Linguistic Approach to Segmenting Source Code

Aviel J. Stein, Daniel Schwartz, Yiwen Shi, Spiros Mancoridis
College of Computing and Informatics
Drexel University, Philadelphia, Pennsylvania 19144
Email: ajs568@drexel.edu

***Abstract*—Source code segmentation is the process of dividing the source code of a program into meaningful pieces, such as in preparation for source code analysis (SCA) tasks. Our goal is to segment code based on the semantics of its content. Specifically such that the segments reflect logical locations that are good candidates for the insertion of manually composed comments or automatically generated comments. Instead of focusing on syntactic boundaries for segmentation, such as function and class declarations, we exploit the semantic content of the code. We use code snippets mined from Github as known semantic segments to train a LSTM Neural Network model. It is able to infer locations in the code where a programmer would likely insert comments. The model can operate on any text and performs well across multiple programming languages for detecting candidate segment boundaries within a program. This semantic code segmentation is especially useful for incomplete code repositories under development, which may be also written in more than one programming language. Additionally, our technique supports a detection threshold parameter so users can adjust the number of suggestions provided by our tool.**

Key words: Deep Learning, Natural Language, Big Data, Source Code Analysis, Segmentation

I. Introduction

Comments in source code help developers document and communicate important information about a program. While frequent and descriptive comments are encouraged as a best practice, they take time to write. Thus, the task of writing comments is often neglected and old comments may become defunct or misleading as the source code changes. Automatically generated comments could keep documentation up to date but knowing where to add comments is hard to automate in long code files. Code segmentation is a technique that can determine the boundaries of source code segments that are semantically cohesive. These boundaries provide good candidate locations in the source code to insert a manually composed comment or a automatically generated comment. We create models for segmenting code into short, medium, and long code snippets. We refine the code segmentation method of [1]. Our enhancements to this method include using a new source corpus for training, NLP enhanced preprocessing methods, and scope level target models as follows: (1) we use a natural language processing (NLP) text tokenizer, called Sentencepiece [2], which is a tool that learns what is a token without the use of formal regular expressions; thus, enabling its users to parse source code using tokens without processing the text on a per character basis making tokens more informative and condensed; (2) we use the accessible and vast "Source Code Analysis Dataset" (SCAD) [3], which consists of code-comment pairs mined from reputable Github repositories instead of using Stack Overflow snippets; and, (3) we use several granularities of segments for different levels of comment specification. Our source code segmentation pipeline is capable of operating on arbitrarily long source code files. Additionally, it can return a variable number of potential segments by changing a configurable threshold for more or fewer source code segment recommendations.

II. Data and Prepossessing

We choose to use the "Source Code Analysis Dataset" (SCAD), which is a dataset of code-comment pairs mined from 108,568 projects downloaded from Github that have a redistributable license and at least 10 stars in C, C++, Java, and Python. We limit our samples to those with more than four and fewer than twenty lines of code. We create 9 datasets of code snippets representing different languages and by length as measured by the number of new lines. The short data set includes snippets with 1 to 5 lines of code, the medium snippets have 3 to 10 lines of code, and the long code snippets have 10-20 lines of code. We also limited the text and comment length to be less than 500 and 250 characters, resulting in millions snippets samples from each language.

To extend the NLP analogy to the problem of source code segmentation, we are able to take advantage of an NLP tokenizer for our code. We use Sentencepiece [2], a language-independent sub-word tokenizer designed for Neural-based text processing, from Google. As code is written using a programming language, our tokenizer pays special attention to spaces, tabs, and new lines which are "SPACE", "TAB", and "NEWLINE" tokens, respectively. We use the entire SCAD corpus to train the tokenizer. This creates a vocabulary which we limit to 10,000 words. While there are many more words in the corpus, this vocabulary contains sufficient sub-words to parse any text it receives. This method is useful for large vocabularies with the potential of unknown words and makes source code processing more robust.

We generated blocks of code from sample segments within SCAD. These generated multi-segment code (MSC) blocks are our ground truth for training. Each MSC consists of three segments from the dataset. We then tokenize each MSC with the custom tokenizer and label the beginning of each segment with a 1 and all others with a 0. This process has the advantage of avoiding the inclusion of a line break between segments as in [3]. Each MSC consists of segment samples from SCAD from the same language so there are no segments with multiple languages within one MSC. We do this to avoid our model learning to separate between languages since this is unlikely in real world code. We have a vast amount of data to use but only need a small fraction to achieve competitive results. We generate 10,000 MSCs for each language and at each length. From this we use 5000 for training, 1,500 for validation, and 1,500 for testing. To train the multi-lingual model, we use 2,500 from each language, 1500 for training, 500 for validation, and 500 for testing.

III. Learning Model

Once we have reprocessed the data we can train our neural network. Recurrent neural networks, and in particular bidirectional long short term memory (LSTM) [4] models have been shown to be effective for NLP tasks [5], [6]. We chose to use a LSTM model because it is effective for the sequence-to-sequence data we have generated. We built our model with the PyTorch library. We chose to focus on classifying the center token and use the 20 tokens before and after as context, resulting in a window size of 41 tokens, compared to the 100 character window size of [1]. We label each of the tokens in the code as either a break point (1) or not a break point (0) and we choose to use Sentencepiece to learn code tokens that can be used to parse the code. We used a sliding window with a center token and twenty tokens before and twenty tokens after. If a window included an area outside the code tokens, we replaced them with the vocabulary data token (unk). We randomly sample one window from each MSC as training for our model which may include one, two, or no breakpoints. This is desirable because it means that it is more likely to generalize when working on programs of arbitrary length. The LSTM model had a hidden dimension of 200, 2-LSTM layers, a dropout of 0.05, learning rate of 0.0005, and clipping threshold of 0.25. We batched the data in sets of 50 samples windows.

IV. Results

We trained the LSTM until there was no improvement in the validation loss for 5 consecutive epochs. Models typically required between 60-80 epochs to complete training or about 30-40 minutes on a personal computer. Also, when evaluating the accuracy and ROC, we only consider NEWLINE tokens, since breakpoints are always known to be at a NEWLINE (NL) token. Java consistently scored above 99% in NL accuracy, Python scored above 98% NL accuracy, but C++ did not perform as well. It scored around 90% NL accuracy and 96% for long segments, which may be in part due to a smaller portion of NL which represent a point of segmentation. It is also worth noting that this data has far more tokens and even NL that do not indicate a break point than do, but the ROC show that the results are balanced between True and False positive results are still significant.

We also train a multi-lingual model that is capable of segmenting C++, Java, and Python indiscriminately. This suggests that its easy to use a tokenizer and model across most common code, making it more viable for code found in the wild. The multi-lingual model scored between 95-97% and was limited to similar constraints as the single language models, (*i.e.,* same vocabulary size and training data, actually a bit less).

V. Discussion

Regarding source code segmentation, we found that the LSTM presented with a byte-pair encoding tokenizer worked well for segmenting code. It offered several advantages including a sliding window, fuzzy parsing, multi lingual and language agnostic properties, and a customizable threshold for more or less granular comments. A sliding window is important for being able to evaluate the code of long programs. The fuzzy parsing means we can analyze code even if the code is not correctly written (*i.e.,* the source code contains syntax errors). Since byte-pair encoding is a compression algorithm, it can easily be adapted to work on entire documents, regardless of the programming language. And finally, we were able to change the value of our threshold for comment insertion to add fewer or more comments.

VI. Conclusion

Our method for segmenting source code accounts for its semantic content by leveraging NLP tools for learning where comments are recommended. These methods can operate on any text and may be trained as a multi-lingual model. We segment source code files into functionally distinct snippets by training on a generated dataset of multi-segment code blocks that have been processed with a Sentencepeice tokenizer. This model is capable of being used off the shelf for recommending comment locations on full multi-lingual code repositories common in places such as Github and different levels of context. In addition to recommending locations for comments, it can be used in down stream SCA tasks such as paraphrase discovery and comment generation. Even though we used less than 5% of available data, we achieve high accuracy for Python and Java. C++ was harder to learn but could be resolved by training with more data. Additionally the multi-lingual model scored higher than 95% on test data across all levels of segment size.

Acknowledgment

This research was funded by the Auerbach Berger Chair in Cybersecurity held by Spiros Mancoridis.

References

[1] J. Dormuth, B. Gelman, J. Moore, and D. Slater, “Logical segmentation of source code,” *arXiv preprint arXiv:1907.08615*, 2019.

[2] T. Kudo and J. Richardson, “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing,” *arXiv preprint arXiv:1808.06226*, 2018.

[3] B. Gelman, B. Obayomi, J. Moore, and D. Slater, “Source code analysis dataset,” *Data in brief*, vol. 27, p. 104712, 2019.

[4] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] K. M. Tarwani and S. Edem, “Survey on recurrent neural network in natural language processing,” *Int. J. Eng. Trends Technol*, vol. 48, pp. 301–304, 2017.

[6] V. Preethi *et al.*, “Survey on text transformation using bi-lstm in natural language processing with text data,” *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 9, pp. 2577–2585, 2021.