Knowledge-Based Systems 248 (2022) 108741

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Enhancing bug localization with bug report decomposition and code hierarchical network

Ziye Zhu [a], Hanghang Tong [c], Yu Wang [a], Yun Li [a,b,*]

[a] Jiangsu Key Lab. of Big Data Security and Intelligent Processing, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu, People's Republic of China
[b] State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, People's Republic of China
[c] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA

ARTICLE INFO

Article history:
Received 18 November 2021
Received in revised form 1 April 2022
Accepted 1 April 2022
Available online 27 April 2022

Keywords:
Software mining
Bug localization
Bug report
Program behavior
Hierarchical network
Network of networks

ABSTRACT

Bug localization, which aims to locate buggy source code files for given bug reports, is a crucial yet challenging software-mining task. Despite remarkable success, the state of the art falls short in handling (1) bug reports with diverse characteristics and (2) programs with wildly different behaviors. In response, this paper proposes a graph-based neural model BLoco for automated bug localization. To be specific, our proposed model decomposes bug reports into several bug clues to capture bug-related information from various perspectives for highly diverse bug reports. To understand the program in depth, we first design a code hierarchical network structure, Code-NoN, based on basic blocks to represent source code files. Correspondingly, a multilayer graph neural network is tailored to capture program behaviors from the Code-NoN structure of each source code file. Finally, BLoco further incorporates a bi-affine classifier to comprehensively predict the relationship between the bug reports and source files. Extensive experiments on five large-scale real-world projects demonstrate that the proposed model significantly outperforms existing techniques.

1. Introduction

Bug localization aims to locate the corresponding buggy source code files for given bug reports, and is crucial for ensuring software trustworthiness [1,2]. In practice, the effectiveness of manual bug localization depends on the experience of programmers and their understanding. Thus, bug localization is a time-consuming and labor-intensive task, particularly for complex, large-scale software systems [3]. To alleviate the burden on programmers and accelerate the development process, automated bug localization has drawn widespread attention as a software-mining task in the past decade.

A simplified bug report and related buggy source files are shown in Fig. 1. A bug report is a technical document that contains all necessary information about a software bug [4–7]. Among the automated bug localization approaches, one representative direction is based on information retrieval (IR) technology [1,8–11], which automatically locates buggy source files of given bug reports by considering the textual or semantic similarity between the bug report and source file. However, the IR-based approach suffers from a *lexical mismatch* [12] between the natural language text in bug reports and the programming language terms in source files. With the prevalence of deep neural models, several recent efforts have investigated the potential of deep neural networks (i.e., another representative direction) for automated bug localization. Most existing neural-based approaches [12–16] treat this task as a structured learning problem to learn the relationship patterns between the bug report and corresponding source file. Because the neural-based approach has its own advantages in language understanding (for both the natural language and programming language), we extend prior research and address the bug localization task in this work. Generally, neural-based approaches address this task by answering the following two key questions: (Q1) How to capture bug-related information from the bug report? and (Q2) How to understand the program in depth from the source code file? We summarize recent studies from each perspective and highlight their limitations.

In terms of **Q1**, the main content in the bug report (i.e., the summary and description items) is written in natural language (Fig. 1). Most existing approaches treat each bug report as a complete sequence to extract bug-related information. For example, Huo et al. [16] simply mapped each bug report into a sequence of one-hot vectors and used the Convolutional Neural Network (CNN) [17] to learn a global semantic representation.

* Corresponding author at: Jiangsu Key Lab. of Big Data Security and Intelligent Processing, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu, People's Republic of China.
E-mail address: liyun@njupt.edu.cn (Y. Li).

https://doi.org/10.1016/j.knosys.2022.108741
0950-7051/

Fig. 1. Simplified bug report from the project SWT and two related fixed source files.

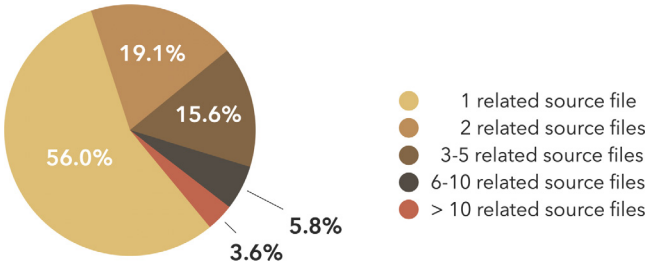

Fig. 2. Statistics on number of source files related to bug report in project JDT. There are 3538 (56.0%) bug reports related to only one source file, 1204 (19.1%) bug reports related to two source files, and 985 (15.6%) bug reports related to three to five source files. In general, numerous bug reports are related to multiple source files, especially for large-scale software projects.

Zhu et al. [18] further applied the attention mechanism [19] to filter the noise in bug reports to capture bug-related information. Unfortunately, these methods ignore the diverse characteristics of the bug report. From the bug report #230118 shown in Fig. 1, we can observe that it provides a wealth of information about a bug from various perspectives, and related to multiple source files. This phenomenon is quite common in real-world scenarios. As shown in Fig. 2, 44% of the bug reports in the project JDT are related to at least two source files. This drawback in the design of previous studies limits their potential to capture richer bug-related information from bug reports with diverse characteristics.

In terms of the more challenging **Q2**, some studies [14,16, 18] consider a source code file as flat sequences and employ neural models to learn the program semantics for bug localization. However, source files that are primarily intended for machines to interpret and execute are actually structured and logical. Thus, current advanced studies [20–22] exploit graphical representations to further manifest the structural information in the program. For instance, Liang et al. [20] and Zhang et al. [21] utilized the Abstract Syntax Tree (AST), a faithful representation of the grammatical structure of the program, rather than flat sequences to represent source files. Alternatively, Huo et al. [22] applied the Control Flow Graph (CFG), a representation that illustrates possible execution paths, to provide additional control flow information. Accordingly, we can first observe that the graph data structure is one of the most appropriate structures for expressing programming languages. In addition, all existing graph-based approaches consider AST and CFG separately, although these two graphical representations are inherently complementary. More importantly, these approaches neglect that many programs exhibit wildly different behaviors. In practice, the execution process of the program is controlled by conditional statements (e.g., if, for, and switch statements), resulting in different execution paths with different program behaviors. Therefore, processing a source file as a unit is inconsistent with the actual execution of the program and cannot achieve deep program understanding.

Before elaborating, we further explore this task and propose a third question: (Q3) How can we model the relationship patterns between bug reports with diverse characteristics and programs with wildly different behaviors? As this question has not been considered in previous research, we need to rethink the manner of relationship prediction, supposing that we are capable of obtaining richer bug-related information and a more complete view of the program.

Armed with these observations, we present a graph-based neural model, **BLoco** (Bug Localization based on bug clue and Code-NoN), for bug localization. We pertinently design distinct processing procedures to capture abundant bug-related information from bug reports and program behaviors from source files, and adequately incorporate this rich knowledge. To be specific, we introduce a decomposition strategy to decompose bug reports into multiple bug clues and apply a CNN-based model to extract bug-related information for each clue. We retain the representations of all bug clues rather than fusing them into one global

representation. We believe that multiple representations can express bug reports more precisely, from different perspectives. To understand the program in depth, we first design a hierarchical network (or graph) structure, **Code-NoN** (a Network of Networks for Code), which integrates the properties of CFG and AST to represent the source file. In this Code-NoN, the main network is the CFG, which illustrates all possible execution paths in the source file. Each node in the main network denotes a basic block, which can be further represented as an AST that characterizes the fine-grained structural information inside. This allows Code-NoN to embrace the advantages of CFG and AST in embodying the program at different granularities. Correspondingly, we customize a multilayer Dense Graph Propagation (DGP) to infer the program behaviors of different execution paths from the hierarchical Code-NoN. Finally, we employ a bi-affine classifier to comprehensively predict the relationship between bug reports and source files by further combining all bug clue representations and program behavior representations. Extensive experiments on five real-world open-source projects demonstrate that BLoco achieves the state-of-the-art performance on all evaluation measures. The main contributions of this work are presented as follows,

- We present a graph-based neural model (BLoco) to handle complex, large-scale software systems for automated bug localization.
- To the best of our knowledge, Code-NoN is the first hierarchical network structure for bug localization that can represent programs with wildly different behaviors in a more complete view.
- Considering the diversity of bug reports, we introduce a decomposition strategy to produce multiple bug clues from each bug report, with advantages for bug-related information extraction.

The remainder of this paper is organized as follows. Section 2 introduces the problem statement and Section 3 presents the proposed model. Section 4 presents and analyzes the experimental results. Section 5 identifies the main threats to validity. Section 6 briefly reviews related work and Section 7 presents the conclusions.

2. Problem statement

First, we introduce notations and concepts that are used throughout. We use calligraphic letters for sets (e.g., $\mathcal{B}$, $\mathcal{S}$), capital letters to denote matrices (e.g., Y), and bold lower case for vectors (e.g., $\boldsymbol{r}$). For a project, we denote $\mathcal{B} = \{b_1, b_2, \ldots, b_{N^b}\}$ as the set of bug reports and $\mathcal{S} = \{s_1, s_2, \ldots, s_{N^s}\}$ as the set of source code files, where N^b and N^s are the number of the bug reports and source files in the project, respectively. In addition, an indicator matrix $Y \in \{0,1\}^{N^b \times N^s}$ is used to indicate the relationship (i.e., buggy or clean) between all pairs of bug reports and source files. That is, $Y(i,j) = 1$ indicates that source code file s_j is buggy for bug report b_i, whereas $Y(i,j) = 0$ indicates clean source code file. Using these notations, the bug localization problem can be formally defined as follows.

Problem 1. Bug Localization Problem

Given: (1) a collection of bug reports $\mathcal{B}$ containing N^b bug reports, (2) a collection of source files $\mathcal{S}$ containing N^s source files, (3) an indicator matrix $Y \in \{0,1\}^{N^b \times N^s}$ that indicates the relationship of all (b_i, s_j) pairs, (4) a new bug report in the current project $b_{new} \notin \mathcal{B}$;

Find: the buggy source file $\tilde{s}$ associated with b_{new}, where $\tilde{s} \in \mathcal{S}$.

We instantiate the bug localization problem as a structured learning problem to model the relationship patterns between bug reports and their corresponding buggy source code files. In the training stage, our model aims to learn a relationship prediction function $F : \mathcal{B} \times \mathcal{S} \rightarrow \mathcal{Y}$ to evaluate the overall relevancy of all pairs of bug reports and source files, where $\mathcal{Y} = \{0, 1\}$. After the model is fully trained, the relationship prediction function $F(\cdot, \cdot)$ learned in the training stage is used to locate buggy source files during testing. The buggy source file $\tilde{s}$ related to the new bug report b_{new} is the source file with the maximum overall relevancy to b_{new}, calculated as $\tilde{s} = \arg\max_{s \in \mathcal{S}} F(b_{new}, s)$.

3. The proposed model

The overall architecture of the BLoco model is depicted in Fig. 3, which contains three integral components, including bug report processing, source code processing, and relationship prediction. Considering the inherent differences between natural and programming languages, we design processing modules with distinct structures for bug reports and source files. The implementation details for each component are explained in the following sections.

3.1. Bug report processing module

For highly diverse bug reports, only one global representation cannot adequately express the rich bug-related information contained therein. To address this issue, the bug report processing module is designed to extract several bug clue representations from the bug reports, each of which expresses only one piece of bug-related information. As shown in Fig. 3, the bug report processing module consists of bug report decomposition, an embedding layer, and a bug clue encoder.

3.1.1. Bug report decomposition

To capture various bug-related information, we first decompose each bug report into bug clues instead of treating it as a complete sequence. The bug report decomposition strategy is primarily based on the following two rules: (R1) The summary item in the bug report is used as a bug clue. (R2) Two adjacent short sentences ($\leq$ 10 in length) in the description item are combined into one sentence, and each sentence is defined as a bug clue. As shown in Fig. 3, for a given bug report b, we obtain all bug clues $\{c_1, c_2, \ldots, c_{N^c}\}$ using the bug report decomposition strategy, where N^c is the number of bug clues in the bug report.

3.1.2. Embedding layer

In the embedding layer for bug reports, we use pretrained word embeddings (i.e., Word2Vec [23]) to initialize the words in each bug clue. A bug clue with u words can be represented by a two-dimensional embedding matrix $E \in \mathbb{R}^{u \times d}$, where d denotes the dimension of word vectors. In addition, with the varying lengths of bug clues, we set a padding size K^p to adjust the dimension of the embedding matrix [17]. After this step, we obtain the representation $E \in \mathbb{R}^{K^p \times d}$ for each bug clue. In this way, a bug report with N^c bug clues is expressed as a series embedding matrices $\{E_1, E_2, \ldots, E_{N^c}\}$ that is passed to the subsequent bug clue encoder.

3.1.3. Bug clue encoder

To extract bug-related information from a bug clue, we use a CNN to encode the embedding matrix of each bug clue. CNNs have been widely used to extract semantics from natural languages [17,24]. We apply K^c convolutional filters (with varying window sizes) to slide across words to generate K^c feature maps in the convolutional layer. Subsequently, a max-pooling layer

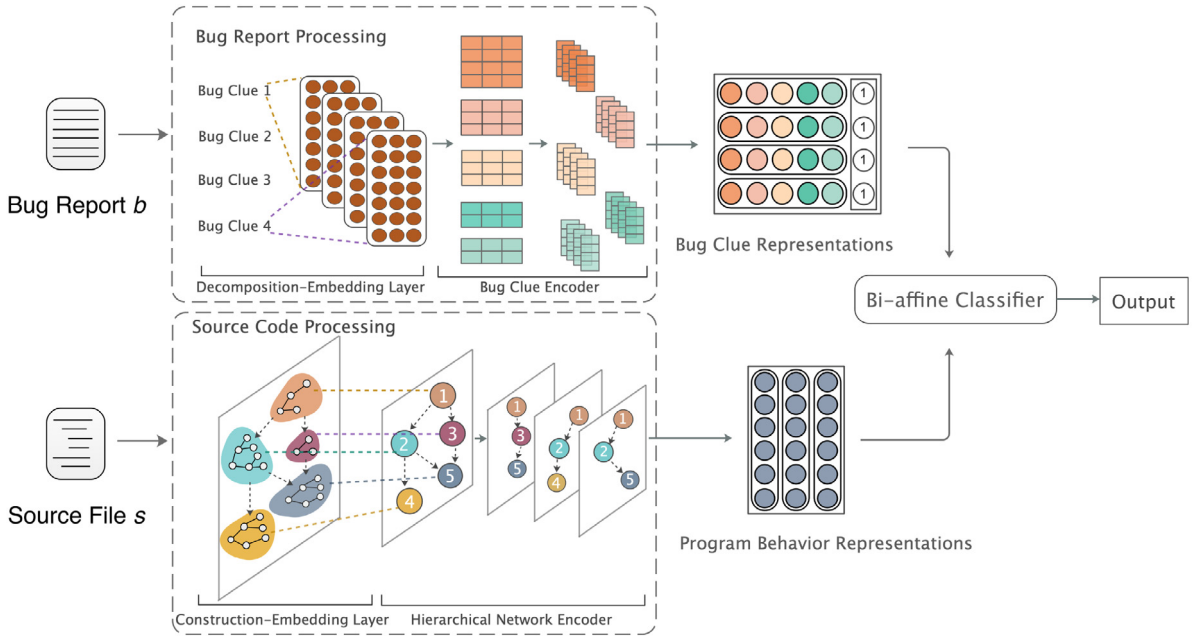

Fig. 3. Overall diagram of BLoco. The bug report processing module extracts bug-related information using a decomposition-embedding layer (i.e., bug report decomposition and embedding layer) and a bug clue encoder. The source code processing module models the program behavior of the execution path using a construction-embedding layer (i.e., Code-NoN construction and embedding layer) and a hierarchical network encoder. The relevance prediction module contains a bi-affine classifier to carefully predict the relationship by exploiting bug clue representations and program behavior representations.

subsamples the output of the convolutional layer by applying a max operation to the result of each filter. With respect to each bug clue, we acquire a d-dimensional bug clue representation using the following equation,

$$\boldsymbol{c}_i = \text{CNNs}(E_i), \forall i = \left\{1, 2, \ldots, N^c\right\}. \tag{1}$$

A bug report is expressed as a series of bug clue representations $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{N^c}\}$ after processing by convolutional and pooling operations.

3.2. Source code processing module

Understanding the functionality of the source code file is the foundation of the bug localization task. In this subsection, we focus on the core of BLoco, the source code processing module responsible for understanding programs with wildly different behaviors. Overall, this core module provides insight into all possible program behaviors from each source file by exploiting its corresponding Code-NoN structure. As illustrated in Fig. 3, the source code processing module incorporates the Code-NoN construction, an embedding layer, and a hierarchical network encoder.

3.2.1. Code-NoN construction

As we expect to learn program behaviors from source files, we utilize a graph data structure to represent the program instead of simply treating a program as a set of sequential statements. Graphical representations (e.g., Parse Trees, AST, and CFG) differ in their level of abstraction, in the relationship between the graph and the program, and in the structure of the graph. In this work, we take one step further and design a hierarchical network (or graph) for code, Code-NoN, which can integrate the complementary knowledge provided by the CFG and AST structures.

Essentially, our proposed Code-NoN is a Network of Networks (NoN) for representing code. In a NoN model [25,26], each node of the main network can be represented as another network. We elaborate on the Code-NoN with a code snippet example, as shown in Fig. 4(a). For the corresponding Code-NoN structure shown in Fig. 4(b), the main network is represented by dashed nodes and edges and the block-specific networks are represented by solid nodes and edges. The main network consisting of four nodes is the CFG structure of the code snippet example, where each node denotes a basic block (i.e., a sequence of statements with no branches into or out of the block). The directed edges between blocks correspond to possible transfers of control from one block to another. Each node in the main network corresponds to a block-specific network, which is the AST structure of the block. Each node in the block-specific network denotes a component (i.e., code token) occurring in the block, and the edge denotes the abstract relationships between the components. In this work, we refer to nodes in the main network (i.e., CFG network) as block nodes, and nodes in block-specific networks (i.e., AST networks) as token nodes. In a nutshell, the Code-NoN structure shown in Fig. 4(b) is a (CFG) network of (AST) networks.

Based on the above explanation, a Code-NoN can be formally expressed as follows. Given a source file s, its corresponding Code-NoN structure is defined as $\mathcal{N} = \langle G, \mathcal{T} \rangle$, where $G \in \{0, 1\}^{N^t \times N^t}$ is the adjacency matrix of the main network and $\mathcal{T} = \{t_1, t_2, \ldots, t_{N^t}\}$ denotes a set of main nodes. The main network is the CFG structure of the source file s containing N^t basic blocks. The non-zero element $G(i, j)$ denotes a possible transfer of control from block node t_i to block node t_j, whereas the zero element denotes no transfer of control from block node t_i to block node t_j. We set up a unique entry node and a unique exit node corresponding to the procedure entry point and exit point, respectively. Furthermore, each block node (except the entry and exit nodes) corresponds to a block-specific network, which is the AST structure for the block. For simplicity, each block-specific network is further represented as $t = \langle Q, \mathcal{V} \rangle$, where $Q \in \{0, 1\}^{N^v \times N^v}$ is the adjacency matrix of the block-specific network and $\mathcal{V} = \{v_1, v_2, \ldots, v_{N^v}\}$ denotes N^v token nodes. The non-zero element $Q(e, f)$ indicates that token node v_f is a child of token node v_e, whereas the zero element indicates that it is not.

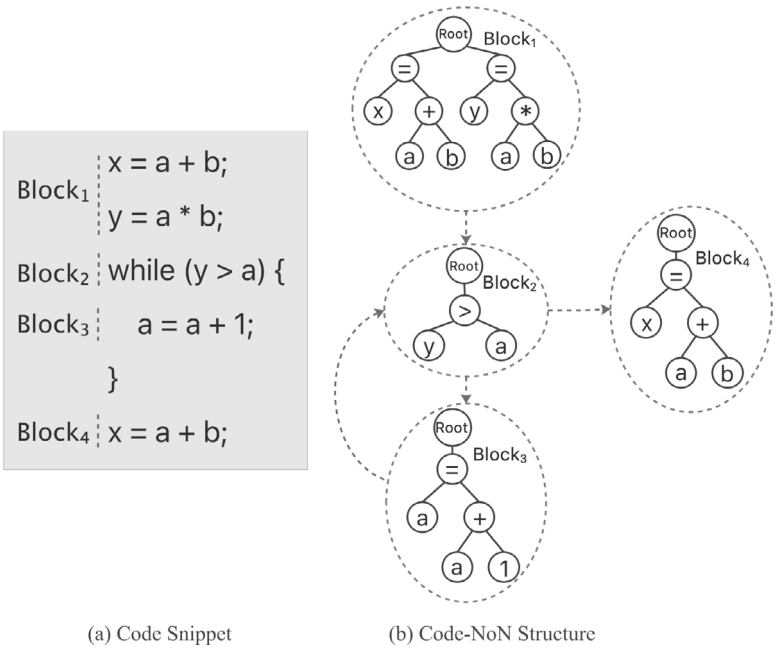

Fig. 4. Code-NoN example. The main (CFG) network is represented by dashed nodes and edges. The block-specific (AST) networks are represented by solid nodes and edges.

3.2.2. Embedding layer

For a given source file, we obtain the network $\mathcal{N} = \langle G, \mathcal{T} \rangle$ after Code-NoN construction. We assign node features to each token node in this graph using the embedding layer in the source code processing module. Owing to the particularity of software projects, directly using pre-trained Word2Vec embeddings [23] for tokens results in many unknown words. We notice that in most well-managed projects, developers generally follow CamelCase naming convention [27] in naming variables, methods, and classes. We split these compound tokens into separate real words based on capital letters. We randomly initialize the remaining unknown tokens and fine-tune their embeddings during the training. In terms of a block-specific network $t = \langle Q, \mathcal{V} \rangle$, we generate a node embedding matrix $X \in \mathbb{R}^{N^v \times d}$. For simplicity, we denote the block-specific network with node features as $t = \langle Q, X \rangle$.

3.2.3. Hierarchical network encoder

From the Code-NoN structure shown in Fig. 4(b), it is evident that the CFG network has multiple execution paths over the complete execution of the program. To understand the program behavior of each execution path, we customize a hierarchical network encoder for the constructed Code-NoN. To be specific, the hierarchical network encoder utilizes DGP [28], a variant of the Graph Convolutional Network (GCN) [29,30], to effectively capture directional information between nodes by treating the information propagation process from the parent classes and child classes differently. As shown in Fig. 3, the hierarchical network encoder employs DGPs to aggregate structural information and generate multi-grained features of the program. Considering the (main) network of (block-specific) networks structure of Code-NoN, we introduce the hierarchical network encoder by the following two main steps.

(S1) Block-specific network encoder. In a block-specific network $t = \langle Q, X \rangle$, DGPs compute the node vectors in the block based on the properties of their neighborhoods. From the adjacency matrix Q, we use two separate adjacency matrices: Q_a denotes the connections from nodes to their ancestors, and Q_d denotes the connections from nodes to their descendants. Note that both these adjacency matrices include self-loops. The learning of the DGP consecutively considers both directions, using the following message-passing architecture,

$$H^{(l+1)} = \sigma(D_a^{-1} Q_a \sigma(D_d^{-1} Q_d H^{(l)} W_d^{(l)}) W_a^{(l)}), \tag{2}$$

where $H^{(l+1)}$ and $W^{(l)}$ are the hidden node representations and the trainable weight matrix in the lth layer respectively. For the first layer, $H^{(0)} = X$. $D_\alpha(e,e) = \sum_f Q_\alpha(e,f)$ is a degree function, where $\alpha \in \{a, d\}$. $\sigma(\cdot)$ denotes a nonlinear activation function. A full DGP module runs K iterations of Eq. (2) to generate the output node vector matrix $Z = H^{(K)}$.

For simplicity, we abstract away the internal structure of the DGPs and denote a DGP module running K iterations of message passing for a block-specific network $t = \langle Q, X \rangle$ using

$$Z = \mathrm{DGPs}(Q, X). \tag{3}$$

Note that the output token node vectors Z are generated by aggregating all token node information in this block-specific network, which maintains the inherent structure (i.e., child–parent relations) of the AST network. In addition, a pooling layer that involves a mean pooling operation is connected to the DGP, which aims to fuse the token node vectors Z into an abstract block representation $\boldsymbol{t}$ for block-specific network t. For a Code-NoN with several block-specific networks $\{t_1, t_2, \ldots, t_{N^t}\}$, we perform the learning of DGP in parallel, yielding a set of block representations $\{\boldsymbol{t}_1, \boldsymbol{t}_2, \ldots, \boldsymbol{t}_{N^t}\}$.

(S2) Main network encoder. To further understand the program behavior for each execution path in the main network, the block representations $\{\boldsymbol{t}_1, \boldsymbol{t}_2, \ldots \boldsymbol{t}_{N^t}\}$ obtained in S1 are stacked in a block representation matrix T and then input to the DGP module. DGPs update the block representations based on the adjacency matrix G of the main network as follow,

$$M = \mathrm{DGPs}(G, T). \tag{4}$$

After the iterations of the message passing on the main network, the new block representation matrix M preserves the control transfer knowledge provided by the CFG network. We also apply a mean pooling layer to fuse the block representations that belong to the same execution path, resulting in a d-dimensional

program behavior representation $\boldsymbol{p}$ for that execution path. With respect to a program containing N^p possible execution paths, the main network encoder generates a set of program behavior representations $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots \boldsymbol{p}_{N^p}\}$.

3.3. Relationship prediction module

At this point, we obtain a set of bug clue representations $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_{N^c}\}$ and a set of program behavior representations $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots \boldsymbol{p}_{N^p}\}$ for a pair of bug report and source file (b, s). We construct clue-behavior representation pairs $(\boldsymbol{c}_i, \boldsymbol{p}_j)$ through pairwise combination, where $\boldsymbol{c}_i$ and $\boldsymbol{p}_j$ denote the ith bug clue representation of bug report b and jth program behavior representation of source file s, respectively. All clue-behavior representation pairs are finally fed into a bi-affine classifier [31] to estimate the relevancy of each pair. The clue-behavior relevancy is predicted as follow,

$$r_{ij} = (\boldsymbol{p}_j)^\top W_1 \boldsymbol{c}_i + (\boldsymbol{p}_j \oplus \boldsymbol{c}_i)^\top W_2 + b, \tag{5}$$

where $\oplus$ denotes concatenation operation, W_1 and W_2 denote weight matrices, and b denotes bias. For all clue-behavior pairs in a (b, s), we will obtain the clue-behavior relevancy matrix $R = (r_{ij}) \in \mathbb{R}^{N^c \times N^p}$. If the program behavior $\boldsymbol{p}_j$ is highly related to the bug clue $\boldsymbol{c}_i$, their corresponding source file and bug report also tend to be related. To predict the overall relationship of the (b, s) pair, we further convert the clue-behavior relevancy matrix R to the overall relevancy $\tilde{r}$ by summing the maximum clue-behavior relevancy of each bug clue. The final objective function of the proposed model is defined as follows,

$$\mathcal{L}(\Theta) = \sum_{i=1}^{N^b} \sum_{j=1}^{N^s} \ell(b_i, s_j, y_{ij}) + \lambda_\Theta \|\Theta\|^2, \tag{6}$$

$$\ell = -(y_{ij} \log(\sigma(\tilde{r}_{ij})) + (1 - y_{ij}) \log(1 - \sigma(\tilde{r}_{ij}))), \tag{7}$$

where Θ denotes all parameters of the model and λ_Θ is the regularization hyperparameter; y_{ij} denotes the label of the pair (b_i, s_j); $\sigma(\tilde{r}_{ij})$ denotes the prediction of the pair (b_i, s_j). In this way, we adapt Adaptive Moment Estimation (Adam) method [32] to directly minimize the loss function $\mathcal{L}(\Theta)$ in our model.

4. Experimental results and analysis

4.1. Dataset

The experimental dataset used in this work is public, and is commonly used in bug localization studies [12,21]. The dataset is created from open-source projects by using Bugzilla as the issue tracking system and GIT as a version control system (earlier versions are transferred from CVS/SVN to GIT). Each project in this dataset includes the bug reports, source code links, buggy files, API documentation, and the oracle of bug-to-file mappings. A brief introduction to the four real-world projects:

- AspectJ[1]: an aspect-oriented extension for the Java programming language.
- Platform UI[2]: a set of frameworks and common services for Eclipse.
- JDT[3]: a set of plug-ins supporting the Java application development.
- Tomcat[4]: an open-source implementation of the Java Servlet and JavaServer Pages technologies.
- SWT[5]: a graphical Eclipse-based widget toolkit for use with the Java platform.

4.2. Metrics

We evaluate the bug localization performance using two evaluation metrics, Top-K and Mean Average Precision (MAP), both of which are broadly adopted in the bug localization task [1,9,12,33,18].

Top-K. Top K is widely used for the evaluation of information retrieval. For a given bug report, if one of the buggy source files related to the bug report is in the top K of the file list, it is counted as a successful localization. Otherwise, we consider it to be a miss. Top-K is measured by the percentage of success for the total number of recommendations in all folds. Following prior studies, we consider three values of K (1, 5, and 10) in our experiments.

Mean Average Precision (MAP). To consider the cases where a given bug report is related to multiple source files, we use MAP to provide a single-figure measure of the quality of information retrieval, as shown in Eq. (8):

$$MAP = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N^s} \frac{Prec(j) * t(j)}{n_i}, \quad \text{where} \quad Prec(j) = \frac{Q(j)}{j}, \tag{8}$$

where N and N^s denote the number of tested bug reports and source files, respectively; n_i is the number of buggy files to bug report b_i; $t(j)$ is the vector that indicates whether the source file in the position j is the buggy file or not. $Prec(j)$ is the precision at the given cut-off j, and $Q(j)$ is the number of buggy source files in top j positions.

4.3. Comparison methods

We compare our proposed model with the following state-of-the-art bug localization methods,

- BugLocator [1]: a well-cited IR-based method that considers information about similar bug reports that have been fixed.
- DNNLOC [12]: a neural model combining IR technology [1] and metadata of the bug-fixing history.
- LS-CNN [16]: a neural model employing a CNN for bug report representation and a combination of CNN and LSTM for source file representation.
- KGBugLocator [21]: a code knowledge graph-based model with rich semantic information of the source code.
- CG-CNN [22]: a CFG-based method that captures additional structural and functional information to represent program semantics.
- DeMoB [18]: a deep multimodal model that exploits the multi-grained nature of programming language and reduces noise in bug reports.

BugLocator is an IR-based method; the others are based on deep neural networks. In particular, KGBugLocator utilizes an additional graph structure from AST. CG-CNN exploits control flow information from CFG (built with single statements). DeMoB splits the source file to learn multi-grained structural features and filters noise in learning bug report representations. In this work, we randomly divide the bug reports into ten folds for each project, where each fold has an equal size. The bug reports of a single fold are retained to test the model, and the remaining nine folds are used for training with a 10-fold cross-validation

1 https://www.eclipse.org/aspectj/
2 https://www.eclipse.org/eclipse/platform-ui/
3 https://www.eclipse.org/jdt/
4 http://tomcat.apache.org/
5 https://www.eclipse.org/swt/

Table 1
Effectiveness results of all approaches.

Project	Model	Top-1	Top-5	Top-10	MAP
AspectJ	BugLocator	0.202	0.487	0.576	0.224
	DNNLOC	0.431	0.694	0.803	0.295
	LS-CNN	0.468	0.745	0.831	0.440
	KGBugLocator	0.511	0.771	0.828	0.426
	CG-CNN	0.537	0.790	0.856	0.484
	DeMoB	0.523	0.788	0.861	0.486
	BLoco	**0.542**	**0.796**	**0.867**	**0.491**
Platform UI	BugLocator	0.261	0.449	0.575	0.302
	DNNLOC	0.446	0.664	0.801	0.416
	LS-CNN	0.458	0.701	0.803	0.459
	KGBugLocator	0.472	0.713	0.819	0.466
	CG-CNN	0.503	0.734	0.846	0.487
	DeMoB	0.496	0.722	0.841	0.484
	BLoco	**0.507**	**0.736**	**0.866**	**0.497**
JDT	BugLocator	0.197	0.413	0.512	0.233
	DNNLOC	0.401	0.648	0.737	0.336
	LS-CNN	0.455	0.706	0.776	0.415
	KGBugLocator	0.439	0.692	0.759	0.400
	CG-CNN	0.479	0.723	0.802	0.441
	DeMoB	0.482	0.738	0.804	0.448
	BLoco	**0.484**	**0.746**	**0.827**	**0.464**
Tomcat	BugLocator	0.362	0.624	0.717	0.433
	DNNLOC	0.536	0.727	0.803	0.520
	LS-CNN	0.542	0.734	0.820	0.551
	KGBugLocator	0.514	0.753	0.828	0.561
	CG-CNN	0.548	0.746	0.841	0.573
	DeMoB	0.535	0.740	0.827	0.569
	BLoco	**0.553**	**0.768**	**0.855**	**0.582**
SWT	BugLocator	0.195	0.379	0.521	0.257
	DNNLOC	0.345	0.675	0.794	0.365
	LS-CNN	0.377	0.712	0.824	0.427
	KGBugLocator	0.383	0.716	0.832	0.432
	CG-CNN	0.404	0.735	0.844	0.451
	DeMoB	**0.452**	**0.763**	0.842	**0.478**
	BLoco	0.414	0.751	**0.853**	0.466

strategy. The cross-validation process is then repeated 10 times, with each of the 10 folds used exactly once as the validation data for testing. The 10 results from 10 repetitions are then averaged to produce a single estimation. The advantage of this strategy is that all observations are used for both training and testing, and each observation is used for testing exactly once. We apply the traditional word embedding technique Word2Vec [23] to initialize words in bug reports and tokens in source files, following the work of KGBugLocator [21] and CG-CNN [22]. That is, we do not use any deep contextual word embeddings (e.g., ELMo and BERT) in any of the experiments. We sort the source files based on the relevance degree of the tested bug reports and used the ranking list as the output. We implement our model using PyTorch and perform our experiments on NVIDIA 1080ti GPU and Intel i7-8700K CPU.

4.4. Main results

The experimental results compared with state-of-the-art bug localization approaches are shown in Table 1; the best performance in terms of each metric on each project is highlighted in bold. From the experimental results, we observe that BLoco consistently achieves the best performance on all five projects with respect to the Top-10 accuracy. BLoco surpasses the current state-of-the-art models (e.g., CG-CNN and DeMoB) by 0.6% on AspectJ, 2.0% on Platform UI, 2.3% on JDT, 1.8% on Tomcat, 0.9% on SWT. Similar results can be found from the Top-1 and Top-5 results. Note that BLoco is the only model that considers multiple representations of both the bug reports and source files. BLoco should present superiority when dealing with complex bug reports (i.e., one bug report related to multiple source files). The results support that exploring multiple representations from both bug reports and source files in the mainstream projects is worthwhile.

BLoco still performs competitively in terms of MAP on all projects, although the model DeMoB (i.e., the best competitor) is slightly better than ours on project SWT. For example, BLoco yields 58.2% of MAP on Tomcat project, which surpasses DeMoB by 1.3%. On the Platform UI project, the localization performance of our model is still optimal; i.e., 49.7% accuracy in MAP. It is worth mentioning that Top-K only considers whether there is a buggy file within the top K list, whereas MAP considers whether all of the relevant source files tend to be ranked highly. However, when source files related to the same bug report are significant different, it is difficult to assign high rankings to all of the files. Furthermore, most existing methods focus on exploring appropriate source code processing methods. Apart from the early IR-based methods, only a few methods carefully analyze and process bug reports. In our opinion, the bug report is one of the key components in bug localization; effective bug report processing methods will significantly improve the bug localization task.

We especially compare our BLoco with graph-based methods and find that it achieves significant improvements in all evaluation metrics and projects. To be specific, in comparison with KGBugLocator (AST-based), BLoco achieves 2.1–6.8% improvement in Top-10 accuracy. In Top-5 accuracy, the improvement is ranges 1.5% to 5.4%. Also, BLoco outperforms CG-CNN (CFG-based) by 0.9–2.3% in terms of MAP. These results indicate that the Code-NoN structure leveraged in BLoco can embrace the advantages of both the CFG and AST in embodying the inherent structure of the program at different granularities.

4.5. Research questions and analysis

In this section, our experimental goal is to elucidate the main components of the proposed BLoco. The research questions are as follows,

RQ1: Is the bug report decomposition strategy beneficial for improving the localization performance?
RQ2: How do different code network structures affect the performance of BLoco?
RQ3: Which relationship prediction manner is best for our proposed model?

4.5.1. RQ1: Is the bug report decomposition strategy beneficial for improving the localization performance?

In this work, we apply the bug report decomposition strategy in processing bug reports to capture rich bug-related information, as introduced in Section 3.1.1. We further experimentally explore the effectiveness of the bug report decomposition strategy. In this group of experiments, we examine the bug localization performance based on different bug report processing methods, including (1) without decomposition, (2) simplified decomposition strategy, and (3) decomposition strategy. Bug report processing without the decomposition step directly learns a global representation by treating the summary and description items in the bug report as a complete sequence. The simplified decomposition strategy denotes that we only decompose a bug report into bug clues according to sentences and learn the representation of each bug clue. The third is the decomposition strategy used in BLoco model. Compared with the simplified decomposition strategy, the decomposition strategy additionally combines two adjacent short sentences into one sentence to alleviate excessive zero padding. In this group of experiments, the resulting representation carrying the information in the bug report is fed into a subsequent bi-affine classifier; all other components in BLoco are fixed.

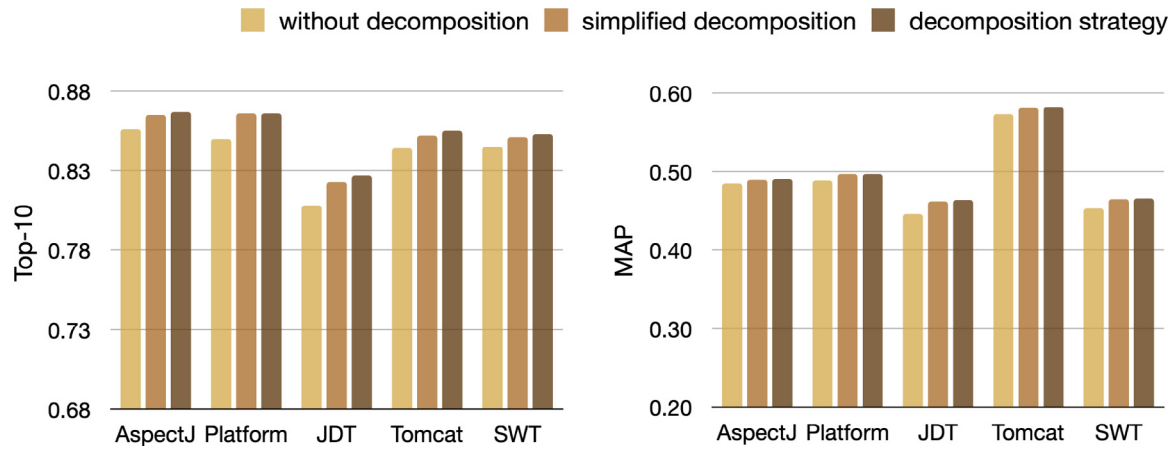

Fig. 5. Performance comparison of bug report processing with different methods, including (1) without decomposition, (2) simplified decomposition strategy, and (3) decomposition strategy.

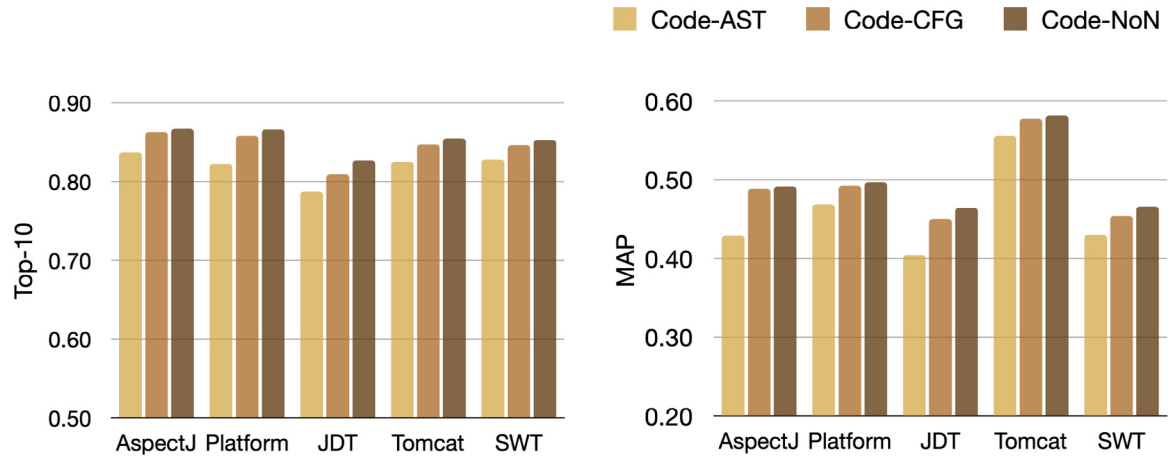

Fig. 6. Performance comparison of different code network structures, including Code-AST, Code-CFG, and Code-NoN.

The corresponding experimental results are shown in Fig. 5. We can first observe that using the decomposition strategy in bug report processing achieves the best performance on all projects in terms of both Top-10 and MAP. For example, on the project Platform UI, the decomposition strategy obtains a 1.6% additional performance improvement in Top-10 accuracy compared to learning without decomposition. We further find that both the decomposition strategy and simplified decomposition strategy have certain advantages over the method without decomposition. These results corroborate that introducing bug clues to handle highly diverse bug reports is indeed necessary for bug-related information extraction. Furthermore, the performance difference between the simplified decomposition strategy and decomposition strategy is relatively small. The main reason is that cases in which two adjacent sentences need to be merged are not common in this project. In future work, we intend to explore different decomposition strategies for processing bug reports.

4.5.2. RQ2: How do different code network structures affect the performance of BLoco*?*

In this work, we design a hierarchical network structure (Code-NoN) to represent source files, which naturally integrates the properties of CFG and AST. To explore the performance impact of different code network structures on BLoco, we present two simplifications of Code-NoN for comparison: (1) Code-CFG, and (2) Code-AST. In Code-CFG, each code file is represented simply by its corresponding CFG structure. We replace the block-specific network encoder with a CNN-based encoder and simply treat each statement as a sequence. With Code-CFG, we can still obtain multiple program behavior representations after source file processing. In Code-AST, each code file is directly expressed by its corresponding AST structure. Considering the structure of Code-AST, we remove the hierarchical network encoder in BLoco, and only use a DGP module to capture fine-grained syntactic information in the AST network. In this group of experiments, the obtained source file representation is fed into the following bi-affine classifier; all other components in BLoco are fixed.

From the experimental results shown in Fig. 6, we can observe the following. First, Code-NoN surpasses the best competitor, Code-CFG, on five projects by 0.3–1.8% and 0.3–1.4% with respect to Top-10 and MAP, respectively. Similarly, Code-NoN generally performs better than Code-AST. This validates that hierarchical Code-NoN integrating CFG and AST better expresses the inherent structure of programs than using CFG or AST independently. Second, both Code-NoN and Code-CFG surpass Code-AST on all projects and both Top-10 and MAP by a large margin. It is worth noting that the Code-AST method can only provide a global source file representation, whereas Code-NoN and Code-CFG can learn multiple program behavior representations. These results reveal that assessing each program behavior separately is a feasible for understanding the program in depth, especially for programs with wildly different behaviors.

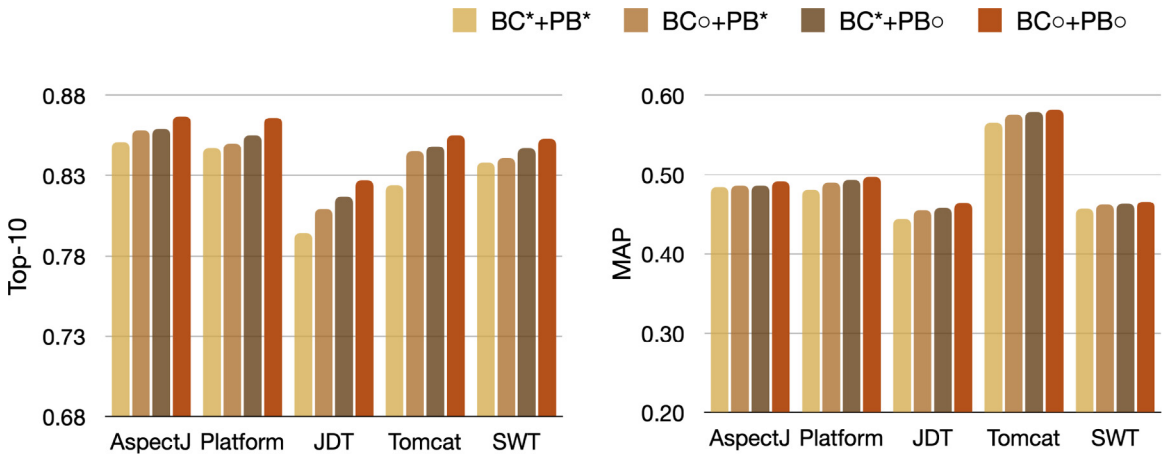

Fig. 7. Performance comparison of multi-representation strategy (∘) and global-representation strategy (∗) for bug clues (BC) and program behaviors (PB).

4.5.3. RQ3: Which relationship prediction manner is best for our proposed model?

In the final relationship prediction stage, we need to predict the relationship between a bug report and a source file based on a set of bug clue representations and a set of program behavior representations extracted from them. Therefore, we perform a series of comparative experiments to determine a relationship prediction manner that is most suitable for BLoco from the following two perspectives.

(1) Necessity of the multi-representation strategy. Our approach retains multiple representations for both bug reports and source files. Thus, we conduct this group of comparison experiments to further investigate the necessity of the multi-representation strategy in relationship prediction. We separately fuse the bug clue representations and program behavior representations to a global one (i.e., global-representation strategy) via an average pooling operation to analyze their contributions. For convenience, 'BC' is short for bug clue; 'PB' is short for program behavior; '∘' indicates multi-representation strategy; '∗' indicates global-representation strategy. BC$^{\circ}$ + PB*, for instance, indicates using multiple bug clue representations in bug report processing, while using a global program behavior representation in source file processing. The experimental results for different strategy combinations are shown in Fig. 7, including BC*+PB*, BC$^{\circ}$+PB*, BC*+PB$^{\circ}$, and BC$^{\circ}$+PB$^{\circ}$. The representations obtained in these cases are all fed into the subsequent bi-affine classifier. From the experimental results, we first observe that BC$^{\circ}$+PB$^{\circ}$ surpasses BC$^{\circ}$+PB* on all projects and both evaluation metrics. These results demonstrate that the global source file representation is inferior to the multiple behavior representations in terms of capturing the wildly different behaviors of the source file. In addition, BC$^{\circ}$+PB$^{\circ}$ obtains 0.7–1.1% improvements over BC*+PB$^{\circ}$ in Top-10. Even if each bug clue is fully considered, using a global representation in the relationship prediction stage will not significantly improve the bug localization performance. Furthermore, these experimental results again illustrate the fact that, despite the varying source file sizes, only a small fraction of source files (i.e., code blocks) might be relevant to some content in the bug report.

(2) Impact of classifier structure. In this group of experiments, we use different methods instead of the bi-affine classifier in the relationship prediction module to investigate the impact of classifier structure on BLoco. We intend to measure the semantic similarity or relatedness of each pair of bug clues and program behaviors. Thus, we naturally choose 'cosine similarity', which is widely used in many sequence matching problems. Another natural choice is a fully connected layer that consists of a linear transformation followed by a non-linear activation function, denoted as 'FC'. For these methods, the subsequent prediction method for each pair of bug reports and source files is identical to that used in BLoco. From the experimental results presented in Fig. 8, we can observe the following. Overall, the bi-affine classifier achieves the best performance on all five projects in terms of both metrics. The comparison method FC is superior to cosine similarity by a large margin. One reasonable explanation is that the final representation spaces of both bug reports and source files are quite different; directly predicting their relationship cannot perform well. In addition, the bi-affine classifier achieves 1.5–3.5% and 1.1–3.3% relative improvements over the FC structure in terms of Top-10 and MAP, respectively. The main advantage over the FC structure is that the tensor in the bi-affine classifier can directly relate input representations. Based on the experimental results, we retain multiple representations of both bug reports and source files in the design of the relationship prediction module, and exploit all bug clues and program behaviors through a deep bi-affine classifier to locate buggy files associated with bug reports.

5. Threats to validity

In this section, we discuss potential threats to the validity of the experiments conducted in this study.

5.1. Threats to internal validity

The threats to internal validity in this study involve factors that may affect our experimental results. A limitation of our approach is that we rely on the quality of bug reports and source files. Bug reports provide crucial information for identifying bugs in a project. If a bug report does not provide sufficient information or provides misleading information, the performance of BLoco is adversely affected. In terms of the source file, good programming practice in naming variables, methods, and classes facilitates the bug localization model to learn the functionality of code, whereas meaningless names might potentially affect localization performance. Our investigation observes that in the majority of projects, developers generally adhere to proper naming conventions. On the other hand, both bug reports and source files are converted into vectors based on word embedding techniques. Thus, to some extent, the performance of our proposed approach relies on the capabilities of the word embedding techniques. In addition, the choice of filter size in the bug clue encoder is related to the writing style of the developers. These potential threats will be investigated in future studies.

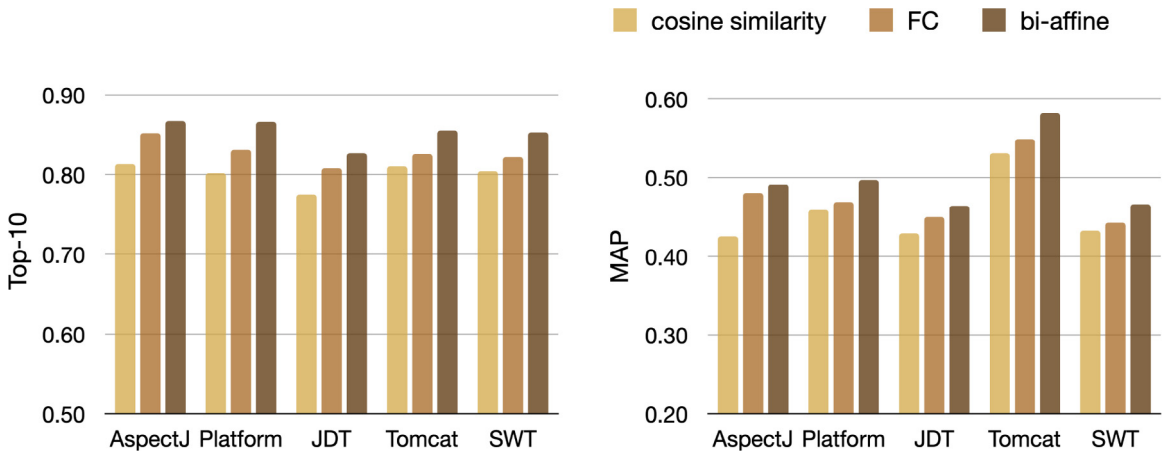

Fig. 8. Performance comparison of relationship prediction module with different methods, including cosine similarity, a fully connected layer (FC), and a bi-affine classifier.

5.2. Threats to external validity

Threats to external validity concern the possibility of generalizing our results. We validated the performance of BLoco on five large-scale real-world software projects that are frequently investigated in bug localization studies. While the five software projects are developed independently and have different writing styles, all of these projects are developed as open-source projects. Because the nature of data in open-source projects may be different from that in closed-source projects, the evaluation results of BLoco on open-source projects might not be generalizable to closed-source projects. Although recent open-source projects have their own quality assurance teams, their support may not be complete compared to commercial projects. However, the relative improvement in BLoco over other bug localization methods is less likely to be affected by these threats. In future work, we intend to improve the model for use in more open- and closed-source projects in actual scenarios. Furthermore, all projects used in our experiments are written in Java. From a language perspective, the principles of the proposed approach can also be applied well to other object-oriented languages. In addition, because the original implementations of the comparison models were not released, we re-implemented our own versions strictly following the procedures described in their work. We tested our implementations and obtained results similar to those of DNNLOC [12] and KGBugLocator [21], which provided their results on the same dataset. We are confident that our implementation reflects the original comparative model.

5.3. Threats to construct validity

Threats to construct validity depend on the dataset used in this study. First, the dataset for the bug localization task is naturally imbalanced because there are far fewer buggy source files than clean ones. We selected a simple sampling strategy to alleviate the class imbalance problem in our experiments, as this problem is not the focus of this study. A better technique for solving the class imbalance should be helpful for bug localization, which should be investigated in a future study. Second, we split each project into a training set (90%) and a testing set (10%), and tested our model using a 10-fold cross validation strategy. This strategy has been employed in most bug localization-related studies [21,22,18]. Although n-fold cross validation is a widely used model validation strategy, different validation strategies may yield different prediction accuracies. Our experimental results based on the 10-fold cross validation may be different from those obtained using other validation techniques. The effects of different splitting strategies on deep learning-based bug localization methods still require further investigation.

6. Related work

To date, IR methods have been widely used in bug localization task by extracting important features from given bug reports and source files. Early approaches [34,35,1,10,36,37,8,9] typically consider surface lexical similarity in correlation estimation, making it impractical to model the relationships between source files written in a programming language and bug reports written in a natural language. For example, Marcus et al. [34] first used the information retrieval method for concept location, in which LSI is used to map concepts expressed in natural language to the relevant parts of the source code. Lukins et al. [35] first found that LDA can be successfully applied for bug localization. Zhou et al. [1] proposed the BugLocator based on the rVSM. Based on these, Wang et al. [10] presented the STMLocator, which uses topic modeling to learn both textual and semantic similarities between bug reports and source files.

Recently, deep learning has become increasingly popular, and has been proven to perform better than traditional machine learning models in the areas of image processing [38,39], speech recognition [40,41], natural language processing [42,43], and graph learning [44,45]. Deep learning has been applied to software data to solve various software engineering problems, such as bug number prediction [46], patch classification [47], and bug prediction [12–16,48,49,33]. Most deep learning-based bug localization approaches consider source files as flat sequences and then use a neural network model to extract the feature from it to be correlated to the bug reports. DNNLOC [12] combines the rVSM [1] with the Deep Neural Network (DNN) for bug localization. NP-CNN [14] is a novel CNN-based model that leverages both lexical and program structure information to learn unified features from natural and programming languages. LS-CNN [16], an improved model of NP-CNN, combines CNN with LSTM networks to enhance the unified features extraction by exploiting the sequential nature of the source code. To manifest deeper semantics behind the textual code, a few advanced approaches that leverage different intermediate representation forms of code have recently been proposed. CAST [20] utilizes customized AST with the TBCNN [50] to enrich the semantic information via program structure information for bug localization. KGBugLocator [21] applies a constructed code knowledge graph based on the AST to represent programming language. The source code

representation learning method, ASTNN [51], demonstrates that AST-based models can better represent source code by capturing the lexical and syntactic knowledge of statements. CG-CNN [22] uses the CFG (built with single statements) to capture additional structural and functional information to represent the semantics of the source code.

7. Conclusions

Bug localization is a crucial task in software engineering. In this study, we propose a novel graph neural model named (BLoco) for this task. We design a hierarchical network structure for code that integrates the properties of CFG and AST to understand the program behavior in depth. We also consider the diversity of bug reports by modeling bug clues to extract bug-related information. In addition, we retain all representations of bug clues and program behaviors, and employ a bi-affine classifier to comprehensively predict the relationship between bug reports and source files. The experimental results demonstrate the effectiveness of the proposed BLoco which achieves better results than existing bug localization methods. Future research explore other efficient graph structures and/or additional software metrics (e.g., static code metrics and object-oriented metrics) to further improve the performance of BLoco.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (No. 61772284), and the State Key Lab. for Novel Software Technology, China (KFKT2020B21). Hanghang Tong is partially supported by NSF, USA (1947135, 2134079, and 1939725).

References

[1] J. Zhou, H. Zhang, D. Lo, 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, 2012, pp. 14–24.
[2] X. Li, H. Jiang, D. Liu, Z. Ren, G. Li, Unsupervised deep bug report summarization, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 144–155.
[3] Q. Wang, C. Parnin, A. Orso, Evaluating the usefulness of IR-based fault localization techniques, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 1–11.
[4] Y. Tian, D. Wijedasa, D. Lo, C. Le Goues, Learning to Rank for Bug Report Assignee Recommendation, in: Proceedings of the 24th International Conference on Program Comprehension, 2016, pp. 1–10.
[5] X. Du, Z. Zheng, G. Xiao, B. Yin, The automatic classification of fault trigger based bug report, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops, 2017, pp. 259–265.
[6] X. Du, Z. Zheng, G. Xiao, Z. Zhou, K.S. Trivedi, DeepSIM: Deep semantic information-based automatic mandelbug classification, IEEE Trans. Reliab. (2021).
[7] G. Xiao, Z. Zheng, B. Yin, K.S. Trivedi, X. Du, K.-Y. Cai, An empirical study of fault triggers in the linux operating system: An evolutionary perspective, IEEE Trans. Reliab. 68 (4) (2019) 1356–1383.
[8] D. Kim, Y. Tao, S. Kim, A. Zeller, Where should we fix this bug? A two-phase recommendation model, IEEE Trans. Softw. Eng. 39 (11) (2013) 1597–1610.
[9] X. Ye, R. Bunescu, C. Liu, 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, 2014, pp. 689–699.
[10] Y. Wang, Y. Yao, H. Tong, X. Huo, M. Li, F. Xu, J. Lu, Bug localization via supervised topic modeling, in: Proceedings of the 2018 IEEE International Conference on Data Mining, 2018, pp. 607–616.
[11] R. Gharibi, A.H. Rasekh, M.H. Sadreddini, S.M. Fakhrahmad, Leveraging textual properties of bug reports to localize relevant source files, Inf. Process. Manage. 54 (6) (2018) 1058–1076.
[12] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Bug localization with combination of deep learning and information retrieval, in: Proceedings of the 25th International Conference on Program Comprehension, 2017, pp. 218–229.
[13] Y. Xiao, J. Keung, Q. Mi, K.E. Bennin, Improving bug localization with an enhanced convolutional neural network, in: Proceedings of the 24th Asia-Pacific Software Engineering Conference, 2017, pp. 338–347.
[14] X. Huo, M. Li, Z.-H. Zhou, Learning unified features from natural and programming languages for locating buggy source code, in: Proceedings of the 25th International Joint Conference on Artificial Intelligence, 2016, pp. 1606–1612.
[15] Y. Xiao, J. Keung, Q. Mi, K.E. Bennin, Bug localization with semantic and structural features using convolutional neural network and cascade forest, in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering, 2018, pp. 101–111.
[16] X. Huo, M. Li, 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, 2017, pp. 1909–1915.
[17] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of Conference on Empirical Methods in Natural Language Processing, 2014, pp. 1746–1751.
[18] Z. Zhu, Y. Li, Y. Wang, Y. Wang, H. Tong, A deep multimodal model for bug localization, Data Min. Knowl. Discov. 35 (4) (2021) 1369–1392..
[19] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, in: Proceedings of the 3rd International Conference on Learning Representations, 2015, pp. 1–15.
[20] H. Liang, L. Sun, M. Wang, Y. Yang, Deep learning with customized abstract syntax tree for bug localization, IEEE Access 7 (2019) 116309–116320.
[21] J. Zhang, R. Xie, W. Ye, Y. Zhang, S. Zhang, Exploiting code knowledge graph for bug localization via bi-directional attention, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 219–229.
[22] X. Huo, M. Li, Z.-H. Zhou, Control flow graph embedding based on multi-instance decomposition for bug localization, in: Proceedings of the 34th AAAI Conference on Artificial Intelligence, 2020, pp. 4223–4230.
[23] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in Neural Information Processing Systems, 2013, pp. 3111–3119.
[24] G. Lee, J. Jeong, S. Seo, C. Kim, P. Kang, Sentiment classification with word localization based on weakly supervised learning with a convolutional neural network, Knowl.-Based Syst. 152 (2018) 70–82.
[25] J. Ni, H. Tong, W. Fan, X. Zhang, Inside the atoms: Ranking on a network of networks, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014, pp. 1356–1365.
[26] J. Ni, H. Tong, W. Fan, X. Zhang, Flexible and robust multi-network clustering, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2015, pp. 835–844.
[27] D. Binkley, M. Davis, D. Lawrie, C. Morrell, To camelcase or under_score, in: Proceedings of the 17th International Conference on Program Comprehension, 2009, pp. 158–167.
[28] M. Kampffmeyer, Y. Chen, X. Liang, H. Wang, Y. Zhang, E.P. Xing, Rethinking knowledge graph propagation for zero-shot learning, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 11487–11496.
[29] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: Proceedings of the 5th International Conference on Learning Representations, 2017, pp. 1–14.
[30] L. Zhang, Z. Kang, X. Sun, H. Sun, B. Zhang, D. Pu, KCRec: Knowledge-aware representation graph convolutional network for recommendation, Knowl.-Based Syst. 230 (2021) 107399.
[31] T. Dozat, C.D. Manning, Deep biaffine attention for neural dependency parsing, in: Proceedings of the 5th International Conference on Learning Representations, 2017, pp. 1–8.
[32] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: Proceedings of the 3rd International Conference on Learning Representations, 2015, pp. 1–15.
[33] Z. Zhu, Y. Li, H. Tong, Y. Wang, COOBA: Cross-project bug localization via adversarial transfer learning, in: Proceedings of the 29th International Joint Conference on Artificial Intelligence, 2020, pp. 3565–3571.
[34] A. Marcus, A. Sergeyev, V. Rajlich, J.I. Maletic, An information retrieval approach to concept location in source code, in: Proceedings of the 11th Working Conference on Reverse Engineering, 2004, pp. 214–223.

[35] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, Source code retrieval for bug localization using latent dirichlet allocation, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 155–164.
[36] T. Hoang, R.J. Oentaryo, T.-D.B. Le, D. Lo, Network-clustered multi-modal bug localization, IEEE Trans. Softw. Eng. 45 (10) (2018) 1002–1023.
[37] Z. Shi, J. Keung, K.E. Bennin, X. Zhang, Comparing learning to rank techniques in hybrid bug localization, Appl. Soft Comput. 62 (2018) 636–648.
[38] G. Algan, I. Ulusoy, Image classification with deep learning in the presence of noisy labels: A survey, Knowl.-Based Syst. 215 (2021) 106771.
[39] C. Tian, Y. Xu, W. Zuo, B. Du, C.-W. Lin, D. Zhang, Designing and training of a dual CNN for image denoising, Knowl.-Based Syst. 226 (2021) 106949.
[40] S. Zhang, M. Chen, J. Chen, Y.-F. Li, Y. Wu, M. Li, C. Zhu, Combining cross-modal knowledge transfer and semi-supervised learning for speech emotion recognition, Knowl.-Based Syst. 229 (2021) 107340.
[41] Q. Wang, C. Feng, Y. Xu, H. Zhong, V.S. Sheng, A novel privacy-preserving speech recognition framework using bidirectional LSTM, J. Cloud Comput. 9 (1) (2020) 1–13.
[42] Y. Wang, Y. Li, Z. Zhu, H. Tong, Y. Huang, Adversarial learning for multi-task sequence labeling with attention mechanism, IEEE/ACM Trans. Audio Speech Lang. Process. 28 (2020) 2476–2488.
[43] G. Beigi, K. Shu, R. Guo, S. Wang, H. Liu, Privacy preserving text representation learning, in: Proceedings of the 30th ACM Conference on Hypertext and Social Media, 2019, pp. 275–276.
[44] X. Li, R. Zhang, Q. Wang, H. Zhang, Autoencoder constrained clustering with adaptive neighbors, IEEE Trans. Neural Netw. Learn. Syst. 32 (1) (2021) 443–449.
[45] K. Ding, J. Li, H. Liu, Interactive anomaly detection on attributed networks, in: Proceedings of the 12th ACM International Conference on Web Search and Data Mining, 2019, pp. 357–365.
[46] S.K. Pandey, A.K. Tripathi, BCV-predictor: A bug count vector predictor of a successive version of the software system, Knowl.-Based Syst. 197 (2020) 105924.
[47] T. Hoang, J. Lawall, Y. Tian, R.J. Oentaryo, D. Lo, PatchNet: Hierarchical deep learning-based stable patch identification for the linux kernel, IEEE Trans. Softw. Eng. 47 (11) (2021) 2471–2486.
[48] X. Huo, F. Thung, M. Li, D. Lo, S.-T. Shi, Deep transfer bug localization, IEEE Trans. Softw. Eng. 47 (7) (2021) 1368–1380.
[49] X. Wan, Z. Zheng, F. Qin, Y. Qiao, K.S. Trivedi, Supervised representation learning approach for cross-project aging-related bug prediction, in: Proceedings of the 30th International Symposium on Software Reliability Engineering, 2019, pp. 163–172.
[50] L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang, TBCNN: A tree-based convolutional neural network for programming language processing, 2014, arXiv preprint arXiv:1409.5718.
[51] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: Proceedings of the 41st International Conference on Software Engineering, 2019, pp. 783–794.