Control Flow Graph Embedding Based on Multi-Instance Decomposition for Bug Localization

Xuan Huo, Ming Li, Zhi-Hua Zhou
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{huox, lim, zhouzh}@lamda.nju.edu.cn

Abstract

During software maintenance, bug report is an effective way to identify potential bugs hidden in a software system. It is a great challenge to automatically locate the potential buggy source code according to a bug report. Traditional approaches usually represent bug reports and source code from a lexical perspective to measure their similarities. Recently, some deep learning models are proposed to learn the unified features by exploiting the local and sequential nature, which overcomes the difficulty in modeling the difference between natural and programming languages. However, only considering local and sequential information from one dimension is not enough to represent the semantics, some multi-dimension information such as structural and functional nature that carries additional semantics has not been well-captured. Such information beyond the lexical and structural terms is extremely vital in modeling program functionalities and behaviors, leading to a better representation for identifying buggy source code. In this paper, we propose a novel model named CG-CNN, which is a multi-instance learning framework that enhances the unified features for bug localization by exploiting structural and sequential nature from the control flow graph. Experimental results on widely-used software projects demonstrate the effectiveness of our proposed CG-CNN model.

Introduction

As software systems grow rapidly, the scale of software project becomes larger and more complex, leading to increasing in difficulty to identify software bugs before its formal release due to the inadequate testing resources and tight development schedule. Thus software systems are often shipped with bugs.

Therefore, developers usually facilitate fast and efficient identification and fixing of bugs in a released software system based on bug reports. *Bug reports*, which are documents written in natural language specifying the situations in which the software fails to behave as it is expected or follow the technical requirements of the system. Unfortunately, many projects often receive more bug reports than what they can handle. To resolve each report, developers usually need to spend much time and effort.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

To alleviate the burden of developers, various effective models for bug localization have been proposed (Gay et al. 2009; Zhou, Zhang, and Lo 2012; Ye, Bunescu, and Liu 2014; Huo, Li, and Zhou 2016). Most existing methods treat the source code as natural language by representing both bug reports and source files based on traditional feature representations, and correlate their relevance by measuring similarity in the same feature space. For example, (Gay et al. 2009) represented both source code and bug reports using the vector space model (VSM), where the similarities between buggy source files and bug reports are computed for bug localization. (Zhou, Zhang, and Lo 2012) proposed a revised vector space model (rVSM), where similar historical bug reports information is exploited to improve the bug localization results obtained by measuring the similarity between bug reports and source files. Recently, to overcome the difference between natural and programming languages, Huo *et al.* (2016) proposed a unified framework that employs different convolutional neural networks to extract features from bug reports and source code respectively. The unified features are further enhanced by exploiting the sequential nature of source code via combining a LSTM model (Huo and Li 2017), which is proved effective in modeling source code and improving the performance of locating buggy source files.

However, previous models have not fully extracted semantics from source code, since the structural and functional nature of the programming language have not been well-captured, which may cause the loss of semantic information from the source code. Huo *et al.* (2016) designed a model based on convolutional neural network to generate high-level semantics based on the relationships among local statements, and the model in Huo *et al.* (2017) was built to exploit the sequential nature of source code. However, only modeling the local and sequential nature of source code are not enough, programming language contains more complex structure that involved multi-dimension information, such as “branch” and “loop” structure, which carries more structural and functional information. For example, a branching structure “`if-then-else`” defines two parallel groups of statements. Each group of statements interacts with the context before and after the branching block and

forms two different execution paths. However, there are no interactions between the two groups and the semantics of the two paths are different. Directly extracting semantic features among statements sequentially or locally may lose the structural semantics.

Therefore, in order to generate a more representative feature, more information about source code such as structural and functional semantics should be carefully considered. Control Flow Graph (CFG) (Allen 1970) is a kind of representation using graphic notations to represent source code, which carries rich structural and functional information, and all paths of CFG are generated by traveling through the program during its execution process. The control flow graph contains rich information reflecting the structural and functional nature of the source code, which is able to provide a better representation of source code and improve bug localization efficiency. However, there is no previous work that focuses on extracting semantic features from a control flow graph for bug localization.

One question arises here: how to utilize the information from CFG to generate the representation of source code? Extracting features from CFG contains two main challenges. First, how to generate the semantic feature of each node of CFG (i.e., each statement in source code). Different from previous graph based data, CFG is a directed graph and contains many execution paths of source code. Each statement of the source code is represented as a node in the CFG. The semantics of each statement is not only relative with the tokens it contains, but also relative with the neighboring statements in the same execution paths of source code. Another challenge is that how to represent the whole source code considering the structural and functional information from the CFG. It can be noticed that, each path of CFG contains different semantics that needs to be processed separately. In addition, the statements in each execution paths may contain sequential relationships, which should be carefully modeled for feature extraction.

In this paper, we propose a novel unified model based on multi-instance decomposition framework named CG-CNN (Control flow Graph embedding based Convolutional Neural Network), which aims to generate a better unified feature from bug reports and source code for bug localization. As aforementioned, learning semantics from source code contain big challenges and is the key of CG-CNN. To address the first challenge in programming language processing, CG-CNN firstly employs a convolutional neural network to incorporate local semantics of each statement to preserve the integrity, and further enhance feature representation by using DeepWalk to consider the relationships between neighboring statements. To address the second challenge, we design a multi-instance learning framework to generate the semantics from the CFG of source code. The CFG of a program contains multiple paths and each path in the control flow graph is considered as an instance while the whole code is a bag. If any path is relevant with the bug report that labeled as buggy (positive), the source code file is buggy and relevant to the bug report. Therefore, learning semantic feature representation from CFG of source code can be regarded as a multi-instance learning task. Experimental results on widely-used software projects show that CG-CNN performs significantly better than state-of-the-art methods, indicating that exploiting control flow graph based information from source code is beneficial for improving bug localization performance.

The contributions of our work are summarized as follows:

- We are the first to employ control flow graph of source code for bug localization based on multi-instance decomposition to extract semantic features from the control flow graph of a program, where both structural and sequential nature from the source code can be carefully captured and modeled.
- We propose a novel deep model named CG-CNN to learn unified features from control flow graph for bug localization. CG-CNN firstly applies CNN and DeepWalk to learn a representation of each statement and then utilizes multi-instance learning framework to learn representation of the source code from the multiple execution paths.

The rest of this paper is organized as follows. In Section 2, some related work about bug localization are discussed. In Section 3, we introduce the details of our proposed model CG-CNN. The experiments and some discussions are presented in Section 4, and finally in the last section, we conclude the paper and issue some future work.

Related Work

To maintain software quality assurance, many bug localization approaches have been studied in recent years. Bug localization, which identifies and locates source files potentially responsible for the bug reported in bug reports, is an extremely vital but costly task in software maintenance. Most traditional approaches treat the source code as documents and formalize the bug localization problem as a document retrieval problem, which calculate the relevancy between a bug report and a source file to identify buggy source code (Poshyvanyk et al. 2007; Lukins, Kraft, and Etzkorn 2008). For example, Gay *et al.* (2009) employed Vector Space Model (VSM) based on concept localization to represent bug reports and source code as feature vectors, which are used to measure the similarity between bug reports and source files. Zhou *et al.* (2012) also proposed BugLocator approach using revised Vector Space Model, which is based on document length and similar bugs that have been solved before as new features. Recently, more information and features from bug reports and source code have been investigated for identifying bugs. Saha *et al.* (2013) utilized structured information from source code, such as class and method to enable more accurate bug localization. Wang *et al.* (2014) proposed AmaLgam that combines version history, similar report and structure to further improve bug localization performance.

Recently, deep learning models are very popular and have achieved enormous success in many software engineering tasks (Mou et al. 2016; White et al. 2015). For example, Mou *el al.* (2016) proposed a novel tree-based convolutional neural network for programming language, which is effective in several software tasks such as program functionality classification and bubble sort program detection. Wei and Li (2017) studied software functional clone detection by applying a

novel tree-based LSTM model, which is able to exploit the lexical and syntactical information from source code. Shi *et al.* (2019) proposed a specific network to employ autoencoder to learn the feature of revision from an original-new source codes pair. There are also some work using deep learning models to identify buggy source code according to bug reports. For example, Lam et al. (2015), proposed a novel deep model that combines a kind of deep neural network (Auto-Encoder) and the revised Vector Space Model (rVSM) to improve bug localization performance. Shi *et al.* proposed a special type of deep neural networks, which employ autoencoder to learn the feature of revision from an original-new source codes pair. To overcome the structural difference between natural and programming languages, Huo *et al.* (2016) designed particular convolutional operations for programming language and proposed a CNN-based model to learn unified features from bug reports and source code for identifying buggy source code. The unified features are further enhanced by combining LSTM and CNN to generate richer information by exploiting the sequential nature of source code (Huo and Li 2017). Different from previous work, the goal of this paper is to show the rich information respecting to the structure of source code can be leveraged to improve bug localization. We consider to extract features from CFG and formalize this problem as a multi-instance learning problem, where a specific model CG-CNN is designed to address this challenge.

In addition, many models have been proposed to learn latent representation of nodes in a graph. For example, Perozzi *et al.* (2014), presented DeepWalk to learn latent social representations of vertices. Using local information from truncted random walks as input, DeepWalk learns a representation which encodes structural regularities. Experiments on a variety of different graphs illustrate the effectiveness of our approach on challenging multi-label classification tasks. Grover and Leskovec (2016) further extended DeepWalk by proposing a novel method named node2vec, which learns a mapping of nodes to low-dimensional space of features that maximizes the likelihood of preserving network neighboring of nodes. Node2vec defines a flexible notion of a node's network neighborhood and design a biased random walk procedure, which efficiently explores diverse neighborhoods. Generating from CFG is different from previous work, since CFG is directed and contains multiple paths reflecting the execution procedure of the source code. In our work, we design a multi-instance learning framework to extract features from CFG.

Our Method

In this paper, we introduce multi-instance decomposition for bug localization, which aims to locate the potentially buggy source files that produce the program behaviors specified in a given a newly finished bug report. Each source code is considered as a bag and the execution paths in the source code are considered as instances. Let $\mathcal{C} = \{C_1, C_2, \ldots, C_{N_1}\}$ denote the set of source code files, where $C_i = \{p_{i1}, p_{i2}, \ldots, p_{ini}\}$ is a source file with n_i execution paths. Let $\mathcal{R} = \{r_1, r_2, \cdots, r_{N_2}\}$ denotes the collection of bug reports, where N_1, N_2 denote the number of source files and bug reports, respectively. The learning task of bug localization aims to learn a prediction function $f : \mathcal{R} \times \mathcal{C} \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a source file $C_j \in \mathcal{C}$ is relevant to a bug report $r_i \in \mathcal{R}$. We instantiate the learning task based on multi-instance learning decomposition by proposing a novel deep model named CG-CNN (Control flow Graph based Convolutional Neural Network), which takes bug reports and source code as inputs and learns a unified feature mapping the $\psi(\cdot, \cdot)$ for a given r_i and C_j, based on which the prediction can be made with a subsequent output layer.

The General Framework of CG-CNN

The general framework of CG-CNN is shown in Figure 1. Specifically, CG-CNN consists of four main components: input and processing layer, language specific feature extraction layer, joint feature fusion layer and the last is output and prediction layer. In training process, pairs of source code and bug reports and their relevant labels are fed into the network, and the model is trained iteratively to optimize the loss. In testing process, a new bug report and some source code that potentially contains the corresponding bug is fed into the model, which outputs their relevant score indicating which code is highly relevant with the given bug report and are located as buggy.

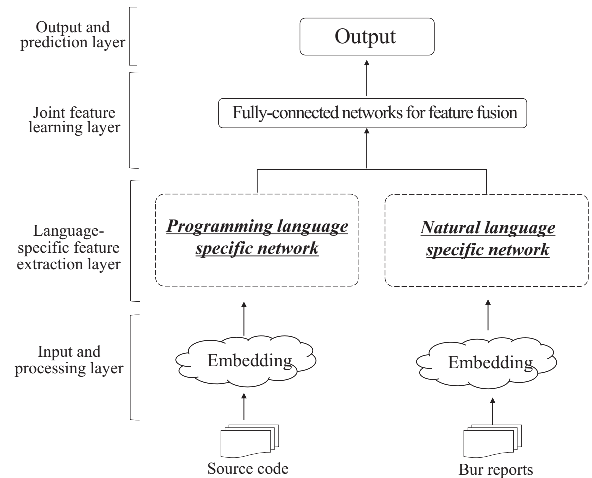

Figure 1: The general framework of CG-CNN. The model contains 4 main parts: input and processing layer, language specific feature extraction layer, joint feature fusion layer and the last is output and prediction layer.

To extract high-level semantic features for identifying buggy files, source code and bug reports are firstly encoded as feature representation, and then feed into the network in the input and processing layer. Since bug reports are written in natural language while source code is in programming language, they contain different structure and semantics which should be processed from different ways. Therefore, CG-CNN designs two networks to process source code and bug reports separately in language specific feature extraction layer: natural language specific feature extraction

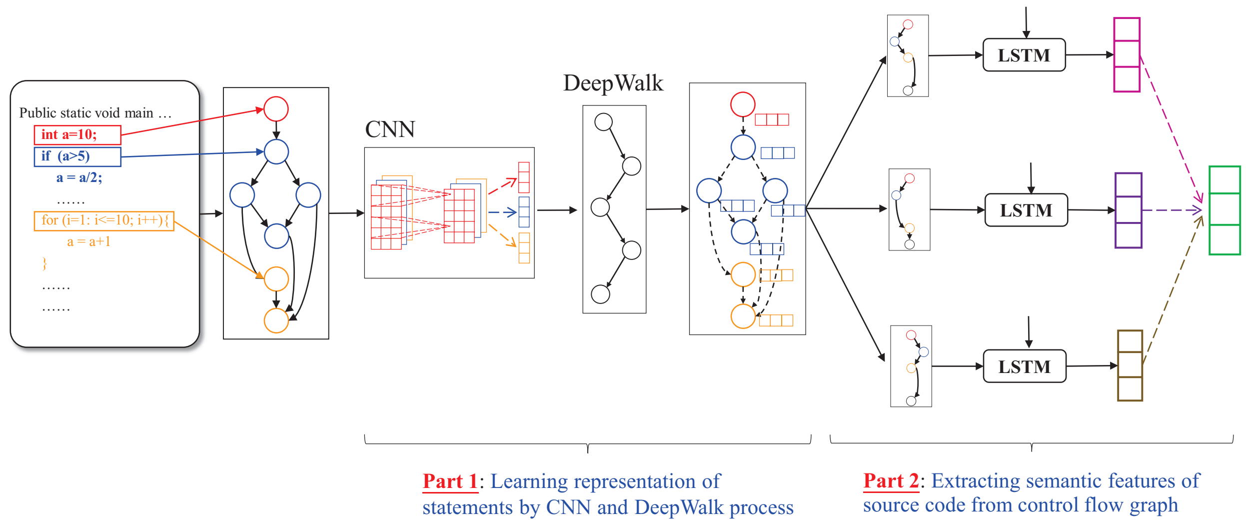

Figure 2: The overall structure of programming language specific feature extraction network in CG-CNN, which contains two parts to learn semantic feature representation of source code. The first part aims to learn representation of each statement, which employs CNN to extract the inner information of statements and a DeepWalk process to model the relationships among neighboring statements. Afterwards, the second part aims to extract the semantic feature based on multi-instance learning setting. The LSTM layer is used to exploit the sequential nature along with the statements in the execution paths and the last pooling layer is connected to generate high-level feature representation among all execution paths.

network and programming language specific feature extraction network. Following the standard approaches (Kim 2014), natural language specific feature extraction network is built based on CNN to extract semantic features $\mathbf{z}^r$ from bug reports.

To extract features from source code, the programming language specific feature extraction network mainly contains two parts. The first part aims to learn semantic feature representation of each statement: CNN is utilized to capture the information locally within each statement while DeepWalk is used to model the relationships among neighboring statements. The second part aims to learn semantic feature vector $\mathbf{z}^c$ to represent the whole source code while considering the structure from the control flow graph, where a multi-instance learning framework is used to model the structural feature. The programming language specific feature extraction component is the key part of CG-CNN, which will be carefully introduced in the following subsection.

Then, language-specific features of bug reports and source code are fused into a unified feature representation via joint feature learning layer, followed by a linear output and prediction layer mapping the unified features to $\mathcal{Y}$ which predicts whether source code C_j is related to bug report r_i.

Programming Language Processing by Control Flow Graph Embedding based on Multi-Instance Decomposition

In this section, we introduce the details of programming language specific feature extraction for source code, where a novel framework is designed based on multi-instance decomposition to extract the structural and functional nature from the control flow graphs of the source code.

There have been some models that focus on learning semantic features from source code for bug localization. However, the structural and functional nature of the programming language has not been well-captured by previous methods, which may cause the loss of semantics. For example, Huo *et al.* proposed NP-CNN (Huo, Li, and Zhou 2016), which focuses on learning semantic feature of source code considering the local information based on CNN. Furthermore, LS-CNN (Huo and Li 2017) is built to exploit the sequential nature of source code by combining CNN and LSTM. However, only modeling local and sequential relationships among statements are not enough, and the source code in programming language contains a more complex structure that involved multi-dimension information, such as "branch" and "loop" structure. Therefore, we aim to show that the rich structural information can be leveraged to improve bug localization. In this paper, we choose Control Flow Graph (CFG), which is a kind of graph to represent source code and carries rich structural and functional information. It is very important to extract semantic features from CFG reflecting the structural and functional nature of the program.

As aforementioned, learning semantic representation of CFG contains two main challenges. First, how to generate the semantic feature of each node of CFG (i.e., each statement of source code). Another challenge is that how to represent the whole code considering the structural and functional information from CFG. CG-CNN designs two specific structures to deal with these challenges. The general

framework of programming language specific network is illustrated in Figure 2. The first part of the network aims to learn the semantics of each statement and deal with the first challenge, where CNN and DeepWalk are used to extract information within and between statements, respectively. The second part of the network aims to learn the representation of the whole code from the control flow graph via multi-instance decomposition, which aims to deal with the second challenge.

Learning feature representation of statements The first part of programming language specific feature extraction layer aims to learn the representation of each statement. Different from previous graph based data, the CFG is directed and contains many execution paths of source code. Each statement of the source code is represented as a node in the CFG. The semantics of each statement is not only relative with its tokens, but also with the neighboring statements in the same execution paths of source code. CG-CNN firstly applies CNN to represent each statement by incorporating the semantic of the tokens, where convolutional filters are applied to slide across tokens to generate semantic features. Inspired by Huo's work (Huo, Li, and Zhou 2016), the filters should be designed to slide within each statement and stop when meet the end of each statement, respecting to the atomicity of statements in semantics explicitly. The semantic features of each statement can be extracted via convolutional operations in this way and the integrity of each statement will be also preserved. Suppose each path contains l statements and each statement contains n tokens. After processing by convolutional and pooling operations, each statement will generate a middle-level feature representation $\mathbf{x}^m$, where m is the number of filters in the CNN layers.

After processed by CNN, CG-CNN then introduces DeepWalk to learn the semantic representation by considering the neighboring statements. DeepWalk uses Skip-Gram (Mikolov et al. 2013a), a widely-used distributed word representation method, into the social network for the first time to learn node representation according to network structure, which can also be used to learn node representation in control flow graph. Since the paths in the control flow graph represent the execution procedure and contains semantic relationships. DeepWalk firstly generates short random walks from the control flow graph. Given a sequence of node $S = \{v_1, v_2, \dots, v_{|S|}\}$ generated by random walks, the node $v \in \{v_{i-t}, \dots, v_{i+t}\} \setminus \{v_i\}$ is regarded as the context of the center node v_i, where t is the window size. Following the idea of Skip-Gram, DeepWalk aims to maximize the average log probability of all node-context pairs in the random walk node sequence S:

$$\frac{1}{|S|} \sum_{i=1}^{|S|} \sum_{-t \leq j \leq t, j \neq 0} \log p(v_{i+j}|v_i) \tag{1}$$

Afterwards, DeepWalk uses Skip-Gram to learn representations of node from sequence generated by random walks, which refers to each statement in the source code. By introducing random walks on control flow graph, DeepWalk is able to learn the semantic representation of each statement considering the relationships among neighboring statements. The semantic feature of each statement are generated during first part.

Extracting features from control flow graph based on multi-instance decomposition The second part of CG-CNN aims to learn semantic representation of the whole code from the control flow graph. How to utilize the information to generate the semantic features of a program from the control flow graph is a big challenge. In this work, we formalize this problem as a multi-instance task: The control flow graph of a program contains multiple paths. Each path in the control flow graph is considered as an instance and the whole program is considered as a bag. If any path is relevant to the bug report and labeled as buggy (positive), the source code file is buggy and relevant with the bug report.

Noticing that the middle-level features are generated from each statement of source code, which maintain inherent sequential nature from the original execution paths. Therefore, LSTM is then concentrated to model the sequential relationships between statements. A pooling layer that involves a mean pooling operation is connected to LSTM, which aims to fuse the outputs $\mathbf{z}_t$ from each time step. We generate the semantic feature $\mathbf{z}^c_{ij}$ of the j-th path of source code C_i by averaging $\mathbf{z}_t$ from each time step: $\mathbf{z}^c_{ij} = (\sum_{t=1}^{T} \pi_{it} \mathbf{z}^c_{it}) / \sum_{t=1}^{T} \pi_{it}$, where $\pi_{it} \in \{1, 0\}$ is the indicator variable, $\pi_{it} = 1$ if statement t of source code C_i is present in time step t, and $\pi_{it} = 0$ otherwise. The last pooling layer is connected to generate the final representation of the source code among all expaths: $\mathbf{z}^c_i = \text{Pooling}(\mathbf{z}^c_{i1}, \mathbf{z}^c_{i2}, \dots, \mathbf{z}^c_{in})$, which are then fed into the joint feature learning layer for further fusion.

After processing from language-specific feature extraction layer, the generated features $\mathbf{z}^r$ from bug reports and $\mathbf{z}^c$ from source code are then fed into joint feature learning layer, where a fully-connected network is employed for learning a unified feature and followed by an output layer mapping to the predictions $\mathcal{Y}$. However, a bug report may be only relevant to one or a few source code, while a large number of source code are irrelevant and this imbalance nature should be considered. Similar to (Huo and Li 2017), some negative instances are randomly dropped in the joint feature learning layer, which can decrease the computational cost and counteract the negative influence.

Specifically, the parameters of natural language specific network for bug reports can be denoted as $\Theta_r = \{\theta^r_1, \theta^r_2, \dots, \theta^r_l\}$ and the parameters of programming language specific network can be denoted as $\Theta_C = \{\theta^C_1, \theta^C_2, \dots, \theta^C_l\}$. Let W denotes the parameters in the joint feature learning layers. Therefore, the loss function implied in CG-CNN is:

$$\mathcal{L}(\Theta_C, \Theta_r, W) = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \ell(\mathbf{z}^r_i, \mathbf{z}^C_j, y_{ij}) + \lambda L_r \tag{2}$$

where

$$\ell(\mathbf{z}^r_i, \mathbf{z}^C_j, y_{ij}) = -(\tilde{y}_{ij} \log \tilde{y}_{ij} + (1 - y_{ij}) \log(1 - \tilde{y}_{ij}))$$

Here, y_{ij} denotes the label of pairs of the bug report r_i and the source code C_j and $\tilde{y}_{ij}$ denotes the prediction. L_r is the

Table 1: Comparisons results in terms of MAP. The best performance of each data set is boldfaced.

Methods	BugLocator	AmaLgam	NP-CNN	LS-CNN	DeepWalk	mLS-CNN	CG-CNN
PDE	0.342 ∘	0.374 ∘	0.452 ∘	0.462 ∘	0.470 ∘	0.467 ∘	**0.487**
PU	0.389 ∘	0.381 ∘	0.396 ∘	**0.408**	0.401 ∘	0.406	0.406
JU	0.433 ∘	0.449 ∘	0.471 ∘	0.495 ∘	0.482 ∘	0.507 ∘	**0.531**
Tomcat	0.430 ∘	0.427 ∘	0.488 ∘	0.481 ∘	0.501	0.489 ∘	**0.510**
Avg.	0.399	0.408	0.452	0.462	0.464	0.467	**0.484**

Table 2: Comparisons results in terms of MRR. The best performance of each data set is boldfaced.

Project	BugLocator	AmaLgam	NP-CNN	LS-CNN	DeepWalk	mLS-CNN	CG-CNN
PDE	0.425 ∘	0.431 ∘	0.542 ∘	0.545 ∘	0.552 ∘	0.549 ∘	**0.566**
PU	0.471 ∘	0.471 ∘	0.496 ∘	0.510	0.502 ∘	0.509	**0.518**
JU	0.527 ∘	0.553 ∘	0.581 ∘	0.581 ∘	0.571 ∘	0.592 ∘	**0.615**
Tomcat	0.480 ∘	0.465 ∘	0.529 ∘	0.536 ∘	0.547 ∘	0.542 ∘	**0.560**
Avg.	0.476	0.480	0.537	0.543	0.543	0.548	**0.565**

regularization term and the parameter λ controls the trade-off between the loss and regularization. All the parameters are learned by minimizing the loss function using stochastic gradient descent (SGD) based method.

Experiments

To evaluate the effectiveness of CG-CNN, we conduct experiments on open source software projects and compare it with several state-of-the-art bug localization models.

Experiment Settings

The datasets used in the experiments are extracted from widely-used open source projects. All the bug reports and software code can be extracted from bug tracking system (Bugzilla) and version control system (Git), which have been widely used in previous studies (Zhou, Zhang, and Lo 2012; Saha et al. 2013; Ye, Bunescu, and Liu 2014; Huo, Li, and Zhou 2016). The first project PDE[1] (Plug-in Development Environment) is a tool to create and deploy features and plug-ins of Eclipse. Another project is *Platform*[2] that contains a set of frameworks and common services which make up Eclipse infrastructures. We use component "UI" and refer as PU in this paper. We also investigate project *JDT*[3] (Java Development Tools), which is an Eclipse project used for plug-ins support and development of any Java applications. We also use source code from the component "UI" as our experiments, which is referred as *JU*. The last Tomcat is a web application server and servlet container[4]. Specifically, as suggested in (Kochhar, Tian, and Lo 2014), some bug reports are *fully localized* are filtered, i.e., developers have already identified all buggy source code files in the bug reports. For such bug reports, bug localization tool is no longer needed.

We consider three evaluation metrics: Top-k, MAP (Mean Average Precision) and MRR (Mean Reciprocal Rank), which has been widely-used in previous studies (Zhou, Zhang, and Lo 2012; Huo, Li, and Zhou 2016; Huo and Li 2017). We compare our proposed model CG-CNN with following baselines:

- BugLocator (Zhou, Zhang, and Lo 2012): a state-of-the-art bug localization method which employs revised Vector Space Model to measure the similarity between reports and locates buggy files related to a given bug report.
- AmaLgam (Wang and Lo 2014): a state-of-the-art bug localization model which combines version history, similar bug reports and structure information for bug localization.
- NP-CNN (Huo, Li, and Zhou 2016): a state-of-the-art CNN-based bug localization model, which employs two different CNNs to learn unified features from source code and bug reports for locating buggy source files.
- LS-CNN (Huo and Li 2017): a state-of-the-art deep bug localization model, which extends NP-CNN by combining CNN and LSTM to enhance the unified features by exploiting both sequential nature of source code.
- DeepWalk (Perozzi, Al-Rfou, and Skiena 2014): a state-of-the-art method for learning latent representations of vertices in a network by treating walks as the equivalent of sentences. DeepWalk has been shown effectiveness in several network classification tasks.
- mLS-CNN: a variant of LS-CNN, which is based on multi-instance learning. mLS-CNN firstly learns representation of each path in CFG of then employs pooling operation to generate the unified feature representation.

[1] http://www.eclipse.org/pde/
[2] http://projects.eclipse.org/projects/eclipse.platform
[3] http://www.eclipse.org/jdt/
[4] http://tomcat.apache.org

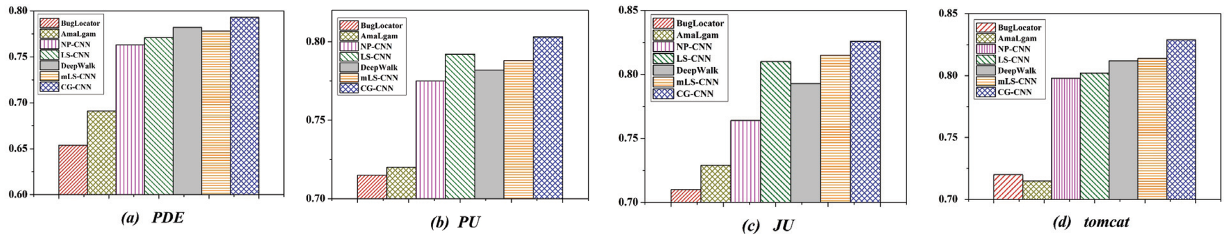

Figure 3: Comparison results of CG-CNN in terms of Top-10.

To compare with previous methods, we follow same parameter settings suggested in their studies (Zhou, Zhang, and Lo 2012; Wang and Lo 2014; Huo, Li, and Zhou 2016). In CG-CNN, pre-trained word embedding model (Mikolov et al. 2013b) is firstly applied to encode each token, which has been shown effective in many text processing tasks (Kim 2014). We then use traditional ReLU as activation function and the window size of convolutional filters is set as 3,4,5 and 100 feature maps is generated each filter. The number of nodes in LSTM is the same as the output of CNN. In addition, the drop-out method (Hinton et al. 2012; Krizhevsky, Sutskever, and Hinton 2012) is applied in the fuly-connected layers to prevent overfitting.

Experimental results

The experimental results are clearly shown in this section. In our experiments, 10-fold cross validation is repeated 10 times for each data set. We also conduct Mann-Whitney *U*-test and Bonferroni correction on each setting, and summarize the results in each result table. If CG-CNN significantly outperforms a compared method, the inferior performance of the compared method would be marked with "∘", and "•" if CG-CNN performs significantly worse.

Table 1 and Table 2 show the average performance of all compared methods in terms of MAP and MRR, respectively. The best performance on each data set is boldfaced. It can be observed from the tables that CG-CNN almost achieves the best performance except that on *PU* in terms of MAP. The reason that CG-CNN does not show significant improvement in *PU* may be that the source code is not complex and control flow graph does not provide further information and semantics. Besides, CG-CNN achieves the best average performance in terms of MAP. For example, the average MAP of CG-CNN is 0.484, which improves the average MAP of BugLocator (0.399) by 21.3%, AmaLgam (0.408) by 18.6%, NP-CNN (0.452) by 7.0%, LS-CNN (0.463) by 4.5% , DeepWalk (0.464) by 4.3%, mLS-CNN (0.467) by 3.5%. It can be observed that CG-CNN performs significantly better than DeepWalk and LS-CNN, which means the multi-instance based setting could be a better solution to extract features from CFG. The performance of CG-CNN is also better than mLS-CNN, which can be regarded that the semantic feature generated from CG-CNN is more representative and contains richer information for bug localization.

Similar trends are also observed in terms of MRR. CG-CNN achieves the best performance among the compared methods and the best average performance. For example, the average MRR of CG-CNN is 0.565, which improves the average MAP of BugLocator (0.476) by 18.7%, AmaLgam (0.480) by 17.7%, NP-CNN (0.537) by 5.2%, LS-CNN (0.543) by 4.0%, DeepWalk (0.543) by 4.0%, mLS-CNN (0.548) by 3.1%. We also conduct Mann-Whitney *U*-test on each setting and summarize the results in Table 1 and Table 2. We can find that among the 24 different comparisons (6 compared methods on 4 data sets) in terms of MAP, CG-CNN performs significantly better on 21 settings. In terms of MRR, CG-CNN performs significantly better on 22 settings and shows improvement against all compared methods.

For better illustration, comparison results in terms of Top-10 are illustrated in Figure 3. It can be noticed that CG-CNN achieves the best performance among the compared methods, which means CG-CNN can to identify the most buggy code when the same number of files are examined.

In summary, the goal of this paper is to show that structural information in the code can be leverage to improve bug localization. The key technical challenge we aim to address is how to model these structural information provided with the textual information in bug reports. We formalize this problem as a multi-instance learning problem and design a specific model CG-CNN. The results on wide-used data sets shows that, CG-CNN performs significantly better than state-of-the-art bug localization and graph embedding methods, which demonstrates the effectiveness in extracting semantic features improve bug localization.

Conclusion

In this paper, we are the first using multi-instance decomposition to learn semantic features from control flow graph for bug localization. We propose a novel deep model named CG-CNN to learn the unified features by exploiting rich structural information from control flow graph. CG-CNN firstly use CNN and DeepWalk model to learn feature representation of each statement and further extract semantics from control flow graph based on multi-instance decomposition. The results of experiments on large scale of open-source project show that CG-CNN performs significantly better than several state-of-the-art bug localization models, which demonstrates that the control flow graph is able to provide more information and CG-CNN is effective in extracting semantic features of source code.

In addition, we aim to show that structural information can be leveraged for bug localization in this paper and we choose to extract semantic features from control flow graph. More structures such as data flow graph and abstract syntax tree can be tried to improve bug localization in the future.

Acknowledgments This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61921006).

References

Allen, F. E. 1970. Control flow analysis. In *ACM Sigplan Notices*, volume 5, 1–19. ACM.

Gay, G.; Haiduc, S.; Marcus, A.; and Menzies, T. 2009. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th International Conference on Software Maintenance*, 351–360.

Grover, A., and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 855–864. San Francisco, CA, USA: ACM.

Hinton, G. E.; Srivastava, N.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. R. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*.

Huo, X., and Li, M. 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 1909–1915.

Huo, X.; Li, M.; and Zhou, Z.-H. 2016. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 1606–1612.

Kim, Y. 2014. Convolutional neural networks for sentence classification. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*, 1746–1751.

Kochhar, P. S.; Tian, Y.; and Lo, D. 2014. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 803–814.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 1097–1105.

Lam, A. N.; Nguyen, A. T.; Nguyen, H. A.; and Nguyen, T. N. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports. In *Proceedings of the 28th International Conference on Automated software Engineering*, 476–481.

Lukins, S. K.; Kraft, N. A.; and Etzkorn, L. H. 2008. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of 15th Working Conference on Reverse Engineering*, 155–164.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013a. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, 1287–1293.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710. New York, NY, USA: ACM.

Poshyvanyk, D.; Guéhéneuc, Y.-G.; Marcus, A.; Antoniol, G.; and Rajlich, V. C. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 33(6):420–432.

Saha, R. K.; Lease, M.; Khurshid, S.; and Perry, D. E. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering*, 345–355.

Shi, S.-T.; Li, M.; David, L.; Ferdian, T.; and Xuan, H. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the 19th AAAI Conference on Artificial Intelligence*.

Wang, S., and Lo, D. 2014. Version history, similar report, and structure: putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, 53–63.

Wei, H., and Li, M. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, 3034–3040.

White, M.; Vendome, C.; Linares-Vásquez, M.; and Poshyvanyk, D. 2015. Toward deep learning software repositories. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories*.

Ye, X.; Bunescu, R. C.; and Liu, C. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 689–699.

Zhou, J.; Zhang, H.; and Lo, D. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, 14–24.