Journal of Systems Engineering and Electronics
Vol. 34, No. 1, February 2023, pp.236 – 246

Bug localization based on syntactical and semantic information of source code

YAN Xuefeng[1,2,*], CHENG Shasha[1], and GUO Liqin[3]

1. College of Computer Science Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China; 2. Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 211106, China; 3. State Key Laboratory of Intelligent Manufacturing System Technology, Beijing Institute of Electronic System Engineering, Beijing 100854, China

Abstract: The existing software bug localization models treat the source file as natural language, which leads to the loss of syntactical and structure information of the source file. A bug localization model based on syntactical and semantic information of source code is proposed. Firstly, abstract syntax tree (AST) is divided based on node category to obtain statement sequence. The statement tree is encoded into vectors to capture lexical and syntactical knowledge at the statement level. Secondly, the source code is transformed into vector representation by the sequence naturalness of the statement. Therefore, the problem of gradient vanishing and explosion caused by a large AST size is obviated when using AST to the represent source code. Finally, the correlation between bug reports and source files are comprehensively analyzed from three aspects of syntax, semantics and text to locate the buggy code. Experiments show that compared with other standard models, the proposed model improves the performance of bug localization, and it has good advantages in mean reciprocal rank (MRR), mean average precision (MAP) and Top N Rank.

Keywords: bug report, abstract syntax tree, code representation, software bug localization.

DOI: 10.23919/JSEE.2023.000010

1. Introduction

Quality and performance are important to the software system [1]. The defect tracking systems such as Bugzilla are used to store bug reports that can be modified by relevant maintenance team. Bug localization is a technique for locating codes that may contain defects, thus alleviating the working pressure of the developer [2]. However, there are a variety of operating modes in the practical equipment [3], and the problem number is large and growing for complex software [4]. Thus, the software dependability and usability can be impacted by the availability and promptness of bug localization.

The focus of bug localization is to analyze the correlation among the exception described in bug report and the source code that realizes the relevant functionality [2]. Majority of the existing software bug localization techniques are based on popular machine learning or information retrieval (IR), which ignore the program structure and regard the code as natural language. The bug report and the source code are represented by using the bag-of-words approach, and the similarity is calculated in the uniform lexical feature vector space [2]. BugLocator [1] improves the vector space model (VSM) by using the file length and adjusts the corresponding source file weight by utilizing the fixed bug reports to detect the defect. Youm et al. [5] presented method of the bug location information analysis that analyzed the keywords and stack information of the bug report, the structure and the composition of the source code, version information, comments, and other text information to realize bug localization synthetically. Ye et al. [6] used reference documents, application programming interface (API) documents and tutorials as corpus to train the word2vec model to calculate the semantic similarity. For the past few years, deep learning is widely capitalized on bug localization [2,7,8]. Yan et al. [7] greatly improved the performance of bug localization by taking advantage of convolution neural network (CNN). Huo et al. [2] proposed a uniform modeling which used diverse CNNs to obtain information from source codes and bug reports. They further analyzed the sequence of source codes by combining it with long short-term memory (LSTM), so that the semantic information can be extracted from the source code, which is used to reflect the sequence naturalness of codes. And the long-distance call relationships between codes are analyzed [8].

However, programming languages usually follow strict syntax structure. It also has complicated information and control flow. And there may be call relationship between code elements. Therefore, regarding source codes as a

Manuscript received February 22, 2021.
*Corresponding author.
This work was supported by the National Key R&D Program of China (2018YFB1702700).

natural language will result in the miss of significant grammatical information. Recent studies have shown that the neural model based on the abstract syntax tree (AST) represents source codes more accurately, and programming languages can benefit from the syntax and structured representation. Mou et al. [9] proposed tree-based CNNs (TBCNNs), which is of great significance to represent the programming language. TBCNNs convert the source file into an AST and acquire the relationship between the code statements, which show good effect in programming language processing. Wei et al. [10] introduced clone detection with learning to hash (CDLH) which parsed the code fragment into AST, considered the lexical and syntactic information of the source file, and used tree-based LSTM (Tree-LSTM) [11] to obtain the code fragment representation for clone detection. Some studies have shown that tree-based source file representation methods have the problems of gradient vanishing and gradient explosion.

To enhance the efficiency of software bug localization, we analyze the syntactical and semantic feature of source codes, and solve the limitation of using AST to represent source codes mentioned above, a newfangled software bug localization model based syntactical and semantic information of source code is proposed. For our proposed model, an integrated AST of the source file is divided into statement trees based on the statement granularity. The statement tree is encoded into vectors by using the recursive autoencoder to capture the lexical and syntactic knowledge of the statement. And then, LSTM is utilized to express the naturalness of the statement to generate the embedding of the source code. At the same time, VSM, token matching and stack trace are employed to analyze the textual relevance among source file and bug report. Finally, the differential evolution (DE) algorithm is used to integrate the similarity, and then the source files are sorted by similarity. Experiments illustrate that our model has significant enhancement for bug localization.

The contributions of our research are mainly shown as below:

(i) The code representation based on syntactical and semantic information of source codes (CRSS) is proposed. And the lexical syntactic information at the sentence level and the naturalness of code are obtained.

(ii) A software bug localization model based on syntactical and semantic information of source codes is proposed. The relevance among source codes and bug report is analyzed from three aspects: syntax, semantics and text.

(iii) Experiments illustrate that compared with other standard models, our model improves the performance and accuracy of bug localization.

The remaining content of our research is arranged as follows: Section 2 introduces the research motivation. Section 3 presents the code representation based on syntactical and semantic information of source codes in detail. Next, a software bug localization model based syntactical and semantics of source codes is introduced in Section 4. In Section 5, our model and standard approaches are exhibited and contrasted by experiments. In the end, the conclusions and following research content are presented in Section 6.

2. Motivation

2.1 Difference between natural language and programming language

Complexity, diversity as well as flexibility are the characteristics of programming languages, but most of the codes written by developers are concise and short. Programming languages distinguish from natural languages chiefly in the following two ways [2].

Firstly, words or terms are the basic linguistic components of natural language, and the simple bag model is used to acquire the relationship between them. However, for programming languages, the statement is the smallest semantic unit. The functionalities of programming language are concluded from the semantics of multi-statements and the interaction of them along the execution path. Secondly, natural languages organize words in a "flat" way, which are always written or used in one dimension. However, programming languages organize statements in a "structured" way, programmers always write their source codes with proper indentations, indicating branches, loops, and even nested structures [12]. Because of the structural distinctions among natural language and programming language, the existing natural language processing (NLP) representation learning algorithm, such as skip-gram and continuous bag of words, are unsuitable for programming language directly.

At the same time, the complicated information interactions exist between code symbols, which can influence the functionality expressed by the source code [13]. The syntax and semantics of different levels in source files are often expressed in a tree structure. The compiled source code abides by strict syntax structure and is explicitly analyzed into syntax tree. Therefore, according to the above description, AST is utilized to analyze programming language and explore its syntactical and semantic information for representation.

2.2 Limitations of the existing code representation work

For TBCNNs [9], Tree-LSTM [11] and CDLH [10], tree-based source file representation has two major limitations. Firstly, if the gradient is calculated by back propagation in the training based on deep neural network, then there will be problems of gradient disappearance and deep structure explosion [12], specifically when the scale of source codes is large and the nested structure is complex. Secondly, for TBCNNs and CDLH, the number of sub-nodes is different in different AST nodes, which may lead to different number of parameters and parameter sharing problems. To solve these problems, AST is converted into binary tree, which will not only impact the original syntax structure, but also lead to a deeper AST structure.

In addition, Huo et al. [14] used the structure and sequence characteristics of control flow graph (CFG) to enhance the unified feature of bug localization, and further captured the structure and functional properties of source files. However, a majority of CFGs merely contain control flows in the codes and cannot represent complete data information. And the CFG is more difficult to obtain than the AST [15,16].

Therefore, for obtaining the syntactical and semantic information of source codes, recursive autoencoders (RAE) and AST are used to encode each node recursively in AST, and the vector representation of each node can be gained for each statement tree.

3. CRSS

3.1 Preprocessing method for source code

First, the source code is transformed into an integrated AST through the existing parsing library (for example, javalang and pycparser), as is shown in Fig. 1 and Fig. 2 . Then, the AST is split based on the statement granularity to form a group of statement sequences, as is shown in Fig. 3. In this paper, source files are converted to ASTs. Some content will be filtered out, such as comments, blank lines, and punctuation. Code elements are represented by nodes in the AST. For example, a complete source file is represented by the first node, and its child nodes may be method, variable declaration and so on.

```
public class BindingScope extends SimpleScope {
    public UnresolvedType lookupType(String name, IHasPosition location) {
        if (m_enclosingType != null) {
            String pkgName = m_enclosingType.getPackageName();
            if (pkgName != null && !pkgName.equals("")) {
                String[] currentImports = getImportedPrefixes();
                String[] newImports = new String[currentImports.length + 1];
                for (int i = 0; i < currentImports.length; i++) {
                    newImports[i] = currentImports[i];
                }
                newImports[currentImports.length] = pkgName.concat(".");
                setImportedPrefixes(newImports);
            }
        }
        return super.lookupType(name, location);
    }
}
```

Fig. 1 A piece of source code in AspectJ dataset

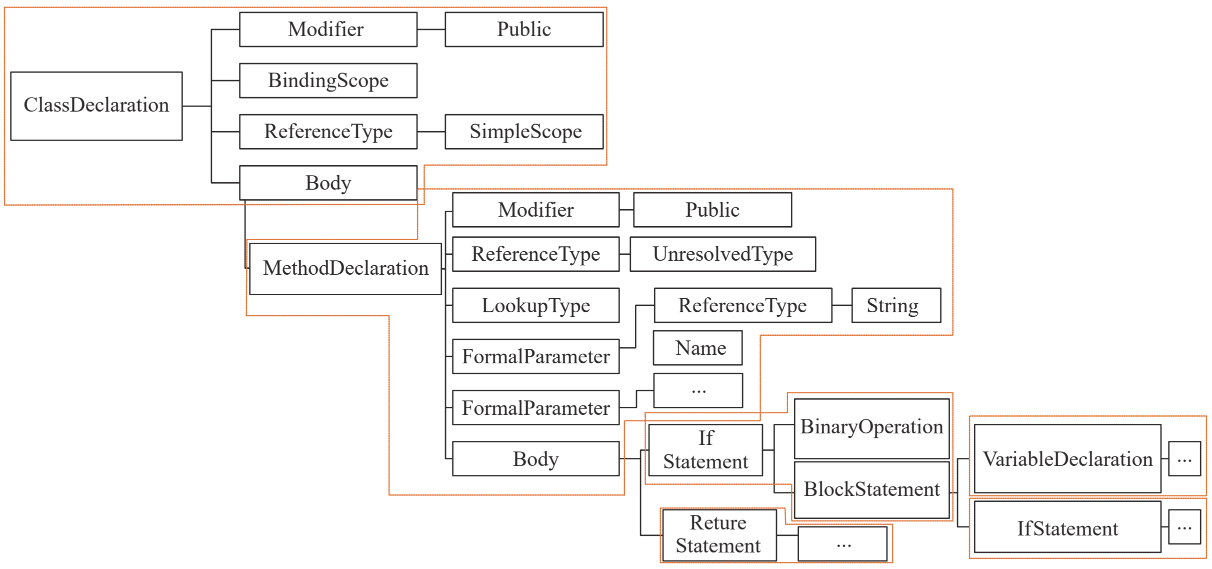


Fig. 2 Conversion of code in Fig. 1 to AST

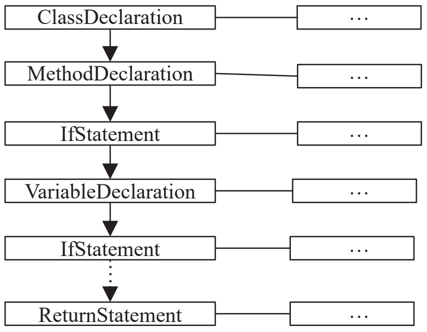


Fig. 3 Statement sequence of code in Fig. 1

For complete AST of source code T and the statement tree $\boldsymbol{ST}$, every statement node $\boldsymbol{ST}$ in T is relevant to a statement in codes. Class declaration, condition statement, switch statement and loop statement are added to the statement sequence as special statements. According to the node type of AST, the specific splitting rule is shown in Table 1.

Table 1 Splitting rule of the statement tree

Type	Definition
Class declaration	Type1={head, body}
Condition statement	Type2={control, block}
Try catch statement	Type3={block, catches}

For Type1, the "head" represents the statement of class declaration, and the "body" part is the content in brackets. As shown in Fig. 2, the head part of class declaration is {Modifier, BindingScope, ReferenceType}. For Type2, "control" is the judgment condition part, and "block" represents the statement executed after the judgment condition is met. In Fig. 2, the control of IfStatement is a binary operation statement, and the block part is VariableDeclaration and IfStatement. In the try-catch-finally, "block" is the statement in try, "catches" is the statement in catch.

For ReturnStatement, VariableDeclaration, BreakStatement and other statements, they are converted into the statement tree directly. Similar syntax structures can also be expressed as above. For example, the method declaration can be expressed as Type1, and the switch statement and loop statement can be expressed as Type2.

The AST of a source file is analyzed by breadth first traversal, and the AST is split according to the above method to attain the statement sequence, as shown in Fig. 3.

In the same position of the same batch, each parent may have a different number of children, which may cause problem in parameter-sharing [10]. Since the structure of the statement tree will not be too large or deep, the statement tree is converted into a binary tree. The conversion approach is akin to the method described in [10], which is adopted to refashion statement tree into complete binary tree (CBT). For details, see the following steps of converting statement tree into CBT and the ST2CBT is proposed as follows in Algorithm 1:

(i) More than two child nodes are separated to produce a new neo-sub-node. The previous left node and the new right child node are reorganized as the child nodes of the original parent node, and then all the child nodes except the original left most are the child nodes of the new child node.

(ii) Repeat this operation from top to bottom until the nodes degree either zero, one or two.

(iii) In order not to further deepen the statement tree, a node will be added to the node with only one child node, such as {New3, New4} in Fig. 4(b). Taking the ClassDeclaration node in Fig. 4(a) as an example, the statement tree is transformed into a binary tree. As shown in Fig. 4(b), four new nodes {New1, New2, New3, New4} are added to convert the statement ClassDeclaration into a CBT.

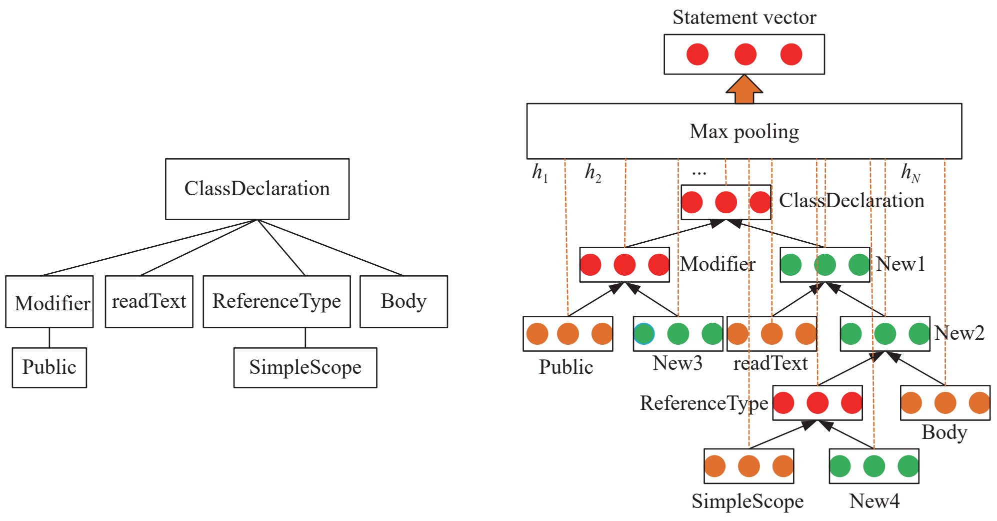


Fig. 4 An example of transforming a statement tree into a CBT

3.2 Encoder with lexical and syntactical information

Given a statement tree, we use the idea of RvNN [17] to encode recursively the leaf nodes in ***ST*** from bottom to top, and then every node in ***ST*** can be converted into vector.

Algorithm 1 ST2CBT

```
Input  The node array of statement tree: ST
Output  The node array of complete binary tree: CBT
    Initialize Node = $\{n_1,n_2,\cdots,n_N\}$ is the node of ST, $N$ is the number of nodes
For each node $n_u \in$ Node
    Initialize $C_i=\{c_1,c_2,\cdots,c_t\}$ is the child node of $n_i$
    If $t>2||t==1$
        Generate a node $c_{\rm new}$
        Let $C_i=\{c_1,c_{\rm new}\}$
        Insert $c_{\rm new}$ in the position $i+1$ of Node, Node = $\{n_1,\cdots,n_i,n_{\rm new},\cdots,n_N\}$
        If $k>2$
            $c_{\rm new}=\{c_2,c_3,\cdots,c_t\}$
        End if
    End if
End for
CBT=ST
Return CBT
```

Since leaf nodes (such as identifiers) of AST contain lexical information of source code, non-leaf nodes of AST contain syntax information similar to syntax structure such as whileStatement. Thus, the preorder traversal is used to get the symbols of each node to form the corpus. The skip-gram of word2vec is used to train the corpus, which is expressed as a real value vector, so as to capture the features of symbols and use them as the initial value of statement tree coding. The initial value of the vector is 0 for the nodes added after being transformed into a CBT.

Given a statement tree ***ST***, n represents a non-leaf node. The vector matrix $\boldsymbol{L}\in\mathbf{R}^{d\times|V|}$, d is the vector dimension after using word2vec training, and $|V|$ is the code elements number in the training set. According to the position p_k of node n, the word n vector representation of $\boldsymbol{x}_n=\boldsymbol{L}p_k\in\mathbf{R}^d$ can be acquired. Then the vector representation of n can be calculated by the following formula:

$$\boldsymbol{V}_n=\sigma(\boldsymbol{W}_n^{\rm T}\boldsymbol{x}_n+\boldsymbol{V}_{n1}+\boldsymbol{V}_{n2}+\boldsymbol{b}_n) \tag{1}$$

where $\boldsymbol{W}_n\in\boldsymbol{V}^{d\times k}$ represents the weight matrix, k represents the vector dimension of code and $\boldsymbol{b}_n$ is the offset term, $\boldsymbol{V}_{n1}$ and $\boldsymbol{V}_{n2}$ are the vector representation of two sub nodes of node n, and σ is the activation function.

The encoder traverses the statement tree which is transformed into a CBT, and then constantly calculates the code element information of the node and the vector representation of its child nodes as the new input to obtain the vector representation of the current node. Through the traversal of the hierarchical structure of the statement tree, the vector representations of the nodes in ***ST*** are obtained. Ultimately, the important features of each node are extracted by max pooling. Then the peek value of each dimension (k dimensions in total) is obtained in all nodes and then ***ST*** is transformed into vector representation.

3.3 Code representation based on sequence naturalness

Hindle et al. [18] have proved that most codes written by developers are concise and short, which can extract the corresponding features to be learned by the language model.

The source code is not only natural, but also better than natural language. Hu et al. [15] proposed that the bug characteristic in the source code can be depicted based on the perspectives of lexical, semantic, and syntax accurately. Therefore, based on the sequence naturalness and feature of the source code, we input each coding statement into LSTM to procure the whole source code for code embedding.

Based on the above-mentioned statement tree vector sequence, LSTM is used to effectively obtain the dependency and sequence naturalness of source codes. LSTM is a recurrent neural network (RNN). At time t, $\boldsymbol{V}_t\in\mathbf{R}^k$ is taken as input, $\boldsymbol{V}_t$ is the t statement in the statement sequence.

$$\begin{cases}\boldsymbol{i}_t=\sigma(\boldsymbol{W}_{xi}\boldsymbol{V}_t+\boldsymbol{W}_{hi}\boldsymbol{h}_{t-1}+\boldsymbol{b}_t)\\ \boldsymbol{f}_t=\sigma(\boldsymbol{W}_{xf}\boldsymbol{V}_t+\boldsymbol{W}_{hf}\boldsymbol{h}_{t-1}+\boldsymbol{b}_f)\\ \boldsymbol{o}_t=\sigma(\boldsymbol{W}_{xo}\boldsymbol{V}_t+\boldsymbol{W}_{ho}\boldsymbol{h}_{t-1}+\boldsymbol{b}_o)\\ \boldsymbol{c}_t=\boldsymbol{f}_t\circ\boldsymbol{c}_{t-1}+\boldsymbol{i}_t\circ\tanh(\boldsymbol{W}_{xc}\boldsymbol{V}_t+\boldsymbol{W}_{hc}\boldsymbol{h}_{t-1}+\boldsymbol{b}_c)\\ \boldsymbol{h}_t=\boldsymbol{o}_t\circ\tanh\boldsymbol{c}_t\end{cases} \tag{2}$$

where $\boldsymbol{h}_t$ is the output vector of time t, σ indicates the sigmoid function, and $\circ$ represents the point multiplication by element. Mean pooling is associated with the LSTM to extract the output $\boldsymbol{h}_t$ of each time step. LSTM remembers long-term states by adding a memory unit c, which makes the network learn to update the hidden state of a given data at the right time. Consequently, the sequence characteristics of source codes can be used to enrich high-level semantic features [8].

4. Bug localization based CRSS

4.1 Framework of the proposed bug localization model

In this paper, software bug localization is regarded as a learning task [2,8,14]. For source files $S_i \in \{s_1, s_2, \cdots, s_{N_1}\}$ and bug reports $b_j \in \{b_1, b_2, \cdots, b_{N_2}\}$, N_1 and N_2 represent the quantity of source files and bug reports, respectively. The constant label y_{ij} indicates the correlation of the sample pair (s_i, b_j). For some sample pairs with known correlation, a training set $D = \{(s_i, b_j, y_{ij})\}$ is constructed. The goal of this paper is to train a deep learning model and study a method f which is used to map the source code to the eigenvector v. For any sample pairs (s_i, b_j), their similarity $\text{score}_{ij} = f(s_i, b_j)$ is close to the corresponding label y_{ij}.

The integrated framework of our model is demonstrated in Fig. 5. Firstly, three kinds of text features of bug report and source code are analyzed, including VSM, token matching and stack trace. Then, according to the CRSS proposed in Section 3, the source file is transformed into vector. The syntactical and semantic feature of the source code is analyzed. And the semantic and syntactical similarity among source file and bug report will be computed. Finally, the similarity integration will be performed by using the DE method and the LSTM is trained.

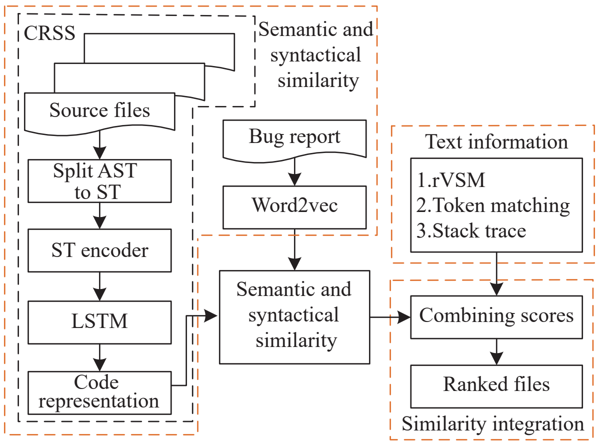


Fig. 5 Framework of bug localization model

4.2 Similarity integration oriented by syntax, semantics and text

In our research, the source file is converted into an AST, we can excerpt some code elements, such as class name, grammatical structure, variable, comment and so on. Some important information can be extracted from the bug report, such as summary, detailed description, stack trace and pervious fixed file. The compositional words are divided by the method of Camel Case splitting. For instance, "packageName" can be divided to "package" and "Name". The stop words, such as "public", "protected", "are" and "of" are evacuated in bug report and source file. The derivation of each word is brought back to the root formalization by employing the typical Porter Stemmer.

According to the CRSS introduced in Section 3, the source file is transformed into vector: $\boldsymbol{V}_S = \{\boldsymbol{V}_{s_1}, \boldsymbol{V}_{s_2}, \cdots, \boldsymbol{V}_{s_{N_1}}\}$. After preprocessing, the bug report is trained by skip-gram and represented by vector: $\boldsymbol{V}_R = \{\boldsymbol{V}_{r_1}, \boldsymbol{V}_{r_2}, \cdots, \boldsymbol{V}_{r_{N_2}}\}$. The cosine distance $\cos(\boldsymbol{r}_i, \boldsymbol{s}_j) = \dfrac{\boldsymbol{r}_i \cdot \boldsymbol{s}_j}{|\boldsymbol{r}_i| \times |\boldsymbol{s}_j|}(\boldsymbol{r}_i \in \boldsymbol{V}_R, \boldsymbol{s}_j \in \boldsymbol{V}_S)$ denotes the syntactical and semantic similarity between them.

Based on the existing IR-based bug localization models, the surface similarity is evaluated by adopting VSM, token matching and stack trace in this paper. The lexical correlation is calculated based on the rVSM method which is introduced in [1]. The code elements of source files (variable, method name, comments) are employed to severally map the content of the bug reports. The amount of exactly word matches is used to express the token matching similarity. A study [19] about the stack trace displays that only ten percent of defected files are not existed in the topped stack list. And it can be extracted from the bug reports by utilizing the regular equation (.*?)\((.*?)\). The stack trace similarity is measured by the inverse of the source files ranking in the stack list [20].

According to the above description, the four kinds of similarity (syntactical and semantic similarity, VSM, token matching, and stack trace) can be obtained. They are linearly built-up to procure the eventual similarity that is the fundament to sort the source file. The objective of linear combination is to maximize the objective function: $\text{obj} = \text{MAP} + \text{MRR}$ where MAP and MRR represent mean average precision and mean reciprocal rank respectively [21]. The objective function can be maximized and the value of parameters can be assessed by the DE algorithm.

Therefore, according to the DE algorithm, the linear combination weight of four groups of correlation is obtained and the correlation y° is measured by the following equation:

$$y^\circ = \sum_{i=1}^{4} w_i \text{Sim}_i \tag{3}$$

where Sim_i and w_i represent the weight corresponding to the ith similarity respectively. Use $y^* = \text{sig mod}\,(y^\circ) \in [0,1]$ to make the similarity range between 0 and 1. Accordingly, the correlation y^* between source file and bug report is obtained. The loss function is defined as the binary cross entropy, and the goal is to minimize the loss. The formula is shown as follows:

$$J(\Theta, y^*, y) = \sum(-(y \cdot \log_2(y^*) + (1-y) \cdot \log_2(1-y^*))). \tag{4}$$

The loss can be minimized when the bug localization model is trained. Because AdaMax is computationally efficient, AdaMax is used as the optimizer of model training in this paper.

The model will be saved when the whole parameters reach the best state. For a new bug report, source files are preprocessed with the statement tree sequence, and the surface text similarity is calculated by using VSM, and the token matching and stack trace between them are calculated as well. And then the above data are input into the trained model for prediction. The output is the probability that the source file may contain defects, and a sorting list of source files can be obtained.

5. Experiment

5.1 Experimental setting

The proposed model is evaluated by empirically assessing its performance based on the four datasets in comparison to the existing advanced bug localization techniques.

Benchmark dataset: The dataset is shown in Table 2, which has been adopted to evaluate different bug localization models (such as learning rank and word embeddings LR+WR [6] and DeepLocator [7]).

Table 2 Statistics of the benchmark dataset

Project	Time	#Bug report	#Source file
AspectJ	03/02-01/14	593	4 439
Tomcat	07/02-01/14	1 056	1 552
SWT	02/02-01/14	4 151	2 056
JDT	10/01-01/14	6 274	8 184

Actually, it is difficult to train whole sample pairs due to the largely source files quantity. For this reason, some sample pairs are chosen for experiments, which are ranked among the top three hundred in the similarity score list based on the information retrieval [22]. The bug reports will be arranged in chronological order by the data they are committed. The bug reports are spited into three parts, with training set accounting for 60%, validation set for 20%, and test set for 20%.

In this paper, javalang is used to convert the source file into AST. Using the skip-gram algorithm in word2vec, the term in bug report (embedding size is set to 300), and terminal node and non-terminal in AST are embedded (embedding sizes are set to 200). The hidden layer dimensions of the statement encoder and LSTM are set to 300. AdaMax is used for training with the optimizer and the learn ratio is 0.001.

5.2 Evaluation metrics

Three metrics are employed to evaluate the achievement of our introduced bug localization model.

(i) Top N Rank

The Top N Rank indicates the bug report quantity in which the actual error source file appears at the first N files in the acquired ranking list.

(ii) MAP

Because of many relevant documents in a single query, its mean accuracy can be calculated as follows:

$$\text{Avg}P_i = \sum_{i=1}^{M} \frac{P(i)\times \text{pos}(i)}{\text{number of positive instances}} \tag{5}$$

where i indicates the ranking, M represents the searching elements quantity, pos(i) represents whether i is related to the query. $P(i)$ is calculated as below:

$$P(i) = \frac{\text{number of positive instances in top } i \text{ position}}{i}. \tag{6}$$

The MAP is the average value for the mean accuracy of each query.

(iii) MRR

MRR is adopted to measure the overall accuracy of bug localization for all bug reports in the dataset. The query’s reciprocal rank represents the reciprocal of the first correctly located source file, which is calculated as

$$\text{MRR} = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{\text{rank}_i} \tag{7}$$

where M represents the amount of bug reports, rank$_i$ is the rank of the buggy source file containing the error in the ranked similarity list returned by the ith bug reports.

The above three metrics are proportional to the accuracy of bug localization.

5.3 Results and analysis

To generally assess the effectiveness of our introduced code representation method CRSS and bug localization model based syntactical and semantic information of source code, the following issues should be focused on research and analysis.

Question 1: How efficient is our introduced model, and whether it is better in comparison to five existing bug localization models?

Question 2: Does the code representation method CRSS can really enhance the effectiveness of bug localization?

Our introduced model in this research will be exerted to four dataset (Table 2) to analyze the above two research questions. The performance of our proposed model is estimated by three metrics.

(i) Questions 1 (Comparative analysis with five existing bug localization models)

In order to substantiate the effectiveness of our model, five software bug localizations are compared with BugLocator [1], bug location based on deep neural networks (DNNLOC) [23], DeepLocator [7], a novel CNN to process peogramming lanaguage (NP-CNN) [2] and customized abstract syntax trees (CAST) [24]. BugLocator is a bug localization method based on information retrieval, and the last four are based on deep learning. In general, our bug localization model has enhanced the effectiveness on three metrics, as shown in Table 3.

Compared with BugLocator, Table 3 demonstrates that our model performance has increased by more than 50% on three metrics. This is because our model not only uses the CRSS to analyze the syntactic and semantic information of source files, but also considers the textual similarity of stack trace and token matching. Therefore, on the basis of IR, further analysis of the programming structure, text attributes can significantly increase the accuracy of bug localization.

Table 3 Experimental results for our model, BugLocator, DNNLOC, DeepLocator, NP-CNN and CAST

Project	Method	Top@1	Top@5	Top@10	MAP	MRR
AspectJ	BugLocator	0.216	0.473	0.571	0.276	0.369
	DNNLOC	0.478	0.712	0.804	0.320	0.520
	DeepLocator	0.400	0.660	0.780	0.340	0.490
	NP-CNN	0.460	0.730	0.810	0.401	0.531
	CAST	0.500	0.766	0.803	0.418	0.536
	Our model	**0.573**	**0.808**	**0.884**	**0.503**	**0.578**
SWT	BugLocator	0.246	0.408	0.533	0.284	0.325
	DNNLOC	0.352	0.690	0.803	0.370	0.450
	DeepLocator	0.360	0.600	0.750	0.390	0.480
	NP-CNN	0.365	0.700	0.812	0.381	0.475
	CAST	0.372	0.701	0.825	0.425	0.503
	Our model	**0.464**	**0.771**	**0.898**	**0.468**	**0.549**
JDT	BugLocator	0.183	0.427	0.508	0.307	0.389
	DNNLOC	0.403	0.650	0.743	0.340	0.450
	DeepLocator	0.400	0.640	0.730	0.390	0.470
	NP-CNN	0.420	0.669	0.749	0.383	0.461
	CAST	0.432	0.681	0.757	0.388	0.467
	Our model	**0.515**	**0.726**	**0.826**	**0.434**	**0.516**
Tomcat	BugLocator	0.354	0.645	0.709	0.431	0.485
	DNNLOC	0.539	0.729	0.804	0.520	0.600
	DeepLocator	0.520	0.720	0.800	0.540	0.600
	NP-CNN	0.530	0.700	0.792	0.529	0.597
	CAST	0.507	0.766	0.825	0.556	0.612
	Our model	**0.625**	**0.814**	**0.877**	**0.601**	**0.658**

Compared with NP-CNN, Top@1, Top@10, MAP and MRR are enhanced by 23.1%, 10.2%, 18.5% and 10.7% respectively. NP-CNN adopts vocabulary and program structure features to study uniform characteristics from bug report and source code. Although CNN is used to extract the features according to the program structure of the source file, the syntax characteristic of the programming language is unconsidered, and the nesting and dependency relationship between statements are not well analyzed. At the same time, the text attributes of source files and bug reports are not analyzed. Our model can locate defects in syntax, semantics and text, and analyze the correlation among source file and bug report more comprehensively, so as to increase the effectiveness of our introduced model.

Compared to CAST, MAP and MRR enhanced by 8.1%−20.3% and 7.8%−10.5% respectively, Top@1, Top@5 and Top@10 increase by 20.4%, 7.7% and 10.0%

respectively. The CAST enhances the TBCNNs model by customizing AST, but it does not consider that when the source file has complex nested structure, the AST structure will become very large and deep, which may lead to the problems of gradient disappearing and difficulty to capture the dependency between statements. In this paper, CRSS is proposed. The AST is split based on statement granularity, which can avoid the above problems and obtain the syntactical and semantic information of source codes, and make the bug localization more accurate. At the same time, compared with CAST, our model takes more into account the text attributes.

Therefore, through above experiments, the overall performance of the proposed software bug localization model is improved. And CRSS, considering the syntactical and semantic information of the source code, can further enhance the comprehensive software bug localization effectiveness.

(ii) Question 2 (Analysis of CRSS method in bug localization)

To validate the influence of the introduced code representation approach CRSS, our model is contrasted with two methods. The first one is to use VSM, token matching and stack trace to analyze the relevance among the source file and the bug report to analyze defects (NoneCRSS). The other is to change the CRSS into Word2Vec, and use skip-gram to convert the source file into vector representation.

As shown in Fig. 6–Fig. 11, the proposed model is better than the other two methods in all aspects of evaluation metrics. Compared with NoneCRSS, the proposed model is increased by 18.8%, 14.2% and 13.1% on Top@1, Top@5 and Top@10. The other two metrics are improved by 9.1%–39.3% and 9.1%–41.3%, respectively. Further analysis of NoneCRSS and Word2Vec shows that Word2Vec in the Top@N, MAP and MRR is better than NoneCRSS. Therefore, this result shows that the accuracy of software bug localization can be improved by further considering the semantic correlation between source file and bug report text.

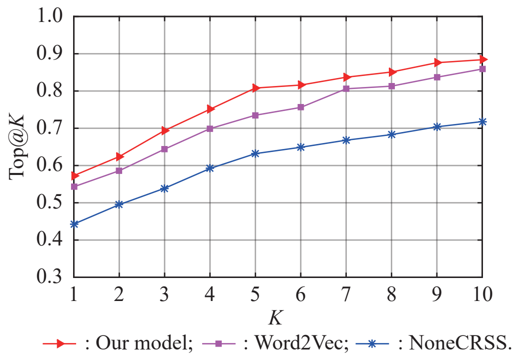


Fig. 6 Top@*K* comparison on AspectJ

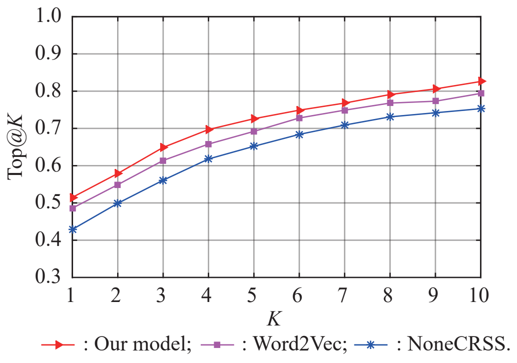


Fig. 7 Top@*K* comparison on JDT

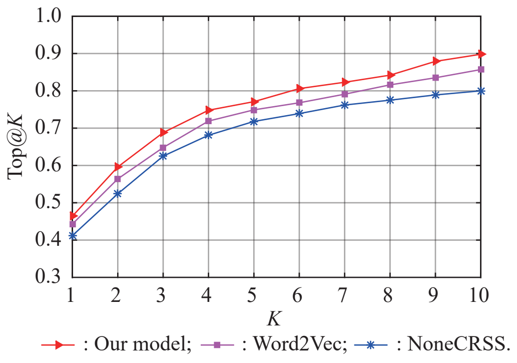


Fig. 8 Top@*K* comparison on SWT

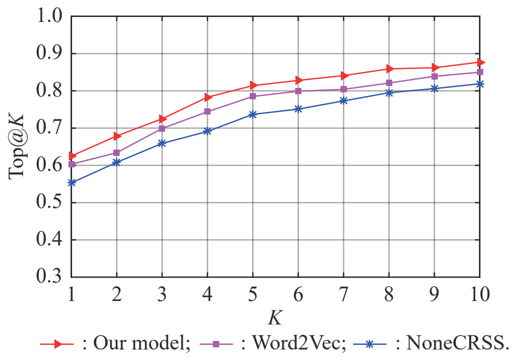


Fig. 9 Top@*K* comparison on Tomcat

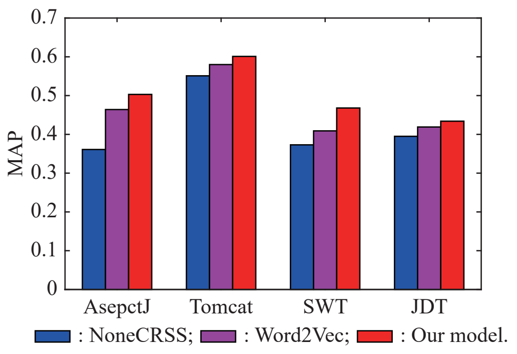


Fig. 10 MAP comparison on four datasets

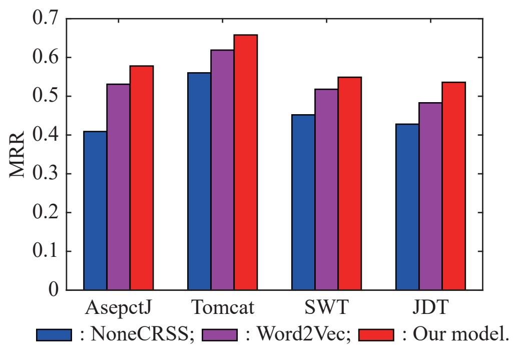


Fig. 11 MRR comparison on four datasets

Compared with Word2Vec, the MAP and MRR of our model are improved by 7.5% and 7.0% respectively, Top@1 and Top@5 are enhanced by 3.6%−5.9% and 2.9%−9.9% respectively. The results show that the CRSS is better than that of skip-gram in the code representation. Because the former transforms the source code into AST, it considers not only the program structure, but also considers the semantic and syntactic information of the source code, and the CRSS more comprehensively represents the source code, which plays a good role in software bug localization.

Therefore, it is of considerable implication to research the grammatical information of source codes by using AST. Generally, CRSS can increase the bug localization accuracy.

6. Conclusions

The developers and testers of software projects need to modify relevant buggy codes based on the content of bug report. However, for complex and large project codes, the locating process can be time-consuming. Therefore, it is of great significance to reduce the time and cost of software developer to utilize the bug localization model to locate the defected files automatically.

In this paper, a software bug localization model based on source file syntactical and semantic information is proposed. Firstly, source file representation method CRSS is proposed, which captures lexical, syntactic knowledge at sentence level, and the naturalness of code. Then we analyze the correlation between source file and bug report from three aspects of syntax, semantics and text. Finally, our experiments show that the capability of the proposed model is superior to the existing methods based on IR and deep learning. Experiments also show that CRSS not only considers the program structure, but also the semantic and syntactical information of the source code. Therefore, it is reasonable that the performance of bug localization can be enhanced by CRSS substantially.

The invocation relationship between statements and the long distance dependency relationship are not reflected in the AST, so in the future, the data flow diagram and control flow diagram of the source file can be added to represent the characteristic of the source code more completely, and the feature of the source file can be accurately represented. And the generalizability of the proposed model will be verified in other software projects.

References

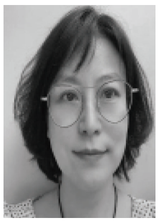
[1] ZHOU J, ZHANG H Y, LO D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. Proc. of the IEEE 34th International Conference on Software Engineering, 2012: 14–24.
[2] HUO X, LI M, ZHOU Z H. Learning unified features from natural and programming languages for locating buggy source code. Proc. of the 25th International Joint Conference on Artificial Intelligence, 2016: 1606–1612.
[3] LIU Y H. Optimal selection of tests for fault detection and isolation in multi-operating mode system. Journal of Systems Engineering and Electronics, 2019, 30(2): 425–434.
[4] KHATIWADA S, TUSHEV M, MAHMOUD A. Just enough semantics: an information theoretic approach for IR-based software bug localization. Journal of Information and Software Technology, 2018, 93: 45–57.
[5] YOUM K, AHN J, LEE E. Improved bug localization based on code change histories and bug reports. Journal of Information and Software Technology, 2017, 82: 177–192.
[6] YE X, SHEN H, MA X, et al. From word embeddings to document similarities for improved information retrieval in software engineering. Proc. of the 38th International Conference on Software Engineering, 2016: 404–415.
[7] YAN X, JACKY K, KWABENA E B, et al. Improving bug localization with word embedding and enhanced convolutional neural networks. Information and Software Technology, 2019, 105: 17–29.
[8] HUO X, LI M. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. Proc. of the 26th International Joint Conference on Artificial Intelligence, 2017: 1909–1915.
[9] MOU L L, LI G, ZHANG L, et al. Convolutional neural networks over tree structures for programming language processing. Proc. of the 30th AAAI Conference on Artificial Intelligence, 2016: 1287–1293.
[10] WEI H H, LI M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. Proc. of the 26th International Joint Conference on Artificial Intelligence, 2017: 3034–3040.
[11] TAI K S, SOCHER R, MANNING C D. Improved semantic representations from tree-structured long short-term memory networks. Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, 2015, 1: 1556−1566.
[12] MOU L L, LI G, LIU Y, et al. Building program vector representations for deep learning. Proc. of the International Conference on Knowledge Science, Engineering and Management, 2014: 547–553.
[13] JAYASUNDARA V, BUI N D Q, JIANG L, et al. TreeCaps: tree-structured capsule networks for program source code processing. https://doi.org/10.48550/arXiv.1910.12306.
[14] HUO X, LI M, ZHOU Z H. Control flow graph embedding based on multi-instance decomposition for bug localization.

Proc. of the AAAI Conference on Artificial Intelligence, 2020, 34(4): 4223–4230.
[15] HU J C, CHEN J F, ZHANG L, et al. A memory-related vulnerability detection approach based on vulnerability features. Journal of Tsinghua Science and Technology, 2020, 25(5): 604–613.
[16] WANG W H, LI G, MA B, et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. Proc. of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, 2020: 261–271.
[17] SOCHER R, LIN C C, MANNING C D, et al. Parsing natural scenes and natural language with recursive neural networks. Proc. of the 28th International Conference on Machine Learning, 2011: 129–136.
[18] HINDLE A, BARR E T, SU Z, et al. On the naturalness of software. Proc. of the India Software Engineering Conference, 2012: 837–847.
[19] SCHRTER A, SCHROTER A, BETTENBURG N, et al. Do stack traces help developers fix bugs? Proc. of the 7th IEEE Working Conference on Mining Software Repositories, 2010: 118–121.
[20] WONG C P, XIONG Y, ZHANG H, et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. Proc. of the IEEE International Conference on Software Maintenance and Evolution, 2014: 181–190.
[21] GHARIBI R, RASEKH A H, SADREDDINI M H. Leveraging textual properties of bug reports to localize relevant source files. Information Processing & Management, 2018, 54(6): 1058–1076.
[22] YE X, BUNESCU R, LIU C. Learning to rank relevant files for bug reports using domain knowledge. Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014: 689–699.
[23] LAM A N, NGUYEN A T, NGUYEN H A. Bug localization with combination of deep learning and information retrieval. Proc. of the IEEE/ACM 25th International Conference on Program Comprehension, 2017: 218–229.
[24] LIANG H L, SUN L, WANG M L, et al. Deep learning with customized abstract syntax tree for bug localization. IEEE Access, 2019, 7: 116309–116320.

Biographies

YAN Xuefeng was born in 1975. He received his Ph.D. degree from Beijing Institute of Technology, China, in 2005. He is currently a professor with the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. His research interests include intelligent modeling, big data analysis, model based system engineering, complex system modeling and simulation theory and method.
E-mail: yxf@nuaa.edu.cn

CHENG Shasha was born in 1996. She received her B.S. degree in computer science technology from Anhui University of Technology in 2018. Now she is pursuing her M.S. degree at Nanjing University of Aeronautics and Astronautics, China. Her research interests are bug localization, system engineering, and big data analysis.
E-mail: shasha@nuaa.edu.cn



GUO Liqin was born in 1976. She is currently a senior engineer at State Key Laboratory of Intelligent Manufacturing System Technology. Her research interests include intelligent manufacture, complex system modeling and simulation theory and method.
E-mail: guo_96@163.com