Automated Code Comment Generation

James Kim, Jeffrey Paulraj, Run Qian Gong
Carnegie Mellon University
Pittsburgh, Pennsylvania
{gyeongwk,jpaulraj,runqiang}@andrew.cmu.edu

1 Introduction

Code comment generation or source code summarization is an important and useful task with real world applications. Code is mainly written in a programming language, but well-written code also includes the natural language descriptions. This allows other programmers to understand the code effectively, which is key to building large, robust software systems. However, for software engineers, writing code comments can be a tedious task. Hence, there is a need for an automated system that involves a trained code comment generation model.

Naturally, the code summarization task is a machine translation task. For code summarization, the goal is to translate sequences in a programming language to sequences in natural language (NL). In this project, we focus on translating Python code snippets to English.

Although code comment generation is inherently a machine translation task, it differs in that programming languages follow a syntactic structure as defined by their Abstract Syntax Tree (AST). Hence, we can extract more information about each unit of the code by understanding its role in the overall structure. As a result, previous papers have improved performance of their models by augmenting the input data with the AST representation of the code [9, 8, 5]. We extend this approach to the Python programming language, while the above papers focused mostly on Java. Since Python has more flexible structure than Java, we expect the impact of the AST to be more crucial in the ability of the model to understand the input code snippet.

A similar approach proposed in numerous papers [8, 7] is to include API documentation for external functions used in the code snippet. Many code snippets make calls to internal and external functions, which impact its behavior. Hence, it is necessarily to understand the purpose of these referenced functions in order to accurately produce a summary of the code snippet. Previous papers take advantage of function descriptions from the Java API documentation. In our implementation, the limitation is that it is more difficult to identify the source of external functions in Python. Thus, we modify our approach to search for functions referenced from the same repository and include their associated docstring in the input.

Overall, we create a model for Python adapting the ideas mainly from the reference paper [8]. For evaluation, we use BLEU scores, which measure the n-gram similarity between the reference comment and the generated comment, and human evaluation. We expect our model to perform worse than results from our reference paper [8] due to limited computing resources. However, we expect the conclusions to be similar in that when augmented with more information (AST and docstring), the model will perform better than the baseline model that uses only the code snippet as input. Furthermore, we expect the AST to have a greater impact on performance than shown in [8]. However, the docstring will have less impact due to the difficulty of identifying helper functions in Python.

2 Dataset/Task

Similar to the original paper, the dataset we are using for the project is the CodeSearchNet dataset (Figure 1). CodeSearchNet contains tuples of functions and associated comments in various programming languages. The language we chose to focus on for this project is Python. CodeSearchNet contains tokenized code snippets with their associated tokenized comments.

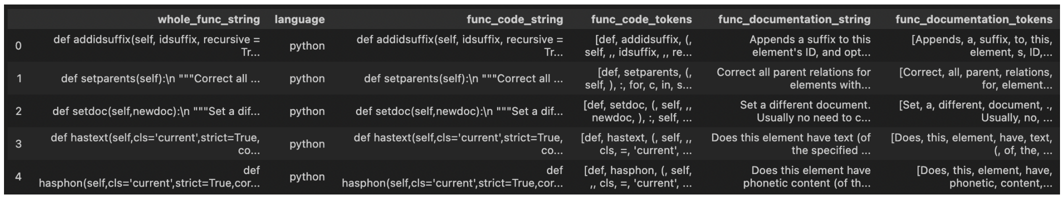
We further enriched the dataset with the AST tree traversal of the code snippets generated through Python's ast library. The motivation behind add the AST tree traversal is to allow the model to learn structural information about the function without having to rely on the much higher vocab size of the actual syntax. The ast tree traversal contains only around 90 different tokens whereas the tokenized function code contains around 500k different tokens, therefore requiring much more tokens to be out of vocabulary.

We also enriched the dataset with the documentation strings of within repository function calls. The goal of this new source of information is for the model to be able to extract information about otherwise unknown functions.

The training dataset that we are using contains around 100k training examples and the testing dataset contains around 5k examples. The vocab size for ast tree that we will use it 93 (to include <pad>, <sos>, <eos> tokens). The vocab size for function code tokens and documentation string tokens is 10k. We will truncate and pad sequence lengths to 200 for ast and function code tokens and truncate and pad the final output documentation strings to a sequence length of 200.

The task of interest is code comment generation or source code summarization. The goal of this task is to take source code and generate English text that accurately describes the function, purpose, and structure of the given code. Alternatively, we can view this task as a variant of Neural Machine Translation (NMT) task where the goals is to translate a programming language to English.

A common evaluation metric for the task is the BLEU score. The BLEU score measure the similarity between the generated text and reference text. In this case, we will use CodeSearchNet's comments as reference. For context, the BLEU score is a percentage score, where higher the value, more similar the texts are. The metric determines the score by measuring the average n-gram precision of the generated sequence to target sequence.



	whole_func_string	language	func_code_string	func_code_tokens	func_documentation_string	func_documentation_tokens
0	def addidsuffix(self, idsuffix, recursive = Tr...	python	def addidsuffix(self, idsuffix, recursive = Tr...	[def, addidsuffix, (, self, ,, idsuffix, ,, re...	Appends a suffix to this element's ID, and opt...	[Appends, a, suffix, to, this, element, s, ID,...
1	def setparents(self):\n """Correct all ...	python	def setparents(self):\n """Correct all ...	[def, setparents, (, self,), :, for, c, in, s...	Correct all parent relations for elements with...	[Correct, all, parent, relations, for, element...
2	def setdoc(self,newdoc):\n """Set a dif...	python	def setdoc(self,newdoc):\n """Set a dif...	[def, setdoc, (, self, ,, newdoc,), :, self, ...	Set a different document. Usually no need to c...	[Set, a, different, document, ., Usually, no, ...
3	def hastext(self,cls='current',strict=True, co...	python	def hastext(self,cls='current',strict=True, co...	[def, hastext, (, self, ,, cls, =, 'current', ...	Does this element have text (of the specified ...	[Does, this, element, have, text, (, of, the, ...
4	def hasphon(self,cls='current',strict=True,cor...	python	def hasphon(self,cls='current',strict=True,cor...	[def, hasphon, (, self, ,, cls, =, 'current', ...	Does this element have phonetic content (of th...	[Does, this, element, have, phonetic, content,...

Figure 1: Image of dataset

3 Related Work

Neural Models for Machine Translation The idea of using neural models for the machine translation task was formalized in [4], where the authors introduced the architecture for the RNN encoder-decoder model. The first RNN component encodes the input sequence into a vector representation, then the second RNN component decodes the vector into the output sequence. This architecture was improved in [3] by incorporating an attention mechanism, which considers the hidden vectors of the encoder at each time step and calculates a weighted average based on which time steps to "pay attention" to. Another paper [1] introduced the use of attention with a convolutional neural network for the same task.

Neural Models for Code Summarization Gaining motivation from the general machine translation task, the use of neural models for code summarization was introduced in [6], where authors propose a model called CODE-NN. This sequence-to-sequence model is based on a Long Short Term Memory network with attention, which is used to transform C# code and SQL queries into NL summaries. In this approach, the LSTM cells are guided by global attention, which is calculated from the current LSTM state.

Abstract Syntax Trees as Data Building upon the use of encoder-decoder models, further work focused on representation of code as data. Hu's paper [9] identifies the main problems with viewing code as plain text as the inability to capture the inherent structure of code and the unlimited vocabulary size in code due to unrestricted identifier names. They address the issue by transforming the input to an Abstract Syntax Tree (AST) representation and replacing unknown tokens with their type. The authors of [2] take a similar approach of representing code using ASTs. Each pairwise path between terminal values of the AST is used to encode the code. The model encodes each AST path into a vector using a bidirectional LSTM. Then, concatenates the vector with embeddings of the two terminal tokens to feed in to a fully connected layer, which combines representations from all paths of the AST.

Documentation as Data More recent studies have focused on augmenting additional information with the code data in order to help models better summarize code [5, 8, 7]. In [8], authors include API documentation for Java methods used in the sample code as additional data. Surprisingly, despite the addition of external data, the authors found negligible increase in performance of the models. Building on this idea, the authors of [7] improve the method in [8] by including API parameters' types and names in addition to the API description. Furthermore, they propose a ranking system to filter out uninformative API documentation. All together, the input used is the code, AST, API descriptions, and API definitions, which increased BLEU score of the output code comments.

4 Approach

As the baseline method, a pre-trained text generation model was used. This is a simple baseline that views code comment generation as a text generation task where the input is the code snippet and the output is the text description. Specifically, we use a baseline GPT-2 model from huggingface and prompt the model to generate the comments given the code snippet.

Our main approach aims to incorporate the strategies posed in previous papers in a different code comment generation setting. Since code comment generation is fundamentally a sequence-to-sequnce task, the base architecture of the models is an encoder-decoder model. For the encoder and decoder, we implement two main architectures: LSTM and Transformer.

As seen in previous literature, we propose two method to augment the input data in order to boost the performance of these base models.

1. **AST** ASTs encode the hierarchical ordering of the features of a code block, including variable assignments, loops, and function calls. This specified code structure can be more helpful to help a model learn patterns between certain hierarchical elements and associated comments, as literal code snippets which need similar comments can look very different but have similar structures. We will be focusing on generating comments for Python code, so we will use the Python AST library to create trees for the code snippets in our training dataset.
2. **Documentation** Often times, code from a single function relies on calling helper functions. We implement a method to extract descriptions of these helper functions, which can be used as additional information for comment generation. In the target paper, authors use Java API documentation as additional information for methods used in the code. However, in Python, it is difficult to know which library functions are used. Instead, we modify the approach to extract and include function documentation for functions written in the same repository.

Combining the two methods above, the input sequences of the model will consist of the code snippet, AST, and documentation. The goal of the model is to output a sequence of tokens that align with the reference code comment. The encoder component is composed of 3 separate encoders for each input sequence. Given code snippet, AST, and documentation sequences, the encoders each generate a context vector, which is concatenated and run through a linear layer to get a combined context vector for the decoder. A diagram of the model architecture is attached in Appendix A.

For the LSTM-based model, the attention mechanism computes coefficients for the current tiem step, which is used to compute the weighted average of the hidden vectors. For the Transformer model, we pass the input vectors into a transformer layer, with the vectors masked so the predicted next tokens will be based only on previously seen tokens. This output is passed into a linear layer to get a final sequence of tokens which is the predicted comment for the input source code.

5 Experiments

5.1 Experiment Details

For the LSTM-based model, we used embedding dimension of 64 and hidden dimension of 512 for all encoders and the decoder. A dropout of 0.1 was used to prevent overfitting. During training, a teacher force ratio of 0.5 was used, which determines whether or not to override the model's output with the ground truth token for each time step of the decoder. The model was trained with batch size 32 and for 200 epochs with starting learning rate of 0.01, which decayed by 0.5 every 50 epochs.

For the Transformer-based model, we used a model dimension of 512, denoting the expected number of features in the embeddings for the input and output. The transformer used six encoder layers and six decoder layers, with eight attention heads. To prevent overfitting, we incorporated a dropout of 0.1. Similar to the LSTM model, we trained with a learning rate of 0.01 with a batch size of 32, but over 100 epochs due to differences in training time per epoch.

5.2 Results

We use a baseline GPT-2 model for text-generation as the baseline method. For our main method, we implement a family of LSTM-based models and Transformer-based models. The Full model uses the code snippet, AST, and documentation string as input. For the ablation study, we measure the performance of AST model (code + AST), Doc model (code + documentation), and Base model (only code). The results are shown in Table 5.2

Model		Metrics			
		BLEU-1	BLEU-2	BLEU-3	BLEU-4
Baseliine	GPT-2	0.276	0.283	0.276	0.276
Base	LSTM				
	Transformer				
AST	LSTM				
	Transformer				
Doc	LSTM				
	Transformer				
Full	LSTM				
	Transformer				

Table 1: Baseline and Main results

6 Plan

Our plan for the project is to divide the work evenly among the 3 group members. The remaing work and timeline is as follows:

- 4/21: Add AST and Documentation component to input (Run)
- 4/23: Train Full version of LSTM and Transformer models (James, Jeff)
- 4/23: Evaluation script for BLEU scores and Human evaluation setup (Run)
- 4/25: Train ablations of LSTM and Transformer models (James, Jeff)
- 4/28 Poster Draft (All)
- 5/1: Poster Final version / Execute Summary draft (All)
- 5/3: Poster Presentation / Final Executive Summary (All)

References

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code, 2016.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[4] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[5] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[6] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[7] Ramin Shahbazi and Fatemeh Fard. Apicontext2com: Code comment generation by incorporating pre-defined api documentation, 2023.

[8] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. Api2com: On the improvement of automatically generated code comments using api documentations, 2021.

[9] Xin Xia David Lo Zhi Jin Xing Hu, Ge Li. Deep code comment generation. pages 200–210, 2018.

Appendix A: Model architecture

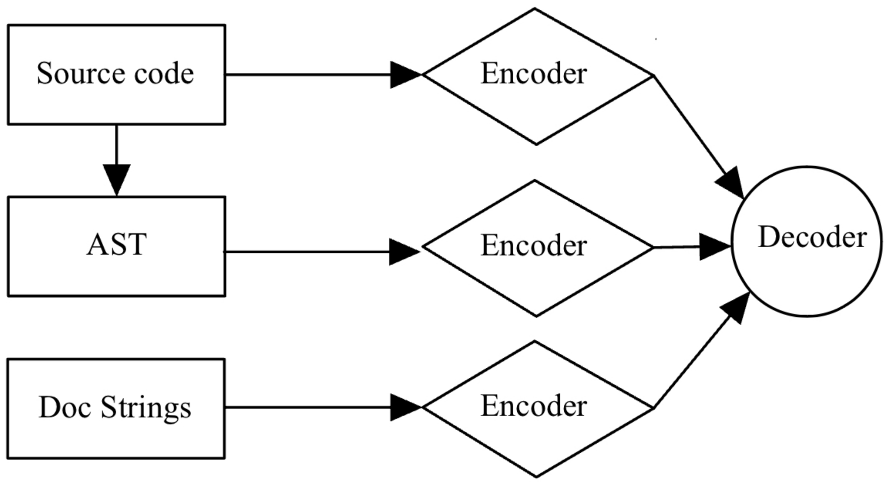


Figure 2: Model Architecture Diagram