Software defect prediction based on gated hierarchical LSTMs

Hao Wang, Weiyuan Zhuang, and Xiaofang Zhang

***Abstract*—Software defect prediction, aimed at assisting software practitioners in allocating test resources more efficiently, predicts the potential defective modules in software products. With the development of defect prediction technology, the inability of traditional software features to capture semantic information is exposed, hence related researchers have turned to semantic features to build defect prediction models. However, sometimes traditional features such as lines of code also play an important role in defect prediction. Most of the existing researches only focus on using a single type of feature as the input of the model. In this paper, a defect prediction method based on gated hierarchical Long Short-Term Memory networks (GH-LSTMs) is proposed, which uses hierarchical LSTM networks to extract both semantic features from word embeddings of Abstract Syntax Trees (ASTs) of source code files, and traditional features provided by the PROMISE repository. More importantly, we adopt a gated fusion strategy to combine the outputs of the hierarchical networks properly. Experimental results show that GH-LSTMs outperforms existing methods under both non-effort-aware and effort-aware scenarios.**

***Index Terms*—software defect prediction, LSTM, hierarchical model, abstract syntax tree**

I. Introduction

WITH the scale of software increasing continuously, software defect prediction technology has attracted more and more attention [1], [2], [3]. The results given by defect prediction models can help developers or testers to determine whether a software module is defective in the early stage of the software development life cycle so that they can better allocate test resources, arrange the test process more efficiently, and consequently improve the quality of software products.

The traditional software defect prediction models mainly predict whether the program is defective based on the statistics of program characteristics, which assumes that the flawless modules and the defective modules have distinguishing features or feature combinations, such as the total number of lines of code, the number of added or deleted lines of code, the number of changed files, quality features, system complexity features [4], Halstead features [5] and so on. By taking these features as inputs of traditional classification algorithms, traditional defect prediction models divide software modules into two categories: buggy or clean, so as to predict defects. However, these approaches ignore the semantic information of the source code, and programs with different semantic features may have similar or even the same traditional features [6]. Under such circumstances, traditional software features are incapable of distinguishing two programs.

H Wang, W Zhuang, and X Zhang are with the School of Computer Science and Technology, Soochow University, Suzhou, 215006, China (e-mail: 20185227067@stu.suda.edu.cn; 20204227007@stu.suda.edu.cn; xfzhang@suda.edu.cn). Corresponding author: Xiaofang Zhang

In recent years, with the wide application of deep learning, researchers have begun to leverage deep learning to perform effective feature generation for defect prediction. A common practice is utilizing AST parser to extract semantic information from source code files [6], and then encoding the ASTs into numeric vectors in order to feed them to deep neural networks, which extract semantic features further. But semantic features are not omnipotent and can not represent some information included in traditional features such as the number of lines of code, which is proved rather significant for defect prediction [7].

Therefore, we propose a GH-LSTMs model, which extracts both semantic features from word embeddings of ASTs and traditional features provided by the PROMISE[1] repository using the hierarchical architecture of LSTMs. Additionally, to appropriately determine the combination ratio of the two kinds of features, we introduce the gated fusion mechanism to merge the output of two LSTMs, which utilizes self-learning gates to filter the information passing through.

Our work focuses on with-in project file-level software defect prediction and evaluates prediction performance by means of metrics under both non-effort-aware scenario and effort-aware scenario. Experimental results on 10 projects show that, under non-effort-aware scenario, our proposed GH-LSTMs method outperforms the state-of-the-art methods in terms of $F-measure$. Additionally, under effort-aware scenario, GH-LSTMs achieves better performance than the state-of-the-art methods in terms of $PofB20$ and IFA, and comparable performance with the state-of-the-art methods in terms of P_{opt}.

The main contributions of this paper are:

- We propose a GH-LSTMs model to extract both semantic and traditional software features and effectively combines the extracted features using gated fusion mechanism.
- We conduct rigorous experiments to evaluate the performance of the GH-LSTMs model under both non-effort-aware and effort-aware scenarios.
- We demonstrate that proper feature fusion can significantly boost the performance of defect prediction.

The rest of this paper is summarized as follows: Section 2 introduces the background, including Deep Belief Network (DBN), word embedding, Recurrent Neural Network (RNN), LSTM and the motivation of this paper; A hierarchical gated networks model is introduced in Section 3; Section 4 describes

[1]http://openscience.us/repo/defect

the experimental setup and results, and then demonstrates the potential threats to validity; Section 5 discusses the related work; The conclusion of this paper and the future work are presented in Section 6.

II. Background

A. Deep belief network and word embedding in software defect prediction

Software defect prediction is able to help developers allocate testing resources by predicting potential defects in software modules. Since software defect prediction is based on the feature extraction of software modules, there are various kinds of software features proposed by existing studies, which can be categorized into two main types: code metrics and process metrics. In recent years, some researchers have proposed semantic features, which are generated by using deep learning technology to extract features from ASTs of source code files automatically. Based on existing studies, semantic features are usually obtained from two techniques: Deep Belief Network [6] and Word Embedding [8], [9].

DBN [10] is a probability generation model, which establishes a joint distribution between observation results and labels, and performs evaluations in both observation/label direction and label/observation direction. DBN is composed of several Restricted Boltzmann Machines (RBM). Each RBM has two layers of neurons: One layer is called visible layer, which is composed of visible units and used to accept the training data. The other layer is called hidden layer, which consists of hidden units and serves as feature detectors. The training process of an RBM can be described as:

$$P(h_j|v) = \sigma(b_j + \sum_{(i=1)}^{N} W_{ij} v_i) \tag{1}$$

$$P(v_i|h) = \sigma(c_i + \sum_{(j=1)}^{M} W_{ij} h_j) \tag{2}$$

Equation (1) is the probability function of the activation result of the neural unit h_j in hidden layer, when the state values of all the neurons in visible layer are given. Equation (2) is the probability function of the reconstruction result of a neural unit v_i in visible layer, when the state values of all the neurons in hidden layer are given. In these two equations, the common parameter σ is the sigmoid function, and W_{ij} is the weight between ith neuron in visible layer and jth neuron in hidden layer. In addition, h_j is the value of jth neuron in hidden layer, v_i is the value of ith neuron in visible layer, b_j is the bias of jth neuron in hidden layer, c_i is the bias of ith neuron in visible layer, N is the number of neurons in the visible layer, M is the number of neurons in the hidden layer. The weight and bias matrices are automatically learned through log-likelihood stochastic gradient descent.

DBN is made up of multiple RBMs. To be specific, the hidden layer of the last RBM will be treated as the visible layer of the current RBM. In the training process, each layer of the RBM network is trained independently to ensure that as much feature information as possible is preserved when the feature vectors are mapped to different feature spaces. Then the backpropagation mechanism is set up, where the output vector of an RBM is treated as its input vector to tune the weights between layers in order to make a better reconstruction of the input data before. When all the iterations are finished, a trained DBN model can be obtained to extracted features from the original input data. Semantic software features can be generated according to the process above using the DBN model.

On the other hand, some researchers [8], [9] suggest that semantic features can also be generated by word embedding techniques. Compared with DBN, word embedding can make better use of the context of AST node sequences. Normally word embedding is accomplished by word2vec [11] or GloVe (Global Vectors) [12].

GloVe is used in this paper to transform an AST node token(as a word) into a vector composed of real numbers, which is calculated based on the semantic relation between words, such as frequency and analogy. First of all, a corpus of the AST node sequences is build and used to construct the co-occurrence matrix. Each element X_{ij} in the matrix represents the number of times that word i and word j appear together in a specific size of the context window. Furthermore, GloVe proposes a decreasing weighting function to calculate the number of co-occurrence: $decay = 1/d$, based on the distance d between two words in the context window, to calculate the weight, i.e., the longer the distance, the less the weight of two words in the total count. After the co-occurrence matrix is built, the GloVe model can be trained based on this loss function:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(\omega_i^T \tilde{\omega}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2 \tag{3}$$

where V is the size of the vocabulary, ω_i and $\tilde{\omega}_j$are the expected word vectors, b_i and $\tilde{b}_j$ are the bias term of the two word vectors respectively, $f(X_{ij})$ is the weighting function, which is defined as follows:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha} & if\ x < x_{max} \\ 1 & oterwise \end{cases} \tag{4}$$

where α and x_{max} are adjustable parameters of the weighting function.

Semantic software features presented in the form of word embedding can be obtained according to the process above using GloVe.

B. Recurrent Neural Network in software defect prediction

Recurrent Neural Network(RNN) achieves outstanding performance in processing sequential data with the help of its inner recurrent architecture. RNN processes the input sequence by traversing all elements and saving a state memory that contains information about what has been viewed. LSTM [13], as an improved variant of basic RNN, is designed to solve the problem of vanishing gradient and exploding gradient during the training of traditional RNN. A typical LSTM cell is shown in Fig. 1. LSTM makes use of three gates to control the input and output streams of information. The forget gate determines how much of the state memory from last time step will be reserved to the current time step; the input gate determines how much of the current input will be saved to the current state memory; and the output gate is used to control how much of the current state memory should be output in the current time step. There are a weight matrix and a bias term for each gate, which will be automatically learned in the training phase.

With the development of software defect prediction technology, many studies [8], [14], [15] use RNN to establish classifiers and perform feature extraction with the purpose of achieving better prediction results.

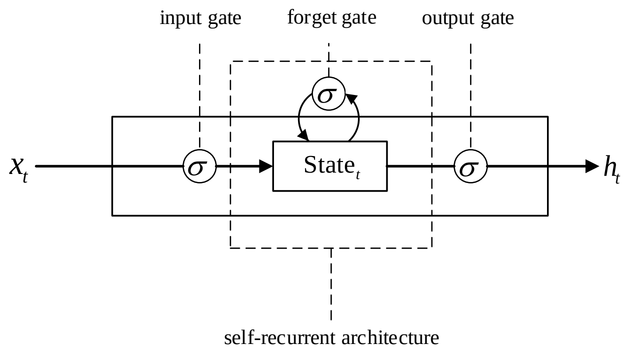


Fig. 1. LSTM model

C. Motivation

Based on previous studies, there are two types of software defect prediction features: traditional features and semantic features. On the one hand, semantic features can capture the semantic information of code snippets that traditional features are unable to represent. For example, there are two Java files in Fig. 2, file 1(clean) and file 2(buggy) have the same code statements only with different positions of the statement 'start=1;'. If we use traditional features such as code complexity features to describe these two code snippets, we will get the identical results, thus unable to distinguish the buggy file. On the other hand, only using semantic features will also lose some statistical information about source code, such as the LOC feature, which is significant for defect prediction [7] but can not be obtained from semantic features. To sum up, traditional features are shown to be informative in distinguishing buggy code while semantic features perform better in capturing semantic information of code. This encourages us to design a defect prediction method that makes use of both kinds of features.

The challenge to proper feature combination is how to determine the reasonable ratios of two kinds of features since the performance of each feature in defect prediction may vary in different software. To solve this problem, we propose a novel model named Gated Hierarchical LSTMs. It leverages a powerful hierarchical LSTMs architecture to ensure the full extraction of each feature and utilizes a self-learning gated fusion mechanism to automatically determine the optimal ratio in feature combination.

III. Approach

Most of the existing studies only focus on one kind of feature, or simply do concatenation of different features. In order to make the best use of both kinds of features, inspired by our previous work [16], we designed a gated hierarchical LSTMs model for defect prediction, which is capable of taking full advantage of both semantic features and traditional features. Fig. 3 shows the whole framework of our method.

First, we manage to extract semantic features and traditional features during data preprocessing. Then the two kinds of features will be taken as inputs to a hierarchical LSTMs to make a further extraction of the hidden information of each feature. After that, the outputs of hierarchical LSTMs will be fed into a gated merge layer where feature fusion happens under the control of gated mechanism. Finally, we input the combined feature into a fully-connected layer to get the defect distribution, i.e., clean or buggy.

A. Data preprocessing

To extract the semantic feature of source code files, we firstly parse source files into AST token sequences, which reserves the semantic information of code. For each source code file, an AST token sequence is generated. Then we join the sequences from the same project together to produce a semantic code corpus, which will be used by GloVe to train the word embedding model for this project. Once the word embedding model is acquired, it can be used to generate the vector representation of AST token sequence, i.e., the semantic feature that used in our work.

For traditional features, we selected 18 code metrics provided by the PROMISE repository according to previous studies [6], [8], [17].

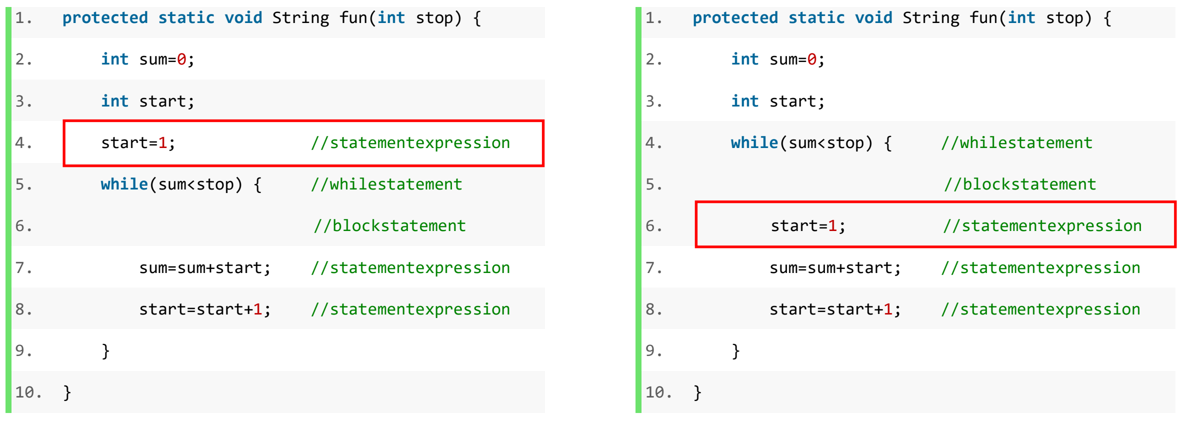
B. Hierarchical LSTMs

In order to utilize both semantic features and traditional features, we construct a hierarchical LSTMs model composed of a semantic-level LSTM and a traditional-level LSTM. As Fig. 3 shows, the input of the semantic-level LSTM is a sequence made up of semantic word vectors. Each LSTM unit receives the output vector from the last LSTM unit and input vector of the current time step, which is defined as

$$E_s = W_h \cdot h_{(i-1)} \oplus W_x \cdot x_i \tag{5}$$

In which $h_{(i-1)}$ refers to the output of previous hidden layer, W_h refers to the weight matrix of the hidden layer, x_i refers to the ith input vector, W_x refers to the weight matrix of the input vector, and $\oplus$ refers to the concatenation operation. With the iteration of i continuing, the extracted information will be added into E_s sequentially.

Similarly, for traditional features, we treat the 18 code metrics as a sequence containing 18 time-steps and feed the sequences into the traditional-level LSTM to extract hidden information in the same way as semantic-level LSTM.



```java
protected static void String fun(int stop) {
    int sum=0;
    int start;
    start=1;              //statementexpression
    while(sum<stop) {     //whilestatement
                          //blockstatement
        sum=sum+start;    //statementexpression
        start=start+1;    //statementexpression
    }
}
```

File1.java

```java
protected static void String fun(int stop) {
    int sum=0;
    int start;
    while(sum<stop) {     //whilestatement
                          //blockstatement
        start=1;          //statementexpression
        sum=sum+start;    //statementexpression
        start=start+1;    //statementexpression
    }
}
```

File2.java

Fig. 2. A concrete example

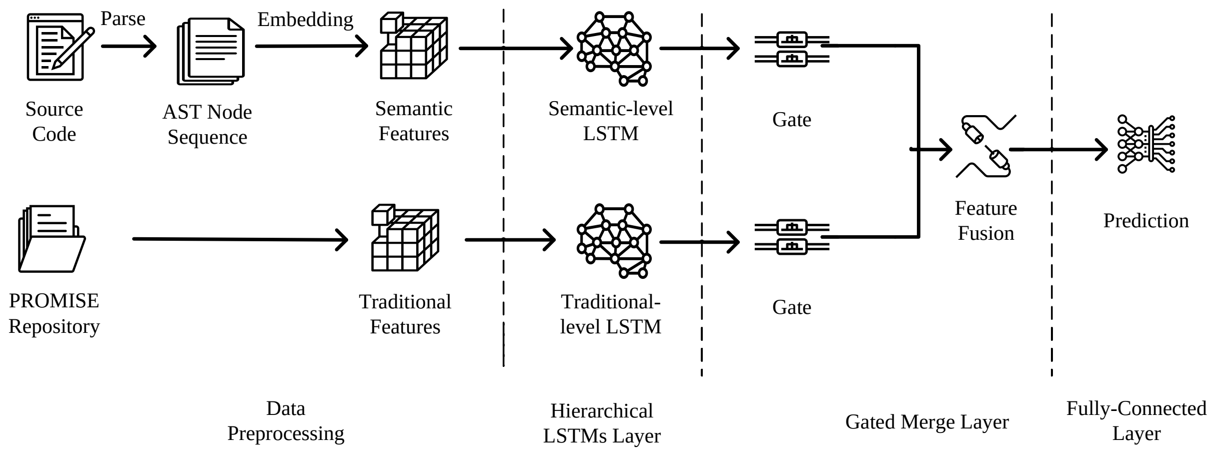


Fig. 3. GH-LSTMs

C. Gated merge layer

The highlight of our method is that we figure out a proper way to automatically determine the best proportion of each kind of feature during the fusion phase. We feed the outputs of semantic-level LSTM and traditional-level LSTM into a gate function separately, where a fully-connected layer is used to generate a filter for the information passing through. Then the gated results will be merged by simple concatenation to generate the combined feature, defined as

$$H_m = concatenate(h_t \otimes \sigma(W_t \cdot h_t + b_t), h_s \otimes \sigma(W_s \cdot h_s + b_s)) \quad (6)$$

where h_t represents the final output of the traditional-level LSTM, h_s represents the final output of the semantic-level LSTM, W_t, b_t, W_s and b_s are learned parameters of the gates, σ is the sigmoid function, $\otimes$ is element-wise product, and *concatenate* represents simple concatenation function.

D. Model training

In this step, a softmax layer is used to receive the output H_m of gated merge layer and predict whether the file is defective or not:

$$y = softmax(W \cdot H_m + b) \quad (7)$$

In which W is the weight matrix of the softmax layer, and b is the bias item.

The backpropagation algorithm is used during model training, trying to minimize the cross-entropy error of defect classification to optimize the model:

$$loss = -\sum_{(i \in D)} \sum_{(j \in C)} \hat{y}_i^j \log y_i^j + \lambda ||\theta||^2 \quad (8)$$

where D represents all training instances, C is the result categories (buggy or clean), $\hat{y}$ refers to the correct distribution of results, y is the predicted distribution of result, and $\lambda ||\theta||^2$ represents l_2 regularization.

IV. Experiments Settings

A. Evaluated projects and datasets

It is necessary to get the correct defect labels of the source code files to build training/testing datasets. We choose to use publicly available data from the PROMISE [18] repository, which is a widely used repository in defect prediction studies [6], [8], [9], [18], [19], [20]. Specifically, based on these previous studies, we selected 10 open-source java projects from PRMOISE repository for the convenience of estimating the performance of our proposed method. PROMISE datasets provide file names, defect labels, and traditional code metrics for different versions of various software projects. According to the projects and version numbers in the repository, we locate the corresponding source code files from GitHub[2] and Apache[3] official websites. Table I shows the basic information of all the projects we used, including version numbers, the average number of files, and the average bug rate of each project. The average number of source code files of all projects is 414, and the average bug rate of all projects is 29.5%.

TABLE I
DATASET DESCRIPTION

Project	Versions	Avg files	Avg bug rate (%)
ant	1.5, 1.6, 1.7	463	19.8
camel	1.2, 1.4, 1.6	815	23.9
jedit	3.2, 4.0, 4.1	296.7	27.6
log4j	1.0, 1.1	122	29.6
lucene	2.0, 2.2, 2.4	260.7	54.9
xalan	2.4, 2.5	763	31.7
xerces	1.2, 1.3	446.5	15.7
ivy	1.4, 2.0	399	6.7
synapse	1.0, 1.1, 1.2	220.3	22.6
poi	1.5, 2.5, 3.0	354.7	62.5
Avg	-	414.9	29.5

The semantic features are learned from the AST token sequences of the source code files. We use *javalang*, an open-source Python library, to extract AST token sequences from the source code files. According to existing study[6], we only extracted three categories of AST nodes: method invocation type, declaration type and control-flow type with the intention of avoiding the noise that might be introduced by some specific AST node types such as intrinsic type declaration. The details of selected AST nodes are presented in Table II. Then the extracted AST token sequences will be used to build semantic code corpus for training a GloVe model as described in Section III-A, which later will be used to generate semantic features. As for the traditional feature, we selected 18 code metrics provided by the PROMISE repository like the existing researches [6], [8] and use them as the traditional feature inputs for our GH-LSTMs model, the details of these code metrics are presented in Table III.

B. Baseline setting

In this paper, we select the following three baseline methods as comparative methods to estimate the performance of our proposed GH-LSTMs:

- DBN [6](DBN Feature + LSTM): LSTM model with semantic features generated by DBN.
- SCE [14](Semantic-Code-Embedding + LSTM): LSTM model with semantic features generated by word embedding.

[2]https://github.com/
[3]https://www.apache.org/

TABLE II
THE SELECTED AST NODES

ReferenceType	MethodInvocation
MethodDeclaration	TypeDeclaration
ClassDeclaration	EnumDeclaration
IfStatement	WhileStatement
DoStatement	ForStatement
AssertStatement	BreakStatement
ContinueStatement	ReturnStatement
ThrowStatement	SynchronizedStatement
TryStatement	SwitchStatement
BlockStatement	StatementExpression
TryResource	CatchClause
CatchClauseParameter	catchclauseparameter
switchstatementcase	forcontrol
enhancedforcontrol	

- DP-AM [8]: A single bidirectional recurrent neural network based on attention mechanism with semantic features generated by word embedding and auxiliary input of traditional features provided by PROMISE repository. Note that DP-AM does not leverage a hierarchical architecture of networks, which means that the traditional features are combined with the extracted semantic information by simple concatenation.

For each project listed in Table I, we choose every two consecutive versions to build the training and testing datasets to evaluate the performance of a defect prediction method. Specifically, we use the earlier version of two versions to build the training set to train the defect prediction model, and then use the later version to build the testing set for evaluation. In total, we conduct 16 sets of experiments on 10 projects for every method.

C. Evaluation

1) Metrics for non-effort-aware evaluation: In non-effort-aware evaluation, we choose $F-measure$ [21] as the evaluation metric, which is widely used in software defect prediction [6], [8], [19], [22]. $F-measure$ is defined as the harmonic mean of $Precision$ and $Recall$. The detailed definitions are shown as follows:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{9}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{10}$$

$$F-measure = \frac{2 * P * R}{P + R} \tag{11}$$

In defect prediction tasks, higher $Precision$ indicates that there are more real buggy instances among all instances that model predicts to be buggy, which will reduce the time and energy wasted on checking false positive instances. Higher $Recall$ means that the model is capable of detecting more real buggy instances, which is often vital to the safe-critical software products. However, $Precision$ and $Recall$ have their own limitations, and can not fully reflect the overall performance of the model. For example, a model can get a $Recall$ of 1 by simply predicting all instances as $buggy$, which will lead to a rather low $Precision$. So in this paper, we choose $F-measure$

TABLE III
TRADITIONAL FEATURES SELECTED FROM PROMISE DATASETS

Measure of Functional Abstraction (MFA)	Inheritance Coupling (IC)
Coupling Between Methods (CBM)	Response for a Class (RFC)
Data Access Metric (DAM)	Efferent couplings (Ce)
Coupling between object classes (CBO)	Measure of Aggregation (MOA)
Lines of Code (LOC)	Weighted methods per class (WMC)
Afferent couplings (Ca)	Depth of Inheritance Tree (DIT)
Number of Children (NOC)	Lack of cohesion in methods (LCOM3)
Lack of cohesion in methods (LCOM)	Cohesion Among Methods of Class (CAM)
Average Method Complexity (AMC)	Number of Public Methods (NPM)

to serve as a trade-off metric of *Precision* and *Recall* under non-effort-aware scenario.

2) Metrics for effort-aware evaluation: Considering that developers or testers always want to inspect less code while detecting as many defects as possible, i.e., to find bugs more effectively, we choose three effort-aware metrics that can measure cost efficiency of code inspection according to results given by defect prediction models, which are $PofB20$ [2], IFA [23] and P_{opt} [24]. These metrics are frequently used to evaluate the performance of defect prediction models under effort-aware scenarios [6], [7], [25], [26].

$PofB20$ is defined as the percentage of captured buggy instances from all buggy instances when 20% of whole inspection efforts are completed. $PofB20$ is rather meaningful in the real-life scenario since not every test team has enough resources to test all the source files. A higher $PofB20$ performance of the defect prediction model means a higher percentage of captured bugs when only checking a limited number of LOC.

IFA is defined as the number of false alarms encountered before testers find the first real defect according to the buggy probabilities given by prediction models. The existing study shows that developers would be frustrated and are not likely to continue inspecting the other changes when the IFA is high [6], [7], [27].

P_{opt} is the normalized version of the effort-aware performance indicator and is based on the concept of the Alberg diagram [28], which demonstrates the effort-aware performances of a prediction model [6], [7], [25], [26]. An example is shown in Fig. 4. The x-axis represents the percentage of the amount of inspected LOC, and the y-axis represents the *Recall* achieved by a prediction model. The **Optimal** line in Fig. 4 means the case where all files are sorted in descending order by defect-density. The **Worst** line means the case where all files are sorted in ascending order by defect-density. The **Prediction** line means the actual prediction model. The **Random** line represents the results of random prediction. Obviously, the curve of a good prediction model should be above the **Random** curve and as close to the **Optimal** curve as possible. The function $Area(x)$ is defined as the area under the curve corresponding to the prediction model. Given a prediction model m, P_{opt} can be calculated as:

$$P_{opt} = 1 - \frac{Area(Optimal) - Area(m)}{Area(Optimal) - Area(Worst)} \quad (12)$$

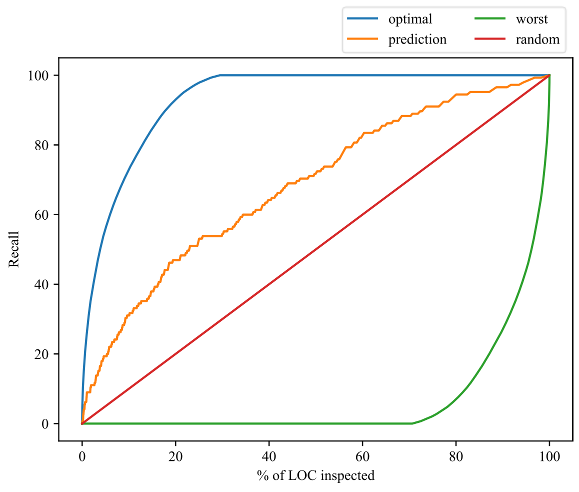


Fig. 4. An example of Alberg diagram

3) Win/Tie/Loss indicator: We also apply the Win/Tie/Loss indicator to compare the performance of different models further, which has been used in prior works for performance comparison between different methods [8], [15]. For each task, we repeat the model training and testing of our GH-LSTMs and other baseline models for 30 times. Then we conduct Wilcoxon signed-rank test and Cliff's delta test to analyze the performance of GH-LSTMs and other methods.

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to determine whether two matched samples have the same distribution. If the p value of Wilcoxon signed-rank test less than 0.05, then the difference between two matched samples is considered to be significantly different and otherwise not. To mitigate the effect caused by multiple tests, we conduct Benjamini-Hochberg correction [29] and report the BH-corrected p values, which are used to calculate the Win/Tie/Loss indicator later.

The Cliff's delta test is a non-parametric effect size test that measures the effective levels of difference between two sets of observation data, and can be regarded as a complementary analysis for the Wilcoxon signed-rank test. Table IV shows the mappings between the Cliff's delta values($|\delta|$) and their effective levels.

To be specific, we make the following comparisons to determine the result of Win/Tie/Loss indicator: for a baseline method M, if GH-LSTMs outperforms M with the p value of Wilcoxon signed-rank test less than 0.05 and the Cliff's delta

TABLE IV
MAPPINGS BETWEEN THE CLIFF'S DELTA VALUES($|\delta|$) AND THEIR EFFECTIVE LEVELS

Cliff's delta	Effective levels
$|\delta| < 0.147$	Negligible
$0.147 \leq |\delta| < 0.33$	Small
$0.33 \leq |\delta| < 0.474$	Medium
$0.474 \leq |\delta|$	Large

value greater than or equal to 0.147, the difference between these two models is statistical significant and can not be ignored. At this time, we mark the GH-LSTMs as a 'Win'. In contrast, if the model M outperforms GH-LSTMs with a p <0.05 and a Cliff's delta≥0.147, GH-LSTMs will be marked as a 'Loss'. Otherwise, we mark the case as a 'Tie'. After all cases are marked, we calculate the Wins, Ties and Losses for GH-LSTMs against each technique on every task.

D. Parameters setting

Every defect prediction method implemented in this paper can be divided into two steps: feature extraction and classifier training. In the feature extraction phase, we set the same training parameters of GloVe model(used by SCE, DP-AM and GH-LSTMs) to generate the semantic features. To be specific, we set the embedding dimensions to be 40 and keep all other parameters as their default values[4]. As for the DBN model that generates DBN features, we stick to the original study [6], i.e., we set the number of hidden layers to 10, the number of nodes in each hidden layer to 100 and the number of iterations to 200. In the classifier training phase, we set the same parameters for all the LSTM networks used in GH-LSTMs and the baseline methods. Details of these parameters are shown in Table V. Note that DP-AM uses a bidirectional RNN instead of LSTM and employs attention mechanism to boost performance, so we set the parameters of DP-AM according to the original study [8].

V. EXPERIMENTAL RESULTS

In this section, we mainly focus on the following three research questions:

RQ1: How does GH-LSTMs perform under non-effort-aware scenario?

RQ2: How does GH-LSTMs perform under effort-aware scenario?

RQ3: How do external parameters affect the performance of GH-LSTMs?

The purpose of RQ1 and RQ2 is to verify the performance of GH-LSTM under the non-effort-aware scenario and the effort-aware scenario, respectively. RQ3 aims to explore how external parameters influence the performance of GH-LSTMs.

A. Answer to RQ1: How does GH-LSTMs perform under non-effort-aware scenario?

The results of $F-measure$ of all 16 sets of experiments are listed in the Table VI, and Fig. 5 shows the distribution of $F-measure$ values of 4 models on all tasks in the form of box-plot, from which we can have the observation that our GH-LSTMs outperforms all baseline methods in terms of median $F-measure$. In Table VI, the 'Task' column stands for the project name and the corresponding versions used as training and testing sets. For example, '*ant_1.5_1.6*' means this task uses the source files from version 1.5 of project *ant* to build the training set that is used to train the defect prediction model, and uses source files from version 1.6 of project *ant* to build the testing set to evaluate the performance of the trained model.

The '$F-measure$' column shows the $F-measure$ values of our proposed GH-LSTMs method and the other three baseline methods. For each task, the result of the best method is presented in bold. Generally speaking, on average, our GH-LSTMs model performs better than all the baseline models in terms of $F-measure$. Specifically, compared with DBN and SCE, our method makes an improvement in terms of $F-measure$ by 16.7 percentage points and 11.1 percentage points on average, which demonstrates the superiority of taking traditional features into consideration. On the other hand, our method outperforms DM-AP by 5.2 percentage points in terms of $F-measure$. Note that DM-AP leverages a more advanced bidirectional RNN based on attention mechanism and also makes use of traditional features by simply concatenating them and the extracted semantic information. This means that under non-effort-aware scenarios, our hierarchical architecture can extract more hidden information from traditional features, and our gated fusion strategy does a better job in feature combination than simple concatenation.

The '$p(\delta)$' column shows p values of Wilcoxon signed-rank test and Cliff's delta values. For the p value of Wilcoxon signed-rank test, when the value is not less than 0.05, the original value is displayed; otherwise, it is replaced by '<0.05'. For Cliff's delta value, it is replaced by the effective level of Cliff's delta value. For the delta value of cliff, replace it with the effective level of cliff delta value. We add '+' or '-' before the effective level to discriminate the positive and negative Cliff's delta values. For example, compare GH-LSTMs with DBN on *ant_1.5_1.6*, the p value is less than 0.05 and the Cliff's delta value is 1.000(≥0.474), so the '$p(\delta)$' of 'GH-LSTMs vs. DBN' is '<0.05(+Large)', and according to the Win/Tie/Loss indicator, we can mark the GH-LSTMs as a 'Win'. From the row 'Average & Win/Tie/Loss' in Table VI, it can be seen that our GH-LSTMs model significantly outperforms other models in most tasks.

In conclusion, the results of $F-measure$ show that: Feature fusion can effectively improve the performance of the model under non-effort-aware scenario: DP-AM and GH-LSTMs perform better than DBN and SCE, which is resulted from the ability of DP-AM and GH-LSTMs to combine both traditional features and semantic features; GH-LSTMs outperforms DP-AM(which combines two kinds of features by simple concatenation) because the hierarchical architecture is more effective in feature extraction and the gated fusion mechanism performs a better combination of two kinds of features.

[4]https://github.com/stanfordnlp/GloVe/blob/master/demo.sh

TABLE V
HYPER PARAMETERS FOR ALL LSTM NETWORKS

Parameter	Description (value)
LSTM units	The number of the LSTM units in each layer (128).
Batch size	The number of training samples that are fed to LSTM at a time (2048).
Epoch	One forward/backward pass of all the training samples (200).
Optimizer	The function that optimizes trainable parameters during training (adam).
Activation	The activation function used in fully connected layers. (sigmoid).

TABLE VI
$F-measure$ VALUES OF DBN, SCE, DP-AM AND GH-LSTMS

	$F-measure$				$p(\delta)$		
Task	DBN	SCE	DP-AM	GH-LSTMs	GH-LSTMs vs. DBN	GH-LSTMs vs. SCE	GH-LSTMs vs. DP-AM
ant_1.5_1.6	0.497	0.284	0.557	**0.582**	<0.05(+Large)	<0.05(+Large)	<0.05(+Medium)
ant_1.6_1.7	0.305	0.341	0.522	**0.584**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
camel_1.2_1.4	0.228	0.340	0.393	**0.415**	<0.05(+Large)	<0.05(+Large)	0.134(+Negligible)
camel_1.4_1.6	0.303	0.398	0.399	**0.441**	<0.05(+Large)	0.075(+Small)	<0.05(+Medium)
ivy_1.4_2.0	0.086	0.183	0.321	**0.393**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
jedit_3.2_4.0	0.374	0.465	0.509	**0.561**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
jedit_4.0_4.1	0.421	0.549	0.553	**0.634**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
log4j_1.0_1.1	0.507	0.621	0.644	**0.739**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
lucene_2.0_2.2	**0.759**	0.710	0.667	0.734	<0.05(-Large)	<0.05(+Medium)	<0.05(+Large)
lucene_2.2_2.4	0.482	0.678	0.707	**0.743**	<0.05(+Large)	<0.05(+Large)	<0.05(+Medium)
poi_1.5_2.5	0.779	0.829	0.790	**0.842**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
poi_2.5_3.0	0.782	0.776	0.759	**0.822**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
synapse_1.0_1.1	0.349	0.370	0.440	**0.464**	<0.05(+Large)	<0.05(+Large)	0.086(+Small)
synapse_1.1_1.2	0.299	0.428	0.517	**0.605**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
xalan_2.4_2.5	0.513	0.584	0.614	**0.658**	<0.05(+Large)	<0.05(+Large)	<0.05(+Medium)
xerces_1.2_1.3	0.213	0.239	**0.347**	**0.347**	<0.05(+Large)	<0.05(+Large)	0.992(-Negligible)
Average & Win/Tie/Loss	0.431	0.487	0.546	**0.598**	15/0/1	15/1/0	13/3/0

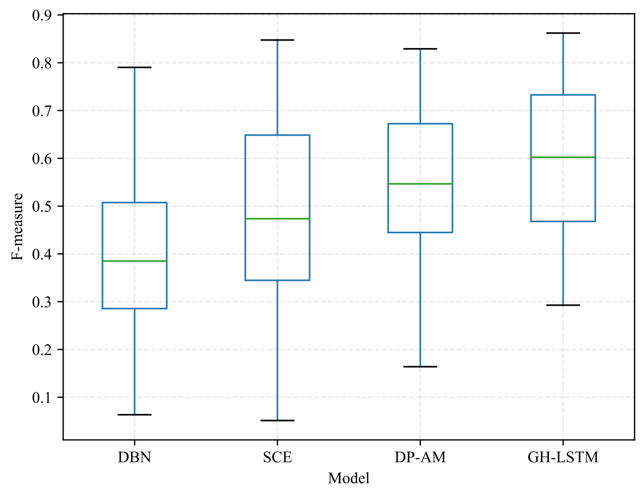


Fig. 5. Overall $F-measure$ comparison of DBN, SCE, DP-AM and GH-LSTMs

B. Answer to RQ2: How does GH-LSTMs perform under effort-aware scenario?

To answer this question, we compare the performance of our proposed GH-LSTMs method with other baseline methods under effort-aware scenario using the $PofB20$, IFA and P_{opt}. Table VII, Table VIII and Table IX respectively show the performance of the four methods on $PofB20$, IFA and P_{opt}.

Table VII shows the $PofB20$ values of all 16 sets of experiments of our method and three baselines. For example, the $PofB20$ score of DBN on task 'ant_1.5_1.6' is 0.261, which means by inspecting source code files in the decreasing order of defect probability given by the DBN method, we can detect 26.1% real bugs when 20% of the total lines of code are inspected. For each task, the highest $PofB20$ value is shown in bold. As we can see, GH-LSTMs outperforms other methods on most of the experiments(11 of 16). On average, our method outperforms DBN, SCE and DP-AM by 2.1 percentage points, 0.3 percentage points and 5.9 percentage points respectively. Additionally, according to the result of Win/Tie/Loss comparison, our GH-LSTMs model significantly outperforms SCE, i.e., the best baseline model, with 5 'Wins', 10 'Ties', and only 1 'Loss'. These results validate that GH-LSTMs method can detect more bugs under effort-aware scenarios. Moreover, the overall $PofB20$ comparison of 4 models is demonstrated in Fig. 6, which shows that GH-LSTMs outperforms SCE in terms of minimum $PofB20$.

From Table VIII, we can have the following observations: Firstly, on average, our GH-LSTMs model significantly outperforms the other three models in terms of IFA with the average value of 0.617, while the IFA values of other three models are 5.004, 1.667 and 4.546. Secondly, SCE and GH-LSTMs perform much better than the other two baselines, and SCE is actually a part of GH-LSTMs, i.e., the semantic-level LSTM without the gated fusion mechanism. This indicates that LSTM is a better learning algorithm to further extract the semantic information from word embeddings under effort-aware scenarios. This conclusion is also supported by the results shown in Fig. 7, where the overall distribution of IFA of all 4 methods is presented. Note that the lower the

TABLE VII
$PofB20$ SCORES OF DBN, SCE, DP-AM AND GH-LSTMS

	PofB20				$p(\delta)$		
Task	DBN	SCE	DP-AM	GH-LSTMs	GH-LSTMs vs. DBN	GH-LSTMs vs. SCE	GH-LSTMs vs. DP-AM
ant_1.5_1.6	**0.261**	0.249	0.259	0.260	0.519(-Negligible)	<0.05(+Medium)	0.978(+Small)
ant_1.6_1.7	0.253	0.279	0.243	**0.281**	<0.05(+Large)	0.172(+Small)	<0.05(+Large)
camel_1.2_1.4	**0.471**	0.469	0.362	0.469	0.135(-Small)	0.779(+Negligible)	<0.05(+Large)
camel_1.4_1.6	0.508	**0.559**	0.359	0.555	<0.05(+Large)	0.172(-Small)	<0.05(+Large)
ivy_1.4_2.0	0.161	0.190	**0.196**	**0.196**	<0.05(+Large)	<0.05(+Small)	0.580(+Negligible)
jedit_3.2_4.0	0.351	0.390	0.298	**0.395**	<0.05(+Large)	0.511(+Negligible)	<0.05(+Large)
jedit_4.0_4.1	0.327	0.362	0.316	**0.377**	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
log4j_1.0_1.1	0.265	0.299	0.275	**0.320**	<0.05(+Large)	<0.05(+Medium)	<0.05(+Large)
lucene_2.0_2.2	**0.601**	0.593	0.550	0.594	<0.05(-Large)	0.511(+Small)	<0.05(+Large)
lucene_2.2_2.4	**0.631**	0.619	0.583	0.618	<0.05(-Large)	0.196(-Small)	<0.05(+Large)
poi_1.5_2.5	0.567	**0.586**	0.524	0.578	<0.05(+Large)	<0.05(-Large)	<0.05(+Large)
poi_2.5_3.0	0.543	0.546	0.502	**0.548**	<0.05(+Large)	<0.05(+Small)	<0.05(+Large)
synapse_1.0_1.1	0.267	**0.289**	0.249	0.279	0.062(+Small)	0.172(-Small)	0.121(+Small)
synapse_1.1_1.2	0.298	**0.313**	0.263	0.311	<0.05(+Large)	0.779(-Negligible)	<0.05(+Large)
xalan_2.4_2.5	0.598	0.635	0.532	**0.636**	<0.05(+Large)	0.779(+Negligible)	<0.05(+Large)
xerces_1.2_1.3	0.657	0.665	0.637	**0.668**	<0.05(+Large)	0.172(+Small)	0.251(+Negligible)
Average & Win/Tie/Loss	0.422	0.440	0.384	**0.443**	11/3/2	5/10/1	12/4/0

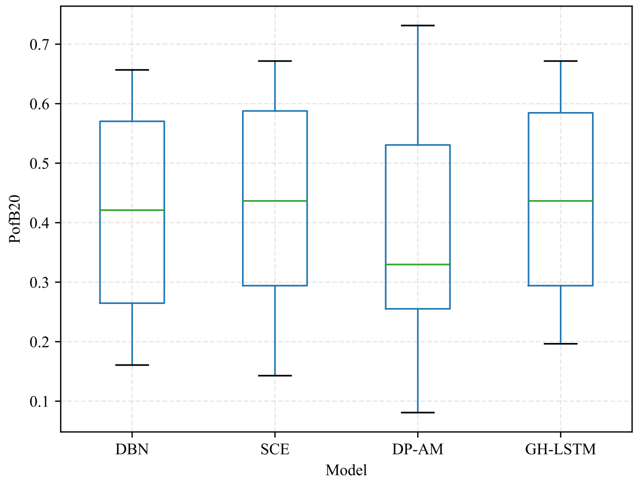


Fig. 6. Overall $PofB20$ comparison of DBN, SCE, DP-AM and GH-LSTMs

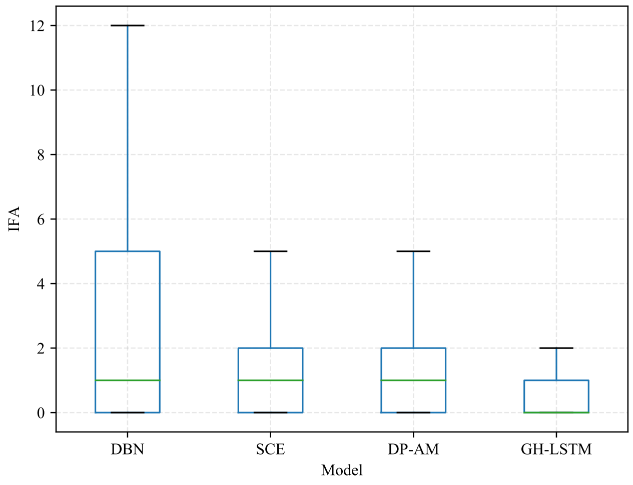


Fig. 7. Overall IFA comparison of DBN, SCE, DP-AM and GH-LSTMs

IFA value is, the better the model performs. Therefore, as indicated in Fig. 7, our GH-LSTMs significantly outperforms all baseline methods in terms of IFA.

Table IX shows P_{opt} scores of four models. We can see that GH-LSTMs significantly outperforms DBN and DP-AM according to the results of Win/Tie/Loss. Besides, although the SCE model is slightly better than the other three models in terms of average value, from the Win/Tie/Loss results, our GH-LSTMs model has 5 'Wins' and only 2 'Losses' in comparison to SCE, which means that GH-LSTMs significantly outperforms SCE on more tasks. Fig. 8 shows the distribution of P_{opt} scores of DBN, SCE, DP-AM and GH-LSTMs on all tasks, from which we can also see that GH-LSTMs achieves comparable performance with SCE.

In conclusion, GH-LSTMs significantly outperformed the other three baselines when evaluated by $PofB20$ and IFA, and achieved comparable performance with SCE when evaluated by P_{opt}. These results indicate that benefiting from hierarchical architecture and proper gated feature fusion mechanism, GH-LSTMs is able to give more accurate prediction results

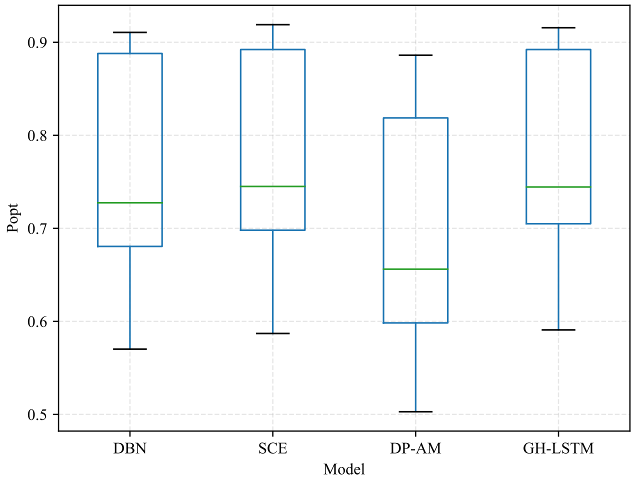


Fig. 8. Overall P_{opt} comparison of DBN, SCE, DP-AM and GH-LSTMs

TABLE VIII
IFA OF DBN, SCE, DP-AM AND GH-LSTMS

	IFA				$p(\delta)$		
Task	DBN	SCE	DP-AM	GH-LSTMs	GH-LSTMs vs. DBN	GH-LSTMs vs. SCE	GH-LSTMs vs. DP-AM
ant_1.5_1.6	2.100	3.133	4.900	**1.133**	<0.05(-Medium)	0.066(-Small)	0.706(+Small)
ant_1.6_1.7	5.433	2.567	16.100	**0.500**	<0.05(-Large)	<0.05(-Small)	0.092(-Negligible)
camel_1.2_1.4	18.967	1.067	3.900	**0.000**	<0.05(-Large)	<0.05(-Medium)	<0.05(-Medium)
camel_1.4_1.6	2.500	2.300	4.300	**1.767**	0.174(-Small)	0.315(-Small)	0.550(-Negligible)
ivy_1.4_2.0	27.567	2.767	13.100	**0.567**	<0.05(-Large)	<0.05(-Large)	<0.05(-Small)
jedit_3.2_4.0	1.567	0.733	5.533	**0.133**	<0.05(-Large)	<0.05(-Medium)	<0.05(-Large)
jedit_4.0_4.1	1.900	2.667	6.900	**0.400**	0.067(-Small)	<0.05(-Large)	<0.05(-Large)
log4j_1.0_1.1	6.367	0.767	1.467	**0.033**	<0.05(-Large)	<0.05(-Large)	<0.05(-Medium)
lucene_2.0_2.2	1.000	0.767	0.400	**0.333**	<0.05(-Large)	0.121(-Small)	0.914(+Negligible)
lucene_2.2_2.4	**0.100**	0.200	0.933	0.200	0.562(+Negligible)	0.785(-Negligible)	<0.05(-Medium)
poi_1.5_2.5	0.167	**0.100**	2.133	0.133	0.854(-Negligible)	0.785(+Negligible)	0.095(-Small)
poi_2.5_3.0	0.767	0.100	0.367	**0.000**	<0.05(-Small)	0.121(-Negligible)	<0.05(-Small)
synapse_1.0_1.1	2.633	2.433	1.567	**0.433**	<0.05(-Large)	<0.05(-Large)	<0.05(-Medium)
synapse_1.1_1.2	**0.733**	2.100	4.333	1.367	<0.05(+Large)	0.213(-Negligible)	0.111(-Negligible)
xalan_2.4_2.5	1.567	0.467	**0.433**	0.733	<0.05(-Medium)	0.735(+Negligible)	0.425(+Negligible)
xerces_1.2_1.3	6.700	4.500	6.367	**2.133**	<0.05(-Medium)	<0.05(-Medium)	<0.05(-Small)
Average & Win/Tie/Loss	5.004	1.667	4.546	**0.617**	11/4/1	8/8/0	9/7/0

TABLE IX
P_{opt} SCORES OF DBN, SCE, DP-AM AND GH-LSTMS

	P_{opt}				$p(\delta)$		
Task	DBN	SCE	DP-AM	GH-LSTMs	GH-LSTMs vs. DBN	GH-LSTMs vs. SCE	GH-LSTMs vs. DP-AM
ant_1.5_1.6	**0.654**	0.642	0.616	0.651	<0.05(-Large)	<0.05(+Large)	<0.05(+Large)
ant_1.6_1.7	0.610	**0.611**	0.574	**0.611**	<0.05(+Medium)	0.105(-Small)	<0.05(+Large)
camel_1.2_1.4	0.708	**0.724**	0.630	0.721	<0.05(+Large)	0.060(-Medium)	<0.05(+Large)
camel_1.4_1.6	0.755	**0.775**	0.593	0.774	<0.05(+Large)	0.926(-Negligible)	<0.05(+Large)
ivy_1.4_2.0	0.574	0.595	0.565	**0.596**	<0.05(+Large)	0.557(+Negligible)	<0.05(+Large)
jedit_3.2_4.0	0.734	**0.748**	0.653	0.747	<0.05(+Large)	0.495(-Small)	<0.05(+Large)
jedit_4.0_4.1	0.699	**0.722**	0.635	0.719	<0.05(+Large)	0.279(-Small)	<0.05(+Large)
log4j_1.0_1.1	0.619	**0.665**	0.592	0.663	<0.05(+Large)	0.591(-Negligible)	<0.05(+Large)
lucene_2.0_2.2	0.895	0.895	0.837	**0.896**	<0.05(+Large)	<0.05(+Medium)	<0.05(+Large)
lucene_2.2_2.4	**0.909**	0.903	0.863	0.906	<0.05(-Large)	<0.05(+Large)	<0.05(+Large)
poi_1.5_2.5	0.904	**0.916**	0.849	0.909	<0.05(+Large)	<0.05(-Large)	<0.05(+Large)
poi_2.5_3.0	0.902	**0.904**	0.849	**0.904**	<0.05(+Large)	0.105(+Small)	<0.05(+Large)
synapse_1.0_1.1	0.721	**0.733**	0.597	0.724	<0.05(+Large)	<0.05(-Large)	<0.05(+Medium)
synapse_1.1_1.2	0.717	0.741	0.651	**0.742**	<0.05(+Large)	0.926(-Negligible)	<0.05(+Large)
xalan_2.4_2.5	0.869	0.890	0.786	**0.891**	<0.05(+Large)	<0.05(+Small)	<0.05(+Large)
xerces_1.2_1.3	0.843	0.847	0.796	**0.848**	<0.05(+Large)	<0.05(+Medium)	<0.05(+Large)
Average & Win/Tie/Loss	0.757	**0.770**	0.693	0.769	14/0/2	5/9/2	16/0/0

under effort-aware scenario.

C. Answer to RQ3: How do external parameters affect the performance of GH-LSTMs?

In this part, we will make a discussion about two external parameters that may affect the performance of GH-LSTMs model: dimensions of semantic code embedding and the size of training set.

To figure out how the dimensions of semantic code embedding influences the prediction performance of GH-LSTMs, we retrain the GloVe models with 5 different dimension parameters of 10d, 20d, 30d, 40d(original), 50d and regenerate semantic features using corresponding GloVe model. On the other hand, the traditional features remain the same as the original experiments. The $F-measure$ values of these 16 tasks with 5 different dimensions are listed in Fig. 9.

As Fig. 9 shows, the dimensions of the semantic code embedding have no significant impact on classification performance: The average performance fluctuation of all 16 tasks is 0.024 in terms of $F-measure$ and the greatest performance fluctuation appears in task 'came_1.4_1.6', where increasing dimensions boost the $F-measure$ by 0.048. Considering that the datasets used to build semantic corpus in our experiments are much smaller than those in common NLP tasks, we can draw a conclusion that semantic code embedding with lower dimensions would not cause obvious performance variation for our GH-LSTM method on datasets with similar scale. Therefore, our advice is: if you use the datasets that have similar scale to our datesets, a fine-tuning of dimensions of semantic code embedding will be helpful to find the lowest usable dimension, which will leads to less consumption in training and testing process with equal levels of performance.

Since GH-LSTMs is a deep learning model, the size of the training set, i.e., the scale of the training version of the project, is possible to have effects on the learning process of the model. Since the defect distribution is usually project-specific, it is inappropriate to compare the performance of all the tasks directly. In order to estimate the performance

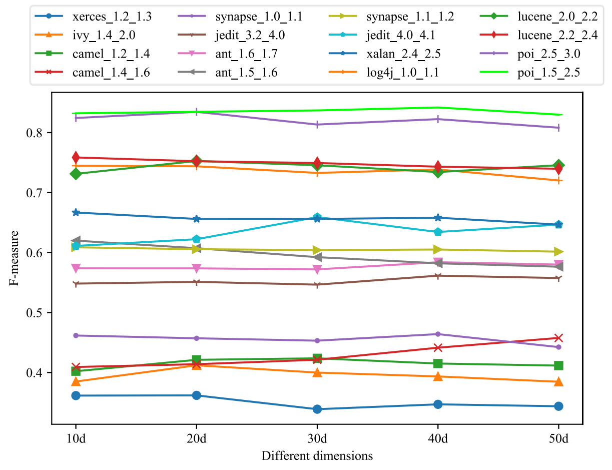


Fig. 9. F-measure of GH-LSTMs under different embedding dimensions

of GH-LSTMs under different sizes of training sets, we built new training sets that are obtained by under-sampling from the original training sets. We choose under-sampling rather than over-sampling because the new training sets generated by under-sampling only consist of real instances, which is more suitable under the setting of this discussion. Specifically, we select 3 sets of experiments that have the largest sizes of training sets and then perform an under-sampling operation on these training sets. We ensure the defect rate remains the same as the original training sets and set the new sizes of training sets to 25%, 50%, 75% and 100% of the original training sets. Fig. 10 shows the performance of GH-LSTMs with different sizes of training sets.

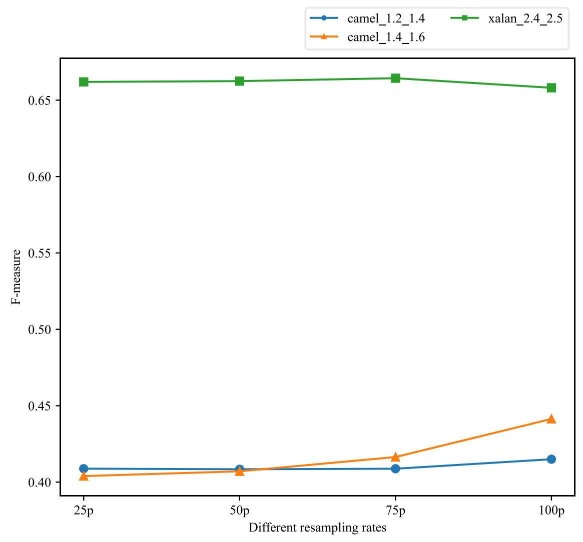


Fig. 10. F-measure of GH-LSTMs under different sizes of training sets

From Fig. 10, we can have the following observation: generally speaking, the GH-LSTMs models trained on under-sampled training sets achieve similar performance when compared with the original GH-LSTMs models. The performance fluctuations caused by changed sizes of training sets are less than 1 percentage points in most cases in terms of $F-measure$(with only one exception for task 'came_1.4_1.6' where the size of training set changing from '75% - 100%' results in 2.8 percentage points of performance variation). Since we adopt this under-sampling strategy, these results can be considered as the approximation of the real applications. Therefore, we can draw a conclusion that the performance of GH-LSTMs is not likely to be seriously influenced by the size of the training set when the defect distribution remains unchanged.

VI. Threats to Validity

A. Internal validity

Internal validity refers to the degree of confidence to which the causal relationship studied in a research is independent of other factors, i.e., is not affected by other variables. Therefore, the correctness of our experimental environment is significant for internal validity. In this study, we build the GH-LSTMs model by Keras and use Tensorflow as the backend of Keras. Additionally, we utilize GloVe to initialize the word vectors. These tools and technologies should be considered reliable since they have been widely used in related researches. Finally, to ensure that the code used to build the model in our experiments is error-free, we also conduct white-box testing on the source code.

B. External validity

External validity refers to the validity of a scientific research result that can be applied to other studies. The main threat to external validity in this study is that we only use 10 Java projects in PROMISE repository. They do not represent all projects (such as other Java projects and non-Java projects). Experiments on other datasets will help to further verify the validity of our method.

C. Construct validity

Construct validity refers to the degree to which the independent variables and dependent variables really represent the concepts that we are interested in measuring. The independent variables in our study can be divided into semantic features and traditional features. The semantic features are extracted from ASTs of source code files using word embedding technique, which is proved effective by previous studies [14], [8]. Additionally, we select 18 code metrics provided by the PROMISE repository as traditional features. These code metrics are calculated by the CKJM tool [30]. For dependent variables, we use bug label data that identified by the BugInfo tool [31] from the PROMISE repository, based on which a lot of studies has been conducted [6], [14], [8]. To sum up, the construct validity of the independent variables and dependent variables of this paper should be acceptable.

D. Conclusion validity

Conclusion validity refers to the extent to which the research conclusion is drawn in a reasonable way. In this study, to mitigate the threat to conclusion validity, we carry out a large number of experiments on enough projects for each model, thus the results obtained from the generated experimental data should be statistically reliable.

VII. Related works

A. Traditional software defect prediction

Software defect prediction is a research hotspot in the field of software engineering, around which a lot of work has been carried out [6], [8], [22]. Most defect prediction techniques are focused on proposing new discriminative features that can be generated from the history information of software. These features, including static features and process features, will be fed into classifiers to identify software defects. Static features such as LOC are based on the static information of software modules, i.e., the source code files, while process features are usually extracted by analyzing changes of software modules such as code delta feature [32] and code turn feature [33].

Among the existing software defect prediction studies, traditional machine learning methods are commonly used [34], [35]. Catalyst and Diri [36] selected Random Forest, C4.5, Naive Bayes and other machine learning algorithms to conduct experiments. The experimental results show that the performance of the Random Forest is the best on the large-scale datasets, while the performance of Naive Bayes is the best on the small-scale datasets. When the feature selection technique is used, the performance of the Random Forest model remains the best. Elish et al. [34] compared the performance of eight modeling techniques with Support Vector Machine(SVM) using NASA datasets. As the experimental results show, the SVM method generally achieves better performance than other comparable models. In cross-project software defect prediction, Ryu et al. [37] proposed a SVM boosting method, which applied imbalance processing technology in the cross-project defect prediction, and the experimental results prove that their method can achieve better performance.

B. Deep learning in software defect prediction

In recent years, with the popularity of artificial neural networks, researchers have begun to apply deep learning techniques to the field of software defect prediction. Yang et al. [38] used DBN to generate expressive features from a set of initial change features in just-in-time defect prediction. Compared with the method proposed by previous work [39], their method performs better in terms of 4 metrics on 6 open-source projects. Liu et al. [15] proposed to utilize the Historical Version Sequence of Metrics (HVSM), which can be extracted from continuous versions software, to build software prediction models based on RNN. Wang et al. [6] designed the semantic feature by using DBN to extract semantic information from token vectors generated from AST node sequences and achieved better performance in both with-in project and cross-project software defect prediction. Liang et al. [14] proposed another semantic feature represented in the form of word embedding and chose LSTM as the classifier to boost the performance of defect prediction. To make use of the combination of semantic features and traditional features, Fan et al. [8] proposed a framework called DP-AM, which leverages a bidirectional RNN with attention mechanism and combines traditional features with semantic features for better performance. Different from their method where traditional features are appended to semantic features by simple concatenation, our GH-LSTMs model leverages a hierarchical architecture to extract both kinds of features efficiently and adopts the gated feature fusion layer to properly combine the extracted features.

Hierarchical network architectures are widely used in typical deep learning areas such as natural language processing and computer vision. For example, in order to make full use of both intra-sentence and inter-sentence relations, Ruder et al. proposed a hierarchical model, which stacks a review-level Bi-LSTM on top of the sentence-level Bi-LSTM. In this paper, we introduce a gated hierarchical LSTMs network to perform defect prediction. To the best of our knowledge, this is the first time that hierarchical architecture is used to perform defect prediction tasks.

VIII. Conclusion and future work

With the increasing complexity and scale of software products, reliability assurance has become a significant challenge. In this paper, we propose a deep-learning-based method called GH-LSTMs, which predicting potential code defects in software modules. GH-LSTMs can further extract semantic features and traditional features simultaneously using the hierarchical LSTMs architecture, and then we utilize gated merge mechanism to automatically learn the optimal ratio of feature fusion to make the best use of both semantic features and traditional features. Finally, the combined features are fed to a fully-connected layer to perform the task of with-in project defect prediction. We conducted 16 sets of experiments on 10 open source projects. The results indicate that our proposed GH-LSTMs method significantly outperforms the state-of-the-art method in terms of $F-measure$ under non-effort-aware scenario. Moreover, under effort-aware scenario, when compared with the state-of-the-art methods, GH-LSTMs significantly outperforms them in terms of $PofB20$ and IFA, and achieves comparable performance in terms of P_{opt}.

As for future work, we would like to further explore the performance of GH-LSTMs in cross-project defect prediction tasks. Additionally, it is of great value to extend our method to other programming languages such as C and C++.

Acknowledgements

This work is partially supported by National Natural Science Foundation of China(61772263, 61772014), Collaborative Innovation Center of Novel Software Technology and Industrialization, Suzhou Technology Development Plan(key industry technology innovation-prospective application research project SYG201807), and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

References

[1] 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.

[2] T. Jiang, L. Tan, and S. Kim, “Personalized defect prediction,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 2013, pp. 279–289.

[3] J. Nam, “Survey on software defect prediction,” *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[4] H. H. Maurice, “Elements of software science (operating and programming systems series),” 1977.

[5] T. J. McCabe, “A complexity measure,” *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[6] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep semantic feature learning for software defect prediction,” *IEEE Transactions on Software Engineering*, 2018.

[7] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, “How far we have progressed in the journey? an examination of cross-project defect prediction,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 1, pp. 1–51, 2018.

[8] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Deep semantic feature learning with embedded static metrics for software defect prediction,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 244–251.

[9] 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.

[10] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[12] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

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

[14] 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.

[15] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, “Connecting software metrics across versions to predict defects,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 232–243.

[16] H. Wang, X. Zhang, B. Liang, Q. Zhou, and B. Xu, “Gated hierarchical lstms for target-based sentiment analysis,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 11n12, pp. 1719–1737, 2018.

[17] C. Ni, X. Chen, F. Wu, Y. Shen, and Q. Gu, “An empirical study on pareto based multi-objective feature selection for software defect prediction,” *Journal of Systems and Software*, vol. 152, pp. 215–238, 2019.

[18] Z. He, F. Peters, T. Menzies, and Y. Yang, “Learning from open-source projects: An empirical study on defect prediction,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 45–54.

[19] 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*, 2014, pp. 414–423.

[20] X. Chen, Y. Mu, Y. Qu, C. Ni, M. Liu, T. He, and S. Liu, “Do different cross-project defect prediction methods identify the same defective modules?” *Journal of Software: Evolution and Process*, vol. 32, no. 5, p. e2234, 2020.

[21] Y. Jiang, B. Cukic, and Y. Ma, “Techniques for evaluating fault prediction models,” *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[22] 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, 2006.

[23] Q. Huang, X. Xia, and D. Lo, “Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 159–170.

[24] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[25] Q. Huang, X. Xia, and D. Lo, “Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction,” *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[26] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, “Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 157–168.

[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[28] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 107–116.

[29] Y. Benjamini and Y. Hochberg, “Controlling the false discovery rate: a practical and powerful approach to multiple testing,” *Journal of the Royal statistical society: series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[30] 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*, 2010, pp. 1–10.

[31] M. Jureczko and D. Spinellis, “Using object-oriented design metrics to predict software defects,” *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pp. 69–81, 2010.

[32] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change bursts as defect predictors,” in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 309–318.

[33] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[34] K. O. Elish and M. O. Elish, “Predicting defect-prone software modules using support vector machines,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[35] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, 2012.

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

[37] D. Ryu, O. Choi, and J. Baik, “Value-cognitive boosting with a support vector machine for cross-project defect prediction,” *Empirical Software Engineering*, vol. 21, no. 1, pp. 43–71, 2016.

[38] 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.

[39] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, “The effects of over and under sampling on fault-prone module detection,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 196–204.

Hao Wang is now a Master degree candidate in the School of Computer Science and Technology, Soochow University, China. He received his B.E. degree at Soochow University.

His research interests include software defect prediction and natural language processing.

Weiyuan Zhuang is now a Master degree candidate in the School of Computer Science and Technology, Soochow University, China. He received his B.E. degree at Soochow University.

His research interests include software defect prediction and deep learning.

Xiaofang Zhang is an Associate Professor in the School of Computer Science and Technology, Soochow University, China.

Her research interests lie primarily in the intersection of Software Engineering and Artificial Intelligence, including intelligent software engineering, software testing, and software defect prediction.