Information and Software Technology xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

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

# Machine translation-based bug localization technique for bridging lexical gap

Yan Xiao*, Jacky Keung, Kwabena E. Bennin, Qing Mi

*Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong*

ARTICLE INFO

*Keywords:*
Bug localization
Deep learning
Machine translation
Lexical mismatch

ABSTRACT

*Context:* The challenge of locating bugs in mostly large-scale software systems has led to the development of bug localization techniques. However, the lexical mismatch between bug reports and source codes degrades the performances of existing information retrieval or machine learning-based approaches.
*Objective:* To bridge the lexical gap and improve the effectiveness of localizing buggy files by leveraging the extracted semantic information from bug reports and source code.
*Method:* We present BugTranslator, a novel deep learning-based machine translation technique composed of an attention-based recurrent neural network (RNN) Encoder-Decoder with long short-term memory cells. One RNN encodes bug reports into several context vectors that are decoded by another RNN into code tokens of buggy files. The technique studies and adopts the relevance between the extracted semantic information from bug reports and source files.
*Results:* The experimental results show that BugTranslator outperforms a current state-of-the-art word embedding technique on three open-source projects with higher MAP and MRR. The results show that BugTranslator can rank actual buggy files at the second or third places on average.
*Conclusion:* BugTranslator distinguishes bug reports and source code into different symbolic classes and then extracts deep semantic similarity and relevance between bug reports and the corresponding buggy files to bridge the lexical gap at its source, thereby further improving the performance of bug localization.

## 1. Introduction and motivation

The high cost of manual *bug localization*, especially for large software systems, has instigated the design of automated techniques to help developers prioritize and focus on potentially buggy files based on bug reports. However, bug reports are written in natural language, whereas source files are represented by code tokens. The differences between them in expression and representation lead to a *lexical mismatch* problem, which stifles the effectiveness and accuracy of proposed bug localization techniques in detecting buggy files [5,8,9].

To improve the accuracy of bug localization, recent techniques [5,8] include the similarity between bug reports and application programming interface (API) entities (class and interface names) to bridge the lexical gap. Ye et al. [9] applied word embedding (WE) to obtain word vectors of bug reports and source code in a shared representation space. These approaches regard the code tokens in source files as the same natural languages used in bug reports, which fails to effectively suppress the effects of lexical mismatch on bug localization.

To address the above issue of lexical mismatch and thereby further improve the performance of bug localization, we distinguish bug reports and source files into different symbolic classes and formulate the bug localization problem as a machine translation problem. For example, during the machine translation process of an English sentence into a French sentence, the two sentences are represented in different languages (symbols), but they represent similar meaning. Likewise, the pairs of API description and API sequence denote similar operations by different representations. Significantly, a machine translation technique achieves outstanding performance in the generation of API sequences given a natural language query [3]. Motivated by this, we propose a novel bug localization model, BugTranslator, based on a recurrent neural network (RNN) Encoder-Decoder with long short-term memory (LSTM) cells by absorbing useful modules from famous machine translation models [1,2,7].

The main contributions of this paper are:

- To the best of our knowledge, we are the first to introduce the machine translation technique to the area of bug localization and to propose a novel method of bridging the lexical gap radically.

* Corresponding author.
*E-mail addresses:* yanxiao6-c@my.cityu.edu.hk (Y. Xiao), Jacky.Keung@cityu.edu.hk (J. Keung), kebennin2-c@my.cityu.edu.hk (K.E. Bennin), Qing.Mi@my.cityu.edu.hk (Q. Mi).

https://doi.org/10.1016/j.infsof.2018.03.003
Received 22 August 2017; Received in revised form 1 March 2018; Accepted 1 March 2018

- Empirically validate the effectiveness of BugTranslator in overcoming the lexical mismatch challenge.

## 2. Lexical mismatch in bug localization

Lexical mismatch means that a similar meaning can be expressed by different vocabulary or languages. In the field of bug localization, bug reports and source code represent similar operations with different expressions. This lexical mismatch challenge also limits the performance of existing bug localization techniques [5,8].

Lam et al. [5] attempted to bridge the lexical gap by combining deep neural networks (DNNs) with information retrieval techniques. However, their experimental results showed that DNNs without information retrieval techniques achieve very poor performance. The WE method was used by Ye et al. [9] to obtain document similarities as two new features added into their previously proposed linear learning-to-rank model (LR) [8]. The natural language in bug reports and code snippets in source files were projected by the WE method into vectors. Their model contained the semantic similarity between the two bags-of-words of bug reports and source codes.

Significantly, the approaches that attempted to bridge the lexical gap were experimentally validated to outperform those that ignored the lexical mismatch, which also revealed the existing challenge caused by lexical mismatch.

## 3. BugTranslator

In this section, we describe the proposed BugTranslator model in detail.

### 3.1. Generating training instances

We first prepare the training set for BugTranslator: API documents, project-specific documents, and older bug reports with corresponding buggy files. We attempt to translate bug reports into corresponding buggy files based on the deep semantic similarity and relevance between them. Thus, the first training instances are the pairs of older bug reports and abstract syntax tree (AST) nodes parsed from corresponding buggy files. During testing, some out-of-vocabulary words never appear in older bug reports and their corresponding buggy files, and this is known to decrease the accuracy of most translation models [2]. In addition to older bug reports and corresponding buggy files, API documents and project-specific documents are included in the training set to enrich the vocabulary and detect some comprehensive information.

The API annotations and corresponding API sequences from API documents in Java SE 7 are extracted as noted in the literature [3]. The source code is parsed into AST nodes that include field declarations and type bindings of all classes and methods. In addition, the method-level code summaries are extracted as corresponding annotations. The project-specific documents are also included in addition to the API documents that are generally invoked by all projects. Paired with annotations of classes, methods, and fields, the source code is parsed into AST nodes of declarations, method invocations, and class instance creations.

### 3.2. Attention-based RNN encoder-decoder with LSTM cells

To learn how to translate natural languages into code tokens, we build an attention-based RNN Encoder-Decoder model with LSTM cells. The workflow is shown in Fig. 1, which illustrates an example of translating the natural language term *audio file player* into a sequence of code tokens. The source sentences are first encoded into several context vectors from which the decoder generates target sentences. The context vectors are the bridge between the source sentences and the target sentences.

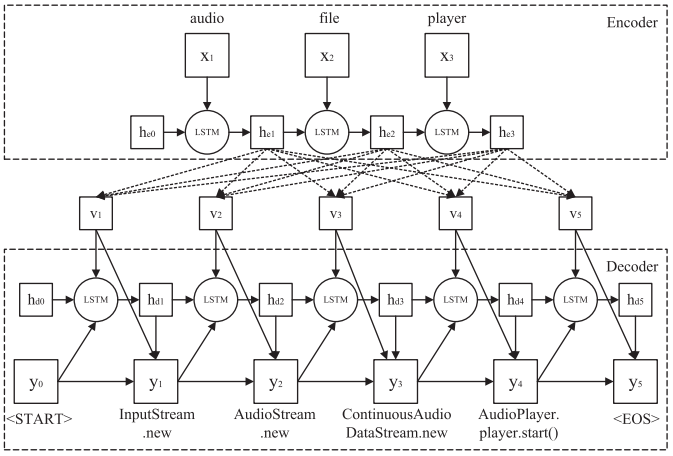


**Fig. 1.** Overall workflow of attention-based RNN Encoder-Decoder with LSTM cells.

#### 3.2.1. Encoder RNN

The source sentences and target sentences are first embedded into 1-of-K (K is the vocabulary size)-coded word vectors [1], $X = (x_1, ...,x_i, ...,x_S)$ and $Y = (y_1, ...,y_j, ...,y_T)$ respectively, where $S$ and $T$ represent the lengths of the source and target sentences. The Encoder first reads the coded word vector $x_1$ embedded by the first word *audio* and then computes the current hidden state $h_{e1}$ by $h_{e0}$ and $x_1$ according to Eq. (1). The initial hidden state $h_{e0}$ is set to 0. The second hidden state $h_{e2}$ is then updated by $h_{e1}$ and word vector $x_2$ of the second word. This process continues until the last hidden state $h_{e3}$ is updated by (1). At each time $t$, the hidden state is updated by:

$$h_{et} = LSTM\,(h_{e(t-1)}, x_t) \tag{1}$$

It has been shown empirically that LSTM works well on machine translation of long sentences [7]. Because bug reports tend to include long sentences, we use RNN with LSTM cells.

In practice, each word in the source sentences has different importance to the word in the target sentences. It is inappropriate to encode the entire source sentence into only one context vector, which has also been verified experimentally in the literature [1]. Therefore, in this paper, the context vector $v_j$ at each step is expressed by the weighted sum of the hidden states of the encoder as discussed in [1].

#### 3.2.2. Decoder RNN

The Decoder is another RNN that is trained to generate the target sentence sequentially based on the context vectors obtained from the encoder RNN. The first word $y_0$ is set as $< START >$, and the initial hidden state $h_{d0}$ is calculated by $h_{d0} = \tanh(W_d h_{e1})$, where $W_d$ is the weight that can be learned during training and $h_{e1}$ is computed by Eq. (1). The Decoder then computes the hidden state $h_{d1}$ using $h_{d0}$, $y_0$, and the context vector $v_1$ by Eq. (2), followed by prediction of the first word *InputStream.new*.

The hidden state $h_{dt}$ at time $t$ is computed by:

$$h_{dt} = LSTM\,(h_{d(t-1)}, y_{t-1}, v_t) \tag{2}$$

The conditional probability of $y_t$ given the previous predicted words and context vector is defined as:

$$p(y_t|y_{t-1}, y_{t-2},...,y_1, v_t) = g(h_{dt}, y_{t-1}, v_t) \tag{3}$$

where $g$ is a *softmax* activation function.

This process continues until the end-of-sentence word $< EOS >$ is predicted.

The Decoder defines a probability over the target sentence $Y$ as:

$$p(Y) = \prod_{t=1}^{T} p(y_t|y_{t-1}, y_{t-2},...,y_1, v_t) \tag{4}$$

The two RNNs are then trained jointly to maximize the following