Information and Software Technology 99 (2018) 58–61

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
Available online 06 March 2018
0950-5849/

- 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.

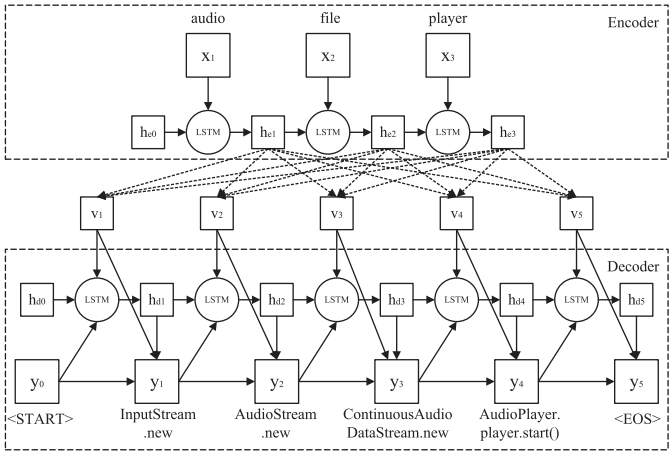

Fig. 1. Overall workflow of attention-based RNN Encoder-Decoder with LSTM cells.

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

conditional likelihood:

$$L(\theta) = \frac{1}{N}\sum_{n=1}^{N} log\,(p_{\theta}(Y_n|X_n)) \quad (5)$$

where (X_n, Y_n) is each pair (a source sentence and a target sentence) in each batch of the training set, and θ is the set of parameters learned during training.

3.3. Translation and generation of scores

After creating BugTranslator, the minibatch stochastic gradient descent algorithm [6] is adopted to train the model to maximize the likelihood in Eq. (5). Given a new bug report, the trained BugTranslator can be used to score the pairs of the bug report and each source file using the probability $p_{\theta}(Y_n|X_n)$ from Eqs. (4) and (5).

4. Experiments

4.1. Preparation of experiments

Experimental settings: When creating the BugTranslator model, the number of hidden cells in both RNNs is set as 1000. To accelerate the training phase, we limit the vocabulary size of the source and target to 20,000.

Datasets: We use the before-fixed version of three open-source Java projects (Eclipse UI, JDT, and SWT) to evaluate the performance of BugTranslator. To make the comparison with existing techniques easier, a strategy similar to that in [9] is adopted. The oldest 1500 bug reports of each project are used for tuning the model while the older 500 bug reports are for training the model and the remaining newest bug reports (1,656, 632, 817 bug reports respectively for Project Eclipse UI, JDT, SWT) are for testing the model. In addition to the aforementioned bug reports, we also collect corpus from Java SE 7 API Reference and project-specific documents.

Evaluation metrics: The Accuracy@k, mean average precision (MAP) and mean reciprocal rank (MRR) are used to evaluate the performance of BugTranslator [5,8,9].

Competitors: Ye et al. [9] enhanced their previously proposed LR [8] with WE, and their results were comparable with the DNNLOC model [5], which outperformed other models [4,8,10]. Therefore, this paper compares the performance of BugTranslator with WE and LR + WE.

4.2. Experimental results and discussions

Table 1 shows the MAP and MRR results of four models, as well as the values of Accuracy@5. It can be observed that the average Accuracy@5 of BugTranslator is about 56.6%. That is to say, BugTranslator can correctly locate buggy files for about 56.6% of the bug reports when recommending five source files to developers. We observe that BugTranslator achieves higher MAP and MRR values than the WE method. The WE method learns the semantic information from bug reports and the source code based on word embedding and calculates the similarity between them. However, if some word pairs never appear in the same context but are relevant to each other, word embedding has a lower probability of assigning them as close vectors, which limits the performance of WE. Moreover, because word embedding focuses on words more than sentences, it is difficult for WE to learn the relative positions between words, which can be learned by BugTranslator by benefiting from the two RNNs with memory capacity. In other words, BugTranslator learns the semantic information from not only words but also the full sentence. It can thus alleviate the effect of lexical mismatch based on the deeper understanding of semantics between bug reports and source code by benefiting from the context vectors between the two RNNs.

Table 1
Results of four models.

Project	Metrics	WE	LR + WE	BugTranslator	LR + BugTranslator
Eclipse UI	MAP	0.26	0.40	0.36	0.41
	MRR	0.31	0.46	0.42	0.48
	Accuracy@5	47.6	60.2	58.1	61.4
JDT	MAP	0.22	0.42	0.34	0.45
	MRR	0.27	0.51	0.41	0.52
	Accuracy@5	45.6	62.3	55.4	63.1
SWT	MAP	0.25	0.38	0.34	0.38
	MRR	0.30	0.45	0.40	0.46
	Accuracy@5	46.2	59.1	56.2	59.9

According to Table 1, both WE and BugTranslator perform worse than LR + WE. In addition to the semantic similarity calculated by the WE method, LR + WE also considers factors such as bug-fixing recency, frequency, and the observation that a previously fixed file may be responsible for similar bugs. These factors are concluded based on the experience of many developers, which is not learned by BugTranslator directly. We also combine BugTranslator with LR in emulation of the approach LR + WE [9]. The observed results of LR + BugTranslator are comparable to those of LR + WE, indicating the effectiveness of BugTranslator. Then we talk about two examples about the ranking results. There are two bug reports in Project SWT: "Bug 369228 Kill pre GTK 2.4 leftovers from Tree (buggy files: *Tree.java, Table.java, List.java, Display.java*, and *Widget. java*)" and "Bug 369227 Kill pre GTK 2.4 leftovers from List (buggy files: *List.java*)". For Bug 369228, the buggy file *Tree.java* was ranked the first by BugTranslator because of related AST nodes (e.g., the method name). However, it did not give the other buggy files (*Table.java, List.java, Display.java*, and *Widget.java*) very high ranks (4th, 6th, 11th and 25th), whereas LR + BugTranslator gave these buggy files higher ranks (5th, 3rd, 7th and 20th). It seems that LR + BugTranslator outperformed BugTranslator. However, for Bug 369227, LR + BugTranslator located and ranked *List.java* 3rd. *Display.java* and *Table.java* were ranked 1st and 2nd because of their greater frequency and earlier recency considered in LR, which instead had a negative effect on the results. But BugTranslator located the buggy file *List.java* and ranked it first, which outperformed LR + BugTranslator.

4.3. Why can BugTranslator bridge the lexical gap?

Lexical mismatch is caused by the difference in the expressions of bug reports and source code files. Existing techniques [5,8,9] regard both in the same language expression. These techniques measure their similarities, which does not solve the root cause of the lexical mismatch problem. Unlike existing approaches, our proposed BugTranslator makes a distinction between bug reports and source code based on the RNN Encoder-Decoder model. The bug reports are encoded into the context vectors from which the source code files are decoded. They are placed in different language platforms, and context vectors are used to bridge them based on the semantic relevance between them. As observed from our experimental results, we argue that BugTranslator can handle and alleviate the issue of lexical mismatch in a radical way.

5. Conclusions and future work

The lexical mismatch between source code files and bug reports degrades the performance of the existing bug localization techniques [9]. This paper proposed a new translation model, BugTranslator, for bug localization, which differs from existing techniques to alleviate the effect of lexical mismatch in a fundamental manner based on the attention-based RNN Encoder-Decoder with LSTM cells. The experimental results show that BugTranslator performs better than the existing single model. When combined with existing models, as with other combined models, BugTranslator achieves comparable results. In a future study,

we will enhance and automate BugTranslator by including the missing factors. Furthermore, RNN with LSTM cells has been noted to be sensitive to word order. Localizing buggy classes or methods instead of buggy files may produce more benefit to our model. We will also investigate this in a future study.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong [No. 11208017], and the research funds of City University of Hong Kong [No. 7004683].

References

[1] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv:1409.0473 (2014).
[2] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using rnn encoder-decoder for statistical machine translation, arXiv:1406.1078 (2014).
[3] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 631–642.
[4] 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.
[5] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Bug localization with combination of deep learning and information retrieval, Proceedings of the 25th International Conference on Program Comprehension, IEEE Press, 2017, pp. 218–229.
[6] M. Li, T. Zhang, Y. Chen, A.J. Smola, Efficient mini-batch training for stochastic optimization, Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2014, pp. 661–670.
[7] I. Sutskever, O. Vinyals, Q.V. Le, Sequence to sequence learning with neural networks, Advances in neural information processing systems, (2014), pp. 3104–3112.
[8] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 689–699.
[9] X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, From word embeddings to document similarities for improved information retrieval in software engineering, Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 404–415.
[10] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports, Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 14–24.