Source Code Summarization with Pre-trained CodeBERT Model and Adversarial Data Augmentation

Adi Apotheker Tovi and Yevgenia Shteynman
`{ sadie941 , jenny075 }@campus.technion.ac.il`

Abstract

Nowadays, large scale development involves many source code repositories from different contributors. For many developers, reading others code is challenging and time consuming, especially when the code does not have proper documentation. One of the solutions is source code summarization, which is a generation of readable summaries that describe the functionality of a program. In this work, we use a pre-trained model called CodeBERT with transformer-based architecture, which is the current state-of-the-art for this task. We propose a simple, but effective, method to improve the robustness and the performances by training the model using data augmentation. We have made our code publicly available [1] to facilitate future research.

1 Introduction

Generating a readable summary that describes the functionality of a program is known as source code summarization. With the advancement of deep learning and the availability of large-scale data through a vast number of open-source repositories, automatic source code summarizing has drawn attention from researchers (Ahmad et al., 2020). Most of the neural network approaches generate source code summaries in a sequence-to-sequence fashion. There are works which used the traditional RNN-based sequence-to-sequence network with attention mechanism on different abstractions of code (Luong et al., 2015; Hu et al., 2018). The RNN-based sequence models have two main limitations in learning source code representations. They do not model the non-sequential structure of source code as they process the code tokens sequentially. Also, source code can be very long, and thus RNN-based models might fail to capture the long-range dependencies between the code tokens (Ahmad et al., 2020). On contrary to RNN-based models, attention based models which leverage self-attention mechanism, can capture long-range dependencies (Shaw et al., 2018). Moreover, transformers as well as using large scale pret-rained contextual embeddings have been shown to perform well on many natural language generation tasks (Farahani et al., 2021; Wang et al., 2020; Feng et al., 2020). Our approach is combining the pret-rained CodeBERT model which achieves state-of-the-art performance on code documentation generation (Feng et al., 2020) with data augmentation methods.This approach allow to enrich the dataset and create a robust model that can also handle semantic-preserving source code transformations, without changing the functionality of the program (Ramakrishnan et al., 2020).

[1] https://github.com/adiap94/CodeDescription

2 Relative work

Large pre-trained contextual embeddings for natural languages have shown dramatic improvements on various natural language processing tasks (Vaswani et al., 2017; Devlin et al., 2019).One of the most representative neural architectures is the Transformer. It contains multiple self-attention layers, and can be conventionally learned with gradient decent in an end-to-end manner as every component is differentiable (Vaswani et al., 2017). BERT improved natural language pre-training by using a de-noising autoencoder. Instead of learning a language model, which is inherently sequential, BERT optimizes for predicting a noised word within a sentence. Such prediction instances are generated by choosing a word position and either keeping it unchanged, removing the word, or replacing the word with a random wrong word. It also pre-trains with the objective of predicting whether two sentences can be next to each other. These pre-training objectives, along with the use of a Transformer-based architecture, gave BERT an accuracy boost in a number of NLP tasks over the state-of-the-art (Devlin et al., 2019). CodeBERT which a bimodal pre-trained model, targets paired natural language (NL) and multi-lingual programming-language (PL) tasks, such as code search and generation of code documentation. It pre-trains a Transformer encoder by treating a natural-language description of a function and its body as separate sentences in the sentence-pair representation of BERT. Results show that CodeBERT achieves state-of-the-art performance on both natural language code search and code documentation generation.(Feng et al., 2020)

Data augmentation (DA) is an explicit form of regularization that is also widely used in the training of deep neural networks in different areas such as images and acoustic domains .(Rebai et al., 2017; Shorten and Khoshgoftaar, 2019). In NLP, where the input space is discrete,how to generate effective augmented examples that capture the desired invariances is less obvious, and hence less popular(Feng et al., 2021). In recent years, DA in NLP gained more attention due to lack of data and generation tasks. For example, learning a robust deep code comment generation models that can maintain the performance of models under augmented inputs.Zhang et al. (Zhang et al., 2020) employed the Metropolis-HASTINGS (MHM) algorithm to rename the code identifiers.In contrary, The work of Ramakrishnan et al. (Ramakrishnan et al., 2020) performed semantic-preserving transformations such as renaming input and local parameters inside the source code without changing the behaviour of the code.

3 Dataset

The dataset we used is CodeSearchNet dataset (Husain et al., 2019) , which includes several millions of codes across six programming languages (Python,PHP,Go,Java,JavaScript and Ruby). The origin of the data are from publicly available open source non-fork GitHub repositories and are filtered with a set of constraints and rules. These consisted of: (1) Each project should be used by at least one other project. (2) Each documentation is truncated to the first paragraph. (3) Documentations shorter than three tokens are removed. (4) Functions shorter than three lines are removed. (5) Function names with substring “test” are removed

In our work we used a "clean" dataset version of CodeSearchNet used in CodeBERT (Feng et al., 2020). This version include additional filtrations such as removing comments in the code, removing examples where the codes couldn’t be parsed into an abstract syntax tree (AST), Removing examples that didn’t had documentation in English, removing examples that had documentation tokens longer than 256. Each example has a raw data structure which includes various fields and metadata. Our focus includes the raw string before tokenization or parsing, tokenized version of code and tokenized version of docstring.

All the examples were divided into train,valid and test datasets. Statistics of the "clean" version python dataset is shown in Table 1 .

3.1 Data Augmentation

Our approach includes data augmentation over the original data in order to increase model robustness and performance. For this task, we adapted AVERLOC framework for python (Ramakrishnan et al., 2020) in order to apply adversarial transformation on code P into semantically equivalent code P′ that fools in the beginning the neural network into making a wrong code documentation.

This stage consists of two steps:Transform and Resolve. The First step creates a sketch (Solar-Lezama et al., 2006) of the original code, marking variable, parameters and additional code attributes as a preparation for semantic code change. The next step is the resolver which given a program sketch, resolves the input sketch into a complete program. It changes the marks with a random token generated by selecting and concatenating a random number of sub-tokens from a chosen fixed vocabulary. We used the vocabulary of CodeBERT model. We utilized 4 transformations on our data and demonstrate them on the following code:

```
def size(self, s):
    leader = self.find(s)
    return self._size[leader]
```

- AddDeadCode: A dead-code statement usually appended to the beginning or the end of program which has no logical effect since it consists an if statement with always "False" expression.

```
def size(self, s):
    if false :
        s = 1
    leader = self.find(s)
    return self._size[leader]
```

PL	Training	Dev	Test
Clean Python	251,820	13,914	14,918
AddDeadCode	171,471	9,507	9,985
RenameLocalVariables	135,097	7,432	7,751
RenameParameters	143,466	8,154	8,356
RenameFields	74,043	4,024	4,466

Table 1: Data distribution of the clean data and each of the transforms

- RenameLocalVariables: Random selection of a single local variable in the code, has its name replaced with different token.

```
def size(self, s):
    ptypeid = self.find(s)
    return self._size[ptypeid]
```

- RenameParameters: Random selection of a single input parameter in the code, has its name replaced with different token.

```
def size(self, ur):
    leader = self.find(ur)
    return self._size[leader]
```

- RenameFields: Random selection of a single referenced field in the code, has its name replaced with different token.

```
def size(self, s):
    leader = self.find(s)
    return self.nam[leader]
```

Nevertheless, the different transformations require minimal conditions in order to apply them on the code. For example, not all functions have input parameters and hence RenameParameters transform will not change the code. Consequently, generating the data augmentation for the different transforms yields non-equal datasets. The distribution of the resulting augmented examples presented in Table 1.

4 Model architecture

We followed CodeBERT architecture (Feng et al., 2020) ,which followed BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019) and used multi layer bidirectional Transformer (Vaswani et al., 2017). The CodeBERT architecture is exactly the same model as RoBERTa's, therefore will not be described. For the experiments, we used encoder decoder,where the encoder of a generative model was initialized with CodeBERT and the decoder was a Transformer with 6 layers, 768 dimensional hidden states and 12 attention heads.

4.1 Pre-Training Input/Output presentation

The input is a concatenation of two segments with a separator token (CLS), which considered as the aggregated sequence representation for classification or ranking. One segment is natural language text (NL), and another is code from a programming language (PL). The NL text was processed as sequence of words, and split it as WordPiece (Wu et al., 2016) ,while the code is a sequence of tokens.

The output includes both contextual vector representation of each token, for both natural language and code, and a representation of separator token, which works as the aggregated sequence representation.

4.2 Objective

In CodeBERT there are two objectives : one is masked language modeling (MLM), which also used in other main works (Devlin et al., 2019; Liu et al., 2019). Given a datapoint of NL-PL pair as an input, we first select a random set of positions for both NL and PL to mask out , and then replace the selected positions with a special mask token. The MLM objective is to predict the original tokens which are masked out.

The second is replaced token detection (RTD),(Clark et al., 2020) This version uses both bimodal and unimodal data for training. There are two data generators, an NL generator and a PL generator , both for generating plausible alternatives for the set of randomly masked positions. The discriminator is trained to determine whether a word is the original one or not, which is a binary classification problem. It is worth noting that the RTD objective is applied to every position in the input.When the generator produce the correct token, the label of that token is “real” instead of “fake”.

Results

In our research, we used the pre-trained CodeBERT model with same architecture and objective as CodeBERT, but trained it with four different data augmentation methods: RenameParameters, RenameLocalVariables, RenameFields and AddDeadCode as described in subsection 3.1 .Those augmentation methods, increase significantly the training

Model/Ground Truth	Code documentation
Ground Truth	Returns the number of elements in the set that s belongs to .
CodeBERT restore	Return the size of a string .
CodeBERT AddDeadCode	Convert a string to a string
CodeBERT RenameLocalVariables	Return the size of a string .
CodeBERT RenameParameters	Return the size of a string
CodeBERT RenameFields	Return the size of the given string .

Table 2: Example's output for the same original source code sample presented in subsection 3.1 for the different models.

samples. In addition to those four data augmentated based models we also train the original CodeBERT model with the original parameters ,except for the batch size which was reduced from 64 to 10 samples per batch due to resources limit. We use the same parameters for all our five experiments.

We use the BLEU-4 metric to evalute the Code-to-Documentation generation performances of the different models. BLEU computes the n-gram overlap between a generated sequence and a collection of references (Papineni et al., 2002).

In Table 3 we summarize the BLEU-4 scores all our five experiments , and the metric score of the original CodeBERT as stated in their paper. The BLEU-4 metric is calculated in all the models on the same Clean Python test set as mentioned in Table 1. The BLEU-4 value of the restored CodeBERT model is close to the original score which indicate we succeeded in the model restoration process and yet, we get lower metric value. We assume it might be related to the bigger batch size they used which might cause less noisy gradient. Moreover, our two data augmented models: RenameParameters and RenameLocalVariables get higher BLEU-4 metric than our CodeBERT restore model and even higher than the CodeBERT original. This indicate those two augmentation methods can create a more robust model. The last augmentation method AddDeadCode gets significantly lower score than all other models, we can deduce that this method does not contribute the model and even confuses it by adding non relevant code which does not affect the function description.

In Table 2, we demonstrate the different generated code-documentations outputs, by applying the models on the same source code data example presented insubsection 3.1 with respect to Ground Truth documentation.

Model	BLEU-4
CodeBERT original	19.06
CodeBERT restore	18.13
CodeBERT AddDeadCode	12.30
CodeBERT RenameLocalVariables	**19.16**
CodeBERT RenameParameters	**19.18**
CodeBERT RenameFields	19.01

Table 3: Results on Code-to-Documentation generation, evaluated with BLEU-4 score.

Conclusion

In our research, we contribute to CodeBERT which is the state of the are in source code summarization task. We propose a simple, but effective method to improve the robustness by training the model with data augmentation. We create different source codes by renaming parameters and local variables without changing the functionality of the program, thus the original source code summary remains. According to our experiments, the models with those two data augmentation methods, get higher BLUE-4 score than our restore model of CodeBERT and even the official score in their paper. Those results leads us to believe that this could be a potential direction for further research on this field. First, we can explore the influence of other augmentation methods and second investigate combinations of several data augmentation methods together in the same model.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. pages 4998–5007.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: pre-training text encoders as discriminators rather than generators. *CoRR*, abs/2003.10555.

Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT 2019 - 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf.*, 1(Mlm):4171–4186.

Mehrdad Farahani, Mohammad Gharachorloo, Marzieh Farahani, and Mohammad Manthouri. 2021. ParsBERT: Transformer-based Model for Persian Language Understanding. *Neural Process. Lett.*, 53(6):3831–3847.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard H. Hovy. 2021. A survey of data augmentation approaches for NLP. *CoRR*, abs/2105.03075.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. *Find. Assoc. Comput. Linguist. Find. ACL EMNLP 2020*, pages 1536–1547.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. *IJCAI Int. Jt. Conf. Artif. Intell.*, 2018-July:2269–2275.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Minh Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. *Conf. Proc. - EMNLP 2015 Conf. Empir. Methods Nat. Lang. Process.*, pages 1412–1421.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2020. Semantic robustness of models of source code. *CoRR*, abs/2002.03043.

Ilyes Rebai, Yessine Benayed, Walid Mahdi, and Jean Pierre Lorré. 2017. Improving speech recognition using data augmentation and acoustic model fusion. *Procedia Comput. Sci.*, 112:316–322.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *CoRR*, abs/1803.02155.

Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. *J. Big Data*.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Adv. Neural Inf. Process. Syst.*, 2017-Decem(Nips):5999–6009.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. 2020. Learning deep transformer models for machine translation. *ACL 2019 - 57th Annu. Meet. Assoc. Comput. Linguist. Proc. Conf.*, pages 1810–1822.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation.

Xiaoqing Zhang, Yu Zhou, Tingting Han, and Taolue Chen. 2020. Training Deep Code Comment Generation Models via Data Augmentation. *ACM Int. Conf. Proceeding Ser.*, pages 185–188.