IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 3, 2019, accepted August 20, 2019, date of publication August 22, 2019, date of current version September 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936948

Deep Learning With Customized Abstract Syntax Tree for Bug Localization

HONGLIANG LIANG[1], (Member, IEEE), LU SUN[1], MEILIN WANG[2], AND YUXING YANG[1]
[1]School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China
[2]China Information Technology Security Evaluation Center, Beijing 100085, China

Corresponding author: Hongliang Liang (hliang@bupt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant U1713212.

ABSTRACT Given a bug report, bug localization technique can help developers automatically locate potential buggy files. Information retrieval and deep learning approaches have been applied in bug localization by extracting lexical features in bug reports and syntactic features in source code files, though they fail to utilize the structural and semantic information of source code files. In this paper, we present a bug localization system CAST, which exploits deep learning and customized abstract syntax trees of programs to locate potential buggy source files automatically and effectively. Specifically, CAST extracts both lexical semantics in bug reports (e.g., words) and source files (e.g., method names) and program semantics in source files (e.g., abstract syntax tree, AST). Moreover, CAST enhances the tree-based convolutional neural network (TBCNN) model with customized ASTs, which distinguish between user-defined methods and system-provided ones to reflect their contributions leading to defects. Furthermore, customized ASTs group the syntactic entities with similar semantics and prune the ones with little or redundant semantics in order to facilitate the learning performance. Experimental results on four widely-used software projects show that CAST significantly outperforms the state-of-the-art methods in locating the buggy source files.

INDEX TERMS Abstract syntax tree, bug localization, convolutional neural network, deep learning, method invocation.

I. INTRODUCTION

For large and evolving software, developers may receive a large number of bug reports, and it is difficult and costly to manually locate the potential buggy source files based on bug reports. Bug localization aims to alleviate the burden of developers by automatically locating potentially buggy files in software repositories for a given bug report. To this end, the key for bug localization is to explore the connection between a bug report written in natural languages and the corresponding source code files written in programming languages.

There are two main approaches to locate bugs automatically. One approach is information retrieval (IR). For instance, Gay *et al.* [25] represent both source files and bug reports using vector space model (VSM) based on the similarities between a bug report and the buggy source files. Zhou *et al.* [28] propose a revised vector space model (rVSM) called BugLocator, which considers the historical reports of similar bugs and their corresponding buggy files. Though being convenient to correlate the heterogeneous data in the same lexical feature space, these approaches usually require much computational capability and mostly depend on the quality of features extracted from bug reports and programs [10].

The other approach for bug localization is deep learning, which requires less human labors and shows better accuracy than IR methods, because the more abstract high-dimensional features which it uses have better representation capabilities. Lam *et al.* [29] present DNNLOC which combines the features built from DNN, rVSM, and the bug-fixing history of a software project, and DNNLOC achieves higher accuracy than the state-of-the-art IR and machine learning techniques. To solve the lexical mismatch between natural and programming languages, Huo and Li[14] and Huo *et al.*[22] employ convolutional neural network (CNN) to learn unified features from natural and programming language. Xiao *et al.* [35] propose DeepLocator which leverages an enhanced CNN considering bug-fixing history and achieves 3.8% higher MAP than DNNLOC.

The associate editor coordinating the review of this article and approving it for publication was Weiping Ding.

Although these tools can model natural and programming language for bug localization, there is room for improvement on accuracy and performance. On one hand, the hierarchical nature of programming language is not well-modeled, and thus the programs are not fully understood by deep learning models. The reason behind this is that existing representation learning algorithms in deep learning are unable to make full use of hierarchical structures, e.g. the AST of a program. Natural language files are always written in one dimension over time. In contrast, developers always write their code files with multiple hierarchical structures, e.g. branches, loops or even nested structures. Hence, natural language is ''flat'' and its representation algorithms are not suitable for learning program structure features directly. On the other hand, for performance, the ASTs of source code files used in existing methods have too many redundant or unrelated nodes, and hence result in longer training time, the curse of dimensionality, overfitting and a complex model. In this paper, we propose CAST to locate buggy files automatically and effectively. CAST leverages CNN to extract rich lexical semantic features, which indicate the relationship between syntactic entities, e.g. words or methods in bug reports and source files, and exploits TBCNN [9] on customized ASTs to capture hierarchical structure features, which contain the structural or semantic relation of program statements in source code files. A customized AST is obtained by 1) differentiating user-defined methods and system-provided ones in order to reflect their contributions leading to defects, 2) grouping the syntactic entities with similar semantics and pruning the ones with little or redundant semantics to improve the learning performance. Experimental results on four widely-used projects, i.e. AspectJ, JDT, SWT, and Tomcat, indicate that CAST significantly outperforms four existing tools for bug localization.

The main contributions of this paper are summarized as follows:

- We propose the customized AST for bug localization. It differentiates user-defined methods and system-provided ones to reflect their contributions leading to defects, which is helpful to improve the accuracy of bug localization. Moreover, it groups the syntactic entities with similar semantics and prunes the ones with little or redundant semantics to facilitate the learning performance.
- We present CAST system which exploits CNN on bug reports and TBCNN on the customized ASTs of source code for bug localization.
- Evaluation results on four widely-used software projects show that customized ASTs are beneficial for bug localization and the proposed CAST system outperforms four state-of-the-art tools for bug localization.

The rest of this paper is organized as follows. We introduce the motivation and preliminaries in section II and III respectively. Section IV presents the details of CAST. Section V describes the evaluation on CAST. We discuss the threats to validity in section VI. We introduce related work in section VII and conclude in section VIII.

TABLE 1. Bug (ID: 55342) report #55342 for Tomcat.

Summary: Lost interruption
Description: Function close should not call Thread.interrupted() because interrupt was already reset by code that thrown InterruptedException.

```
catch (InterruptedException ex) {
  if(getPoolProperties().getPropagateInterruptState()) {
    Thread.currentThread().interrupt();
  } else {
    Thread.interrupted();
  }
}
```

II. MOTIVATION

Abstract syntax trees (ASTs) are ordered trees where inner nodes represent operators (e.g., additions or assignments) and leaf nodes correspond to operands (e.g., constants or identifiers). While these trees faithfully encode how statements and expressions are nested to produce programs, we argue that they are not suitable for deep neural networks to localize defects because of two observations:

1) user-defined methods and system-provided methods are treated equally in ASTs though they obviously have different contribution to defects. In fact, user-defined methods usually call system or library methods whose code may be not available and thus not usable for bug localization. Moreover, the system methods usually are tested by wider community and hence with less bugs than user-defined ones.
2) the original AST contains a large number of redundant or unrelated features. If the original AST is fed directly to a deep neural network, it may lead to longer training time, the curse of dimensionality, and overfitting of the neural network.

Therefore, these observations motivate us to build customized ASTs for defect localization. As an example, bug #55342 report for Tomcat is shown in Table 1 and its buggy code in Listing 1, the code at lines 7-8 should be removed because the interrupted flag has been cleared when the `InterruptedException` is thrown. Fig.1(a) is the original AST of the example code, where the number after each node indicates the line number in code snippet. Fig.1(b) is our customized AST for the example code snippet, which prunes many useless nodes, i.e., those red nodes in Fig.1(a). Moreover, it also distinguishes the invocations of user-defined methods from the ones of system-provided methods. For example, a system method i.e., `interrupted()` at line 8, is called in above code snippet. From Fig.1(a), one cannot differentiate the two types of method calls which are unified as `Method_Invocation`; however, in Fig.1(b), this entity is refined into `Method_Invocation_Usr` and `Method_Invocation_Sys`. As a result, deep learning models can learn these features from the customized AST, which the original AST fails to provide, and pay more attention to the user-defined methods.

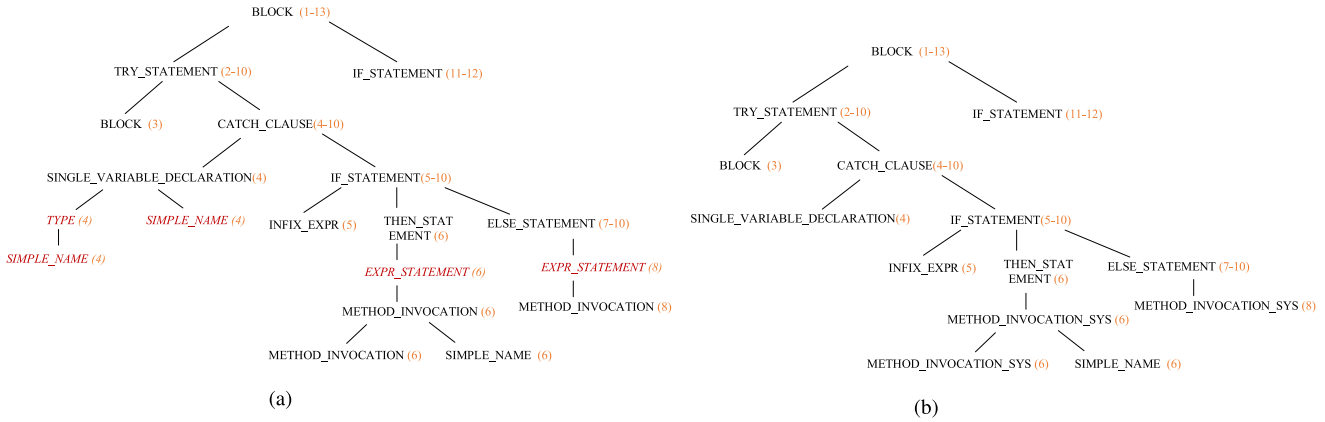


FIGURE 1. (a) Original AST and (b) Customized AST of The Code in Listing 1.

```
1  protected void close(boolean force) {
2    try {
3      ......
4    }catch (InterruptedException ex) {
5      if (getPoolProperties().
           getPropagateInterruptState()) {
6        Thread.currentThread().interrupt();
7      }else{
8        Thread.interrupted();
9      }
10   }
11   if (pool.size()==0 && force && pool!=busy)
12     pool = busy;
13 }
```

Listing 1. Buggy code snippet for report #55342.

III. PRELIMINARIES

A. CONVOLUTIONAL NEURAL NETWORK (CNN)

Convolutional neural networks (CNNs) are inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers, and normalization layers. In deep learning, CNNs are usually applied to analyze visual images.

A convolutional layer applies a convolution operation to the inputs, then passes the result to the next convolutional layer. The parameters of convolution layers consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. We slide each filter across the width and height of the input volume and compute at every spatial position the dot products of the entries in the filter. Then we use an activation function (e.g., Rectified Linear Unit, ReLU) to get a non-linear feature. The formula is as follows:

$$y = f\left(\sum_i w_i x_i + b\right) \tag{1}$$

where f is an activation function, w_i is the weight, x_i is the input, and b is the bias.

Pooling layers combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For instance, Max (Average) pooling uses the maximum (average) value from a cluster of neurons at the prior layer.

B. TREE-BASED CONVOLUTIONAL NEURAL NETWORK (TBCNN)

TBCNN is proposed by Mou *et al.* [9]. Its main components include vector representation and coding, tree-based convolution, dynamic pooling, fully-connected networks, and an output. To represent the AST of a program as vectors, the authors propose a code criterion to ensure that similar AST nodes have similar feature vectors. As tree-based convolution, the authors design a set of fixed-depth subtree detectors sliding over the entire AST to extract structural features of the program, e.g., the structure relation of code tokens. In a fixed-depth window, if there are n nodes with vector representations $x_1, x_2, \ldots, x_n$, then the output of the feature detectors is

$$y = tanh(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}) \tag{2}$$

where $b_{conv} \in R^{N_c}$ is the bias, $W_{conv,i} \in R^{N_c \times N_f}$ is the weight.

To deal with varying children numbers of a node, the "continuous binary tree" method in TBCNN uses three weight matrices as parameters for tree-based convolution, i.e., W^t_{conv}, W^r_{conv}, and W^l_{conv} (superscript t, r, l means "top", "right", "left" respectively). The weight matrix for any node x_i is a linear combination of W^t_{conv}, W^r_{conv}, and W^l_{conv} as follows:

$$W_{conv,i} = \theta^t_i W^t_{conv} + \theta^l_i W^l_{conv} + \theta^r_i W^r_{conv} \tag{3}$$

and the coefficients computed as follows:

$$\eta^t_i = \frac{d_i - 1}{d - 1} \tag{4}$$

$$\eta^r_i = (1 - \eta^t_i)\frac{p_i - 1}{n - 1} \tag{5}$$

$$\eta^l_i = (1 - \eta^t_i)(1 - \eta^r_i) \tag{6}$$

where d_i is the depth of the node x_i in the sliding window, d is the depth of the window, p_i is the position of the node, n is the total number of p's siblings.

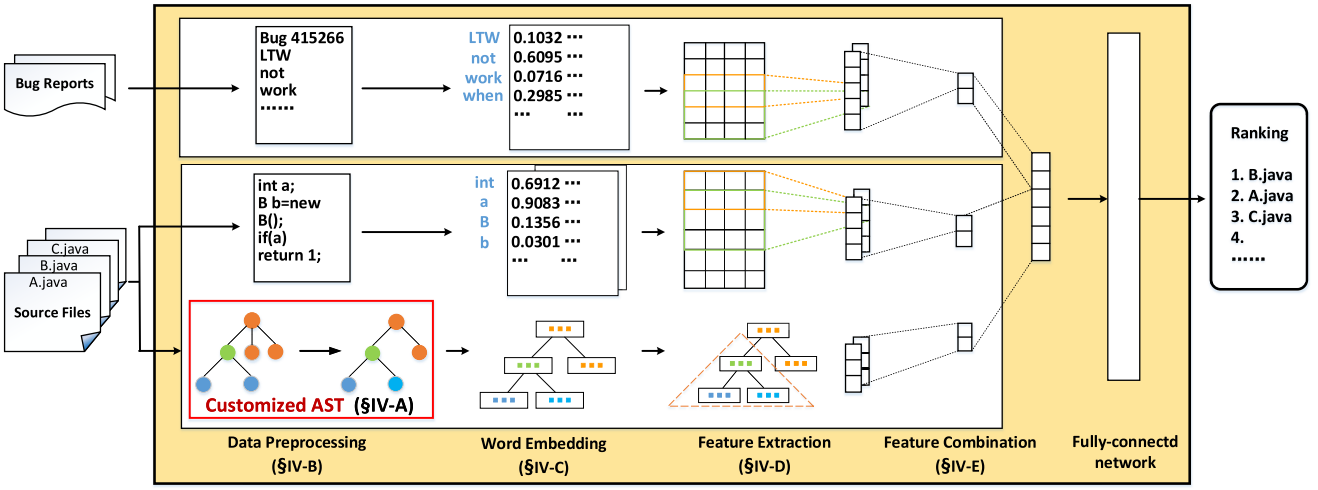


FIGURE 2. CAST's architecture.

IV. APPROACH

The architecture of CAST is shown in Fig.2. CAST takes the bug reports and source files as inputs and deals with them into two partitions for *data preprocessing*, *word embedding*, *feature extraction*, and fuses them in *feature combination*, Finally CAST feeds fusion features into the *fully-connected network* and ranks the given source files. During the training phase, pairs of source code files and bug reports, and their labels are fed into the CAST network which is then trained iteratively to optimize the loss function. At the testing phase, the trained network is given a new bug report and its candidate source files, and outputs the scores of these source files. Each score indicates the relevance between a source code file with the given bug.

We explain the approach via the example, i.e., report #55342 in Table 1. The report #55342 for Tomcat and source files are parsed into words respectively by data preprocessing (IV-B) and then all words are converted to feature vectors (IV-C). Next the feature vectors of report and source files are fed into the CNNs to extract lexical semantic feature (IV-D). In addition, we use the original ASTs like Fig.1(a) to build the customized ASTs like Fig.1(b) (IV-A), and feed them into the TBCNN to extract hierarchical structure feature (IV-D). Finally, all extracted features are combined (IV-E) using a fully connected neural network to get the scores which indicate the relevances between source files (e.g., Listing 1) and the report #55342.

In summary, CAST is formalized as a learning task. For a software project P, let $R = r_1, r_2, \ldots, r_i, \ldots, r_{n_1}$ denotes the set of bug reports received by its bug tracking system and $F = f_1, f_2, \ldots, f_j, \ldots, f_{n_2}$ denotes the set of source files in P where n_1, n_2 denotes the number of bug reports and source files respectively. CAST attempts to learn a *prediction function* $R \times F \rightarrow Y$, $y_{ij} \in Y = \{+1, -1\}$ indicating whether a source file f_i is related to a bug report r_j.

A. CUSTOMIZED AST

To build a customized AST, we first extract the original AST from a source file, then differ the user-defined methods from the system-provided ones, and compact the syntactic entities of the AST. Specifically, we can obtain the original ASTs by using the Eclipse Java Development Tools (JDT) which can parse a Java file into different syntactic entities, 92 kinds of entities in total.

1) DISTINGUISH METHOD INVOCATIONS

In original ASTs, user-defined methods and system-provided methods are treated equally though they obviously have different contribution to defects. In fact, user-defined methods usually call system-provided methods whose code may be not available and thus not usable for bug localization. Moreover, the system methods usually are tested by wider community and hence with less bugs than user-defined ones. As a result, we distinguish them in the customized ASTs. Specifically, user-defined methods and system-provided methods are expressed as `Method_Invocation` in the ASTs extracted by JDT ASTParser, whereas we replace `Method_Invocation` with two refined ones: `Method_Invocation_Usr` and `Method_Invocation_Sys`. This distinction between the user-defined methods and the system ones provides richer information than the original ASTs. In current implementation, CAST adopts a "whitelist" approach which includes all methods provided by system-provided classes, e.g., from Java standard library and those third-party libraries imported by the project under analysis.

2) RECONSTRUCT ASTs

Original ASTs usually contain lots of entities which can provide little even no help for bug localization. For example, JDT ASTParser can produce 92 different syntactic entities.

We reconstruct the structure of AST by *grouping* syntactic entities with same/close meaning and *pruning* redundant syntactic entities. As a result, we reduce syntactic entities from 92 kinds in original ASTs to 70 kinds by grouping syntactic entities and further to 54 kinds by pruning redundant syntactic entities. As a result, the percentage of redundant ones is about 41%.

a: GROUP SYNTACTIC ENTITIES

In original ASTs, entities with the same meaning lead it hard to learn the hierarchical features well in TBCNN. Therefore, we divide all entities into multiple equivalence classes (6 in total) according to the function of each entity, e.g., annotation, loop. The syntactic entities in an equivalence class are then grouped, for example, `Marker_Annotation` and `Normal_Annotation` are grouped as `Annotation`, while `For_Statement` and `While_Statement` are grouped as `Loop`. Certainly we regard a kind of entity as unique, e.g., `Modifier`, if it could not be grouped with other kinds.

b: PRUNE SYNTACTIC ENTITIES

The original ASTs usually include a large number of redundant or unrelated entities. If the original ASTs are fed directly to a deep neural network, it may lead to longer training time, the curse of dimensionality, and overfitting of the neural network. Therefore, we prune those redundant or unrelated syntactic entities in order to learn the unique structural features of each AST. Specifically, on one hand, we cut down common nodes of subtrees based on their semantics. For example, the child of a kind of entity must be another kind of entity. e.g., {`Simple_Type` → `Simple_Name`}, which means that the child of `Simple_Type` nodes must be `Simple_Name` nodes in AST, so we prune the `Simple_Name` nodes. On the other hand, we prune all nodes, each of which is the common parent node, for example, Expression_Statement is the common parent of nodes Assignment or Method_Invocation_Usr. So, we do not care `Expression_Statement` nodes while we are more concerned their children nodes with rich semantics, hence we replace {`Block` → `Expression_Statement` → `Method_Invocation_Usr`} with {`Block` → `Method_Invocation_Usr`}.

B. DATA PREPROCESSING

A bug report usually contains the summary and description for a found bug. A source file includes both code and comments. First, each text in bug reports and source files is filtered for stop words, numbers and punctuation using the NLTK package [39]. Besides, compound words are split into single words based on the CamelCase Naming Convention [36], e.g., "GetClassName" is split into "Get", "Class" and "Name". Finally, all words are reduced to their stem using the Porter stemmer [38]. For example, "setting" and "sets" has the same stem "set", which will make a positive impact on the recall performance.

C. WORD EMBEDDING

After preprocessing, all words in bug reports and source files are converted to feature vectors using a word embedding technique (word2vec [13]). Specifically, we first build a vocabulary V_1 which contains top 5000 words that appear frequently in bug reports and source files, and thus each word in bug reports and source files is represented as a one-hot vector using vocabulary V_1. Then we employ word2vec with skip-gram model [13] to convert each word into a k-dimensional vector. For words not presented in the V_1, we randomly initialize them.

For customized ASTs which contain 54 nodes types, we build a vocabulary V_2 which thus contains 54 words. For a given customized AST, CAST represents its nodes by using a variation of word2vec, which can learn the context information, i.e., its parent and children nodes, of each node in the tree. Finally, the AST of each source file is represented by $vec(.) \in R^{N_c}$, where $vec(.)$ is the feature representation of a node in the AST and N_c is the dimension of $vec(.)$.

D. FEATURE EXTRACTION

Since extracting features from natural language using CNN has been widely studied [12], we follow the approach to extract features from bug reports, denoted as x^r.

For source files, we capture lexical semantic features using different sizes of convolution kernels, as shown in Fig.3. Suppose $x_i \in R^k$ is the k-dimensional word vector corresponding to the i-th word in a statement, n_l is the maximum number of sentences in the source files, and n_w is the maximum number of words in the sentence. The *first* convolutional layer employs a filter, i.e., a window $h \times k$: $w \in R^{h \times k}$, to convolve the vectors. In order to extract different kinds of information, we apply m feature maps for each filter. Hence, after the first convolutional layer, the input dimension converts $n_l \times n_w \times k$ to $n_l \times (n_w - h + 1) \times m$. Next, the first max-pooling layer extracts the most important information from all words in source files and the dimension becomes $n_l \times m$. Each row after convolution and pooling corresponds to a statement in source files, and then the second convolution and pooling layers extract the interaction relation between sentences within h lines. Hence, we represent a source file as $(dm) \times 1$ vector, denoted as x^f, where d is the number of values which h can take.

To extract structural features from an AST, we use TBCNN [9], which has shown excellent performance on extracting the structural features in source code files. Specifically, we feed the feature vectors of all nodes in the AST (see section IV-C) to a tree-based convolutional layer. We apply a set of feature filters with a fixed window size to slide across the entire tree, as discussed in section III-B. Assume a node n_1 with vector vec_1 has children nodes $n_2, \ldots, n_i, \ldots, n_t$ with vectors $vec_2, \ldots, vec_i, \ldots, vec_t$ respectively, the vector vec_1 is updated as:

$$vec_1' = ReLU(\sum_{i=1}^{t} W_{conv,i} * vec_i + b_{conv}) \quad (7)$$

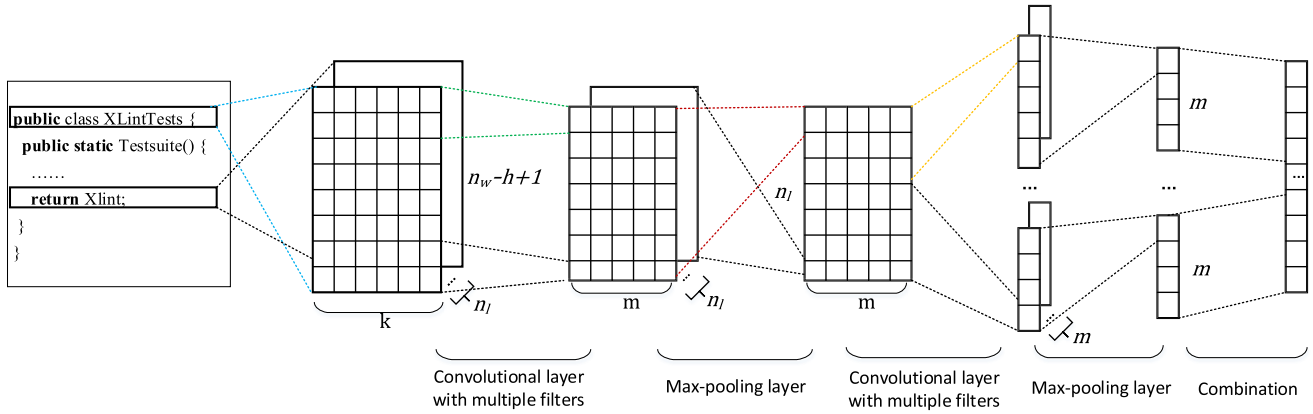


FIGURE 3. Extracting lexical-semantic features of source files.

where $W_{conv,i} \in R^{N_s \times N_c}$ refers to the weight of the vector vec_i, $b_{conv} \in R^{N_s}$ is the bias, N_s is the number of feature filters. As such, we get a new tree whose shape and size are the same as the previous one, but the vectors of all nodes are updated. Finally, through max-pooling operation, we obtain the maximum value of each dimension of all vectors and thus form one vector denoted x^s.

E. FEATURE COMBINATION

In the feature combination phase, all extracted features are combined using a fully connected neural network for supervised training. Given a bug report r_i and a source file f_j, the formulas are as follows:

$$x_{i,j} = (x_i^r, x_{i,j}^f, x_{i,j}^s) \tag{8}$$

$$y'_{i,j} = f(w_{i,j} \cdot x_{i,j}) \tag{9}$$

where x_i^r is the lexical semantics feature of the bug report, $x_{i,j}^f$ is the lexical semantic feature of the source file, and $x_{i,j}^s$ is the structure feature of the source file. $x_{i,j}$ is the fusion of above features. $y'_{i,j}$ is the output of the fully connected network, f is the sigmoid function.

F. OPTIMIZATION FUNCTION

In real-world applications, only a few source files are related to bug reports and most of the source files are irrelevant, which leads to class imbalance. Class imbalance [27] hurts the training accuracy, which will make neural networks not locate the buggy files accurately.

To solve the issue, we follow the optimizaiton function of [22] to consider the cost of wrongly associating a source file to a bug report ($cost_n$) and the cost of missing a buggy file that is responsible for the reported bugs ($cost_p$). $cost_n$ and $cost_p$ are hyper-parameters set by running the network with validation set. The optimization function is defined as follows:

$$Opt = \min_w \sum_{i,j} [cost_n L(y'_{i,j}, y_{i,j}; w)(1 - y_{i,j}) + cost_p L(y'_{i,j}, y_{i,j}; w)(1 + y_{i,j})] + \lambda ||w||^2 \tag{10}$$

where L is the cross-entropy loss function, and λ is the trade-off parameter. The weight w is learned by minimizing the above objective function based on Adam algorithm [37].

V. EVALUATION

To evaluate the performance of CAST, we focus on four research questions (RQ) as follows:

RQ1 What effect do the different model settings have on CAST? When building CAST, we need to determine the suitable values of hyper-parameters.

RQ2 Can CAST outperform other bug localization methods? To evaluate the capability of CAST, we compare CAST with four state-of-the-art tools in bug localization (BugLocator [28], DNNLOC [29], DeepLocator [35], NP-CNN [22]).

- BugLocator uses a revised vector space model (rVSM) and considers information about similar bugs.
- DNNLOC combines the features built from DNN, rVSM, and takes bug-fixing history into account.
- DeepLocator uses CNN and AST to extract features from bug reports and source files that are preprocessed using a revised TF-IDuF [40].
- NP-CNN leverages both lexical and program structure information to learn unified features from natural language and source code in programming language for bug localization, as CAST does.

Since these tools are not available, we use the public available version of BugLocator from Lee et.al. [3] and we implement our version of NP-CNN according to its paper [22]. As to DNNLOC and DeepLocator, their datasets are same as ours, and thus we directly use the results provided in the papers [29] and [41].

RQ3 What effect does the customized AST have on CAST? To answer this question, we conduct two sets of experiments. In the first group, we evaluate whether the customized ASTs can

TABLE 2. The Java projects in datasets.

Project	Time Range	#Bug Reports	#Java Files
AspectJ	03/2002-01/2014	593	6508
SWT	02/2002-01/2014	4151	2056
JDT	10/2001-01/2014	6274	8034
Tomcat	07/2002-01/2014	1056	1550

improve CAST's performance over the original ASTs for bug localization on the subject projects. In the second one, we combine the customized AST with a normal CNN and evaluate its performance on bug localization.

RQ4 What effect does the word embedding over customized ASTs have on CAST? To compare the contributions of word embedding and customized ASTs, we evaluate their effects by holding-out one another from CAST.

A. EXPERIMENTS PREPARATION

To evaluate the effectiveness of CAST, we use four open-source Java projects (AspectJ, SWT, JDT, and Tomcat) shown in TABLE 2, which has been widely studied in the previous studies. Files and bugs of the projects can be obtained by using a bug tracking system (e.g., Bugzilla) or version control system (e.g., Git). Because software bugs are often found in different versions of the source files, we use the before-fixed version of the source files for evaluation. To make the comparison with existing techniques easier, we use the same splitting strategy as in Lam *et al.* [29]. For all datasets but Aspectj, we use 10-fold cross validation. We split the chronologically sorted bug reports of each datasets into 10 equally sized folds $fold_1, fold_2, \ldots, fold_{10}$, where $fold_1$ is the oldest and $fold_{10}$ is the newest. AspectJ is the smallest project of the datasets, so we divide AspectJ into 3 folds. Furthermore, we split $fold_1$ into 60% training and 40% validation, in order to tune CAST's hyper-parameters, e.g., the numbers and sizes of filters. CAST is trained on fold k and tested on fold $k+1$, for all $k \leq 9$, which means that we always train on the most recent bug reports, which are supposed to better match the properties of the bugs in the current fold. We run all experiments on a server with CPU Intel Xeon CPU E5-2650 2.00GHz (16 cores), 96 GB RAM. With the space cost of 16G, CAST can run smoothly.

B. EVALUATION METRICS

To evaluate the performance of CAST, we use Accuracy@k, Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) as metrics, which are frequently used in existing studies of bug localization [22], [29], [35].

- Accuracy@k measures the percentage of bug reports for which we make at least one correct recommendation in the top k ranked files.
- Mean Average Precision (MAP) is a standard metric widely used in information retrieval [31]. The higher the values of MAP, the better the performance of the technique. MAP is formulated as follows:

$$MAP = \frac{1}{n_1} \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \frac{\frac{Q(j)}{j} * bool(j)}{N_i} \tag{11}$$

where n_1, n_2 respectively indicate the number of bug reports and source files. $bool(j)$ denotes whether the source file in rank j is buggy or not. N_i is the number of buggy files for the ith report. $Q(j)$ indicates the number of buggy files correctly located in top j.

- Mean Reciprocal Rank (MRR) is the mean of the accumulations of the inverse of the ranks of the first correctly-located buggy file for each bug report. It is computed by:

$$MRR = \frac{1}{n_1} \sum_{i=1}^{n_1} \frac{1}{first_i} \tag{12}$$

where n_1 indicates the number of bug reports and $first_i$ denotes the position of the first correctly-located buggy file for the ith bug report.

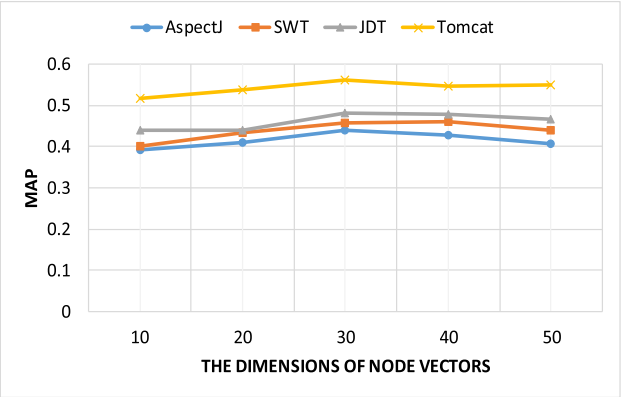


FIGURE 4. Performance of different dimensions of node vectors.

C. EXPERIMENTAL RESULTS AND ANALYSIS

Answer to RQ1: What effect do the different model settings have on CAST?

The hyper-parameters of CAST include the dimension of word vectors (k), the dimension of node vectors (N_c), the size of filters, the number of filters and the depth of the tree convolution. According to the work [35], the dimension of word vectors $k = 100$ can achieve comparable performance. The depth of the tree convolution is 2, based on the setting in [9]. Moreover, in our preliminary experiment, we found that extra low (high) ratio of $cost_p$ and $cost_n$ leads to lower precision (recall) of localizing buggy files. Therefore, we set $cost_n$ and $cost_p$ as 1 and 6 respectively.

First, to observe the effect of different dimensions of node vectors on the validation set, we test different dimensions (N_c) from 10 to 50 on four projects as shown in Fig.4. We consider that $N_c = 10$ is too small to learn deep-level features of AST nodes, while $N_c = 50$ brings a comparable performance but consumes more time. So we set N_c as 30.

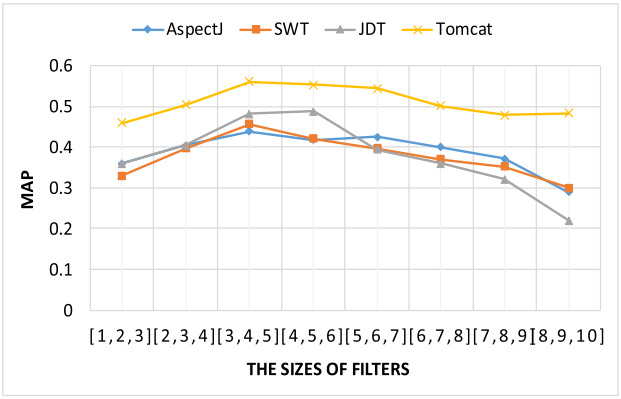


FIGURE 5. Performance of different sizes of filters.

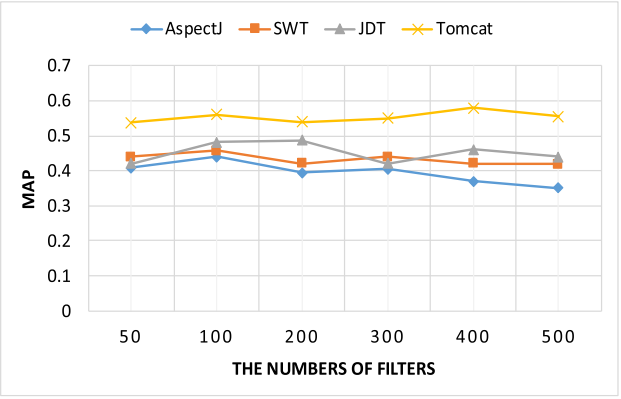


FIGURE 6. Performance of different numbers of filters.

Second, we try different filter sizes from 1 to 10 on four projects as shown in Fig. 5. Obviously if the filter size is very small, e.g, [1, 2, 3], the model cannot extract valid features from one or two words. If the larger filter size is used, e.g, [8, 9, 10], in other words, more words are in a group, it is harder for the model to extract semantic features from a large group. So we choose [3, 4, 5], in this case we get best MAP value.

Then, we compare the MAP value to determine the number of filters on the validation set. We experiment with 50, 100, 200, 300, 400 and 500 filters on four projects respectively. As shown in Fig.6, when the value is 100, the best MAP values are gotten in SWT and AspectJ, while near the best MAP values are in JDT and Tomcat. So we set the number of filters as 100.

Answer to RQ2: Can CAST outperform other bug localization methods? CAST is compared to four state-of-the-art techniques for bug localizaton. Their performances on four projects are shown in Table 3. We can observe that for accuracy CAST always performs better than the four competitors except for Accuracy@1 in Project Tomcat, for MAP and MRR it always is better than the four competitors except in Project JDT, but the loss is negligible.

For Accuracy@1,5,10,15, CAST is better than BugLocator because CAST can extract local abstract features (e.g., the equal semantic) to bridge the lexical gap between bug reports (e.g., "obtain the number of . . . ") and source files (e.g., "getnumber()") while BugLocator cannot.

CAST achieves higher Accuracy@1,5,10,15 on three of four projects than DNNLOC, because some bug reports describe with the symbols (e.g., class A, method B) which DNNLOC cannot handle well.

Compared with DeepLocator, CAST achieves higher Accuracy@1,5,10,15. DeepLocator also employs CNN and AST, but it treats equally bug reports in natural language and source files in programming language. Besides, DeepLocator directly maps an AST to a vector and thus loses the structure information of the program. In contrast, CAST handles the source files in programming language separately and combines a customized AST with TBCNN.

CAST achieves higher Accuracy on all projects than NP-CNN. In fact, both CAST and NP-CNN deal with bug reports and source files in the same way. However, CAST leverages customized ASTs and hence is good at dealing with nested relationship of sentences and those semantic-unrelated statements that are neighbors in real code.

The performance comparisons in terms of MAP and MRR indicate that CAST outperforms than other four competitors in all data sets except JDT project, where DeepLocator is better than CAST because many bug reports in JDT are related to history fixed bugs and DeepLocator considers history bugs.

Moreover, to show the statistical significance of our experimental results above, we conduct the effect size test, i.e., Cliff's delta or δ [42]. It is a measure that quantifies the aumount of difference between two groups:

$$\delta = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 n_2} \tag{13}$$

where x_1 and x_2 are scores from group 1 and group 2, with size of n_1 and n_2 respectively. The symbol # indicates counting. The value of Cliff's Delta lies in the closed interval $[-1, 1]$, where -1 or 1 indicates that all scores from one group is smaller or larger than those from the other. Table 4 shows the average Cliff's delta of CAST against BugLocator and NPCNN approaches on the 10 sets of experiments. According to the table, CAST makes the practical significance of difference compared with the two approaches.

Answer to RQ3: What effect do the customized ASTs have on CAST?

Experiment 1: To evaluate the effect of the customized ASTs, we build a variant CAST (CASTo for short) by replacing customized ASTs with original ASTs, and conduct experiments on the datasets. Fig.7, Fig.8, Fig.9, and Fig.10 present their Accuracy@k results for the four projects, with k ranging from 1 to 20. Fig.11 and Fig.12 show their MAP and MRR results for the projects respectively. Experiments results show that CAST performs better than CASTo. This is because original ASTs have more nodes than customized ASTs, and thus cause feature redundancy and overfitting. In addition, we also record the average prediction time required for CAST and CASTo to locate buggy files per bug report,

TABLE 3. Performance comparison with the state-of-the-art techniques.

Project	Method	Accuracy@1	Accuracy@5	Accuracy@10	Accuracy@15	MAP	MRR
AspectJ	BugLocator	0.216	0.473	0.571	0.643	0.276	0.369
	DNNLOC	0.478	0.712	0.804	0.862	0.320	0.520
	DeepLocator	0.400	0.660	0.780	-	0.340	0.490
	NPCNN	0.460	0.730	0.810	0.852	0.401	0.531
	CAST	**0.500**	**0.766**	**0.830**	**0.870**	**0.418**	**0.536**
SWT	BugLocator	0.246	0.408	0.533	0.595	0.284	0.325
	DNNLOC	0.352	0.690	0.803	0.853	0.370	0.450
	DeepLocator	0.360	0.600	0.750	-	0.390	0.480
	NPCNN	0.365	0.700	0.812	0.865	0.381	0.475
	CAST	**0.372**	**0.701**	**0.825**	**0.873**	**0.425**	**0.503**
JDT	BugLocator	0.183	0.427	0.508	0.562	0.307	0.389
	DNNLOC	0.403	0.650	0.743	0.794	0.340	0.450
	DeepLocator	0.400	0.640	0.730	-	**0.390**	**0.470**
	NPCNN	0.420	0.669	0.746	0.804	0.383	0.461
	CAST	**0.432**	**0.681**	**0.757**	**0.818**	0.388	0.467
Tomcat	BugLocator	0.354	0.645	0.709	0.752	0.431	0.485
	DNNLOC	**0.539**	0.729	0.804	0.859	0.520	0.600
	DeepLocator	0.520	0.720	0.800	-	0.540	0.600
	NPCNN	0.530	0.700	0.792	0.853	0.529	0.597
	CAST	0.507	**0.766**	**0.825**	**0.869**	**0.556**	**0.612**

TABLE 4. Cliff's delta effect size test results of CAST against BugLocator and NPCNN approaches.

Project	Method	Accuracy@10	MAP	MRR
AspectJ	BugLocator	1	0.98	1
	NPCNN	0.8	0.44	0.5
SWT	BugLocator	1	1	1
	NPCNN	0.70	0.72	0.78
JDT	BugLocator	0.94	1	1
	NPCNN	1	0.82	0.0.76
Tomcat	BugLocator	1	1	1
	NPCNN	0.32	0.54	0.58

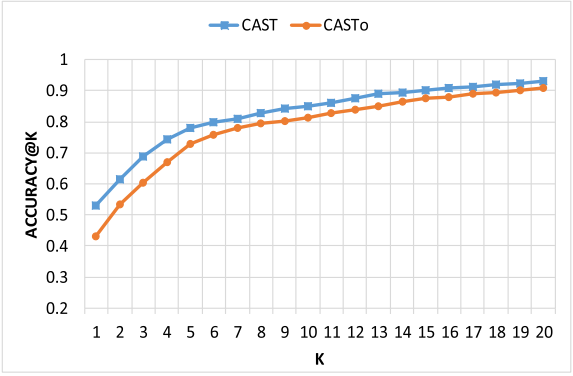


FIGURE 7. Accuracy graphs on AspectJ.

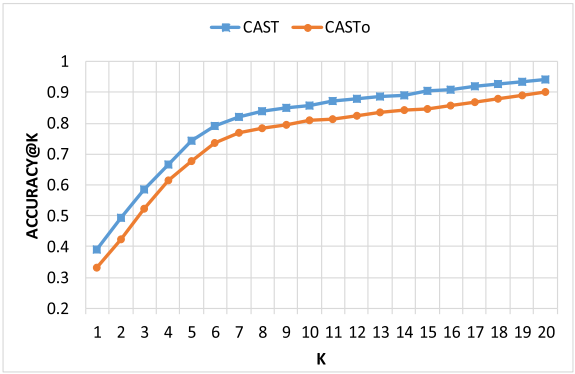


FIGURE 8. Accuracy graphs on SWT.

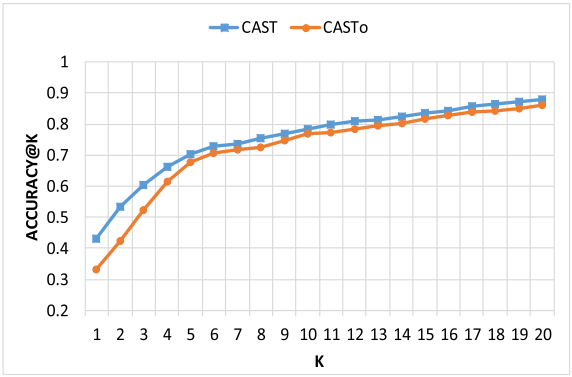


FIGURE 9. Accuracy graphs on JDT.

as shown in Table 5 CAST's prediction time is much lower (20%) than CASTo's, which shows that CAST is more efficient than CASTo. Due to the pruning and grouping AST entities, customized ASTs help reduce the parameters of the learning models, thereby improving the performance of bug localization.

Experiment 2: Although we get better performance by combining customized ASTs and TBCNN, we want to explore whether the customized ASTs can also help other models in bug location. In this experiment, we implement a variant (named CASTp) by replacing TBCNN with a normal CNN and compare it with CAST. Table 6 shows the MAP values of the two models in four projects. As shown in the table, CAST performs better than CASTp in all projects,

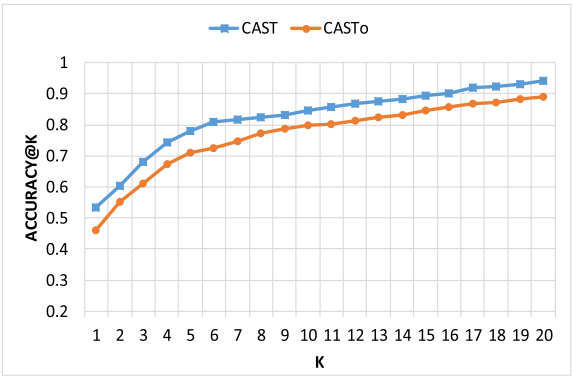


FIGURE 10. Accuracy graphs on Tomcat.

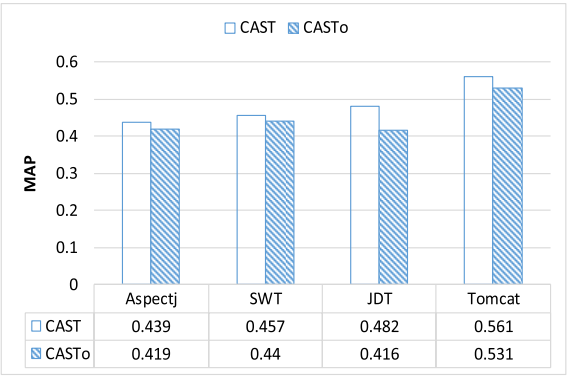


FIGURE 11. MAP comparison on AspectJ, SWT, JDT, and Tomcat.

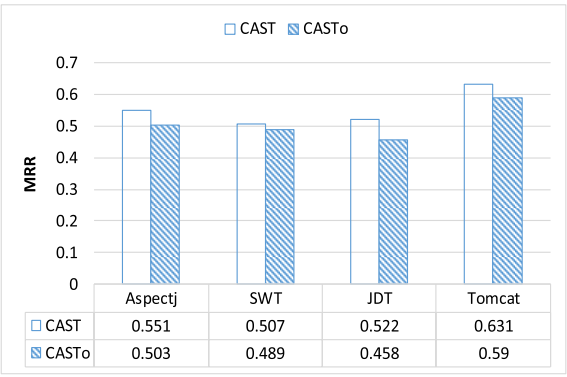


FIGURE 12. MRR comparison on AspectJ, SWT, JDT, and Tomcat.

TABLE 5. Prediction time of CAST and CASTo.

Time(sec.)	AspectJ	SWT	JDT	Tomcat
CAST	150.5	217.3	294.3	61.8
CASTo	190.9	248.9	380.1	82.7

which indicates that combining TBCNN with customized ASTs benefits for bug localization.

Answer to RQ4: What effect does the word embedding over customized ASTs have on CAST?

To explore the contributions of word embedding and customized ASTs, we evaluate their effect by holding-out one another from CAST on the datasets. We build a variant

TABLE 6. Performance comparison of CAST and CASTp.

MAP	AspectJ	SWT	JDT	Tomcat
CAST	0.439	0.457	0.482	0.561
CASTp	0.341	0.368	0.395	0.414

TABLE 7. Performance comparison of CAST, CASTq and CASTo.

MAP	AspectJ	SWT	JDT	Tomcat
CAST	0.439	0.457	0.482	0.561
CASTq	0.431	0.453	0.476	0.550
CASTo	0.419	0.440	0.416	0.531

CAST (CASTq) by replacing word embedding with vectors random initialization, and compare it with CAST and CASTo (built for RQ3). Table 7 shows the MAP values of the three models on four projects. As shown in the table, CAST obtains higher MAPs on four projects than CASTq, which indicates that word embedding can improve the performance of CAST. CASTq performs better than CASTo on all projects, which shows that the contribution of customized ASTs is more than word embedding.

VI. THREATS TO VALIDITY

Although CAST performs well in the experiments, there are still some potential threats to validity of our study.

A. INTERNAL VALIDITY

Firstly, the performance of our proposed model may be somewhat dependent on the performance of the word embedding techniques. Hence we checked the property of the adopted word2vec technique before adopting it in our model. Secondly, the hyper-parameters configuration set could introduce some bias in the experimental results. However, we set the parameters according to the suggestions of the existing studies, which enabled our choices to be reasonable. For example, the dimension of word vectors $k = 100$ were set as the suggestions given in the literature [35]. More fine-tunes might be needed for our model. We leave it for future studies. Thirdly, we cannot get the dataset of NP-CNN though we did contact its authors. We'll evaluate CAST on the dataset after it is publicly released.

B. EXTERNAL VALIDITY

We evaluated our model on four dataset from Java projects as many bug localization studies [22], [28], [29], [35] did and tried to report the general results. However, the selection of only four projects may have potentially limited representativeness. Besides, the performance of CAST on other projects written in other programming languages is still unknown. We plan to examine CAST on more projects, especially the ones written in, e.g. C++, in a future study.

VII. RELATED WORK

A. BUG LOCALIZATION

Bug locating is an essential but still costly activity in software development. IR-based fault localization techniques can help

developers locate faults by exploring the link between bug reports and source files. Poshyvanyk *et al.* [15] employed Latent Semantic Indexing (LSI) to locate bugs, which represents code files and bug reports as vectors and measures the cosine similarity between vectors. Lukins *et al.* [4] located bugs based on another IR technique, Latent Dirichlet Allocation (LDA), whose properties, which include modularity and extensibility, provide advantages over LSI. Gay *et al.* [25] applied Vector Space Model (VSM) to express each document as a vector and compute the similarity between them. Zhou *et al.* [28] proposed BugLocator based on a revised VSM (rVSM), which also utilizes information about similar bugs that have been fixed before to improve the ranking performance. However, the limitation of these aforementioned IR-based bug localization methods is a lexical mismatch between natural language texts in bug reports and terms in source files. These methods are based on the textual vector representation of bug reports and source files but ignore the semantic information in them.

In recent years, bug reports and code files as natural language text are used in some machine learning and deep learning models to extract the lexical semantics. Ye *et al.* [21] leveraged API descriptions to bridge the lexical gap between bug reports and source code and they also use domain knowledge by decomposing source files into methods, using the bug-fixing history etc. Lam *et al.* [7] combined rVSM with Deep Neural Network (DNN) to recommend the potentially buggy files for a bug report. DNN is used to capture high-level abstractions and compute the relevancy between a bug report and a source file. Even these approaches convert the heterogeneous data into the same lexical feature space, but they lose the structural information of the bug reports and source files. Huo *et al.* [22] proposed NP-CNN, to learn the unified features from bug reports and source files to extract the structural information. They also exploited the sequential nature of source code to enhance the unified features [14]. Xiao *et al.* [35] proposed DeepLocator which combines an enhanced CNN considering bug-fixing history with rTF-IDuF method. In CAST, we propose to combine customized ASTs and TBCNN, which can extract both the lexical semantics in bug reports and hierarchical structure features in source files, and thus improve performance and accuracy for bug localization.

B. CONVOLUTIONAL NEURAL NETWORK

CNN has outstanding performance not only in the field of image processing but also in the natural language processing. Kim [8] applied CNN for sentence classification tasks with pre-trained word vectors resulted from the word embedding technique and achieved excellent results on multiple benchmarks. Kalchbrenner *et al.* [1] proposed a convolutional architecture dubbed the Dynamic Convolutional Neural Network (DCNN) which adopts the semantic modeling of sentences. These approaches work well with generic data, but if we directly fed AST structure to a network, we may lose rich structural information of a program. Mou *et al.* [9] applied an unsupervised approach to learn vector representations of a program and proposed a tree-based CNN to detect structural information from programs.

VIII. CONCLUSION

In this paper, we propose CAST for bug localization, which combines tree based convolutional neural network with customized ASTs to locate buggy files effectively. CAST can capture both lexical semantic in bug reports and source files, and hierarchical structure information in AST. The proposed customized ASTs have richer semantics with the help of refining method invocations and less nodes types by pruning and grouping entities in AST, and hence help CAST gain improved performance and accuracy. Experimental results on four open-source projects show that customized ASTs do improve the performance of the model. Moreover, CAST achieved higher MAP (at most 0.044) and MRR (at most 0.033) than the best results of the four current state-of-the-art techniques (BugLocator, DNNLOC, DeepLocator and NP-CNN).

In future, we will evaluate the performance of our models in more Java projects, and explore to add more program information in our models, e.g., control dependency and data dependency. Besides, to further improve the CAST's prediction time, we will explore the combination of the customized AST and relatively simple models.

REFERENCES

[1] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics*, Baltimore, MD, USA, vol. 1, Jun. 2014, pp. 655–665.

[2] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 297–308.

[3] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4BL: Reproducibility study on the performance of IR-based bug localization," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Amsterdam, The Netherlands, Jul. 2018, pp. 61–72.

[4] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 972–990, Sep. 2010.

[5] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Bug localization with semantic and structural features using convolutional neural network and cascade forest," in *Proc. 22nd Int. Conf. Eval. Assessment Softw. Eng.*, Christchurch, New Zealand, Jun. 2018, pp. 101–111.

[6] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-Gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2016, pp. 708–719.

[7] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 476–481.

[8] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Doha, Qatar, Oct. 2014, pp. 1746–1751.

[9] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 13th AAAI Conf. Artif. Intell.*, Phoenix, AZ, USA., Feb. 2016, p. 1287–1293.

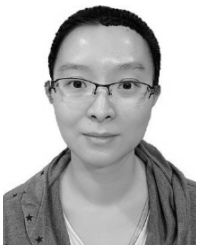
[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[11] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn.*, Beijing, China, vol. 32, Jan. 2014, pp. 1188–1196.

[12] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, Denver, CO, USA, 2015, pp. 103–112.
[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Represent. (ICLR)*, Scottsdale, AZ, USA, May 2013, pp. 1–12.
[14] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proc. IJCAI*, Aug. 2017, pp. 1909–1915.
[15] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.
[16] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Orlando, FL, USA, Dec. 2012, pp. 359–368.
[17] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization," in *Proc. 25th Int. Conf. Program Comprehension, (ICPC)*, Buenos Aires, Argentina, May 2017, pp. 241–250.
[18] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 345–355.
[19] Z.-W. Zhang, X.-Y. Jing, and T.-J. Wang, "Label propagation based semi-supervised learning for software defect prediction," *Automated Softw. Eng.*, vol. 24, no. 1, pp. 47–69, Mar. 2017.
[20] P. Loyola and Y. Matsuo, "Learning graph representations for defect prediction," in *Proc. 39th Int. Conf. Softw. Eng. Companion*, May 2017, pp. 265–267.
[21] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 689–699.
[22] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, New York, NY, USA, Jul. 2016, pp. 1606–1612.
[23] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.
[24] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Providence, RI, USA, 2012, pp. 3642–3649.
[25] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 351–360.
[26] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.
[27] Z.-H. Zhou and X.-Y. Liu, "Training cost-sensitive neural networks with methods addressing the class imbalance problem," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 1, pp. 63–77, Jan. 2006.
[28] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 14–24.
[29] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, Buenos Aires, Argentina, May 2017, pp. 218–229.
[30] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proc. Int. Conf. Knowl. Sci., Eng. Manage.*, Chongqing, China, Oct. 2015, pp. 547–553.
[31] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[32] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*. [Online]. Available: https://arxiv.org/abs/1802.00921
[33] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, Apr./May 2018, pp. 1–17.
[34] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript," in *Proc. 15th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Saclay, France: Springer, Jun. 2018, pp. 303–325.
[35] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Improving bug localization with an enhanced convolutional neural network," in *Proc. 24th Asia–Pacific Softw. Eng. Conf., (APSEC)*, Nanjing, China, Dec. 2017, pp. 338–347.
[36] D. W. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *Proc. 17th IEEE Int. Conf. Program Comprehension, (ICPC)*, Vancouver, BC, Canada, May 2009, pp. 158–167.
[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, San Diego, CA, USA, May 2015, pp. 1–15.
[38] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 40, no. 3, pp. 211–218, Jul. 2006.
[39] S. Bird, E. Klein, and E. Loper, *Natural Language Processing With Python*. Newton, MA, USA: O'Reilly, 2009.
[40] J. Beel, S. Langer, and B. Gipp, "TF-IDuF: A novel term-weighting scheme for user modeling based on users' personal document collections," in *Proc. 12th Conf.*, Wuhan, China, Mar. 2017, pp. 1–7.
[41] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Inf. Softw. Technol.*, vol. 105, pp. 17–29, Jan. 2019.
[42] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychol. Bull.*, vol. 114, no. 3, pp. 494–509, 1993.

HONGLIANG LIANG (M'14) received the Ph.D. degree in computer science from the University of Chinese Academy of Sciences, Beijing, China, in 2002. He is currently an Associate Professor with the Beijing University of Posts and Telecommunications, Beijing. His research interests include system software, program analysis, software security, and artificial intelligence. He is a member of the ACM and a Senior Member of the CCF. He serves as a Reviewer for some prestigious journals, including the IEEE TRel, ACM TIST, and JSS.

LU SUN received the B.E. degree in computer science from the Beijing Electronic Science and Technology Institute, in 2014. She is currently pursuing the master's degree with the Beijing University of Posts and Telecommunications. Her main research interests include deep learning and trusted software.



MEILIN WANG received the B.E. degree in electronic science and engineering from the University of Electronic Science and Technology of China, in 2003, and the master's degree in information science and technology from the University of International Relations, in 2009. She is currently a Researcher with the China Information Technology Security Evaluation Center. Her main research interests include software security and trusted software.

YUXING YANG received the B.E. degree in computer science from the Beijing University of Posts and Telecommunications, in 2016, where he is currently pursuing the master's degree. His main research interests include deep learning and trusted software.