Code Commenter

Edward Nee, Andrew Petropavlovskiy, Vladislav Annenkov

January 2023

Abstract

This document describes our solution to the code2text problem. Here we have described the approaches to learn model and collect our own dataset. Here you can find our repository with notebook: `https://github.com/EdwardNee/CodeCommenter-NLP4Code`.

1 Introduction

The code is written for people and documentation comments are important part to make code more understandable. Popular libraries and frameworks contain documentation for public methods, but this doesn't apply to amateur code that makes up most of the open source code. The same can be said about commercial code because not all companies have a rule to write comments in their code.

So, the main idea behind this project is to solve this problem through docstring generation from source code and remove the task of writing comments from the developers.

1.1 Team

- **Edward Nee** prepared this document;
- **Andrew Petropavlovskiy** prepared this document;
- **Vladislav Annenkov** prepared this document.

2 Related Work

2.1 LAMNER: Code Comment Generation Using Character Language Model and Named Entity Recognition

In the paper *LAMNER* [Rishab Sharma, 2022] (Language Model and Named Entity Recognition) a Named Entity Recognition model is trained to learn the different types of code tokens. code tokens then fed into an encoder-decoder architecture to generate code comments.

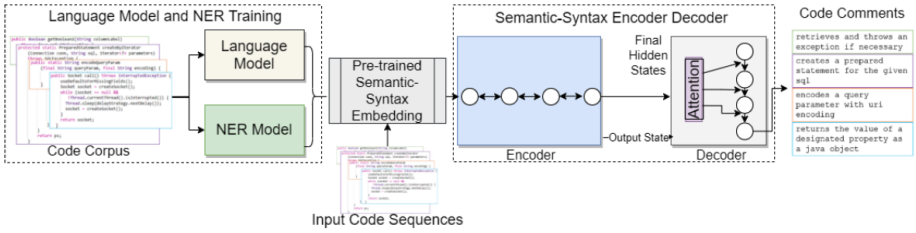

Figure 1: Overview of LAMNER Framework

Figure 1 illustrates the given framework of the LAMNER model. First left box depicts a bidirectional character-based language model and a NER model are trained separately on a code corpus that generates the input code embeddings for the next layer – Semantic-Syntax encoder decoder. Received raw embeddings are used as an input for the encoder. The decoder then uses the attention mechanism to decode the input code snippet into code comments.

Character-Level Language Model Architecture. The authors showed architecture of the language model on figure 2. It employs a single layer *bidirectional LSTM*. The input to each LSTM unit is an embedding of a randomly initialized character. Each LSTM unit processes the embedding to generate the output and the next hidden state for the character. The output of the last unit is used to select the next output character with maximum likelihood. The model predicts the character l for the given sequence of *public Boo*, which makes the last letter of *public Bool* sequence.

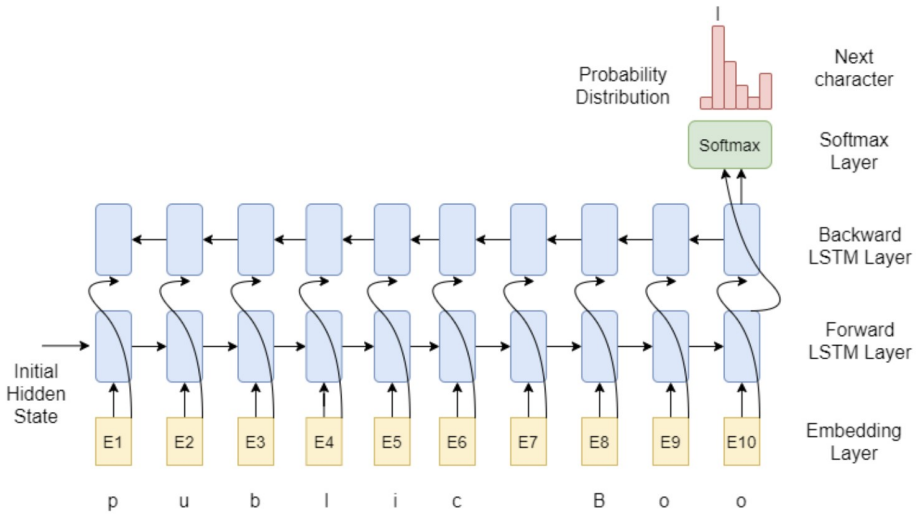

Figure 2: Overview of Character-Level Language Model Architecture.

The article describes processing the code tokens with the code sequence, however, the AST does not have a one-to-one mapping with code tokens, which is distinct in each programming language. NER is used to label the tokens of the code sequences. For example, the given code sequence *public Boolean getBoolean2 ...*, the token *public* is labeled as *modifier*, the token *Boolean* is labeled as *return type*, and the token *getBoolean2* is labeled as *function*.

The Semantic-Syntax encoder takes this semantic-syntactic embedding of the **code tokens** as input and models them together to output the semantic-syntax embeddings of the code sequence. Encoder processes the input using single layer bidirectional GRU. They use it due to its faster training time, and preservation the information for long sequences.
They say that the hidden state of the last token h_{last} contains information of the complete sequence. This token then fed into a fully connected linear layer. The equation is: $y_{fc} = h_{last} * W_t + b$, where W_t is weight matrix and b is bias values matrix.
The final output is: $h_{final} = tanh(y_{fc})$

The decoder with attention is trained in natural language. It implements Bahdanau's mechanism on a unidirectional GRU. Model uses h_{final} to pay attention to the input sequence's important tokens. The decoder will predict the next token until it reaches the maximum sequence length or the end of sentence token $\langle$eos$\rangle$.

2.2 ComFormer: Code Comment Generation via Transformer and Fusion Method-based Hybrid Code Representation

Authors introduced a new way of code commenting by combining transformers with hybrid code representation with AST information for better model learning. They also evaluated the performance of ComFormer [Guang Yang, 2021] on a large Java code corpus with comments for that code.
First of all, authors got rid of OOV. They got tokens, which are the class names, variables and etc. Then they convert them into a sequences by dividing camel case names into a separated words. For example, getUserInfo() method will be converted to a sequence of get, user, info. After that, ComFormer uses BPE to convert each word of a sequence into a sequence of letters. The authors claim that with this mechanism, the generated comments will not contain words that are not in the data set.
Secondly, authors use AST representations in their model. They convert AST into a sequence for processing this data by the model. At start they used SBT method for that, but recently they created their own Sim_SBT, which does not display unnecessary relationships, such as relationships with standard types and generates slightly shorter result, than SBT. ComFormer follows the Transformer architecture.

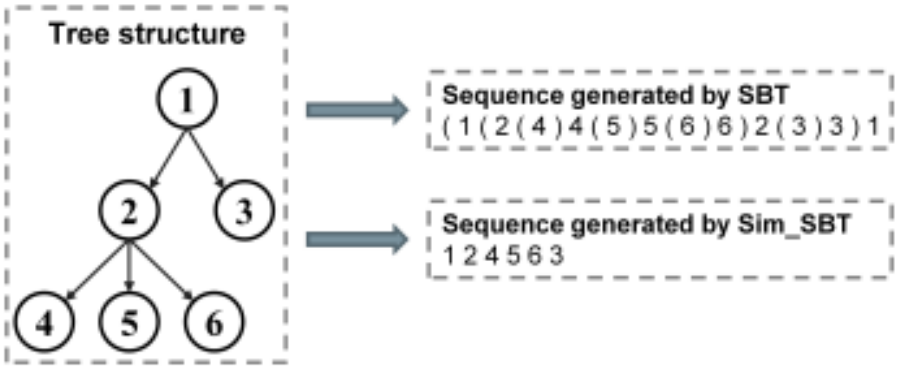

Figure 3: Sim_SBT result

Encoder produces a sequence of context vectors $Z = (z_1, \ldots, z_n)$. Input tokens are passed through a standard embedding layer and then model uses another positional embedding layer. Than, encoded embeddings used as the input to the encoder of N layers. Every layer consists of multi-head attention mechanism and a feed-forward network. Attention calculated as $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})$. Then the model calculates multi-headed attention and concatenate it with feed-forward layer.
$head_i = Attention(QW_I^Q, KW_I^K), VW_I^V$
$MultiHead(Q, K, V) = Concat_i(head_i)W^O$.

The second component of each layer is FFN:
$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$
Authors use three different methods to fuse lexical and syntactical information of code end Encoder.

Jointly Encoder assumes that AST and code are two different inputs.
Shared Encoder encodes two inputs by weight sharing, switches two matrices together, adds linear layer and activates it.
Single Encoder splices inputs and and proceeds through word embeddings.
Then, model use standart Decoder to decode embeddings.
So, the whole model represented on figure 7.

3 Model Description

To implement this task, we used the T5 model. T5 is an encoder-decoder model pre-trained on a multi-task mixture of unsupervised and supervised tasks and for which each task is converted into a text-to-text format. T5 works well on a variety of tasks out-of-the-box by prepending a different prefix to the input corresponding to each task.

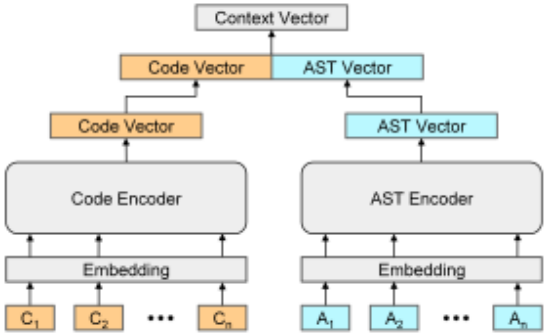

Figure 4: Structure of Jointly fusion method in the Encoder

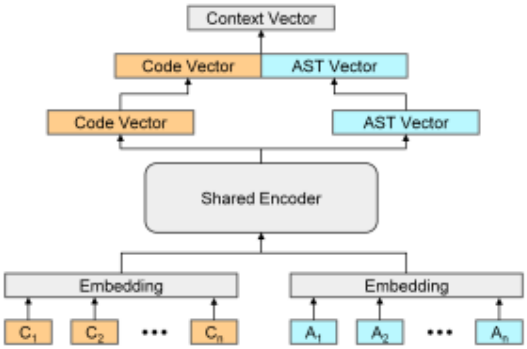

Figure 5: Structure of Shared fusion method in the Encoder

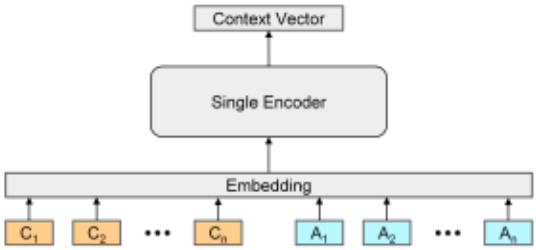

Figure 6: Structure of Single fusion method in the Encoder

We choose this model by two reasons:

- This model achieves good results in different tasks.
- This model fits well for different tasks at the same time. So we don't need to implement difficult architecture for the our task.

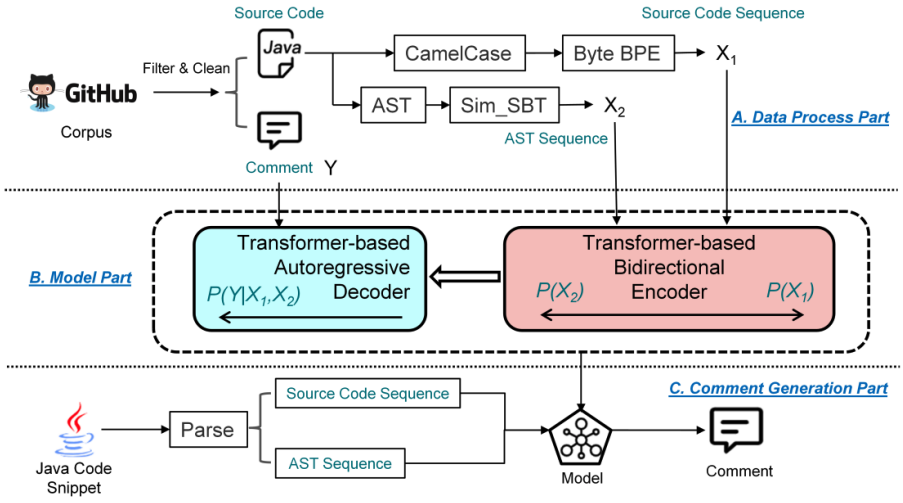

Figure 7: Overview of ComFormer Framework

Ideally, we would like to get a universal model to understand multiple programming languages and generate by that comment docs with various templates. By templates we mean some comment structure, because it can be different in some programming languages. For example, in C# it is a XML-like doc comments while in Java it is simple text with a pinch of HTML tags.

To take this into account, we use two prefixes during training. In our task we've used Java and go programming languages:

- "go docstring: ..." — to translate code into go-like docstring;
- "java docstring: ..." — to translate code into Java-like docstring.

Architecture of the resulting model and pipeline showed on figure 8.

4 Dataset

4.1 Dataset Description

For training the model, we used the CodeXGLUE dataset, which we have further refined for our task.

CodeXGLUE [Shuai Lu, 2021] stands for General Language Understanding Evaluation benchmark for CODE. It includes 14 datasets for 10 diversified programming language tasks covering code-code (clone detection, defect detection, cloze test, code completion, code refinement, and code-to-code translation), text-code (natural language code search, text-to-code generation), code-text (code summarization) and text-text (documentation translation) scenarios.

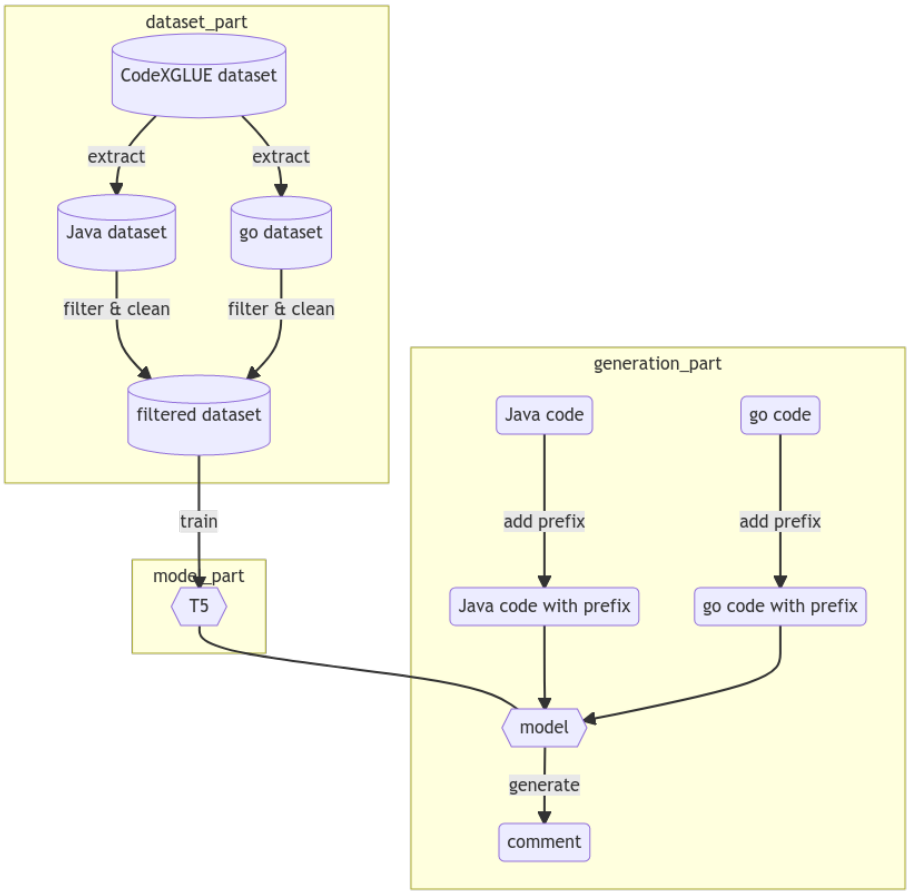

Figure 8: Overview of our training-generation pipeline based on T5

4.2 Getting Dataset

The dataset can be received in two ways: via GitHub and via HuggingFace. We've used HuggingFace that is popular solution for downloading pretrained models and datasets.

So, to download the dataset you can run the following method:

```
import datasets
datasets.load_dataset(
    "code_x_glue_ct_code_to_text", # CodeXGLUE
    "java",                        # Programming language
    split="train")                 # train, valid, test
```

4.3 Dataset Modifications and Markup

To make dataset more usable we made some modifications for it.

First, we've added comment slashes in Java-part of the dataset. This makes it possible to use comments in our tools without any modification after generation. Just use our trained model as a full-fledged algorithm.

Second, we've combined datasets with different programming languages (Java and go) into unified dataset. To do this, we've added additional column with language name.

Statistics of our dataset showed in the table 1. The table shows data split between train, valid and test selections.

	Train	Valid	Test
Snippets	332211	12508	19077
Programming Languages		2	

Table 1: Statistics of our dataset

5 Experiments

5.1 Metrics

We use the cross entropy loss:

$$L = -\sum_{i=1}^{n} t_i \log(p_i),$$

for n tokens, where t_i is the truth label and p_i is the Softmax probability for the i^{th} token.

5.2 Experiment Setup

5.2.1 Models

We have tried two base models: *BERT* and *T5*. We stopped at T5 because this model provides better results for our task. Also T5 is used in articles with the same task as a model with the best training results.

5.2.2 Datasets

We have tried two datasets from CodeXGLUE: *code2text* and *text2code*. As a closer dataset to our task, code2text has gave better results during model training. Comparison of these datasets is shown in the table 2.

	code2text	text2code
First epoch	0.976	1.683
Second epoch	0.945	1.626
Third epoch	...	1.618

Table 2: Dataset losses comparison

5.3 Baselines

...

6 Results

As a result, we got a model which translates programming code into docstring text. We haven't make special comparisons of our model with other models. Generation results showed in the table 3 and table 4.

java docstring: void printMessage(String message) { System.out.println(message); }
// Prints a message. // // @param message the message to print.

Table 3: Java input and output sample

go docstring: func CheckMdsAvailability() error { if conn, err := net.Dial("unix", common.MetadataServiceRegSock); err != nil { return errUnreachable } else { conn.Close() return nil } }
// CheckMdsAvailability checks whether a local metadata service can be reached.

Table 4: Go input and output sample

7 Conclusion

In our work we have received a complete solution for code2docstring task: adapted dataset, model setup and working pipeline.

To get an our own dataset we've worked on CodeXGLUE dataset, removed irrelevant data and concatenated datasets with different languages into one large dataset. Also we've added some programming language-specific decorations for docstring texts, like slashes in Java docstrings.

As a basic model, we have chosen the T5. T5 is a suitable solution to train multiple tasks in one model. We use it to implement different docstring types for various programming languages. Of course, we could use any other

Transformer-based model to achieve this purpose, but we think, that pretrained T5 model with multitask-specific part has helped us achieve better results.

Finally, we have tested our model and got meaningful results. We think, that this solution can be improved later and we are considering various options for this:

- Replace programming code with AST;
- Produce more useful data for model, e.g. transfer different representations of programming code;
- And more experiments with different models, datasets, metrics and hyperparams...

References

[Guang Yang, 2021] Guang Yang, Xiang Chen, J. C. S. X. Z. C. C. Y. K. L. (2021). Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. 10:12.

[Rishab Sharma, 2022] Rishab Sharma, Fuxiang Chen, F. F. (2022). Lamner: Code comment generation using character language model and named entity recognition. *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA. ACM, New York, NY, USA*, 10:12.

[Shuai Lu, 2021] Shuai Lu, Daya Guo, S. R. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation.