Hindawi
Scientific Programming
Volume 2019, Article ID 6230953, 14 pages
https://doi.org/10.1155/2019/6230953

Research Article

Software Defect Prediction via Attention-Based Recurrent Neural Network

Guisheng Fan,[1,2] **Xuyang Diao**,[1] **Huiqun Yu**,[1] **Kang Yang**,[1] **and Liqiong Chen**[3]

[1]*Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China*
[2]*Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 201112, China*
[3]*Department of Computer Science and Information Engineering, Shanghai Institute of Technology, Shanghai 201418, China*

Correspondence should be addressed to Guisheng Fan; gsfan@ecust.edu.cn; Xuyang Diao; y30170698@mail.ecust.edu.cn; and Huiqun Yu; yhq@ecust.edu.cn

Received 14 January 2019; Revised 15 March 2019; Accepted 2 April 2019; Published 15 April 2019

Academic Editor: Autilia Vitiello

In order to improve software reliability, software defect prediction is applied to the process of software maintenance to identify potential bugs. Traditional methods of software defect prediction mainly focus on designing static code metrics, which are input into machine learning classifiers to predict defect probabilities of the code. However, the characteristics of these artificial metrics do not contain the syntactic structures and semantic information of programs. Such information is more significant than manual metrics and can provide a more accurate predictive model. In this paper, we propose a framework called defect prediction via attention-based recurrent neural network (DP-ARNN). More specifically, DP-ARNN first parses abstract syntax trees (ASTs) of programs and extracts them as vectors. Then it encodes vectors which are used as inputs of DP-ARNN by dictionary mapping and word embedding. After that, it can automatically learn syntactic and semantic features. Furthermore, it employs the attention mechanism to further generate significant features for accurate defect prediction. To validate our method, we choose seven open-source Java projects in Apache, using F1-measure and area under the curve (AUC) as evaluation criteria. The experimental results show that, in average, DP-ARNN improves the F1-measure by 14% and AUC by 7% compared with the state-of-the-art methods, respectively.

1. Introduction

With the continuous expansion of modern software, software reliability has become a key concern. The complex source code of software tends to cause software defects which may lead to software failure. In order to help developers and testers locate software defects in time, software defect prediction has become one of the research directions in the field of data mining of software engineering [1].

Software defect prediction [2, 3] is a process of constructing machine learning classifiers to predict defective code snippets, using historical information in software repositories such as code complexity and change records to design software defect metrics [4]. The predicted results can assist developers to locate and fix potential defects, thereby improving software stability and reliability. According to whether source data and target data are homogeneous or heterogeneous, software defect prediction can be divided into within-project software defect prediction [5] and cross-project software defect prediction [6]. In this paper, we focus on within-project software defect prediction. The program granularity can be file level, change level, or function level, and we choose the file-level granularity as the representation of programs in this paper.

Traditional defect prediction methods mainly consist of two stages: extracting software metrics from historical repositories and constructing a machine learning model for classification. Previous research focuses on designing discriminative artificial metrics to achieve higher model accuracy. These manual metrics are mainly divided into Halstead features [7] based on the number of operators and

operands, dependency-based McCabe features [8], and CK features [9] based on object-oriented programs.

However, sometimes static code attributes cannot distinguish whether the code has defects because a clean code snippet and a buggy one may have the same values of static code attributes, which makes classifiers hard to differentiate. Since the syntactic and semantic information between them is different, features which contain such structural and semantic information should improve the performance of defect prediction. Programs have their own particular syntactic structures and rich semantic information hidden in ASTs [10], which help analyzing and locating defects more accurately. AST is the representation of source code. It uses a tree structure to describe the relationship of code context, which contains syntactic structures and semantic information of the program module. Leveraging deep learning methods to mine hidden features of ASTs can generate significant features that better reflect the code context information, leading to more accurate software defect prediction.

To take full advantage of intrinsic syntaxes and semantics of programs, this paper proposes a framework called software defect prediction via attention-based recurrent neural network (DP-ARNN), which can capture syntactic and semantic features of programs and use them to improve defect prediction. Specifically, we build recurrent neural network (RNN) [11] to automatically learn features with syntaxes and semantics from encoded token vectors extracted from programs' ASTs and then use the attention mechanism to generate key features for training a more precise defect prediction model. We select seven open-source projects in Apache as datasets, using F1-measure [12] and AUC [13] as evaluation criteria. The experimental results show that, compared with the state-of-the-art methods, the DP-ARNN proposed in this paper has an average increase of 14% on F1-measure and 7% on AUC. This paper makes the following contributions:

(i) We propose an RNN-based defect prediction framework to learn valuable features which contain syntactic and semantic information of the source code. The empirical studies show that, in average, these deep learning-based features outperform traditional features by 14% on F1-measure and 10% on AUC.

(ii) We apply dictionary mapping and word embedding to convert programs' ASTs into high-dimensional digital vectors as the inputs of RNN to learn code context information.

(iii) We leverage attention mechanism to further generate significant features from the outputs of RNN, leading to better performance of defect prediction. The experimental results show that, compared with RNN, attention-based RNN has an average increase of 3% on F1-measure and 1% on AUC.

The rest of this paper is organized as follows. Section 2 introduces the related work about traditional software defect prediction and deep learning-based defect prediction. Section 3 elaborates our proposed DP-ARNN in detail. Section 4 shows the experimental setup and analyzes the results. Finally, Section 5 concludes our work.

2. Related Work

2.1. Traditional Software Defect Prediction. Software defect prediction is a significant research field in software engineering [1]. Most of references focus on designing new discriminative features, filtering defect data, and building an efficient classifier. Nagappan and Ball [14] proposed churn metrics and combined it with software dependencies for defect prediction. Moser et al. [15] made a comprehensive analysis of the efficiency of change metrics and static code attributes for defect prediction. Besides, Arar and Ayan [16] selected appropriate features by applying the Naive Bayesian method to filter redundant ones. Mousavi et al. [17] used the idea of ensemble learning to solve the class-imbalance problem in software defect prediction. Moreover, Jing et al. [18] proposed a dictionary learning method based on calculating misclassification cost for the prediction of software defects.

To solve the problem of lack of information in the historical repositories of the same project, more and more papers have studied cross-project software defect prediction. In this field, because of the different domains of training samples and test samples, we need to apply transfer learning techniques. By using the transfer component analysis (TCA+) [19], we can build an available target prediction model. However, large irrelevant cross-project data usually lead to low performance. To overcome this challenge, many researchers focus on filtering instances or features of the source project that are irrelevant to the target project. Turhan et al. [20] applied the neighbour filter method to remove those instances of the source project, whose features are not close enough to the ones of the target project. Besides, Yu et al. [21] employed correlation-based feature selection to select features that have strong correlation with the target project. Ma et al. [22] proposed a method called transfer Naive Bayes, using a data gravitation approach [23] to adjust the weights of training instances and build a naive Bayes classifier on them. Recently, some studies have demonstrated that, if we can make full use of the small portion of labelled data in the target project, it may result in higher prediction performance. Chen et al. [24] first initialized the weights of source project data by the data gravitation method and adjusted them with a limited amount of labelled data in the target project by building a prediction model named TrAdaboost [25]. Qiu et al. [26] constructed a novel multiple-components weights learning model with the kernel mean matching (KMM) algorithm. It divides the source project data into several components, and KMM is applied to adjust the source-instance weights in each component. Then, it builds prediction models with both the source instances with weights in each component and a fraction of labelled data in the target project. Finally, it initializes and optimizes the source component weights to construct a more precise ensemble classifier.

Our proposed DP-ARNN differs from the aforementioned traditional defect prediction methods. Instead of

using the static code attributes, we leverage the deep learning technique (i.e., RNN) to automatically generate features from the source code, which can capture syntactic and semantic information of programs, and implement the attention mechanism to generate significant features which can improve the performance of defect prediction.

2.2. Deep Learning in Software Defect Prediction. Datasets of traditional defect prediction are extracted from artificially designed metrics which may be redundant or not be highly correlated with class labels. These all can affect the prediction performance of the model. Besides, manual metrics cannot make full use of code context information to mine the syntactic structure and semantic information of programs.

The syntactic and semantic information of programs can be represented in two ways. One is ASTs and the other is control flow graphs (CFGs) [27]. AST is the abstract tree representation of the source code, which describes the hierarchical relationship among various components in program modules. Wang et al. [28] employed DBN to generate hidden features, which contains syntaxes and semantics of programs and fed them into classifiers to predict the buggy code. Lin et al. [29] employed long short-term memory (LSTM) network [30] to learn the cross-project transfer representation of programs' ASTs for vulnerable function discovery. Dam et al. [31] built a deep tree-based model based on ASTs for software defect prediction. In addition, Li et al. [32] combined artificial metrics with deep learning-based features learned by convolutional neural network (CNN) [33] to build a hybrid model. CFG is the representation of the program control flow graphs, which show all the paths that can be traversed during the program execution. Phan et al. [34] extracted CFGs from the assembly code of projects and designed a graph convolutional network to learn semantic features of programs.

The aforementioned deep learning-based methods consider all the hidden features to be equally significant, and they cannot identify discriminative features that contain key syntaxes and semantics. This may lead to inaccurate defect prediction. Hence, in our proposed method, we employ the attention mechanism to capture these key features and give them higher weights. Besides, we choose ASTs of programs as the representation of programs rather than CFGs, because ASTs can better depict the structure of the source code and reserve more information of source code.

3. Component Design

In this section, we elaborate our proposed DP-ARNN, a framework which automatically learns syntactic and semantic features from the source code and generates key features from them for precise software defect prediction. Figure 1 illustrates the overall framework of DP-ARNN.

As is shown in Figure 1, we first parse the source code of the training set and test set into ASTs. Then we select representative nodes and apply depth-first traversal (DFT) to get ASTs' sequence vectors. A file corresponds to a sequence vector extracted from ASTs. In order to train these token vectors, we not only create a mapping between tokens and integers but also employ word embedding to encode them into multidimensional vectors which are used as the inputs of the network. After that, we build an RNN to automatically learn syntactic and semantic features of the source code. Furthermore, we put them into an attention layer with the attention mechanism to further generate significant features. Finally, these crucial features are fed into fully connected layers, and a logistic regression classifier is built for prediction. After the whole framework is well-trained by the training set, we can get a defect probability for each file in the test set, indicating whether it is buggy or clean.

3.1. Parsing Source Code. In order to represent the source code in each file as a vector, we first need to find the appropriate granularity as the vector representation of the source code. We can extract characters, words, or ASTs from the source code as tokens. According to the former research [35], AST is the suitable representation which can reflect the structural and semantic information of programs.

In our experiments, we apply an open-source Python package named javalang which is available at https://github.com/c2nes/javalang to parse our Java source code into ASTs. It provides a lexical analyzer and parser based on the Java language specification, which can construct ASTs of the Java source code. Following the relevant method [36], we only select three types of ASTs' nodes as tokens: (1) nodes of method invocations, (2) nodes of declarations, including method declarations, constructor declarations, and class declarations, and (3) control flow nodes (i.e., branches, loops, exception throws, and captures). For method invocations, we record them as their plain text in the source code. For all the nodes of declarations, we extract their node names as tokens. Control flow nodes are simply recorded as their node types. Besides, nodes of AssertStatement and TryResource are recorded as their values. All the selected nodes in the experiments are shown in Figure 2. Finally, we employ the DFT method to turn ASTs of each program into a vector. Algorithm 1 describes the procedure of the parsing source code.

3.2. Encoding ASTs and Handling Imbalance

3.2.1. Encoding ASTs. ASTs can effectively store structural and semantic information of the program module. For example, code A in Figure 3(a) has a strong resemblance of code B in Figure 3(b), which means manual features can be totally the same, while the code A's AST in Figure 4(a) has two more nodes (i.e., StatementExpression and MemberReference) than code B's AST in Figure 4(b). Since the vector is a combination of string tokens, we cannot directly use it as an input to DP-ARNN. Hence, we build a mapping dictionary between tokens and integers. Assuming that the number of tokens is m and each token corresponds to a unique integer, then the mapping range is from 1 to m. Firstly, we count the frequency of each token and then sort them based on the token frequency. After that, we establish

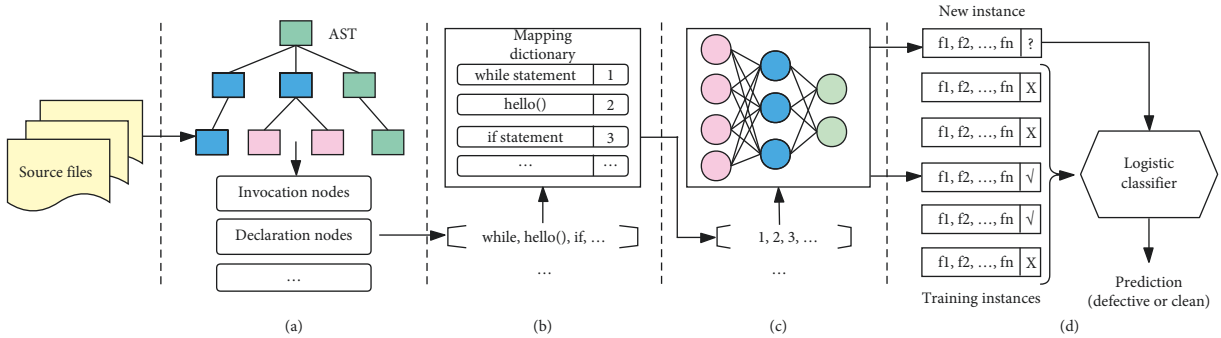


Figure 1: The overall framework of our proposed DP-ARNN. (a) Parsing source code. (b) Mapping string vectors into integer vectors. (c) Generating features via RNN with attention mechanism. (d) Performing defect prediction.

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

Figure 2: The selected nodes of ASTs.

Input: source files $F = \{f_1, f_2, \ldots, f_n\}$,
Representative nodes $S = \{s_1, s_2, \ldots, s_n\}$;
Output: ASTs' vectors $V = \{v_1, v_2, \ldots, v_n\}$;
(1) for $i = 1 \longrightarrow n$ do
(2) AST_i = constructing AST from f_i;
(3) Traversing *node* in AST_i by DFT;
(4) If *node* in S then
(5) Adding node into v_i;
(6) end
(7) Adding v_i into V;
(8) end
(9) return V;

Algorithm 1: Parsing source files into ASTs' vectors.

an index dictionary of the ordered tokens, in which tokens with higher frequency are in front. After the mapping step, we make these digital vectors the same fixed length. In order to avoid vectors being too sparse, the appropriate vector length should be selected. For a vector whose length is less than the specified length, it is filled with 0 because 0 does not have any meaning since we map tokens starting from 1. For a vector whose length is longer than the specified length, the extra part is truncated. Since the token with higher frequency is mapped into smaller integer, the token with the lowest frequency is mapped into the maximum integer. Hence, we locate the index of the maximum integer in the vector and delete it each time until the vector length becomes the same as the fixed length. The pseudocode of encoding ASTs' vectors is shown in Algorithm 2. Finally, we also employ word embedding which is embedded into the network as a trainable word dictionary to represent each token as a high-dimensional vector.

3.2.2. Handling Imbalance. Software defect prediction data are usually class imbalanced. Defective instances usually account for a small part of all the instances. If you put them directly into the model for training, the prediction results will be biased towards the majority class (i.e., clean instances). According to the research [37], there are two popular approaches to solve the class imbalance problem, oversampling, and undersampling. The former replicates instances in the minority class, and the latter deletes instances in the majority class. The undersampling technique may lose part of the data information in the training set, resulting in underfitting. In order to ensure the integrity of the data information, we apply the oversampling method, duplicating training samples from the minority class (i.e., defective instances), to generate a class-balanced training set.

3.3. Bi-LSTM with Attention Mechanism. In order to learn the context information of the source code and generate the key features, we construct a Bi-LSTM network, a variant of standard RNN, with attention mechanism [38], as illustrated in Figure 5. The network architecture mainly consists of five parts: an embedding layer, a Bi-LSTM layer, an attention layer, two fully connected layers, and an output layer.

3.3.1. Embedding Layer. Simple digital integers cannot reflect the content information carried by an AST node. Therefore, we adopt word embedding technique to map each positive integer vector into a high-dimensional real vector with fixed size, which can be defined as follows:

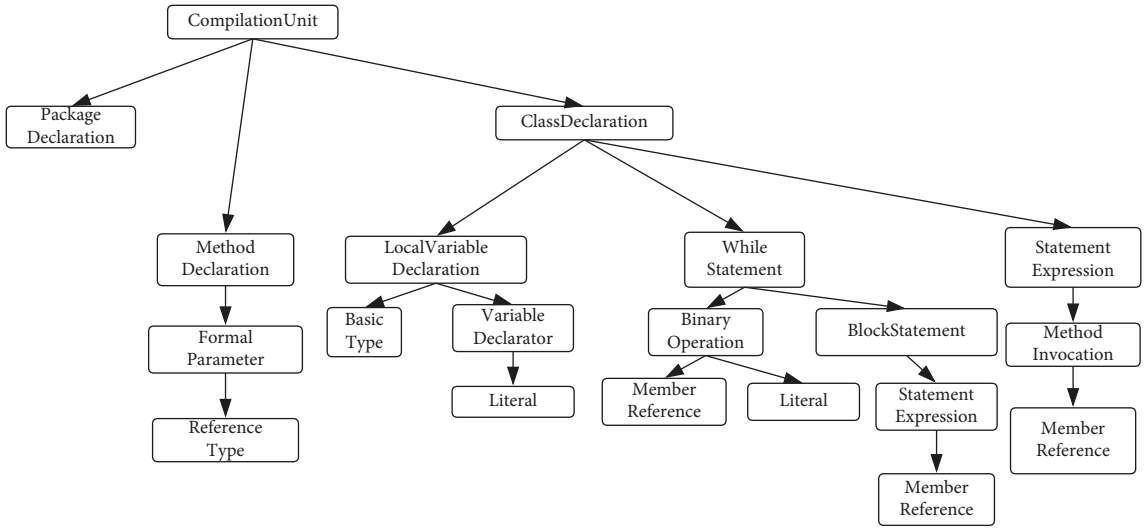
```
1  package com;
2  public class A{
3      public static void main(String[]args){
4          int i=0;
5          while(i<10){
6              i++;
7          }
8           System.out.println(i);
9      }
10 }
```

(a)

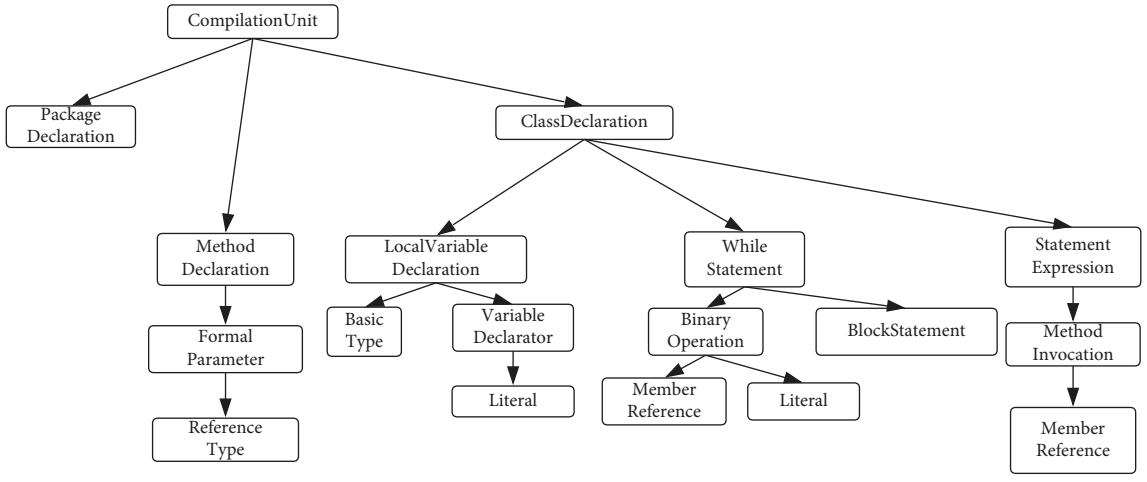
```
1  package com;
2  public class B{
3      public static void main(String[]args){
4          int i=0;
5          while(i<10){
6
7          }
8           System.out.println(i);
9      }
10 }
```

(b)

Figure 3: Source code of two example files. (a) The clean code A. (b) The defective code B.



(a)



(b)

Figure 4: ASTs of two example files. (a) The clean code A. (b) The defective code B.

$$F : M \longrightarrow R^n, \tag{1}$$

where M represents a dictionary formed by AST nodes and R^n is an n-dimensional real vector space. F is a parameterized function that maps each token in M into an n-dimensional vector. The word embedding dictionary is randomly initialized, and it can be updated during the training of the network. Token vectors encoded by word embedding can be fed into the Bi-LSTM network to further explore syntaxes and semantics of programs.

Input: ASTs' string vectors $S = \{s_1, s_2, \ldots, s_n\}$, the fixed length of each vector m;
Output: integer vectors $V = \{v_1, v_2, \ldots, v_n\}$;
Initialize a list V, a dict *tokFreq* and a dict *toktoInt*;
for $i = 1 \longrightarrow n$ **do**
  for $j = 1 \longrightarrow len(s_i)$ **do**
    if s_{ij} not in *tokFreq*.keys **then**
      $tokFreq[s_{ij}] = 0$;
    end
    $tokFreq[s_{ij}] += 1$;
  end
end
Creating a list *sortokFreq* sorted in descending order of token frequency which contains tuples of each token and its corresponding frequency;
for $i = 1 \longrightarrow \text{len}(sortokFreq)$ **do**
  $token = sortokFreq[i][0]$;
  $toktoInt[token] = i$; // establishing a dict of the ordered tokens to map them into integers
end
for $i = 1 \longrightarrow n$ **do**
  for $j = 1 \longrightarrow len(s_j)$ **do**
    $v_{ij} = toktoInt[s_{ij}]$;
  end
  if $len(v_i) < m$ **then**
    Adding $m - len(v_i)$ 0s into v_i;
  end
  else if $len(v_i) > m$ **then**
    for $k = 1 \longrightarrow len(v_i) - m$ **do**
      $z = v_i.index(max(v_i))$; // finding the index of the lowest token frequency *del* v_{iz}; // deleting the token
    end
  end
  Adding v_i into V;
end
return V;

Algorithm 2: Encoding ASTs' string vectors.

3.3.2. Bi-LSTM. Standard RNN splits sequence data into vectors with fixed length. Each element in a vector denotes a certain moment. For a certain moment t, the output $o^{(t)}$ is not only influenced by the current input $x^{(t)}$ but also depends on the accumulated information transmitted from the moment $t-1$ (i.e., $h^{(t-1)}$), which can be formulated as the following equations:

$$\begin{aligned} h^{(t)} &= f\left(U * x^{(t)} + W * h^{(t-1)} + b\right), \\ o^{(t)} &= g\left(V * h^{(t)} + c\right), \end{aligned} \tag{2}$$

where U, W, V, b, and c denote the weights and bias of the network and f and g are the activation functions. The standard RNN can only memorize short-term sequence information and cannot transmit long-term sequence information. Therefore, we select LSTM [30] as the basic unit of RNN. An LSTM unit is mainly composed of an input gate, a forgotten gate, and an output gate. To prevent the gradient of the network from disappearing, information passing from the past is filtered by the forgotten gate, and then LSTM feeds it and information from the input gate into the output gate to generate the current information. Furthermore, to obtain the long dependencies of the surrounding moments close to moment t, the bidirectional LSTM (Bi-LSTM) is built to achieve this purpose.

Contextual information of the source code is significant to detect potential bugs. Each program has its own syntaxes and semantics which are context sensitive. Therefore, the occurrence of a defective code segment is usually relevant to either previous or subsequent code, or even to both of them. In most cases, because of the complexity of the source code, it is hard to exactly locate which line of code actually results in the vulnerability. Hence, in order to efficiently capture the defective programming patterns, we implement Bi-LSTM to make full use of both forward and backward information.

3.3.3. Attention Mechanism. From the output of the Bi-LSTM network, we can get the hidden features of all time nodes in a sequence. Contributions of these nodes to the representation of the sequence meaning are not the same. In order to enhance the effect of critical nodes, we embed an attention layer after the Bi-LSTM Layer. By applying the attention mechanism, critical nodes which are significant to the meaning of the sequence are aggregated together to form a sequence vector. Figure 6 illustrates the entire process of it, and we can describe it as follows:

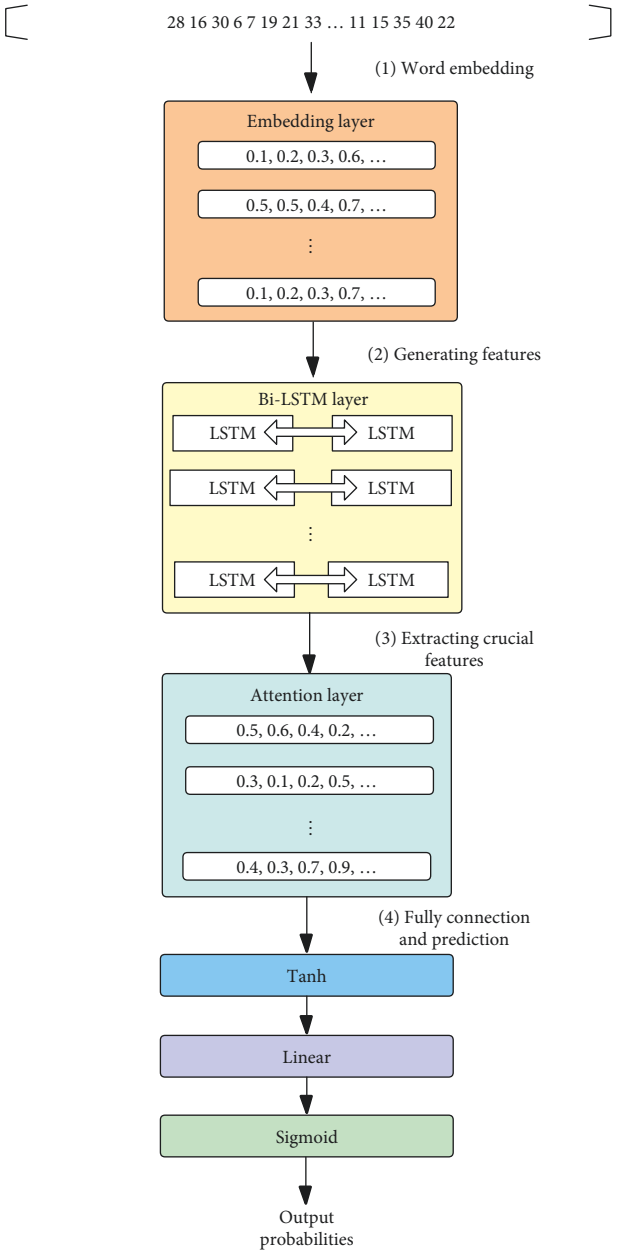


Figure 5: The network architecture of DP-ARNN.

$$u_{it} = \tanh\left(W_n h_{it} + b_n\right),$$
$$\alpha_{it} = \frac{\exp\left(u_{it}^{\top} u_n\right)}{\sum_t \exp\left(u_{it}^{\top} u_n\right)}, \quad (3)$$
$$s_i = \sum_t \alpha_{it} h_{it}.$$

That is, we first input the node annotation h_{it} into a one-layer multilayer perceptron (MLP) to generate u_{it} as a hidden representation of the node, and then we set up a node-level context vector u_n, which can be considered as a high-level representation of a query to search critical nodes in the sequence. After that, we measure the importance of the node as the dot product similarity of u_{it} with u_n and obtain a normalized importance weight α_{it} through a softmax function. Finally, we calculate the sequence vector s_i as a weighted sum of all the nodes with relevant weights. The node level context vector u_n is randomly initialized and can be updated during the training process.

3.3.4. Training Phase. In the training phase, we construct two fully connected layers and an output layer. The first fully connected layer normalizes sequence features through a tanh function, and the second fully connected layer with a linear function further extracts features. At last, in the output layer, we put them through a sigmoid function as a logistic regression classifier to compute the defect probability of the program module.

4. Experiments and Analysis

In this section, we design experiments to verify the effectiveness of DP-ARNN. Four research questions (RQs) are need to be answered as follows:

(i) RQ1: do the deep learning methods improve the performance of defect prediction compared to traditional methods based on static code metrics?

(ii) RQ2: compared with features generated by the classical unsupervised learning methods, do features learned by the deep learning methods better represent syntaxes and semantics of programs?

(iii) RQ3: does DP-ARNN outperform the basic deep learning methods, including CNN and RNN?

(iv) RQ4: how is the prediction performance of DP-ARNN under different parameter settings?

In our experiments, we choose Keras (2.2.4) and Tensorflow (1.11.0) to build attention-based Bi-LSTM network. The implementation of other benchmark methods is mainly based on scikit-learn (0.19.2) and Python 3.6. The experimental operating environment is a server running Ubuntu 16.04 with a 3.60 GHz Intel i7 CPU and RAM of 8 GB.

4.1. Experimental Datasets. We collect seven open-source Java projects in Apache, each of which contains two versions (i.e., preversion and postversion). Datasets that contain static code metrics and defect annotations of source files in each project are from metrics repository, which is a public available repository specializing in software defect prediction research datasets. Specifically, each source file has 20 traditional artificial features, which are carefully extracted by Jureczko and Madeyski, the contributors of CK features for object-oriented programs [39]. We list the detailed description about them in Table 1. These static metrics, including lines of code (LOC), average method complexity (AMC), and number of children (NOC), have been widely used in the previous studies [18, 40, 41]. Table 2 shows the specific information of these projects, including project

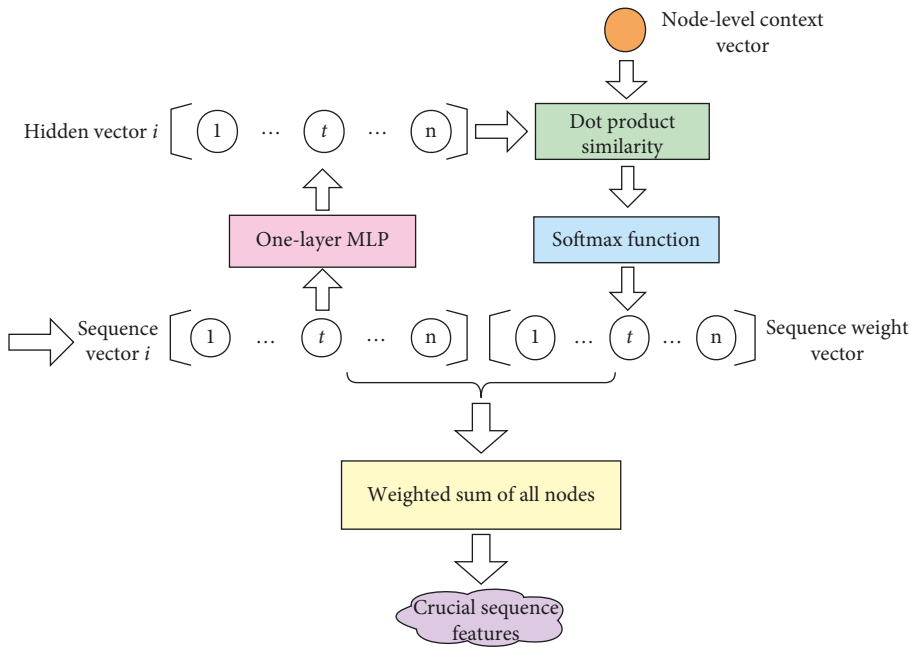


Figure 6: The process of attention mechanism.

Table 1: Description of the 20 static code metrics.

Metric Name	Symbol	Description
Weighted methods per class	WMC	The number of methods in the class
Depth of inheritance tree	DIT	The position of the class in the inheritance tree
Number of children	NOC	The number of immediate descendants of the class
Coupling between object classes	CBO	The value increases when the methods of one class access services of another
Response for a class	RFC	Number of methods invoked in response to a message to the object
Lack of cohesion in methods	LCOM	Number of pairs of methods that cannot share a reference to an instance variable
Lack of cohesion in methods, different from LCOM	LCOM3	If m and a are the number of methods and attributes in a class number and $\mu(a)$ is the number of methods accessing an attribute, then lcom3 = $(((1/a)\sum_j^a \mu(a)) - m)/(1-m)$
Number of public methods	NPM	The number of all the methods in a class that are declared as public
Data access metric	DAM	Ratio of the number of private (protected) attributes to the total number of attributes
Measure of aggregation	MOA	The number of data declarations (class fields) whose types are user-defined classes
Measure of function abstraction	MFA	Number of methods inherited by a class plus number of methods accessible by member methods of the class
Cohesion among methods of 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
Inheritance coupling	IC	The number of parent classes to which a given class is coupled
Coupling between methods	CBM	Total number of new/redefined methods to which all the inherited methods are coupled
Average method complexity	AMC	The number of JAVA byte codes
Afferent couplings	Ca	How many other classes use the specific class
Efferent couplings	Ce	How many other classes are used by the specific class
Maximum McCabe	Max (CC)	Maximum McCabe's cyclomatic complexity values of methods in the same class
Average McCabe	Avg (CC)	Average McCabe's cyclomatic complexity values of methods in the same class
Lines of code	LOC	Measures the volume of the code

name, project version, average code file number, and average defect rate. We use the predecessor version as the training set and the postversion as the test set.

4.2. Evaluation Metrics. We evaluate the performance of our model as F1-measure and AUC. F1-measure is used to measure the stability of DP-ARNN, and AUC is used to assess the discrimination ability of it.

F1-measure is the harmonic mean of the precision and the recall. We define equations (4)–(6) to describe Precision (P), Recall (R), and F1-measure (F) in software defect prediction:

$$P = \frac{N_{d\longrightarrow d}}{N_{d\longrightarrow d} + N_{c\longrightarrow d}}, \tag{4}$$

$$R = \frac{N_{d\longrightarrow d}}{N_{d\longrightarrow d} + N_{d\longrightarrow c}}, \tag{5}$$

Table 2: Java project dataset information.

Project	Versions (pre, post)	Avg files	Avg defect rate (%)
Camel	1.4, 1.6	918	18.1
Lucene	2.0, 2.2	221	53.7
Poi	2.5, 3.0	413	64.0
Xerces	1.2, 1.3	447	15.7
Jedit	4.0, 4.1	309	25.0
Xalan	2.5, 2.6	844	47.3
Synapse	1.1, 1.2	239	30.3

$$F = \frac{2 * P * R}{P + R}. \tag{6}$$

where specifically, symbol c (clean) means files without defects, while symbol d (defective) means files with defects. Three cases are defined as follows: (i) predicting defective files as defective ($d \longrightarrow d$), (ii) predicting defective files as clean ($d \longrightarrow c$), and (iii) predicting clean files as defective ($c \longrightarrow d$). Besides, N denotes the number of each case.

Normally, Precision and Recall cannot be optimal at the same time. For example, if we predict all the program files to be defective, Recall will reach 100%, but Precision will be very low. Therefore, we make a trade-off between Precision and Recall as F1-measure (i.e., the harmonic mean of the two metrics). The range of it is $[0, 1]$, and the higher value means the better stability of the model.

AUC (i.e., area under ROC curve) is based on the area under the ROC (i.e., receiver operating characteristic) curve to evaluate the distinguishing ability of the prediction model. When evaluating the model classifier, the ROC curve first sets different thresholds for classification. The abscissa of the ROC curve is the value of false positive rate (fpr) and the ordinate is the value of true positive rate (tpr). Each classification threshold generates a coordinate (fpr, tpr), and ROC is the curve formed by these coordinate points. AUC is the area under ROC curve. The value of it ranges from 0 to 1, the higher the better. In addition, AUC is appropriate for evaluating class-imbalanced datasets.

Besides, we employ the Friedman test [42–44] as the test of significance of methods. Suppose there are k methods. The Friedman test obeys the chi-square distribution with a $k-1$ degree of freedom, and its original hypothesis is that there is no significant difference in evaluation metrics among the k methods. If the p value of the test result is small enough (i.e., less than 0.05), we can come to the conclusion that the original hypothesis is not established. In other words, there is a significant difference among methods. Moreover, we apply Nemenyi's posthoc test [43] to compare the differences between our proposed DP-ARNN and other baseline methods.

4.3. Baseline Methods. We select the following five baseline methods to compare with our proposed DP-ARNN.

(i) RF: random forest (RF) [45] method based on 20 static code metrics

(ii) RBM + RF: random forest method with hidden features learned by restricted Boltzmann machine (RBM) [46]

(iii) DBN + RF: random forest method with hidden features generated by deep belief network (DBN) [47]

(iv) CNN: a deep learning method based on text sequence convolution, which feeds hidden features learned by CNN to the final classifier.

(v) RNN: a bidirectional recurrent neural network based on LSTM units to generate syntactic and semantic features for defect prediction

We take the same method to generate the inputs of CNN and RNN, which we have mentioned in Section 3.2. When building the network architecture of CNN, we use the same parameter settings as in [32] (i.e., 10 filters each of whose length is 5 and a fully connected layer including 100 hidden nodes). In terms of RNN, its parameter settings are the same as DP-ARNN. As for the inputs of RBM and DBN, we divide each element in each vector by the fixed length (i.e., 2000) for normalization. In addition, RBM's hidden layer has 100 nodes and DBN has 5 hidden layers each of which contains 100 nodes.

The Friedman test is performed on F1-measure among all the methods, whose result is shown in Table 3. The degree of freedom k is 5 since we have 6 methods. The p value of F1-measure among all the 6 methods is 6.37×10^{-5}, which is much less than the baseline 0.05. Therefore, we verify the significant differences among all the methods. Furthermore, Table 4 lists Nemenyi's test result of p values between DP-ARNN and other baseline methods, which indicates that the significant difference between our proposed DP-ARNN and baseline methods is mainly on RBM + RF and DBN + RF. Tables 5 and 6 list F1-measure and AUC comparison of different models. For each project, the result of the best method is shown in bold. The next-to-last row displays the win/tie/loss (W/T/L) statistics between our proposed DP-ARNN and other baseline methods. The last row is the average of the results of the seven projects for each method, and the best is also shown in bold.

4.4. Performance Comparison between Deep Learning Methods and Traditional Methods (RQ1). We first compare three deep learning methods (i.e., CNN, RNN, and DP-ARNN) with two traditional machine learning methods (i.e., RF and RBM + RF). RF is a traditional features-based method with static code metrics, and RBM + RF is a method which first builds a shallow network including two layers (i.e., a visible layer and a hidden layer) to generate hidden features and then feeds them into RF for classification. This comparison is to verify the superiority of deep learning methods in the field of software defect prediction. We conduct the experiments on these projects listed in Table 2. Each project has two versions, each of which the older version is used for model training, and the newer version is used for model evaluation.

Table 3: Friedman test among all the 6 methods.

	k	p value
Baseline	5	0.05
Test result	5	6.37×10^{-5}

Table 4: P values of Nemenyi's posthoc test between DP-ARNN and baseline methods.

	RF	RBM + RF	DBN + RF	CNN	RNN
DP-ARNN	0.104	0.001	0.008	0.900	0.766

Table 5: F1-measure comparison of different models.

Project	DP-ARNN	RF	RBM + RF	DBN + RF	CNN	RNN
Camel	**0.515**	0.396	0.310	0.330	0.473	0.506
Lucene	**0.721**	0.604	0.600	0.623	0.711	0.672
Poi	**0.764**	0.669	0.639	0.652	0.734	0.722
Xerces	**0.270**	0.185	0.128	0.167	0.243	0.262
Jedit	0.560	0.550	0.468	0.500	**0.596**	0.595
Xalan	**0.644**	0.638	0.628	0.623	0.639	0.606
Synapse	0.477	0.414	0.303	0.360	0.424	**0.487**
W/T/L		7/0/0	7/0/0	7/0/0	6/0/1	5/0/2
Average	**0.564**	0.494	0.439	0.465	0.546	0.550

Table 6: AUC comparison of different models.

Project	DP-ARNN	RF	RBM + RF	DBM + RF	CNN	RNN
Camel	**0.790**	0.677	0.674	0.654	0.732	0.766
Lucene	0.680	0.641	0.679	0.682	0.688	**0.693**
Poi	**0.796**	0.636	0.657	0.668	0.745	0.764
Xerces	**0.761**	0.576	0.579	0.560	0.671	0.730
Jedit	0.820	0.797	0.797	0.794	0.841	**0.842**
Xalan	0.674	0.674	**0.676**	**0.676**	0.674	0.654
Synapse	0.645	**0.682**	0.646	0.657	0.632	0.648
W/T/L		5/1/1	5/0/2	4/0/3	4/1/2	4/0/3
Average	**0.738**	0.669	0.673	0.670	0.712	0.728

Table 5 lists the F1-measure values on each project by implementing our proposed DP-ARNN method and other baseline methods. We take project *camel* as an example. The F1-measure values of DP-ARNN, CNN, and RNN are 0.515, 0.473, and 0.506, respectively, while RF and RBM + RF only have 0.396 and 0.310. Obviously, DP-ARNN, CNN, and RNN outperform traditional methods. We can see from the last row of Table 5 that, in average, the deep learning methods achieve higher F1-measure than traditional methods. Especially, DP-ARNN achieves the highest value, indicating the advantage of our proposed DP-ARNN method. These results validate the stability of deep learning-based defect prediction model.

Table 6 lists the AUC values on each project. In most cases, deep learning-based methods including DP-ARNN, CNN, and RNN have higher AUC values than traditional methods. In terms of the average value of the seven projects, DP-ARNN has the best performance and other deep learning-based methods also have an advantage over traditional methods. All these results demonstrate that compared with traditional methods, deep learning methods enhance the discrimination ability between clean code and buggy code.

Based on the analysis above, we come to a conclusion that deep learning methods are superior to traditional machine learning methods for software defect prediction.

4.5. Feature Comparison between Deep Learning Methods and Unsupervised Methods (RQ2). To further demonstrate that features generated by deep learning methods are generally better than typical unsupervised feature extraction methods, we construct an RBM model and a DBN model to extract features from ASTs of programs and feed them into RF for classification. The difference between RBM and DBN is that the former is a two-layer shallow neural network, and the latter is a network that consists of multiple RBMs.

By comprehensively comparing the average F1-measure of RBM + RF and DBN + RF on the seven projects, we can see that the average F1-measure of DBN + RF is higher than the values of RBM + RF, indicating that the information of ASTs of programs can be deeper mined. From the perspective of W/T/L, compared with DBN + RF, DP-ARNN and CNN win 7 times on F1-measure, and RNN also wins 6 times, validating the stability of models based on deep learning methods. As for AUC, the average values of DP-ARNN, CNN, and RNN are all higher than the value of DBN + RF, which means the comprehensive discrimination ability based on deep learning methods outperforms unsupervised learning methods. These results validate the superiority of features extracted from deep learning methods, especially our proposed DP-ARNN method.

4.6. Performance Comparison between DP-ARNN and Other Deep Learning Methods (RQ3). In this section, we compare the performance of our proposed DP-ARNN method with other deep learning methods, including CNN and RNN. We construct a convolutional neural network and a recurrent neural network as our deep learning baseline methods. We implement one-dimensional convolution on elements in each encoded vector in CNN. For RNN, we adopt LSTM as the basic unit and then construct a Bi-LSTM network without attention mechanism.

From the perspective of W/T/L, compared with CNN and RNN, our proposed DP-ARNN wins 6 times and 5 times, respectively, on F1-measure. This indicates that, in terms of the stability of software defect prediction, DP-ARNN has better performance than CNN and RNN. As for AUC, Figure 7 shows the ROC curves of different deep learning methods on the seven projects. DP-ARNN improves CNN an average of 0.03, and RNN an average of 0.01 on AUC. This demonstrates that DP-ARNN improves the distinguishing ability of software defect prediction. In terms of the average F1-measure and AUC of seven projects in Tables 5 and 6, our proposed DP-ARNN improves CNN by 3% on F1-measure and 4% on AUC. In particular, DP-ARNN improves RNN by 3% on F1-measure and 1% on AUC, which indicates that the attention mechanism has a

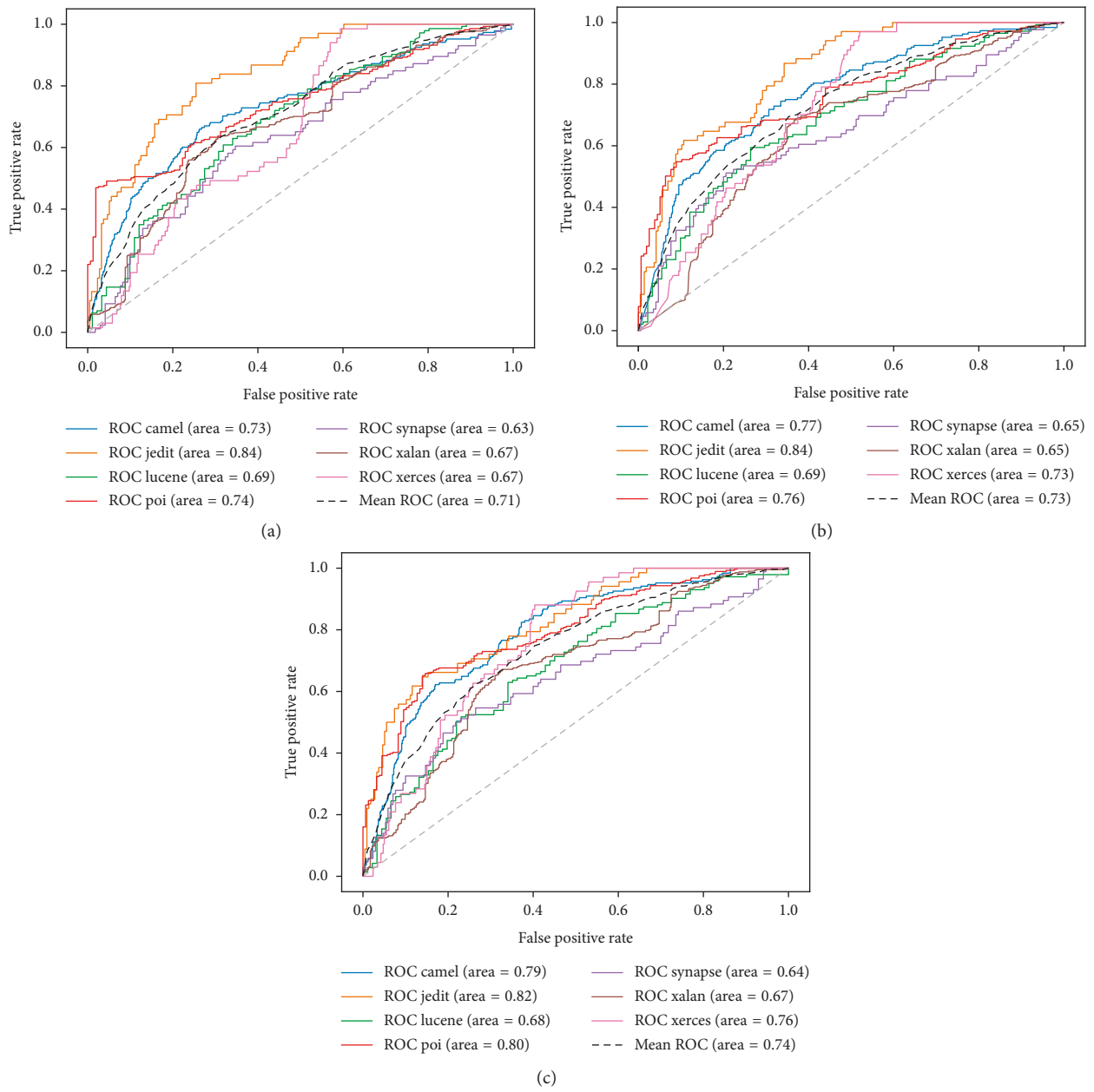


Figure 7: The ROC curves of (a) CNN, (b) RNN, and (c) DP-ARNN, respectively.

positive effect on further generating crucial features which lead to better defect prediction performance.

These results exactly answer our RQ3 that, compared with the typical convolutional neural network and recurrent neural network, our proposed DP-ARNN method can better learn the key syntactic and semantic features of programs with the help of attention mechanism and perform the best.

4.7. Performance under Different Parameter Settings (RQ4). In this section, we discuss how we tune the key parameters in DP-ARNN to achieve the best performance of software defect prediction. We only select part of projects to tune the parameters, considering the cost of training time. We first choose the 90th percentile of AST vector length in the projects as the length of each AST vector. Then we select suitable dimensionality of embedding vectors, and we need to make a trade-off between model precision and training cost. Empirically, the range of it is from 20 to 150. After that, we set the batch size as 32 heuristically and the appropriate epoch is determined by the method of early stopping. That is, the training is stopped when the error of the current model on validation set is worse than the previous one, and we use the parameters in the previous result as the final parameters of the model. More importantly, there are three crucial parameters in our proposed DP-ARNN, including the number of the Bi-LSTM units per layer, the number of the 1st hidden layer nodes, and the number of the 2nd hidden layer nodes. We use F1-measure as the evaluation index. Finally, we calculate the average F1-measure of the projects under different parameter

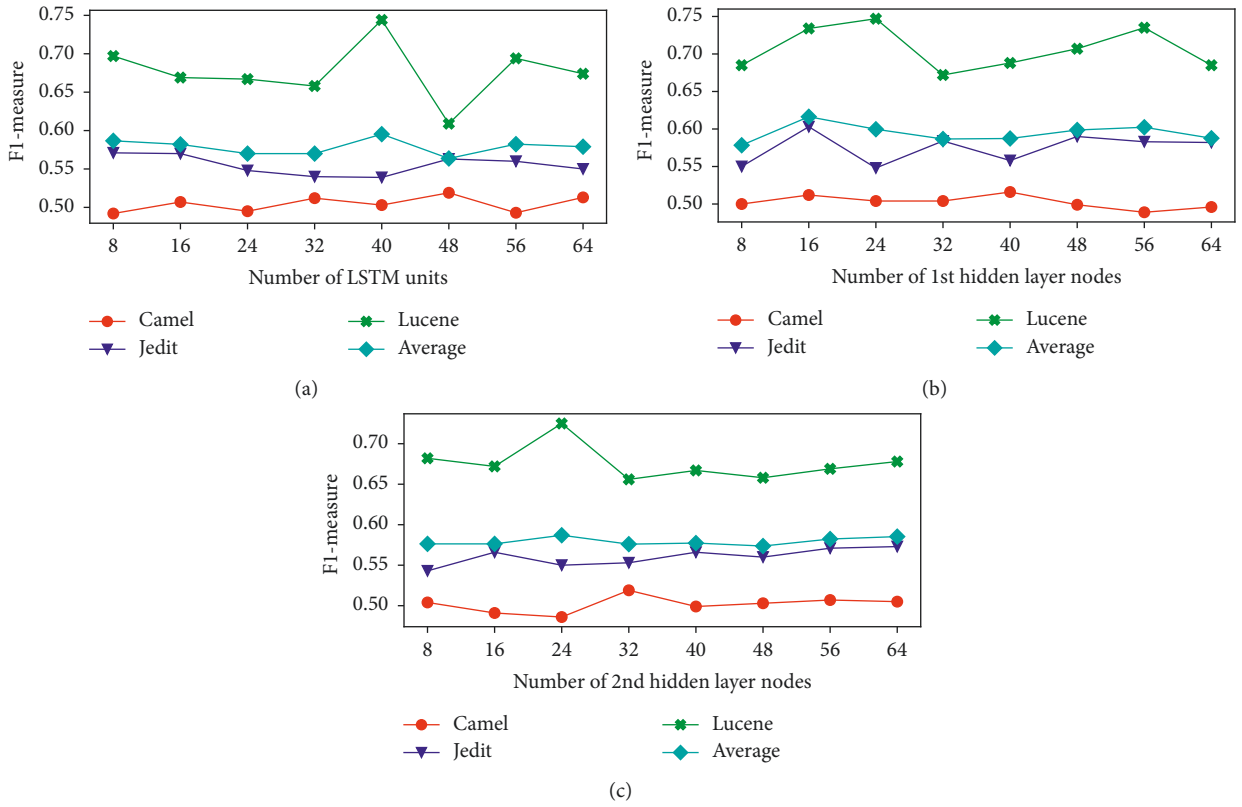


Figure 8: F1-measure of DP-ARNN under different parameter settings. Different numbers of (a) LSTM units, (b) 1st hidden layer nodes, and (c) 2nd hidden layer nodes.

values, choosing the values that the average curve under different parameters reaches the peak.

In our experiments, we select *camel*, *jedit*, and *lucene* for parameter tuning. Figure 8 illustrates the F1-measure of DP-ARNN under different numbers of LSTM units, different numbers of 1st hidden layer nodes and different numbers of 2nd hidden layer nodes. The peak points of F1-measure in average under these three parameters are 40, 16, and 24, respectively. Hence, we use them as the values of the three parameters. Other parameters can also be gained via parameter adjustment, and Table 7 shows all the parameters we have tuned for DP-ARNN with training datasets.

Table 7: Tuned parameters for DP-ARNN.

Parameter	Description (value)
Embedding_dim	The dimensionality of embedding vectors (30)
Vector_length	The length of each AST vector (2000)
Bi-LSTM units	The number of the Bi-LSTM units per layer (40)
1st hidden layer nodes	The number of 1st hidden layer nodes (16)
2nd hidden layer nodes	The number of 2nd hidden layer nodes (24)
Batch_size	The number of training samples that propagated through DP-ARNN at a time (32)
Epoch	One forward/backward pass of all the training samples (20)
Monitor	The evaluation criteria on the validation set (val_acc)
Loss function	The loss function to minimize (binary_crossentropy)
Optimizer	The loss function solver (RMSprop)
Activation	Types of activation used in fully connected layers (tanh, linear, and sigmoid)

5. Conclusion

As the scale and complexity of modern software continue to increase, software reliability has become an important indicator of software quality. To enhance software reliability, in this paper, we propose a deep learning-based method called DP-ARNN (defect prediction via attention-based recurrent neural network), as an aid to software testing and code review, to predict potential code defects in software. Specifically, DP-ARNN leverages RNN to automatically generate syntactic and semantic features from source code. Furthermore, we employ the attention mechanism to capture crucial features, which can further improve our defect prediction performance. Our experiments on seven open-source projects indicate that, in average, DP-ARNN improves the state-of-the-art baseline methods by 14% on F1-measure and 7% on AUC in software defect prediction.

To further evaluate the generality of DP-ARNN in the fields of defect prediction, in the future, we will conduct experiments on more projects, including personal projects

and company projects. Meanwhile, we will implement our method to other programming languages such as Python, Javascript, and C++ to verify the effectiveness of it. Moreover, we will try to embed static code attributes into DP-ARNN, and then test whether the performance of defect prediction can be improved.

Data Availability

There are two different datasets including source code and static code metrics of the seven open-sourced Java projects. The source code of these projects from Apache is available at https://github.com/apache. Datasets which contain static code metrics of theses projects are derived from http://snow.iiar.pwr.wroc.pl:8080/MetricsRepo/.

Conflicts of Interest

There are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was partially supported by the NSF of China under Grant nos. 61772200 and 61702334, Shanghai Pujiang Talent Program under grants no. 17PJ1401900, Shanghai Municipal Natural Science Foundation under Grant nos. 17ZR1406900 and 17ZR1429700, Educational Research Fund of ECUST under Grant no. ZH1726108, and the Collaborative Innovation Foundation of Shanghai Institute of Technology under Grant no. XTCX2016-20.

References

[1] L. L. Minku, E. Mendes, and B. Turhan, "Data mining for software engineering and humans in the loop," *Progress in Artificial Intelligence*, vol. 5, no. 4, pp. 307–314, 2016.

[2] 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.

[3] 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.

[4] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[5] F. Wu, X.-Y. Jing, Y. Sun et al., "Cross-project and within-project semisupervised software defect prediction: a unified approach," *IEEE Transactions on Reliability*, vol. 67, no. 2, pp. 581–597, 2018.

[6] Y. Zhou, Y. Yang, H. Lu et al., "How far we have progressed in the journey? an examination of cross-project defect prediction," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, pp. 1–51, 2018.

[7] M. H. Halstead, "Elements of software science," in *Operating and Programming Systems Series*, Vol. 2, Elsevier, Amsterdam, Netherlands, 1977.

[8] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[9] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," in *Models and Methods of System Dependability*, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, Poland, 2010.

[10] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 359–368, ACM, Orlando, FL, USA, December 2012.

[11] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Proceedings of the Eleventh Annual Conference of the International Speech Communication Association*, Makuhari, Japan, September 2010.

[12] D. M. Powers, "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[13] J. M. Lobo, A. Jiménez-Valverde, and R. Real, "AUC: a misleading measure of the performance of predictive distribution models," *Global Ecology and Biogeography*, vol. 17, no. 2, pp. 145–151, 2008.

[14] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: an empirical case study," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 364–373, IEEE, Madrid, Spain, September 2007.

[15] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 181–190, ACM, Leipzig, Germany, May 2008.

[16] Ö. F. Arar and K. Ayan, "A feature dependent naive Bayes approach and its application to the software defect prediction problem," *Applied Soft Computing*, vol. 59, pp. 197–209, 2017.

[17] R. Mousavi, M. Eftekhari, and F. Rahdari, "Omni-ensemble learning (OEL): utilizing over-bagging, static and dynamic ensemble selection approaches for software defect prediction," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 6, article 1850024, 2018.

[18] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 414–423, ACM, Hyderabad, India, June 2014.

[19] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pp. 382–391, IEEE, San Francisco, CA, USA, May 2013.

[20] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[21] Q. Yu, S. Jiang, and J. Qian, "Which is more important for cross-project defect prediction: instance or feature?," in *Proceedings of the International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 90–95, IEEE, Kunming, China, November 2016.

[22] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[23] L. Peng, B. Yang, Y. Chen, and A. Abraham, "Data gravitation based classification," *Information Sciences*, vol. 179, no. 6, pp. 809–819, 2009.

[24] L. Chen, B. Fang, Z. Shang, and Y. Tang, "Negative samples reduction in cross-company software defects prediction," *Information and Software Technology*, vol. 62, pp. 67–77, 2015.

[25] Y. Yao and G. Doretto, "Boosting for transfer learning with multiple sources," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1855–1862, IEEE, San Francisco, CA, USA, June 2010.

[26] 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.

[27] K. D. Cooper, T. J. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," Technical Report TR02-399, Rice University, Houston, TX, USA, 2002.

[28] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pp. 297–308, IEEE, Austin, TX, USA, May 2016.

[29] G. Lin, J. Zhang, W. Luo et al., "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 7, pp. 3289–3297, 2018.

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

[31] H. K. Dam, T. Pham, S. W. Ng et al., "A deep tree-based model for software defect prediction," 2018, https://arxiv.org/abs/1802.00921.

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

[33] Y. Kim, "Convolutional neural networks for sentence classification," 2014, https://arxiv.org/abs/1408.5882.

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

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

[36] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the International Conference on Knowledge Science, Engineering and Management*, pp. 547–553, Springer, Changchun, China, August 2015.

[37] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 99–108, IEEE, Florence, Italy, May 2015.

[38] 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*, pp. 1480–1489, San Diego, CA, USA, June 2016.

[39] 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 (PROMISE '10)*, p. 9, ACM, Timisoara, Romania, September 2010.

[40] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[41] 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 Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[42] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," *Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[43] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[44] O. Reyes, A. H. Altalhi, and S. Ventura, "Statistical comparisons of active learning strategies over multiple datasets," *Knowledge-Based Systems*, vol. 145, pp. 274–288, 2018.

[45] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and qsar modeling," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[46] I. Sutskever, G. E. Hinton, and G. W. Taylor, "The recurrent temporal restricted Boltzmann machine," in *Proceedings of the Advances in Neural Information Processing Systems*, pp. 1601–1608, Vancouver, Canada, December 2009.

[47] G. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.