The technique of software defect prediction facilitates the identification of defects in the early stages of software development. Therefore, academia [1] and industry [2] have focused their attention on the study of software defect prediction.

Defect prediction plays a significant role in the field of software engineering. In recent years, most researchers aim to extract some static metrics of software, such as the number of lines of code (LOC) [3] and the cyclomatic complexity [4], to predict software defects. Traditional machine learning algorithms [5] are designed as classifiers, such as support vector machines (SVM), logistic regression, and random forests. Static metrics are fed into the classifier as manual features to identify defects. Although these static metrics can measure some statistical properties of the code, most of them are set by artificial rules resulting in a lack of syntactic, semantic, and structural information in the program.

In recent years, several researchers have leveraged deep learning methods to extract semantic features from programs to identify software defects. These models, deep belief networks (DBNs) [6] and convolutional neural networks (CNNs) [7], are novel but suffer from some drawbacks: (1) code tokens, parsed from source code by abstract semantic trees, resulting in unmeasured distances between themselves, (2) DBN can only handle isolated code tokens, result in tokens that cannot generate correlation, and (3) CNN can only handle a limited range of code tokens, lacking wholeness. Subsequent researchers have proposed recurrent neural networks [8] to construct semantic feature dependencies from source code, but it is not able to handle long sequences of code tokens efficiently. Therefore, the currently proposed approaches to extract semantic features from source code are not sufficient to effectively predict software defects.

Inspired by these slights, we propose the GB-ODCN system, consisting of a Bi-LSTM network and an ODCN. LSTM is used by constructing dependencies of semantic features between code marks after AST parsing. ODCN constructs dependencies of local features. Then multiple attentions focus on assigning more weights from location information to features respectively to extract high-level features, which are then fed to the classification layer for defect prediction. Our proposed GB-ODCN is experimentally verified to achieve state-of-the-art performance on the Promise dataset concerning the F-measure metric. The contribution of the proposed approach comes from the following aspects.

(1) Propose a novel network called GB-ODCN to efficiently extract semantic and contextual features from source code.

(2) Introduce firstly multiple attention mechanisms on GB-ODCN to obtain critical information from different perspectives

(3) Verify that GB-ODCN outperforms the currently available mainstream deep learning algorithms on the PROMISE dataset.

The remainder of this paper is organized as follows: Section 2 presents the work on defect detection, Section 3 shows our proposed algorithm, Section 4 obtains experimental results and analysis, and Section 5 summarizes the conclusions of the proposed method.

[1] Hiroshima University, Higashi-Hiroshima, 739-8527, Japan

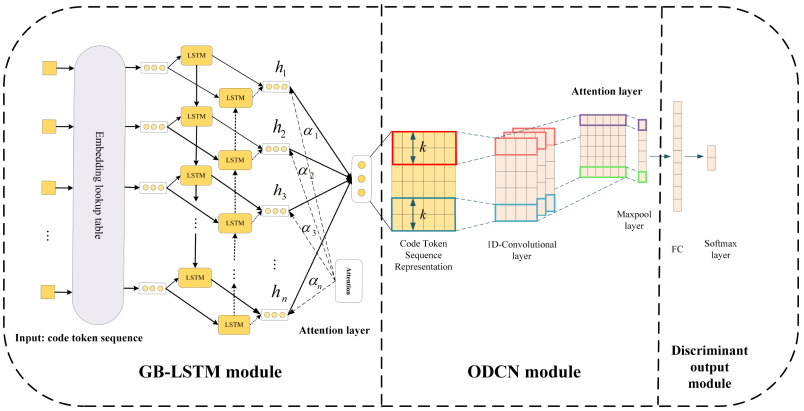


Figure 1. GB-ODCN model framework

2. Related work

2.1 Defect prediction based on static metrics

In the early research on software defect prediction [1]-[5], most of them leverage static metrics as features to train classifiers based on machine learning. Some researchers have used specific rules to design static metrics with statistical properties [5] such as the number of lines of code, the number of methods. In the follow-up, some new metrics are proposed based on the previous ones. For example, Nagappan et al. [9] propose the Churn metric to explore potential defects. Hassan et al. [10] uses changes in entropy to predict defects. However, static metrics lack semantic and structural information in the program. In addition, there is no guarantee of redundancy in the features subsequently fed into the classifier.

2.2 Defect prediction based on deep learning methods

In recent years, deep learning has achieved remarkable success in the fields of computer vision, natural language processing, and speech recognition. Some researchers are becoming enthusiastic about exploring semantic features in programs through deep learning (DL) methods. For example, Wang et al [6]. leverages deep belief networks to learn semantic features from sign vectors extracted from AST trees to identify defects. Li et al [7]. combines semantic features learned by convolutional neural networks with manual features to identify defects. Deng et al [8]. cleverly exploits the learned long-term dependency of LSTM to predict software defects.

3. The method of GB-ODCN

Previous studies in DL on software defect prediction proposed a single model that did not perform satisfactorily on software defect prediction. Our proposed model combines the intrinsic characteristics of the code to propose a robustness model, GB-ODCN, consisting of three modules, GB-LSTM, ODCN, and discriminative output, as shown in Figure 1.

3.1 GB-LSTM module

Most of the program utterances in the source code are repetitive, so it is possible to extract some predictable statistical properties from them by language models, such as word embedding and LSTM [11]. To meet the current scenario of software defect prediction. In particularly, we use a bidirectional LSTM network with gating units to learn semantic and contextual properties in the code.

(1) Word embedding

For subsequent model processing, the token vector parsed by AST needs to be transformed into an integer vector by word embedding for subsequent model processing, as described in the following formula:

$$F : M^{d \times n} \rightarrow \mathbb{R}^{n} \tag{1}$$

where M is the embedding matrix, d is the dimension of the dictionary, and n denotes the maximum length of the code token.

(2) LSTM

In this paper, we use LSTM to construct interdependencies between code signs. LSTM is a variant of a recurrent neural network (RNN). Unlike the RNN structure, it has a storage unit for information memory and a gating unit for filtering information. The forgetting gate g_f mainly selectively forgets the previous moment state cell; the input gate g_i determines the degree of acceptance of the current updated state; the output gate g_o determines whether the current cell state is output or not. The gating cell is calculated by the following formula:

$$g_f = \sigma(w_f[\overrightarrow{h_{t-1}},,x_t] + b_f) \tag{2}$$

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

$$g_o = \sigma(w_o[\overrightarrow{h_{t-1}},,x_t] + b_o) \tag{4}$$

where $(w_f, b_f), (w_i, b_i), (w_o, b_o)$ correspond to the weight and bias matrices of the forgetting gate g_f, the input gate g_i, the output gate g_o. After the filtering of the information by the gating unit, the

memory unit, and the output states are c_t and $\vec{h}_t$, respectively. The equations are described as follows:

$$\widetilde{c_t} = Tanh(w_c[\overrightarrow{h_{t-1}}, \mathrm{x}_t] + \mathrm{b}_c) \quad (5)$$

$$c_t = g_i \odot \widetilde{c_t} + g_f \odot c_{t-1} \quad (6)$$

$$\vec{h}_t = g_o \odot Tanh(c_t) \quad (7)$$

where (w_c, b_c) represents the weight and bias of the updated state $\tilde{c}_t$, $\odot$ denotes an element-wise multiplication, σ (sigmoid) and Tanh denotes two nonlinear activation functions. However, the forward-structured LSTM can only represent the forward information of the code token sequence, but not the backward information. Defective codes are usually related to the preceding and following code tokens, so we use a Bi-LSTM in the current scenario.

In addition, since LSTM is not ideal in dealing with long sequences, we introduce an attention mechanism [12] based on bidirectional LSTM, which can reduce the loss of information. Software defects often appear in some specific locations of the program, which is also the information we focus on. We assign different weights to the output states by the attention mechanism, which is beneficial to catch some key information. The formula for attention is described as follows:

$$\alpha = Softmax(W_h^T f(h)) \quad (8)$$

$$f(h)=h \quad (9)$$

$$S = \alpha h = [\alpha_1 h_1, \alpha_2 h_2, ..., \alpha_n h_n] \quad (10)$$

where $\alpha=(\alpha_1,\alpha_2,\alpha_3,...,\alpha_n)$ denotes the attention weight vector, the W_h is a random initialized attention matrix, and h denotes the fused state information. Function f can be viewed as one of the calculated ways in the attention mechanism, as shown in Figure 2. It can be interpreted as relating to all code tokens for the subsequent calculation of the contribution of the current code token in the corpus. The output states at each position are weighted to form a final state vector S. Thus, we view S as an intermediate representation of the sequence of code tokens.

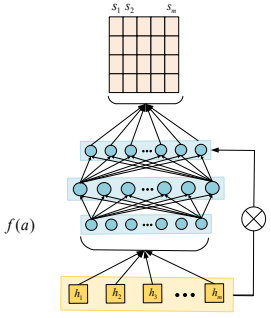


Figure 2. Attention model based on Bi-LSTM network

3.2 ODCN

The intermediate representation generated after GB-LSTM processing contains both contextual and semantic features. The intermediate representation consists of a two-dimensional matrix with temporal and feature dimensions, so we can consider it as an image to explore the dependencies between local features by ODCN. Compared to two-dimensional convolutional networks, ODCN is more suitable for processing each code symbol represented in the temporal dimension. In this paper, we use the depth-separable convolution [13] in ODCN. The advantage of depth-separable convolution is that it reduces the parameters of the operation to a certain extent and makes efficient use of the enhanced feature-dependent representation. During the processing of ODCN, the size of the intermediate representation is kept invariant. In particular, a gated linear unit (GLU) is used to compress the information of a single signature before using ODCN, which preserves the invariance of position and content to promote the processing efficiency of ODCN.

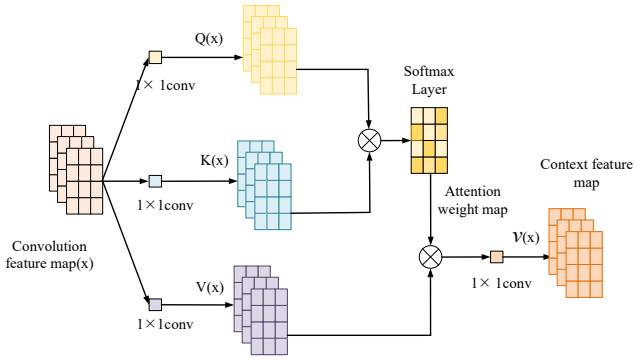


Figure 3. Attention model based on ODCN

In addition, we introduce self-attention [14] on the basis of ODCN. The importance of global features is established by applying weights to local features at arbitrary times as shown in Figure 3. In other words, it enhances the important node information and suppresses the less important ones. The formula of the attention mechanism is calculated as follows:

$$\mathrm{Q}(x) = W_q x \quad (11)$$

$$\mathrm{K}(x) = W_k x \quad (12)$$

$$\mathrm{V}(x) = W_v x \quad (13)$$

$$S_{i,j} = Q(x_i)^T K(x_j) \quad (14)$$

$$\beta_{i,j} = \frac{\exp(S_{i,j})}{\sum_{i=1}^{N} \exp(S_{i,j})} \quad (15)$$

$$o_j = v(\sum_{i=1}^{N} \beta_{j,i} \mathrm{V}(x_i)) \quad (16)$$

where Q(x), K(x), and V(x) denote the semantic feature maps processed by the 1×1 convolutional layers with initial weights W_q, W_k and W_v, respectively. It could compress the number of channels, which are C/8, C/8, and C, respectively, assuming the original number of channels is C. The output at the end of the model is filtered for important information using maximum pooling.

3.3 Trained GB-LSTM

For effective prediction of defect information, we design decision output layers for extracting high-level semantic features, consisting of fully connected layer and output layers with Relu and Sigmoid function, respectively. Finally, for the training of the model, we use a cross-entropy loss function (L) to determine the error between the predicted and true values, as described by the following formula:

$$L=\frac{1}{K}\sum_{k=1}^{K}\sum_{m=1}^{M}\delta(y_m{}^k)\log\frac{e^{W_m{}^T X^{(k)}}}{\sum_{m=1}^{M}e^{W_m{}^T X^{(k)}}} \quad (17)$$

where $X^{(k)}$ denotes the *kth* sample out of the total K samples, $W=(W_1,W_2,W_3,...,W_M)$ denotes the weight vector of all categories, and the total number of categories is M (defective and non-defective). $\delta(y_m{}^k)$ is an indicator function, denoting the label of the sample in class, e.g., when the real label and the predicted label are the same, it is 1, otherwise it is 0.

4. Experiments

In this section, we focus on verifying the performance of GB-ODCN and develop the content around the following issues.

Q1: How effective is the GB-LSTM network compared to the current mainstream deep learning methods in defect prediction?

Q2: How do the parameters of the network affect the GB-LSTM?

4.1 Evaluated data sets

Supervised learning models need to be developed on a large number of labeled datasets. In contrast to the lack of labeled software defects, we use the Promise[1] dataset, which is currently widely used in software defect prediction, and locate the corresponding repository in java language on Github[2]. The specific distribution of the data is shown in TABLE I, using the original version for training the model and the new version for testing the model, e.g., poi2.0-3.0 indicates that poi.2.0 is the training set of the model and poi3.0 is used as the test set of the model.

For source code parsing, we follow the literature [6]. Three different types of nodes are parsed by AST: (1) nodes for method calls and instance creation, (2) declaration nodes, and (3) control flow nodes such as if statements in TABLE II. According to our statistics on the length of all code sequences, the length of the maximum word vector is 2025. For greater than the maximum length of 2000, we remove the code tokens and discard the files that are less than the minimum length of 3. The number of samples with defects is increased by oversampling [15].

TABLE I. Java project dataset

Project	Versions	Avg files	Avg defect(%)
poi	2.0-3.0	918	18.1
lucene	2.0-2.4	170	58.7
xalan	2.5-2.6	221	53.7
camel	1.4-1.6	443	49.7
xerces	1.3-1.4	202	23.0
synapse	1.1-1.2	844	47.3
velocity	1.5-1.6	413	64.0

TABLE II. Select the type of node

MethodInvocation	ClassCreator
SuperMethodInvocation	ConstructorDeclaration
InterfaceDeclaration	IfStatement
VariableDeclarator	AssertStatemen
WhileStatement	ThrowStatement
ContinueStatement	BlockStatement
SynchronizedStatement	SwitchStatementCase
TryResource	MemberReference
EnhancedForControl	PackageDeclaration
SuperMemberReference	MethodDeclaration
ClassDeclaration	ForStatement
FormalParameter	BreakStatement
DoStatement	TryStatement
ReturnStatement	CatchClauseParameter
CatchClause	ForControl
ReferenceType	StatementExpression
SwitchStatement	BasicType

4.2 Baseline Methodology

We reproduce some of the mainstream deep learning algorithms in software defect prediction as a baseline for the model to compare the performance of GB-LSTM.

(1) DP-ALSTM: This model consists only of GB-LSTM and the decision output layer to identify defects.

(2) DP-LSTM: Bi-LSTM networks learn semantic features from codes to predict defects [8].

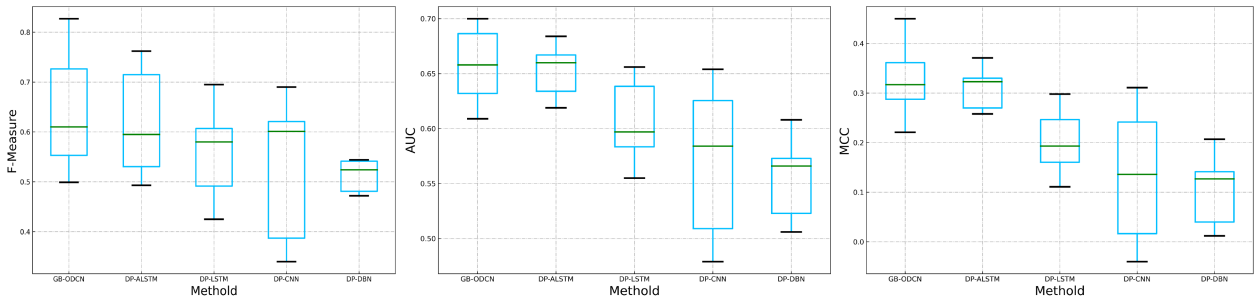
(3) DP-CNN: a software prediction method adapts standard convolutional neural networks [7].

(4) DP-DBN: a DBN formed by multilayer Boltzmann machines uses to predict defects in the software [6].

For the evaluation of the models, we use metrics that are currently widely used in software forecasting [1]-[9], including precision, recall, F-measure, AUC, and MCC, for correlating the predicted results with the true categories

[1] [Online]. Available: https://zenodo.org/communities/seacraft

[2] [Online]. Available: https://github.com/

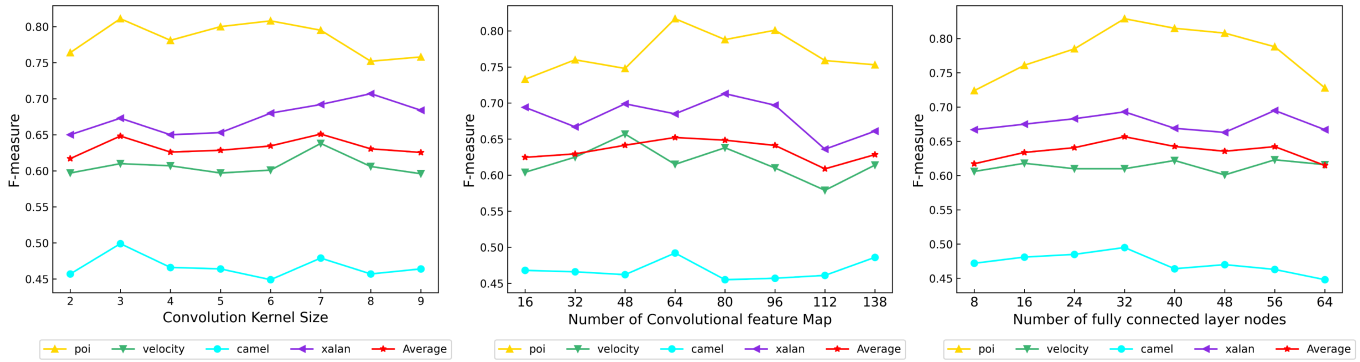


(a) F-measure of different methods **(b) AUC of different methods** **(c) MCC of different methods**

Figure 4. Comprehensive comparison of methods

TABLE III. Precision, Recall, F-Measure of DP-DBN, DP-CNN, DP-LSTM, DP-ALST, AND EPR

	DP-DBN			DP-CNN			DP-LSTM			DP-ALSTM			GB-ODCN		
Task	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
poi2.0-3.0	0.732	0.623	0.673	0.78	0.619	0.69	0.749	0.648	0.695	0.763	0.758	0.761	0.763	0.904	**0.827**
lucene2.0-2.4	0.622	0.453	0.524	0.721	0.547	0.622	0.707	0.571	0.632	0.615	1.00	0.762	0.683	0.882	**0.770**
xalan2.5-2.6	0.538	0.55	0.544	0.652	0.584	0.616	0.567	0.599	0.582	0.609	0.742	0.669	0.581	0.827	**0.683**
camel1.4-1.6	0.22	0.559	0.316	0.261	0.585	0.361	0.328	0.606	0.425	0.503	0.489	0.496	0.483	0.516	**0.499**
xerces1.3-1.4	0.66	0.455	0.539	0.628	0.577	0.601	0.714	0.399	0.512	0.851	0.347	**0.793**	0.793	0.413	0.543
synapse1.1-1.2	0.379	0.581	0.472	0.346	0.512	0.413	0.402	0.51	0.471	0.46	0.733	0.565	0.496	0.651	**0.563**
velocity1.5-1.6	0.412	0.603	0.49	0.474	0.692	0.562	0.465	0.769	0.58	0.458	0.846	0.595	0.485	0.821	**0.61**
Average	0.509	0.546	0.508	0.551	0.588	0.552	0.561	0.586	0.557	0.608	0.701	0.62	0.612	0.752	**0.657**



(a) Different Convolutional kernel sizes **(b) Different convolutional feature maps** **(c) Different FC nodes**

Figure 5. F-measure under different parameters

4.3 Comparison with baseline method (Q1)

To verify the effectiveness of GB-LSTM, we compare the above baseline methods, and the experimental results are shown in Figure 4 and TABLE III. It can be seen from the figure that our proposed GB-LSTM is higher than the current baseline method in the average level of F-measure, AUC, and MCC, indicating the rationality of the GB-LSTM design. In addition, the specific distribution of values is shown in the table. The table shows that GB-LSTM achieves the highest level of mean value in the F-measure. GB-LSTM is 14.9% higher than DP-DBN, 10.5% higher than DP-CNN, 10% higher than DP-LSTM, and 3.7% higher than DP-ALSTM.

Compared with other baseline models, we skillfully design two different attention mechanisms to impose different weights from location information and feature importance, which can facilitate GB-LSTM to express more complex semantic features. It is worth noting that GB-LSTM is based on DP-ALTM with the introduction of ODCN module, and the experimental results show that ODCN enhances the performance of the model.

4.4 Influence of parameters on GB-LSTM (Q2)

In particular, we verify the influences of different types of parameters on the model. For both parameters, the embedding layer and the number of LSTM cells, we refer to previous work as a priori knowledge [8] to set 30 and 40, respectively. We only consider the effect of three parameters on the GB-LSTM, including the convolutional kernel size, the number of filters in the convolutional layer, and the number of hidden layer units. Limited by time and resources, we only consider poi, velocity, camel, and xalan as the test data, and the average value of the test data is used as the basis for our selection of parameters. Because of the differences in the amount of data in different repositories, the optimal parameters are not consistent. According to the different parameter types, the corresponding curve trend is shown in Figure 5. It can be seen from the figure that the average value of the F-measure of the size of the convolution kernel, the number of feature maps of the convolution, and the number of FC nodes are 3, 32, and 64. However, these may not be the best parameters in terms of the trend of the curve, but the increase in F-measure is not very obvious, which means that the use of other parameters does not make much sense.

5. Conclusions

In this paper, we propose a novel system called GB-LSTM, which achieves state-of-the-art performance compared to other related models on the Promise repository. Experimental results show that GB-LSTM can effectively obtain long-term dependencies from the code, followed by two different attention mechanisms to obtain important information from different perspectives.

Regarding future work, our proposed model GB-LSTM will be tested in different scripting languages such as C#, C++, and python. In specially, semi-supervised learning will be an attractive work to improve the robustness of the model.

Acknowledgments

The work was supported by ROIS NII Open Collaborative Research 2021-(21FS02).

Reference

[1] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675-689, Sept.-Oct. 1999.

[2] R. Rana, M. Staron, J. Hansson, M. Nilsson, and W. Meding, "A framework for adoption of machine learning in industry for software defect prediction," in *Proc. Int. Conf. Softw. Eng. Applica.* (ICSOFT-EA), 2014, pp. 383-392.

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

[4] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. 4, pp. 308-320, 1976.

[5] C. Catal, B. Diri, “Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem,” *Inform. Sci.*, vol. 179, no. 8, pp. 1040-1058, 2009.

[6] 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.

[7] J. Li, P. He, J. Zhu and M. R. Lyu, "Software Defect Prediction via Convolutional Neural Network," in *Proc. IEEE Int. Conf. Softw. Quality., Rel. Secur.* (QRS), 2017, pp. 318-328.

[8] J. Deng, L. Lu, S. Qiu “Software defect prediction via LSTM,” in *IET Softw.*, vol. 14, no. 4, pp. 443-450, 2020.

[9] N. Nagappan and T. Ball, “Using software dependencies and churn metrics to predict field failures: an empirical case study,” in *Proc. Int. Sym. Emp. Soft. Eng. Meas.* (ESEM 2007), pp. 364–373, September 2007.

[10] A. E. Hassan, “Predicting faults using the complexity of code changes,” in Proc. 31st Int. Conf. Softw. Eng., 2009, pp. 78–88.

[11] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neur. Compu*., vol. 9, no. 8, pp. 1735–1780, 1997.

[12] T. M. Luong, H. Pham, D. C. Manning, “Effective approaches to attention-based neural machine translation,” arXiv:1508.04025, 2015.

[13] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in *Proc. IEEE Conf. Comp. Visi. Patt. Recog.* (CVPR), pp. 1800-1807, 2017.

[14] H. Zhang, I. Goodfellow, D. Metaxas, et al, “Self-attention generative adversarial networks,” in *Proc. Int. Conf. Machin. Lear.,* PMLR, pp. 7354-7363. 2019.

[15] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in *Proc. IEEE Int. Conf. Softw. Eng.*, pp. 99–108, May 2015.