AST-Enhanced Code Summarization for Java Subroutines

Bradley Ramunas
Department of Computer Science
Stanford University
bradleyr@cs.stanford.edu

Abstract

Code summarization - or generating natural language summaries of code - as a deep learning task is a rapidly growing and developing field in natural language processing. Improvements in code summarization will lead to better automatic code documentation and maintenance for large code bases. Our new model incorporates information encoded in the abstract syntax tree (AST) of Java subroutines in order to provide improved code summaries over traditional Seq2Seq code summarization models. We evaluate our model's performance by training on a corpus of over 2 million Java subroutines and comparing BLEU scores achieved on a test set of approximately 100,000 examples.

1 Introduction

Code summarization is a new and growing field within the realm of natural language processing that has important real-world applications for software engineers at large.

The basic premise of code summarization is to generate human-language descriptions of code in order to improve a code bases' readability and clarity. There are also many other applications of code summarization outside of improving these metrics, such as allowing for human-language queries to be made over a code base thus allowing for more intelligent auto-complete and code search technologies.

We propose a new method of representing programming subroutines that would allow for better code summarization. The intent is that our model will take in as input a string representation of a subroutine (i.e. a Java function) and will output a text description of what the subroutine does (i.e. summarize the function).

2 Related Work

LeClaire et al. [7] provide a comprehensive history of the field of code summarization, reaching back to the pre-deep learning era in which template-based approaches were used to automatically generate basic summaries of code. Of this history, two papers have strongly informed the work done in this paper, specifically the work of Alon et al. [1] and Hu et al. [4].

Alon et al. explore encoding specific subsets of paths in ASTs, with a subset of the path ultimately leading to a singular leaf node in the tree. Then, attention is applied to the specific relevant portions of the AST which is ultimately embedded and used in their model. Hu et al. explore a different method of encoding the AST; in their approach they traverse the AST in what they call a 'structure-based traversal'. Through this traversal, they are able to encode the AST as a sequence of characters that can then be used in generic Seq2Seq model.

We believe that the work of Alon et al. manipulate the AST too heavily, leaning closer to an over-engineered approach that is not necessary within the context of deep learning, while the method

CS230: Deep Learning, Fall 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

implemented by Hu et al. does not necessarily capture any more information than the raw code snippets themselves. As such, we believe that we can iterate on these two techniques in order to create a simple and elegant model with ideas from both.

3 Dataset and Features

LeClaire et al. [6] have curated a dataset of 2.1 million raw Java subroutine and comment pairs; each pair is identified by a unique ID, which also maps to a project ID. This project ID indicates the repository from which a subroutine was retrieved from.

At the suggestion of LeClaire et al., we will be using their provided train/test/validation sets; these sets are created such that the code from a given repository (as identified by the project ID) shall not be split across the different data splits as to ensure that there is no leakage of data between the splits that could potentially influence training. The train/test/validation splits will be approximately 90%, 5%, and 5% of the entire dataset, respectively.

LeClaire et al. provide two forms of their datasets: the first being the comments and subroutine code in a raw format as found in the code repositories, and the second being a pre-split dataset as described above, with the code tokenized and the comments restricted to their first line after removing special reference symbols such as `@return` or `@variableName` (with the assumption that the first line of a comment represents the English summarization of the subroutine).

For the purposes of training our baseline model, we shall only be using the pre-split dataset as provided. For creating our new AST-Enhanced model, we shall be using both datasets in order to have access to the raw Java subroutine for the purposes of extracting out the AST.

4 Methods

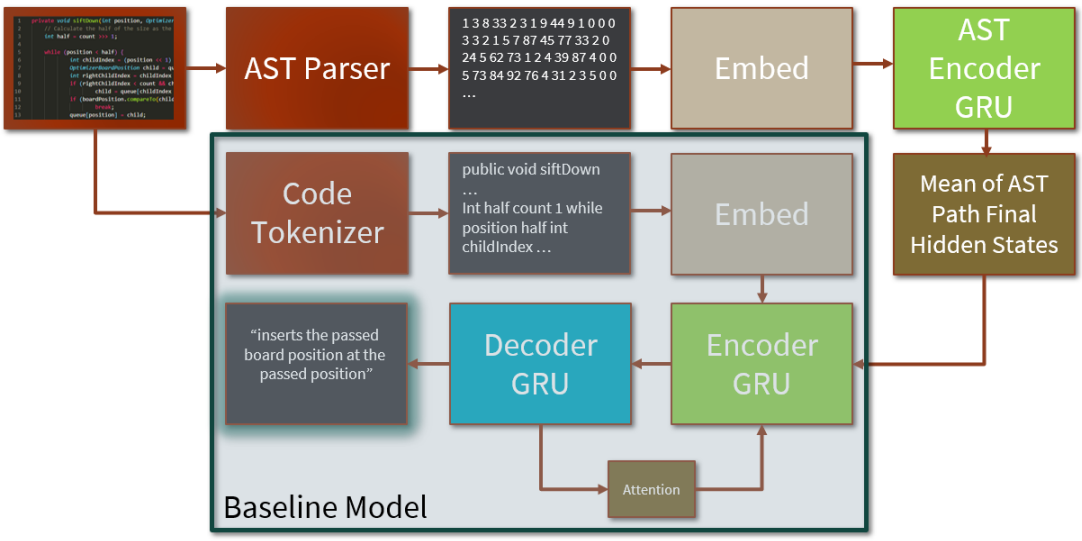

Figure 1: Our model architecture overview, with our baseline model indicated by the blue-shaded rectangular region.

4.1 Baseline Model

Our baseline model is an off-the-shelf implementation of the Seq2Seq model [9] with teacher forcing [5]. The source and target vocabularies used in the Seq2Seq model are created from the tokenized code and tokenized comment snippets, being limited to 5,000 and 10,000 entries respectively.

With these vocabularies, we load each example and prepend and append start- and end-of-sequence tokens, as well as replace out-of-vocabulary tokens with unknown tokens. These sequences are then

embedded and ran through a bi-directional gated recurrent unit (GRU) [2]. The output provided by this encoder GRU is then ran through a decoder GRU with attention. If teacher forcing is not enabled, then the highest probability word generated by the decoder GRU is then used as input into the next pass of the decoder GRU. In the case where teacher forcing is enabled, with a probability of 0.5, the correct next word is passed into the decoder instead. For the purposes of evaluation, teacher forcing is disabled.

4.2 AST-Enhanced Model

Our proposed model, which we call the AST-Enhanced model, is built on top of the baseline model explored previously.

In order to incorporate information found within the AST, we had to parse the raw subroutines provided by LeClaire et al. For each example available in the corpus, we ran a Java Abstract Syntax Tree parser library, which returned a series of AST nodes. An example of a node would be a `block-statement node` which is a node that represents a series of statements, thus meaning that it has many children nodes.

With these nodes, we can reconstruct the AST and traverse it in a deep-first manner. In doing so, we can collect the sequence of nodes visited as we approach a leaf or terminal node, and store each such sequence (henceforth known as an AST path). These AST paths are then mapped into a compressed form (specifically, any given node type is mapped to an integer value) and then stored. Through our analysis, we found that storing 24 AST paths of length 18 for any given Java subroutine would allow for us to keep a vast majority of the AST paths while avoiding having unnecessarily large input sequence sizes. These 24 AST paths are then mapped to the Java subroutine's unique ID, so that they may then be loaded at train/test/evaluation time.

With each Java subroutines mapping to 24 unique AST paths, we have to encode all these AST paths into a singular representation that can then be used with the code token encoder GRU. In order to accomplish this, we employ another GRU. This GRU is uni-directional, and will run over all 24 unique AST paths for a given Java subroutine. The final hidden states calculated by this GRU for each AST path is then averaged over, resulting in a singular representation of all 24 unique AST paths together.

This averaged representation is then used as the initial hidden state in the encoder found in our baseline model (except that this encoder has been changed to be uni-directional as well). From this point onward, the process of decoding with attention is the same.

5 Experiments/Results/Discussion

For the purposes of evaluating our AST-Enhanced model against our baseline model, we trained both models for 9 epochs and used their best-performing weights as determined by the log-likelihood loss against the evaluation set. With these two models, we then used them to generate code summaries of the examples found in the test set, which we could then use in tandem with the provided comments in order to calculate various BLEU scores [8].

Country List				
Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Baseline	32.736	22.944	17.063	13.692
AST-Enhanced	33.901	23.715	17.875	13.800

From this, we see a minor but significant improvement from our AST-Enhanced model over the baseline model.

6 Conclusion/Future Work

Our proposed AST-Enhanced model outperformed our baseline model as determined by the various BLEU scores noted in figure 2.

Moving forward, there are many interesting avenues of research to explore. Firstly, the issue of variable names. As noted earlier, our vocabulary sizes were somewhat limited, resulting in a large

Summaries:	**Actual Comment:**
"initializes the player panel"	"this method is run when player driver is an applet"
"merges the given body in the body"	"this method merge a first code fragment"
"logs an error message to the log"	"logs message with error severity in error log"
"returns the number of items in the series"	"returns the number of data point in a series"
"clears the log store"	"clears all entries out of the store"

Figure 2: Example code summaries generated by our AST-Enhanced model compared to the provided code summaries.

amount of unknown word tokens being generated by our decoder. In order to address this issue, there has been some work by Gu et al. [3] in which the decoder would attend to the raw input sequence, thus allowing unknown word tokens to be replaced by out-of-vocabulary words found in the input sequence itself via copying. We believe that this would be the most important improvement that can be made, as increasing vocabulary size can only help so much as there is an infinite space of variable names.

Further, stronger hardware could benefit our model greatly, as it would allow for larger vocabulary sizes to help circumvent the problem noted above. Further, it would allow us to iterate more, testing different hyper parameters. Due to the model's rather slow performance and the large size of epochs (2 million examples), it was unfeasible to test too many different values in the time allotted.

7 Contribution

As we were a group of one person, all work was done by said person.

Explicitly, the work done for this project includes but is not limited to: acquiring an off-the-shelf Seq2Seq model, testing the Seq2Seq model and understanding how it works, creating a new set of PyTorch / TorchText vocabularies and dataloaders that load in the code tokens and AST paths, created a Java program that parsed out the ASTs, traversed the ASTs to generate AST paths, wrote out the AST paths, wrote a Python Jupyter Notebook to analyze the AST paths and to properly pad and truncate them in order for them to be used in the model, created and iterated over multiple PyTorch custom encoding layers for the model - testing various approaches and ensuring that the model learned, made the model more robust by adding various parameters that can be modified by command-line arguments, improved the saving and loading functionality of the model, added metrics calculation and scoring utilities, and created the video and final report for this project.

References

[1] Uri Alon et al. "code2vec: Learning Distributed Representations of Code". In: *CoRR* abs/1803.09473 (2018). arXiv: 1803.09473. URL: http://arxiv.org/abs/1803.09473.

[2] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[3] Jiatao Gu et al. *Incorporating Copying Mechanism in Sequence-to-Sequence Learning*. 2016. arXiv: 1603.06393 [cs.CL].

[4] Xing Hu et al. "Deep Code Comment Generation". In: *Proceedings of the 26th Conference on Program Comprehension*. ICPC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 200–210. ISBN: 9781450357142. DOI: 10.1145/3196321.3196334. URL: https://doi.org/10.1145/3196321.3196334.

[5] keon. *mini seq2seq*. 2020. URL: https://github.com/keon/seq2seq.

[6] Alexander LeClair and Collin McMillan. *Recommendations for Datasets for Source Code Summarization*. 2019. arXiv: 1904.02660 [cs.CL].

[7] Alexander LeClair et al. *Improved Code Summarization via a Graph Neural Network*. 2020. arXiv: 2004.02843 [cs.SE].

[8] Kishore Papineni et al. "BLEU: A Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://doi.org/10.3115/1073083.1073135.

[9] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.