Data Mining and Knowledge Discovery (2021) 35:1369–1392
https://doi.org/10.1007/s10618-021-00755-7

A deep multimodal model for bug localization

Ziye Zhu[1] · Yun Li[1,2] · Yu Wang[1] · Yaojing Wang[2] · Hanghang Tong[3]

Received: 19 October 2019 / Accepted: 19 April 2021 / Published online: 28 April 2021

Abstract

Bug localization utilizes the collected bug reports to locate the buggy source files. The state of the art falls short in handling the following three aspects, including (L1) the subtle difference between natural language and programming language, (L2) the noise in the bug reports and (L3) the multi-grained nature of programming language. To overcome these limitations, we propose a novel deep multimodal model named DEMOB for bug localization. It embraces three key features, each of which is tailored to address each of the three limitations. To be specific, the proposed DEMOB generates the multimodal coordinated representations for both bug reports and source files for addressing L1. It further incorporates the AttL encoder to process bug reports for addressing L2, and the MDCL encoder to process source files for addressing L3. Extensive experiments on four large-scale real-world data sets demonstrate that the proposed DEMOB significantly outperforms existing techniques.

Responsible editor: Johannes Fürnkranz

✉ Yun Li
liyun@njupt.edu.cn

Ziye Zhu
2016070251@njupt.edu.cn

Yu Wang
2017070114@njupt.edu.cn

Yaojing Wang
wyj@smail.nju.edu.cn

Hanghang Tong
htong@illinois.edu

1 Jiangsu Key Lab. of Big Data Security and Intelligent Processing, Nanjing University of Posts and Telecommunications, Nanjing, People's Republic of China

2 State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, People's Republic of China

3 Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA

Keywords Bug localization · Bug report · Multimodal learning · Attention mechanism · Multi-grained features

1 Introduction

A software system contains bugs, regardless of how much effort is spent on developing (Wong and Debroy 2009). According to the IEEE standard, bugs are the manifestations and results of errors during the program coding (DeMillo et al. 1997). To debug, programmers must first be able to locate the buggy source files (or methods, classes, etc.) (Li and Li 2012), and then fix them. The process of locating the buggy source files is known as bug localization. Furthermore, the effectiveness of manual bug localization depends on the developers' experience and the understanding of the program (Wong and Debroy 2009). Therefore, the bug localization task has historically been regarded as a time-consuming and labor-intensive task, especially for the complex, large-scale software systems (Wang et al. 2015a). The limitations have motivated extensive investigations into partially or fully automated bug localization. To date, these automated approaches are mainly based on program analysis (Sterling and Olsson 2007; Zhang et al. 2005; Wong and Qi 2006) and bug report (Zhou et al. 2012; Kim et al. 2013; Lam et al. 2017). The former approach locates the buggy source code by static or dynamic source code analysis. The latter approach predicts the related buggy source files concerning the bug report (i.e., the users' feedback). This paper focuses on the latter, referred to as bug report-based bug localization task.

In general, a bug report records program defects or failures submitted by programmers, testers, or end-users during software development and maintenance. Figure 1 shows a simplified bug report from SWT project recorded in online Bugzilla system,[1] where the description item in it details the reproduction of an unexpected problem encountered by the reporter. In this case, the bug report records a security issue related to the password field. Recently, various bug report-based bug localization approaches (Zhou et al. 2012; Kim et al. 2013; Lam et al. 2017) have been proposed, mainly focusing on estimating the relevance between the bug report and source file.

Despite great achievements, the state of the art falls short in handling the following three aspects. (L1) The subtle difference between natural language and programming language. Existing bug report-based methods ignore different language properties and directly fuse two language representations by a simple neural network. The localization performance might be limited without narrowing the gap between the two representation spaces. (L2) The noise in the bug reports, especially in non-professional ones. The main content of the bug report belongs to the informal document (i.e., user-generated text). Intuitively, the noise in bug reports will lead to a negative effect on localization performance. (L3) The multi-grained nature of programming language. That is, the information carrier of the programming language is multi-grained (e.g., code token, code statement, code function). Recently, a few approaches can extract structural fea-

[1] A bug tracking system for both free and open-source software, proprietary projects, and products. https://www.bugzilla.org.

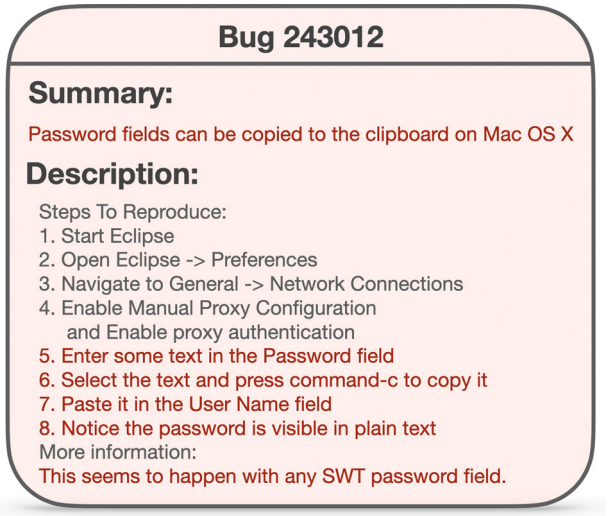

Fig. 1 Take an SWT bug report #243012 for example, the key information is displayed in red, and the noise in black (Color figure online)

tures in statements, but ignore the multi-grained structural features in the program. We will discuss these limitations in more detail in Sect. 2.2—Limitations and Ideas.

To address these limitations, we propose a novel deep multimodal model named DeMoB (Deep Multimodal model for Bug localization) for the bug report-based bug localization task. Inspired by multimodal machine learning (Mroueh et al. 2015), which is capable of processing and correlating information from multiple sources, we treat the bug report (i.e., natural language) and source file (i.e., programming language) as two modalities, and then narrow the gap between the two languages to address L1. For addressing L2, we propose the AttL encoder to automatically focus on valuable information and filter bug-irrelevant part of the bug report, thereby reducing or eliminating the interference caused by noise. For addressing L3, we first split the source file according to the code token, code statement and code function, and then the MDCL encoder extracts the multi-grained features of the source file. Furthermore, the MDCL encoder can process variable-length source files by leveraging dynamic Convolutional Neural Network (DCNN) (Kalchbrenner et al. 2014). Extensive experiments on large-scale real-world data sets reveal that our model significantly outperforms several state-of-the-art techniques on the bug localization task.

The contribution of our work is threefold,

- We propose a novel multimodal model called DeMoB for bug report-based bug localization task. The proposed DeMoB ameliorates the difference between the bug report and source file by mapping their individual representations into a coordinated multimodal space.
- We propose the AttL encoder to extract key information from bug reports while filtering noise.
- We design the MDCL encoder to well capture the multi-grained programming language features, including token-level, statement-level, and function-level.

Table 1 Notations

Symbol	Description
$B = \{b_1, \ldots, b_M\}$	Collection of bug reports
$S = \{s_1, \ldots, s_N\}$	Collection of source files
M	Number of bug reports
N	Number of source files
W	B-S relevance indicator matrix
b_{new}	A new bug report
S^+	Collection of buggy source files to b
s^+	s is a buggy source file to b
S^-	Collection of clean source files to b
s^-	s is a clean source file to b

The rest of this paper is organized as follows. Section 2 describes the problem definition and key ideas. Section 3 presents the proposed DEMOB. Section 4 presents the experimental results and analysis. Section 5 briefly reviews the related work, and Sect. 6 concludes.

2 Problem statement

In this section, we first present the bug report-based bug localization problem and then analyze the limitations of existing methods.

2.1 Problem definition

The goal of the bug report-based bug localization task is to find buggy source files that lead to inappropriate defects described in bug reports. We denote $B = \{b_1, b_2, \ldots b_M\}$ as a set of bug reports, and $S = \{s_1, s_2 \ldots, s_N\}$ as a set of source files, where M, N are the number of the bug reports and source files in the project. Besides, an indicator matrix $W \in \mathbb{R}^{M \times N}$ is used to indicate whether a source file is the buggy file of a bug report. For example, $W_{i,j} = 1$ indicates that source file s_j is the buggy file of the bug report b_i and $W_{i,j} = 0$ indicates not. The main symbols in this paper are shown in Table 1. Based on the above notations, we define the bug report-based bug localization problem as follows.

Problem 1 bug report-based Bug Localization

Given: (1) a collection of bug reports B containing M bug reports, (2) a collection of source files S containing N source files, (3) an indicator matrix W, (4) a new bug report $b_{new} \notin B$;

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

To solve this problem, we instantiate the bug localization as a structured learning task, and propose a two-stage solution containing a training stage and a prediction

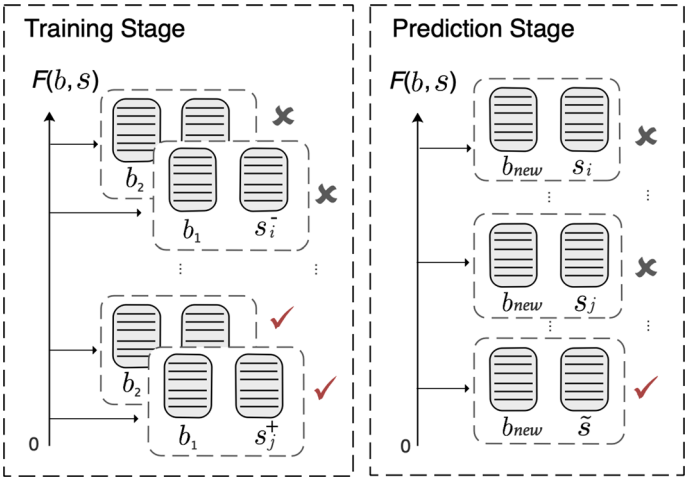

Fig. 2 The two-stage framework contains a training stage and a prediction stage. The training stage aims to learn an evaluation function to estimate the relevance degree of a (b, s) pair, where (b, s^+) and (b, s^-) denote the related pairs and irrelated pairs, respectively. The prediction stage is to find the source file $\tilde{s}$ which is most likely to contain the error described in bug report b

stage, as shown in Fig. 2. The framework of our proposed DEMOB is denoted as

$$F\colon B \times S \to R \Rightarrow f(b_{new}) = \tilde{s} = \arg\min_{s \in S} F(b_{new}, s). \tag{1}$$

In detail, the training stage aims to learn an evaluation function $F\colon B \times S \to R$ to estimate the relevance between the source file s and the bug report b. The prediction stage is to find the source file $\tilde{s}$ which is most likely to contain the error described in bug report b_{new}. The localization is made by $\tilde{s} = \arg\min_{s \in S} F(b_{new}, s)$, where $F(.,.)$ is the evaluation function learned in the training stage.

2.2 Limitations and ideas

Before we present the details of our DEMOB, we summarize the limitations of the state of the art in handling the following three aspects, and the key ideas of the proposed DEMOB to address them.

L1: Subtle difference between natural language and programming language The main subjects in the bug report are written in natural language (e.g., English, Chinese), and the source file is a sequence of programming language statements (e.g., C, Java). Although there are numerous syntactic correspondences between them (Mihalcea et al. 2006), the corresponding representation space of each language is individual due to the different properties of the two languages. Existing bug report-based methods ignore the properties of different languages and directly fuse two language representations by using a simple neural network. For example, Huo et al. (2016) employed two CNNs to learn the feature representations of bug reports and source files respectively, and then directly fused them through a fully-connect neural network. However, inferring the

relevance between the bug report and the source file without narrowing the distance between the two representation spaces might limit the final localization performance.

Key idea in DEMOB *for addressing L1: Narrowing the gap by coordinated representation module* Currently, multimodal representation learning (Baltrušaitis et al. 2019) has gained much attention due to its powerful ability to learn a representation of data using information from multiple modalities, such as text and video (Silberer and Lapata 2014), text and image (Frome et al. 2013). In particular, coordinated representation learning, the main category of multimodal representation learning, processes unimodal data separately, but can enforce certain constraints on them to bring them into a coordinated space. Considering the properties of natural language and programming language in our task, we treat the bug report and source file as two modalities. In the existing methods, the individual representations of the bug report and source file can be well learned by neural networks of different structures. However, their individual representations could be further projected into a coordinated space based on coordinated representation learning. To be specific, the coordinated representation module in our model is responsible for coordinating the individual representations of the bug report and source file into a coordinated space through a constraint (i.e., relevance distance). We encourage the representation of related source file to be as close as possible to the representation of the given bug report by minimizing the relevance distance between them in the coordinated space.

L2: The noise in the bug reports, especially in non-professional ones The main content of the bug report belongs to the informal document (i.e., user-generated text). According to the statistical analysis (Rahman and Roy 2018), apart from some reports submitted by professionals that contain localization hints (e.g., program elements, stack traces), most non-professional reports generally contain bug-irrelevant content. From the bug report #243012 shown in Fig. 1, we can see some bug-irrelevant content displayed in black (e.g., 1. Start Eclipse, 2. Open Eclipse→Preferences). This is because users desired to be as specific as possible when describing the issue, which leads to redundant information. In our task, such information might harm the localization performance (referred to as 'noise' in our paper). Recently, existing methods (Huo et al. 2016; Huo and Li 2017) adopt the CNN, which can effectively learn local features, to make the influence of noise on the model relatively small. However, these methods cannot perform well in learning the sequential features of bug reports.

Key idea in DEMOB *for addressing L2: The AttL encoder for encoding bug reports* Based on this observation, we desire our model to learn sequential features while mitigating the effects of noise in bug reports. Armed with this, we propose the AttL encoder, a Bidirectional Long Short-Term Memory Networks (BiLSTM) (Hochreiter and Schmidhuber 1997) with attention mechanism (Bahdanau et al. 2014), to capture the semantic information in bug reports and automatically focus on a set of words where the most bug-relevant content is concentrated. At present, the attention mechanism has been widely used in many tasks, such as machine translation (Bahdanau et al. 2014), network embedding (Liu et al. 2019). It affords the model a remarkable capacity to selectively learn information by assigning different weights to each part of the input. For our task, soft attention (a popular attention-based network) is employed to consider all words in a bug report, and then assign words that have a decisive effect on our task to a higher weight, and words that are useless to a lower weight. The final representation

of the bug report is computed as a weighted sum of all word-level representations generated by BiLSTM. In this way, the AttL encoder allows our model to concentrate on key information and filter noise in learning bug report representations.

L3: The multi-grained nature of programming language The information carrier of the programming language is multi-grained (e.g., code token, code statement, code function). Abundant multi-grained structural features can be obtained by analyzing source files at different granularities. Such multi-grained features are essential for understanding the functionality performed by complex source files. Unfortunately, most existing methods do not fully exploit the multi-grained nature of programming language. For example, DNNLoc (Lam et al. 2017) only considered the features of tokens, but ignored the more abstract features, such as the structural features provided by statements. Recently, LS-CNN (Huo and Li 2017) explored the structural nature within statements and the sequential nature among statements. However, this method ignored the function-level features.

Key idea in DEMOB *for addressing L3: The MDCL encoder for encoding source files* In order to extract abundant multi-grained structural features of source files, we design a particular network MDCL containing multiple DCNNs (Kalchbrenner et al. 2014) and a BiLSTM as the source file encoder in our model. To be specific, we first split the input source file according to different granularities, including token, statement, and function. Subsequently, multiple DCNN layers learn information within the code function, including structural information in the statement and code block (consisting of multiple statements). Then, we utilize BiLSTM to learn the dependencies between functions. The final source file representation obtained by the MDCL encoder incorporates multi-grained features (including token, statement, and function). It is worth to be mentioned that we employ DCNN instead of traditional CNN in feature extraction, and the inputs of BiLSTM are the function-level representations. Accordingly, the MDCL encoder is endowed with the ability to handle over-length source files effectively.

3 The proposed model

In this section, we present the proposed DEMOB for the bug report-based bug localization task. As illustrated in Fig. 3, there are three integral parts in our model, including a bug report processing module $M^b : b \rightarrow \boldsymbol{v}^b$, a source file processing module $M^s : s \rightarrow \boldsymbol{v}^s$, and a coordinated representation module $C(\boldsymbol{v}^b, \boldsymbol{v}^s)$, where $\boldsymbol{v}^b$ and $\boldsymbol{v}^s$ are the representations obtained from the bug report and source file processing modules respectively.

3.1 Bug report processing module

We first introduce the bug report processing module designed to extract semantic information about the bugs described in the bug report. It contains an embedding layer and a bug report encoder, as illustrated in Fig. 3.

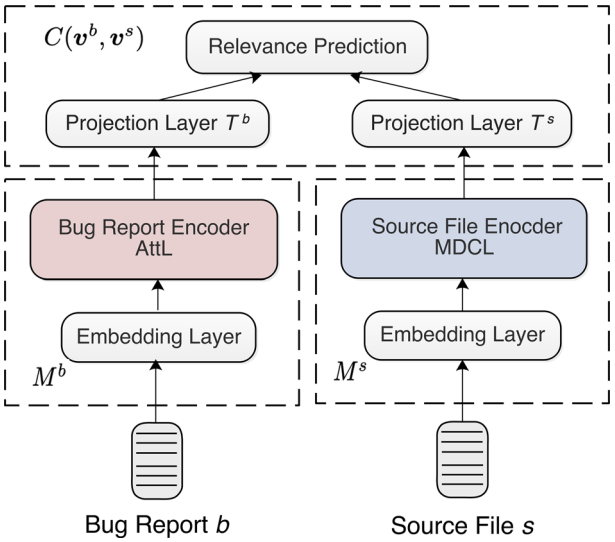

Fig. 3 Overall framework of DeMoB. It consists of (1) bug report processing module M^b, (2) source file processing module M^s, and (3) coordinated representation module $C(\boldsymbol{v}^b, \boldsymbol{v}^s)$

3.1.1 Embedding layer

Given a bug report b, we consider all content from the summary and description items, which are the main part of the bug report, and treat them as a sequence of m words $\{w_1, w_2, \ldots, w_m\}$. Then, we use pre-trained ELMo (Peters et al. 2018) to initialize each word, which can generate a dynamic context-dependent representation for each word depending on the entire context. Specifically, the contextualized word representations are embedded from a deep bidirectional Language Model (biLM). Through the embedding layer, the bug report b is represented as a word embedding sequence $\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_m\}$, where $\boldsymbol{w}_i$ is the word embedding of w_i. The word embedding sequence is then passed to the subsequent bug report encoder.

3.1.2 Bug report encoder

In order to encode the input bug report b while mitigating the effects of noise in it, we design the network AttL as the bug report encoder. The structure of AttL is shown in Fig. 4. To be specific, the BiLSTM can efficiently learn the sequential features in the bug report (Hochreiter and Schmidhuber 1997; Schuster and Paliwal 1997). We denote a BiLSTM and concatenate the forward hidden state $\overrightarrow{\boldsymbol{h}}_i^b$ and a backward hidden state $\overleftarrow{\boldsymbol{h}}_i^b$ into a new vector $\boldsymbol{h}_i^b$ as follows,

$$\overrightarrow{\boldsymbol{h}}_i^b = \text{LSTM}_{forward}^b(\boldsymbol{w}_i, \overrightarrow{\boldsymbol{h}}_{i-1}^b), \tag{2}$$

$$\overleftarrow{\boldsymbol{h}}_i^b = \text{LSTM}_{backward}^b(\boldsymbol{w}_i, \overleftarrow{\boldsymbol{h}}_{i+1}^b), \tag{3}$$

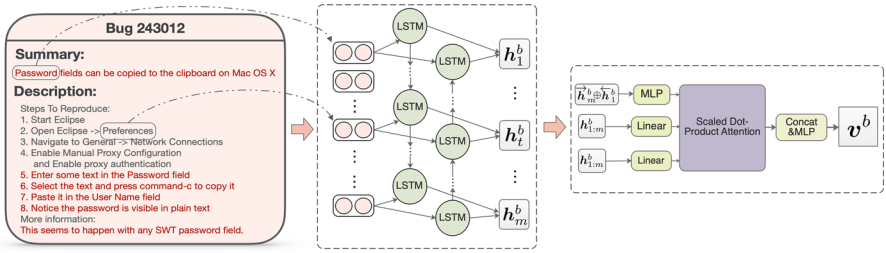

Fig. 4 The overall structure of the AttL encoder. Take an SWT bug report for example, the key information is displayed in red, and the noise in black (Color figure online)

$$\boldsymbol{h}_i^b = \overrightarrow{\boldsymbol{h}}_i^b \oplus \overleftarrow{\boldsymbol{h}}_i^b, \tag{4}$$

where the $\boldsymbol{w}_i$ is the input of the BiLSTM in the time step i, and the $\oplus$ is the concatenate operation. The obtained $\boldsymbol{h}_i^b$ contains information about the whole input sequence with a strong focus on the parts surrounding the ith word w_i of the input sequence. For the input bug report b, BiLSTM generates a hidden state sequence $\{\boldsymbol{h}_1^b, \boldsymbol{h}_2^b, \ldots, \boldsymbol{h}_m^b\}$.

In addition, for filtering the bug-irrelevant noise in the bug report b, we employ soft attention, a popular attention-based network, to consider the influence of all words in the bug report. We calculate the final bug report representation $\boldsymbol{v}^b$ as a weighted sum of these hidden states $\{\boldsymbol{h}_1^b, \boldsymbol{h}_2^b, \ldots, \boldsymbol{h}_m^b\}$ by Eq. 5:

$$\boldsymbol{v}^b = \sum_{i=1}^{m} \alpha_i \cdot \boldsymbol{h}_i^b, \tag{5}$$

where α_i denotes the weight of each hidden state $\boldsymbol{h}_i^b$ indicating the importance of w_i in the input bug report b. That is, words that carry key information are given higher weights, and words that are useless to our task are assigned lower weights. Specifically, α_i is generated by

$$\alpha_i = \frac{\exp(\boldsymbol{g} \cdot \boldsymbol{h}_i^b)}{\sum_{j=1}^{m} \exp(\boldsymbol{g} \cdot \boldsymbol{h}_j^b)}, \tag{6}$$

where $\boldsymbol{g}$ is the global feature from a multilayer perception (MLP) network, denoted as:

$$\boldsymbol{g} = \text{MLP}(\overrightarrow{\boldsymbol{h}}_m^b \oplus \overleftarrow{\boldsymbol{h}}_1^b). \tag{7}$$

Thanks to the attention mechanism, we relieve the bug report encoder from the burden of encoding all sequential features in the bug report into a vector, by selectively retrieving the information distributed throughout the hidden state sequence. At this point, we obtain a final bug report representation $\boldsymbol{v}^b$ from the AttL encoder that will be input to the coordinated representation module for relevance prediction.

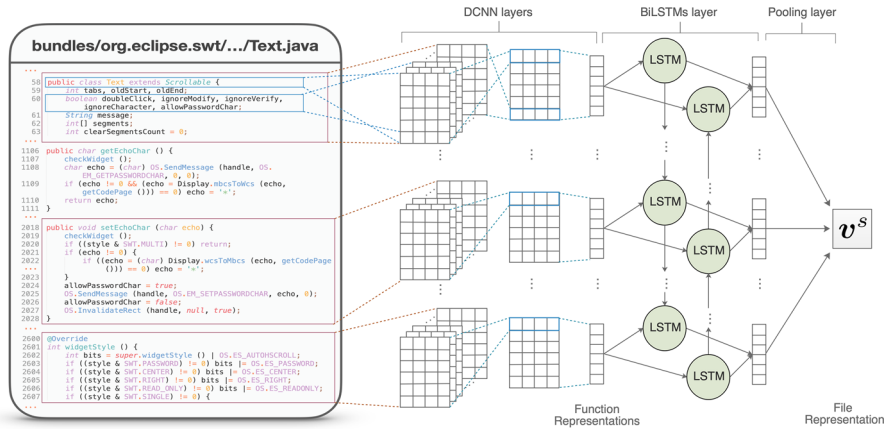

Fig. 5 The overall structure of the MDCL encoder. The input source file is related to the bug report shown in Fig. 4

3.2 Source file processing module

In terms of source files, we employ an embedding layer and a source file encoder to fully extract the multi-grained information. The structure of the source file processing module is shown in Fig. 3.

3.2.1 Embedding layer

Given a source file s, the embedding layer in the source file processing module is responsible for mapping all tokens in it into their corresponding embedding representations. Due to the difference between natural language and programming language, we pre-train another ELMo-based embedding layer, which has the same structure as the embedding layer in the bug report processing module, but no parameters are shared between them. For the source file s, we treat it as a sequence of n code tokens $\{x_1, x_2, \ldots, x_n\}$. The embedding layer converts the tokens into a sequence of embeddings $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$, where $\boldsymbol{x}_i$ is the token embedding of x_i. The token embedding sequence is then passed to the source file encoder.

3.2.2 Source file encoder

After embedding the given source file to a token representation sequence, we split it according to multiple granularities, including code token, code statement, and code function. Then, a particular network MDCL composed of multiple DCNN layers (Kalchbrenner et al. 2014) and a BiLSTM layer is set as the source file encoder to learn multi-grained information and generate a file representation. The overall structure of the MDCL encoder is shown in Fig. 5.

To be specific, we select all token representations belonging to a function as the input of multiple DCNN layers. The first DCNN layer is used to represent the semantics of a statement based on the tokens within the statement. The subsequent DCNN layer

is used to learn the semantics conveyed by multiple statements while preserving the integrity of statements. The dynamic k-max pooling operations can process the varying length code functions, where the k is a set of dynamic values depending on the token number of the input function and the depth of the network. Following the work of Kalchbrenner et al. (2014), the pooling parameter is calculated as follow,

$$k_l = max\left(k_{top}, \left\lceil \frac{L-l}{L} s \right\rceil\right), \tag{8}$$

where l is the number of the current convolutional layer of the applied pooling, L is the total number of convolutional layers in the network, and k_{top} is the pooling parameter of the topmost convolutional layer with a default value. In this way, for the source file s, multiple DCNN layers eventually generate a function representation sequence $\{\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots, \boldsymbol{f}_u\}$, where u is the number of functions in source file s, and each function representation contains the semantic information inside one code function.

Then, the function representation sequence is input to the BiLSTM layer to learn the dependency among functions. For simplicity, we denote the function dependency extraction as the following equations,

$$\overrightarrow{\boldsymbol{h}}^s_t = \text{LSTM}^s_{forward}(\boldsymbol{f}_t, \overrightarrow{\boldsymbol{h}}^s_{t-1}), \tag{9}$$

$$\overleftarrow{\boldsymbol{h}}^s_t = \text{LSTM}^s_{backward}(\boldsymbol{f}_t, \overleftarrow{\boldsymbol{h}}^s_{t+1}), \tag{10}$$

$$\boldsymbol{h}^s_t = \overrightarrow{\boldsymbol{h}}^s_t \oplus \overleftarrow{\boldsymbol{h}}^s_t. \tag{11}$$

Finally, we employ a pooling layer involving a mean pooling operation to fuse the $\boldsymbol{h}^s_t$ and obtain the source file representation $\boldsymbol{v}^s$ of the source file s.

3.3 Coordinated representation module

At this moment, we obtain the bug report representation $\boldsymbol{v}^b$ from bug report processing module, and the source file representation $\boldsymbol{v}^s$ by source file processing module. Due to the properties of different languages, the obtained corresponding representations are located in their individual embedding spaces. To coordinate the individual representations into a coordinated embedding space, the core coordinated representation module is trained by means of a projection layer and a relevance prediction layer. Specifically, the projection layer is a linear transformation derived from the bug report and source file representations by

$$\tilde{\boldsymbol{v}}^b = T^b \boldsymbol{v}^b, \tag{12}$$

$$\tilde{\boldsymbol{v}}^s = T^s \boldsymbol{v}^s, \tag{13}$$

where T^b and T^s are the transformation matrices; $\boldsymbol{v}^b$ and $\boldsymbol{v}^s$ are the representations obtained from bug report processing module and source file processing module respectively; the coordinated representations $\tilde{\boldsymbol{v}}^b$ and $\tilde{\boldsymbol{v}}^s$ have the equal dimensions. In order

to predict the relevance of the bug report and the source file, we calculate the distance between their mappings in the embedding. The relevance distance is defined as

$$F(b, s) = \|\tilde{\boldsymbol{v}}^b - \tilde{\boldsymbol{v}}^s\|_2^2 = \|T^b \boldsymbol{v}^b - T^s \boldsymbol{v}^s\|_2^2. \tag{14}$$

We encourage the representation of the relevant buggy source file to be as close as possible to the representation of the given bug report. Considering the distance of relevant pairs (b, s^+) should should be less than the distance of irrelevant pairs (b, s^-), where $s^+ \in S^+$ and $s^- \in S^-$, the pairwise ranking loss function for training DeMoB is

$$\mathcal{L}(\Theta) = \sum_{b, s^-, s^+} \max(0, \tau + F(b, s^+) - F(b, s^-)) + \lambda_\Theta \|\Theta\|^2, \tag{15}$$

where $F(.,.)$ is computed by Eq. 14; Θ denotes all parameters of DeMoB and λ_Θ is the regularization hyperparameter; τ is a margin, forcing $F(b, s^+)$ to be smaller than $F(b, s^-)$ by τ. In this way, we adapt Adaptive Moment Estimation (Adam) method (Kingma and Ba 2014) to directly minimize the loss function $\mathcal{L}(\Theta)$ in our model.

4 Experimental results and analysis

This section introduces the data sets used for evaluation and then presents the experimental setup and experimental results. Finally, we present the case study and analysis of the experimental results.

4.1 Data sets

We use four well-known real-world projects (Lam et al. 2017) to evaluate our method. The data sets are extracted from the bug tracking system Bugzilla and the version control system Git, including the bug reports, source code links, buggy files, API documentation, and the oracle of bug-to-file mappings. Some brief introductions of these projects are as follows,

- AspectJ[2]: an aspect-oriented programming extension for Java programming language.
- JDT[3]: a suite of Java development tools for Eclipse.
- SWT[4]: an open-source widget toolkit for Java.
- Eclipse Platform UI[5]: contains a set of frameworks and common services for Eclipse.

Additionally, the statistics of each project are shown in Table 2.

[2] https://www.eclipse.org/aspectj/.

[3] https://www.eclipse.org/jdt/.

[4] https://www.eclipse.org/swt/.

[5] https://www.eclipse.org/eclipse/platform-ui/.

Table 2 Basic statistics of data sets

Data sets	Time range	No. bug report	No. source file
AspectJ	03.13.2002–01.10.2014	574	4439
JDT	10.10.2001–01.14.2014	6211	8184
SWT	02.19.2002–01.17.2014	4018	2056
Platform UI	10.10.2001–01.17.2014	6455	3454

4.2 Experimental setup

We randomly divide the bug reports into ten folds for each data set, where each fold has the equal size, and ten-fold cross-validation is used in the experiments. Of the 10 folds, the bug reports of a single fold are retained as the unfixed bug reports for testing the model, and the remaining 9 folds are used as fixed bug reports for training the model. 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 times are then averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and testing, and each observation is used for testing exactly once. We evaluate the bug localization performance relied on two metrics, i.e., Top-K and Mean Average Precision (MAP).

4.2.1 Evaluation metrics

Top-K The Top-K has been widely adopted in bug report-based bug localization methods (Zhou et al. 2012; Huo et al. 2016; Huo and Li 2017; Wang et al. 2018). For a given bug report, if one of the buggy files is within the Top-K list of files, we count it as a hit. Otherwise, we consider that as a miss. The Top-K accuracy is measured by the percentage of hits over the total number of tested bug reports as follow,

$$Top_K = \frac{1}{M_{test}} \sum_{i}^{M_{test}} hit_i^{(K)}, \tag{16}$$

where M_{test} is number of tested bug reports, $hit_i^{(K)} \in \{0, 1\}$ indicates whether the recommended Top-K source files of bug report b_i contains as least one truly buggy file (i.e., take a hit). As to the list size K, we choose K = 1, 5, 10 for Top-K criteria.
MAP To consider the cases of a bug report with multiple buggy files, we adopt MAP, which is also widely used for evaluating the cost-effectiveness of identifying buggy source files (Huo et al. 2016; Huo and Li 2017; Xiao et al. 2018), and computed as follows,

$$MAP = \frac{1}{M_{test}} \sum_{i=1}^{M_{test}} \sum_{j=1}^{N} \frac{Prec(j) * t(j)}{n_i}, \quad \text{where } Prec(j) = \frac{Q(j)}{j}, \tag{17}$$

where M_{test}, N denote the number of tested bug reports and source files in S, respectively; n_i is the number of buggy files to bug report b_i and $t(j)$ is the vector indicates whether source file in the rank position j is related to b_i 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. That is, average precision calculates the precision at the position of every related source file in the ranked results list. MAP is the mean of these average precisions across all tested bug reports. This metric considers whether all of the relevant source files tend to get ranked highly, which matches the need of our bug localization task to show as many relevant source files as possible high up the recommended list.

4.2.2 Methods for comparison

In our experiments, we compare our proposed DEMOB with the following state-of-the-art bug localization methods,

- BugLocator (Zhou et al. 2012): a well-cited Information Retrieval(IR)-based method that ranks all files considering information about similar bugs that have been fixed before.
- BLUiR (Saha et al. 2013): an IR-based model that takes advantage of structural information such as class and method names in code.
- DNNLoc (Lam et al. 2017): a bug localization model combining rVSM (Zhou et al. 2012) with Deep Neural Network (DNN) while considering the metadata of the bug-fixing history and API elements.
- NP-CNN (Huo et al. 2016): a deep learning method based on CNN, which leverages both lexical and program structural information from natural language and programming language.
- LS-CNN (Huo and Li 2017): an improved model of NP-CNN, which employs a CNN model for bug reports representation and a combination of CNN and LSTM to process the source files.

Among these methods, BugLocator and BLUiR are IR-based methods; DNNLoc, NP-CNN and LS-CNN are based on deep neural networks.

4.2.3 Implementation details

We assign the hyper-parameters using the default values based on the experience. In the bug report encoder, the BiLSTM has a depth of 2 and a hidden size of 256, and the MLP has a depth of 2 and a hidden size of 256. In the source file encoder, the total number of convolutional layers of each DCNN layer is 3, and the BiLSTM layer has a depth of 2 and a hidden size of 256. We apply Dropout (Srivastava et al. 2014) to the output of the BiLSTM at the rates of 0.5. The initial learning rate is set to 0.008 and decreases as the training step increases. The batch size is set to 16. A fixed margin $\tau = 0.2$ is used in all experiments. We implement our model under PyTorch.[6] All of our experiments are performed on NVIDIA 1080ti GPU and Intel i7-8700K CPU. In addition, a bug report is generally only related to one or a few source files. Therefore,

[6] https://pytorch.org/.

Table 3 The effectiveness results of all comparison methods for bug localization based on bug report

Project	Model	Top-1	Top-5	Top-10	MAP
AspectJ	BugLocator	0.202	0.487	0.576	0.224
	BLUiR	0.304	0.568	0.624	0.269
	DNNLoc	0.431	0.694	0.803	0.295
	NP-CNN	0.486	0.721	0.807	0.488
	LS-CNN	0.515	0.763	0.834	0.542
	DeMoB	**0.592**	**0.822**	**0.895**	**0.578**
JDT	BugLocator	0.197	0.413	0.512	0.233
	BLUiR	0.266	0.507	0.620	0.312
	DNNLoc	0.401	0.648	0.737	0.336
	NP-CNN	0.441	0.691	0.788	0.426
	LS-CNN	0.479	0.723	0.796	0.430
	DeMoB	**0.547**	**0.749**	**0.824**	**0.498**
SWT	BugLocator	0.195	0.379	0.521	0.257
	BLUiR	0.258	0.518	0.744	0.320
	DNNLoc	0.345	0.675	0.794	0.365
	NP-CNN	0.417	0.711	0.829	0.522
	LS-CNN	0.483	0.732	0.848	0.538
	DeMoB	**0.596**	**0.832**	**0.887**	**0.561**
Platform UI	BugLocator	0.261	0.449	0.575	0.302
	BLUiR	0.342	0.523	0.748	0.309
	DNNLoc	0.446	0.664	0.801	0.316
	NP-CNN	0.448	0.714	0.819	0.478
	LS-CNN	0.463	0.753	0.833	0.525
	DeMoB	**0.583**	**0.807**	**0.903**	**0.554**

The DeMoB's result is significantly better than all compared methods, with p-value < 0.05 according to the t-test

it is a struggle to use all irrelevant source files for training due to the fact that the number of source files in a project is large. To address this problem, we only select part of irrelevant files as the negative samples (Lam et al. 2017) during the training process. During the prediction process, we calculate and rank all the source files in the project according to the relevance to the new bug report.

4.3 Main results

Table 3 shows the results of the accurate comparison for different approaches on all the projects. As we can see, DeMoB outperforms all the compared methods on all evaluation metrics. For the MAP value, DeMoB surpasses its best competitor (i.e., LS-CNN) by 6.8% on JDT, 3.6% on AspectJ, 2.3% on SWT, and 2.9% on Platform, respectively. For the Top-K accuracy, we can observe that our DeMoB can achieve significant improvements in all four projects, and it is more effective for Top-1. To

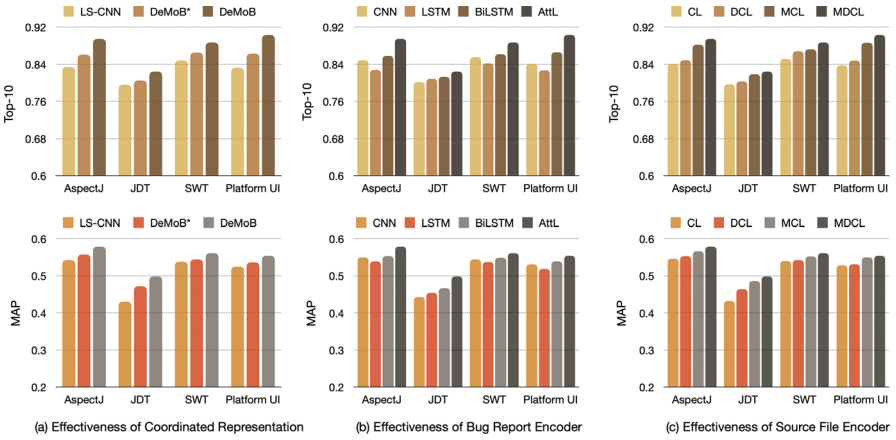

Fig. 6 The effectiveness results with different components in terms of Top-10 and MAP on four projects

be specific, our model achieves from 2.8 to 7.0% improvement overall compared methods for Top-10 accuracy. Meanwhile, the improvement is from 6.8 to 12.0% for Top-1 accuracy. Compared with the deep learning-based methods, our DeMoB further explores the characteristic of our task and designs the advanced neural networks based on the two languages (i.e., natural language and programming language). These results corroborate the effectiveness of our DeMoB. Additionally, the experimental results show that the performance of all deep learning-based methods significantly outperforms the BugLocator and BLUiR. For example, on JDT data set, DNNLoc achieves 22.5% and 11.7% relative improvements w.r.t. Top-10 over BugLocator and BLUiR, respectively. On SWT data set, DNNLoc improves BugLocator and BLUiR by 10.8% and 4.5% w.r.t. MAP. The results demonstrate that deep learning methods can more effectively locate the bugs based on bug reports. Also, we find that our DeMoB performs better than DNNLoc w.r.t. both Top-K and MAP on all data sets, which indicates that DeMoB is better even if it does not consider the manual features of the project.

4.4 Ablation study

In Sect. 2.2, we analyze the limitations of existing methods and present our model DeMoB to address them. In order to prove the effectiveness of our proposed improvements, we construct three sets of experiments to evaluate the impact of different components of DeMoB on its performance, and related experimental results are shown in Fig. 6.

4.4.1 Effectiveness of coordinated representation

As we mentioned in Sect. 2.2, each language has an individual representation space due to different properties. Our proposed DeMoB will narrow the gap between natural language and programming language by learning coordinated representations. This set of

experiments aims to investigate whether coordinated representation learning improves the bug localization task performance. Accordingly, we present a variant of our model, which simply removes the projection layer. That is, the individual representations of the bug report and source file are directly fused through a fully-connected network to make predictions. This revised model is denoted as DeMoB*.

The experimental results are shown in Fig. 6a. We can observe that our DeMoB achieves better performance than DeMoB* on all the four projects. Compare to DeMoB*, DeMoB significantly attains 4.2% relative improvements at MAP accuracy on AspectJ data set. On Platform UI data set, DeMoB achieves 2.1% relative improvements on Top 10. These results validate that the properties of the two languages greatly impact localization performance, and we should coordinate them before measuring relevance. Additionally, we combine the results of this set of experiments with LS-CNN and find that the model without the projection layer is still attained better performance than this method. For example, DeMoB* attains 2.7%, 0.9%, 1.7% and 3.0% relative improvements on each project over LS-CNN in the terms of Top-10. This finding also demonstrates the effectiveness of the bug report encoder and source file encoder in our model.

4.4.2 Effectiveness of bug report encoder

For encoding bug reports, we design the AttL based on the BiLSTM and attention network to capture the indicative information and filter the noise in bug reports. This set of experiments are constructed to investigate the performance of different encoder structures in reducing noise in bug reports and extracting indicative information from bug reports. We choose CNN, LSTM, and BiLSTM structures instead of AttL to encode bug reports, while the other components in DeMoB are fixed.

From Fig. 6b, we can first observe that DeMoB with AttL encoder achieves the best average performance on all four projects. Compare with the suboptimal BiLSTM, AttL still significantly achieves 3.7% and 2.5% relative improvements on Top 10 and MAP on AspectJ data set, respectively. Similarly, on Platform UI data set, it achieves 3.9% and 1.5% relative improvements, respectively. DeMoB with the CNN encoder produces better both Top-10 and MAP results than DeMoB with the LSTM on most data sets, which indicates that CNN also can filter noise because it is more effective in extracting local features. For example, on AspectJ project, CNN achieves 2.1% and 1.0% relative improvements on Top-10 and MAP over LSTM. On SWT data set, CNN improves LSTM by 1.3% on Top-10 and 0.7% on the MAP. Furthermore, the performance of DeMoB with the BiLSTM is better than DeMoB with CNN on all projects. We consider that both semantic and sequential information are essential for learning bug reports.

4.4.3 Effectiveness of source file encoder

For encoding the source file, we propose a network MDCL composed of multiple DCNN and BiLSTM to learn the multi-grained structure information from source files. Moreover, the dynamic k-pooling method in CNN layers and the function-level input improve the performance of our model dealing with the over-length source code

file. To investigate the effect of MDCL encoder, we compare the performance of different encoder structures in this set of experiments, and the results are shown in Fig. 6c. We set CL (a combination of CNN and BiLSTM) as the baseline structure for comparison. Furthermore, we design the DCL (a combination of DCNN and BiLSTM) to evaluate the importance of multi-grained feature extraction, and MCL (a combination of multiple CNN and BiLSTM) to explore the effect of the dynamic k-pooling method. In this set of experiments, other components in the DeMoB are also fixed.

As shown in Fig. 6c, experimental results on four data sets show that our MDCL encoder is significantly superior to others. From the results, DeMoB with DCL is more effective than DeMoB with CL. The main reason we consider is that CNN can effectively extract structural information in programming languages. Simultaneously, the dynamic pooling method also improves the learning capability for over-length source code files. Moreover, we can see that DeMoB with MCL is effective than DeMoB with DCL on all metrics. For example, on project SWT, DeMoB with MCL achieves 0.4% and 1.0% relative improvements w.r.t. Top 10 and MAP than it with DCL, respectively. The results demonstrate that the multi-grained feature extraction indeed plays an essential role in the source code file encoder. Furthermore, the DeMoB with MDCL achieves significant improvements than it with MCL and DCL. For example, in terms of Top-10, it improves DeMoB with MCL and DCL by 1.9% and 0.5% on JDT, respectively. These results demonstrate that the MDCL proposed in this paper could effectively learn the multi-grained structural information in the source files, including over-length files.

4.5 Case study

To further illustrate the effectiveness of our proposed model, we take three fixed bug reports from real-world projects listed in Table 4 for analysis.

The first example is bug report #185841 from Platform UI project. We estimate that this report is likely to be submitted by an end-user since the submitter only presented a suggestion about the color of tags, not an unexpected failure during use. The most important indicative information might be the word *color*. However, we can find many similar source files for such bug reports. For example, in the Platform UI project, there are at least dozens of files that deal with color and have this word in their file names, such as *ColorSelector.java*, *ColorCellEditor.java*, *FormColors.java*, *TestColorFactory.java*, and *BackgroundColorDecorator.java*. Furthermore, the bug report has some other words, such as *workbench*, *new* and *focus*, which are contained in more than one hundred files, such as *Theme.java*, *ShowViewDialog.java* and *output.java*. Fortunately, DeMoB places *LightColorFactory.java* at 8-th position in the Top-10 list. But the compared methods (e.g., LS-CNN) miss the relevant source file in the Top-10 list. For bug reports containing a large amount of content unrelated to bug localization, previous methods attempt to understand the semantics of the content. However, DeMoB focuses on specific keywords (e.g., *color*, *background*, and *new*) and reduces interference caused by unrelated content.

The second example is more challenging in locating the buggy source file. To be specific, the bug report #97229 is about the dialogue operation. The submitter

Table 4 Three fixed bug reports from real-world projects

Eclipse Platform UI Bug Report #185841
Summary Colour treatment of active non-focus tabs
Description We need a new color treatment for active non-focus tabs (the one you get when another window other than the workbench has focus). This goal is a better match with the active tabs, so the transition is less jarring.
Fixed Source File Pathname org/eclipse/ui/internal/themes/LightColorFactory.java
Eclipse Platform UI Bug Report #97229
Summary ListSelectionDialog should be resizable by default
Description Open several editor and leave some dirty. Close all. The 'Save resources' dialog is not not resizable. I think it is always better to make dialogs resizable as this makes it easier for users with different font sizes or using very long names. Make the ListSelectionDialog by default.
Fixed Source File Pathname bundles/org.eclipse.ui.workbench/ EclipseUI/org/eclipse/ui/dialogs/SelectionDialog.java
AspectJ Bug Report #29769
Summary Ajde does not support new AspectJ 1.1 compiler options
Description The org.aspectj.ajde.BuildOptionsAdapter interface does not yet support the new. AspectJ 1.1 compiler options. These need to be added to the interface, any old or renamed options deprecated, and then the correct processing needs to happen within Ajde to pass these options to the compiler.This enhancement is needed by the various IDE projects for there AspectJ 1.1 support.
Fixed Source File Pathname ajbrowser/src/org/aspectj/tools/ajbrowser/BrowserProperties.java
ajde/src/org/aspectj/ajde/BuildOptionsAdapter.java
ajde/src/org/aspectj/ajde/ProjectPropertiesAdapter.java
ajde/src/org/aspectj/ajde/internal/AspectJBuildManager.java
ajde/src/org/aspectj/ajde/internal/CompilerAdapter.java
ajde/src/org/aspectj/ajde/ui/internal/AjcBuildOptions.java
ajde/testdata/examples/figurescoverage/figures/Figure.java
...
org.aspectj.ajdt.core/src/org/aspectj/ajdt/internal/core/builder/AjBuildConfig.java
org.aspectj.ajdt.core/testsrc/org/aspectj/ajdt/ajc/BuildArgParserTestCase.java

also provided indicative information about the buggy source file in its summary (i.e., *ListSelectionDialog.java*). The name of the correct buggy source file is *SelectionDialog.java*, which has the same project path as the *ListSelectionDialog.java*. From this example, one can observe is that the indicative information contained in the bug report is incorrect. Although our DEMOB can locating the right source file in this example, and places *SelectionDialog.java* at 10-position in the Top-10 list, it still cannot achieve satisfactory results in locating such reports. We will investigate whether DEMOB can distinguish the true indicative information from bug reports in future work.

The last example is the bug report #29769 associated with eighteen source files from AspectJ project. This example shows a phenomenon in practice, that is, a bug report might be associated with multiple source files. According to the experimental results, we find that when processing such bug reports, all compared models fail to locate all of related buggy source files. Generally, the functions of these source files might be different. We consider that this functional inconsistency will confuse the bug localization model during the training process. Compared with the existing methods, the performance of our model on this type of bug report is slightly improved. We leave as future work to explore how to address the bug report related to multiple source files.

5 Related work

5.1 Studies on bug report-based bug localization

To date, IR methods are widely used in bug report-based bug localization task by extracting important features from the given bug reports and source files. These IR-based methods consist of four steps (Jie et al. 2015): corpus creation, indexing, query formulation, retrieval and ranking. Specifically, the retrieval models widely used include Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), and Vector Space Model (VSM). For example, Marcus et al. (2004) first used the information retrieval method for concept location, where LSI is used to map concepts expressed in natural language to the relevant parts of the source code. Lukins et al. (2008) first found that LDA can be successfully applied for bug localization. Zhou et al. (2012) proposed the BugLocator based on the revised VSM. Based on these, Wang et al. (2018) further presented the STMLocator, which uses topic modeling to learn both textual and semantic similarities between bug reports and source files. Hoang et al. (2018) presented NetML, which uses multimodal information from both bug reports and program spectra to localize bugs. Since most IR-based methods used extra features in addition to IR similarity, Shi et al. (2018) conducted a brief survey of such hybrid bug localization methods and explored the beneficial features for improving bug localization performance.

Also, machine learning methods are widely used in bug report-based bug localization. For example, Kim et al. (2013) treated the bug localization as a classification problem and proposed a two-phase prediction model applying Naive Bayes. Ye et al. (2014) used an adaptive ranking method for bug localization, where the features are extracted from multiple sources, including source files, API description, bug-fixing, and change history. Nevertheless, the lexical mismatch between natural language in bug reports and programming language in source code is the key factor in limiting these bug localization methods' performance. With the development of deep learning, it can effectively solve the aforementioned lexical mismatch problem. For example, Lam et al. (2017) proposed the DNNLoc combining rVSM (Zhou et al. 2012) with DNN for bug localization, where rVSM evaluated the textual similarity between bug reports and source files, and DNN is learned to associate words in bug reports with potentially different code token in the source files to overcome the lexical mismatch. To address the training problem of DNNLoc (Lam et al. 2017) caused by multiple

DNNs, Xiao et al. (2017) proposed DeepLocator model based on an enhanced CNN and some extra features. Recently, Huo et al. (2016) presented a novel CNN-based model that leveraged both lexical and program structure information to learn a unified feature from natural language and programming language for automatically locating the buggy files. Xiao et al. (2018) proposed CNN_Forest, involving CNN and ensemble of random forests, to process bug reports and source files in different ways. Furthermore, Huo and Li (2017) combined CNN with LSTM networks to enhance the unified feature extraction by exploiting the sequential nature of source code. Huo et al. (2018) applied embeds code comments in generating semantic features from the source code for software defect prediction. Zhang et al. (2019) proposed a feature learning framework called UniEmbed for both natural and programming languages by extracting three levels of information, including global, local, and sequential information.

5.2 Studies on multimodal representation learning

Information in the real-world comes through various sources, such as seeing objects, hearing sounds, feeling texture, smelling odors, and tasting flavors. In general terms, modality is the way in which something happens or is experienced (Baltrušaitis et al. 2019). However, each modality (e.g., text, images, or audio) is characterized by unique statistical properties that make it difficult to neglect the truth that they come from different sources (Srivastava and Salakhutdinov 2012). To process and correlate information from multiple modalities, multimodal machine learning has received a great deal of attention (Poria et al. 2016; Xu et al. 2018). Baltrušaitis et al. (2019) identified five core technical challenges surrounding multimodal machine learning: representation learning, translation, alignment, fusion, and co-learning. Here, we mainly introduce multimodal representation learning.

Joint representations project the unimodal representations together into the same representation space. As neural networks have become a prevalent method for unimodal data representation (Bengio et al. 2013), they can be used to construct a joint multimodal representation. Mroueh et al. (2015) presented methods in deep multimodal learning for fusing speech and visual modalities for Audio-Visual Automatic Speech Recognition. Silberer and Lapata (2014) introduced a new model that uses stacked autoencoders to learn higher-level embeddings from textual and visual inputs. Probabilistic graphical models are another popular way to construct representations by using latent random variables (Bengio et al. 2013). For example, multimodal DBMs learned joint representations from multiple modalities by combining multiple undirected graphs (Srivastava and Salakhutdinov 2012). Moreover, for representing varying length sequences, constructing a multimodal representation using RNNs comes from work by Cosi et al. (1994) and Rajagopalan et al. (2016). As an alternative, coordinated representations learn representations for each modality separately, but enforce certain constraints on them to coordinate them to a coordinated space (Baltrušaitis et al. 2019). According to the enforced constraints, similarity models minimize the distance between modalities in the coordinated space. For example, DeViSE (Frome et al. 2013) encouraged the gap between the representation of the word dog and an image of a dog to be less than the distance from an image of a car. While the above mod-

els enforced similarity between representations, structured coordinated space models impose more structure on the resulting space, such as hashing, cross-modal retrieval, and image captioning, based on the application (Cao et al. 2016; Vendrov et al. 2015; Wang et al. 2015b).

6 Conclusions

Machine learning methods are widely used in bug report-based bug localization. In this paper, we propose a deep multimodal model named DeMoB to automatically locate bugs. The proposed DeMoB utilizes the AttL encoder and MDCL encoder to improve the ability to learn the representation of natural language and programming language. Moreover, the DeMoB is capable of merging these representations through multimodal learning. The experimental results demonstrate the effectiveness of the proposed DeMoB which achieves better results over existing bug localization methods. Future work will explore other efficient algorithms for addressing bug localization, such as cross-project and cross-language bug localization tasks.

Acknowledgements This research was supported by Natural Science Foundation of China (No. 61772284), State Key Lab. for Novel Software Technology (KFKT2020B21), and Postgraduate Research and Practice Innovation Program of Jiangsu Province (SJKY19_0763). Hanghang Tong is partially supported by NSF (1947135 and 2003924).

References

Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473

Baltrušaitis T, Ahuja C, Morency LP (2019) Multimodal machine learning: a survey and taxonomy. IEEE Trans Pattern Anal Mach Intell 41(2):423–443

Bengio Y, Courville A, Vincent P (2013) Representation learning: a review and new perspectives. IEEE Trans Pattern Anal Mach Intell 35(8):1798–1828

Cao Y, Long M, Wang J, Yang Q, Yu PS (2016) Deep visual-semantic hashing for cross-modal retrieval. In: Proceedings of ACM SIGKDD international conference on knowledge discovery and data mining (pp 1445–1454). ACM

Cosi P, Caldognetto EM, Vagges K, Mian GA, Contolini M (1994) Bimodal recognition experiments with recurrent neural networks. In: Proceedings of IEEE international conference on acoustics, speech and signal processing (ICASSP) (vol 2, pp II–553). IEEE

DeMillo RA, Pan H, Spafford EH (1997) Failure and fault analysis for software debugging. In: Proceedings of annual international computer software and applications conference (COMPSAC) (pp 515–521). IEEE

Frome A, Corrado GS, Shlens J, Bengio S, Dean J, Mikolov T et al (2013) Devise: a deep visual-semantic embedding model. In: Advances in neural information processing systems, pp 2121–2129

Hoang T, Oentaryo RJ, Le TDB, Lo D (2018) Network-clustered multi-modal bug localization. IEEE Trans Software Eng 45(10):1002–1023

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Huo X, Li M (2017) Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proceedings of international joint conference on artificial intelligence (IJCAI), pp 1909–1915

Huo X, Li M, Zhou ZH (2016) Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of international joint conference on artificial intelligence (IJCAI), pp 1606–1612

Huo X, Yang Y, Li M, Zhan DC (2018) Learning semantic features for software defect prediction by code comments embedding. In: 2018 IEEE international conference on data mining (ICDM) (pp 1049–1054). IEEE

Jie Z, Wang XY, Dan H, Bing X, Lu Z, Hong M (2015) A survey on bug-report analysis. Sci China Inf Sci 58(2):1–24

Kalchbrenner N, Grefenstette E, Blunsom P (2014) A convolutional neural network for modelling sentences. arXiv preprint arXiv:1404.2188

Kim D, Tao Y, Kim S, Zeller A (2013) Where should we fix this bug? A two-phase recommendation model. IEEE Trans Softw Eng 39(11):1597–1610

Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980

Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2017) Bug localization with combination of deep learning and information retrieval. In: Proceedings of international conference on program comprehension (ICPC), pp 218–229

Li W, Li N (2012) A formal semantics for program debugging. Sci China Inf Sci 55(1):133–148

Liu Z, Zhou D, He J (2019) Towards explainable representation of time-evolving graphs via spatial-temporal graph attention networks. In: Proceedings of the 28th ACM international conference on information and knowledge management, pp 2137–2140

Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent Dirichlet allocation. In: Proceedings of working conference on reverse engineering (WCRE), pp 155–164

Marcus A, Sergeyev A, Rajlich V, Maletic JI (2004) An information retrieval approach to concept location in source code. In: Proceedings of working conference on reverse engineering (WCRE), pp 214–223

Mihalcea R, Liu H, Lieberman H (2006) Nlp (natural language processing) for nlp (natural language programming). In: Proceedings of international conference on intelligent text processing and computational linguistics (CICLing) (pp 319–330). Springer

Mroueh Y, Marcheret E, Goel V (2015) Deep multimodal learning for audio-visual speech recognition. In: Proceedings of IEEE international conference on acoustics, speech and signal processing (ICASSP) (pp 2130–2134). IEEE

Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L (2018) Deep contextualized word representations. arXiv preprint arXiv:1802.05365

Poria S, Chaturvedi I, Cambria E, Hussain A (2016) Convolutional mkl based multimodal emotion recognition and sentiment analysis. In: Proceedings of international conference on data mining (ICDM) (pp 439–448). IEEE

Rahman MM, Roy C (2018) Poster: improving bug localization with report quality dynamics and query reformulation. In: Proceedings of IEEE/ACM international conference on software engineering: companion (ICSE-Companion) (pp 348–349). IEEE

Rajagopalan SS, Morency LP, Baltrusaitis T, Goecke R (2016) Extending long short-term memory for multi-view structured learning. In: Proceedings of European conference on computer vision (pp 338–353). Springer

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: Proceedings of IEEE/ACM international conference on automated software engineering (ASE), pp 345–355

Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681

Shi Z, Keung J, Bennin KE, Zhang X (2018) Comparing learning to rank techniques in hybrid bug localization. Appl Soft Comput 62:636–648

Silberer C, Lapata M (2014) Learning grounded meaning representations with autoencoders. Proc Annu Meet Assoc Comput Linguist 1:721–732

Srivastava N, Salakhutdinov RR (2012) Multimodal learning with deep Boltzmann machines. In: Proceedings of advances in neural information processing systems, pp 2222–2230

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(1):1929–1958

Sterling CD, Olsson RA (2007) Automated bug isolation via program chipping. Softw Pract Exp 37(10):1061–1086

Vendrov I, Kiros R, Fidler S, Urtasun R (2015) Order-embeddings of images and language. arXiv preprint arXiv:1511.06361

Wang Q, Parnin C, Orso A (2015a) Evaluating the usefulness of ir-based fault localization techniques. In: Proceedings of international symposium on software testing and analysis (ISSTA) (pp 1–11). ACM

Wang W, Arora R, Livescu K, Bilmes J (2015b) On deep multi-view representation learning. In: Proceedings of international conference on machine learning (ICML), pp 1083–1092

Wang Y, Yao Y, Tong H, Huo X, Li M, Xu F, Lu J (2018) Bug localization via supervised topic modeling. In: 2018 IEEE international conference on data mining (ICDM) (pp 607–616). IEEE

Wong WE, Debroy V (2009) A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Tech Rep UTDCS-45 9

Wong WE, Qi Y (2006) Effective program debugging based on execution slices and inter-block data dependency. J Syst Softw 79(7):891–903

Xiao Y, Keung J, Mi Q, Bennin KE (2017) Improving bug localization with an enhanced convolutional neural network. In: 2017 24th Asia-Pacific software engineering conference (APSEC) (pp 338–347). IEEE

Xiao Y, Keung J, Mi Q, Bennin KE (2018) 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

Xu Y, Biswal S, Deshpande SR, Maher KO, Sun J (2018) Raim: recurrent attentive and intensive model of multimodal patient monitoring data. In: Proceedings of ACM SIGKDD international conference on knowledge discovery and data mining (pp 2565–2573). ACM

Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 689–699

Zhang X, He H, Gupta N, Gupta R (2005) Experimental evaluation of using dynamic slices for fault location. In: Proceedings of international symposium on automated analysis-driven debugging, pp 33–42

Zhang Y, Zheng W, Li M (2019) Learning uniform semantic features for natural language and programming language globally, locally and sequentially. Proc AAAI Conf Artif Intell 33:5845–5852

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of international conference on software engineering (ICSE), pp 14–24