AUTOMATIC COMMENT GENERATION FOR PYTHON CODE

A PREPRINT

Weifeng Hu , Chengyu Dong, Ye Fan, Rui Yang, Chun Ni, Shang Gao
Department of Computer Science Engineering
University of California, San Diego
{weh031, cdong, y1fan, r3yang, chni, shgao}@eng.ucsd.edu

March 23, 2019

ABSTRACT

Writing comments on source code has benefits in area such as code maintenance in many industries. However, many code bases have inaccurate, outdated or even no comments. In this project, we propose a fully data-driven approach for generating comments of source code. We implement a sequence to sequence model that is commonly used for language translation tasks. Our model uses two GRU [1] networks, one as an encoder and the other as a decoder. In order to obtain structural information on source code, we implement a Structural Based Traversal technique before inputting into the encoder. We also experiment with a simple sequence to sequence model and a sequence to sequence model that uses attention. We find that the sequence to sequence model with attention outperforms other models in generating comments.

1 Introduction & Motivation

It is of great importance to write good comments which can be easily understood by other software developers. However, due to various reasons including tight project schedules and different comment styles, code comments can be hard to understand, out of date or even mismatched, which makes automatically generating code comments a significant help in these situations. Hereby, we would like to use Recurrent Neural Networks (RNNs) to extract structural and semantics information from Python source code and generate readable code comments. We build a sequence-to-sequence model which consists of an encoder and a decoder to map the source code to the descriptive comments. In order to improve the performance of the model, we use structure-based traversal (SBT) to parse the abstract syntax tree of the source code. We also use the attention mechanism between the encoder and the decoder to improve the performance.

Our dataset contains data collected from StackOverflow, which is a question and answer site for professional and enthusiast programmers. We used the Python tags to extract python code from this website and collected all together 31,257 posts. We cleaned these posts and extracted descriptive comments and Python code from each post. For training and evaluating our model, we divided out data into a train set containing 25671 entries, a validation set containing 2792 entries and a test set containing 2794 entries. Figure 1 shows an example entry of our data. The questions / comments start with a <QUE> header, answers / codes start with a <ANS> header and ends with a <END>.

```
<QUE>
Parse entries that is within specified date using feedparser
<ANS>
import feedparser
d = feedparser.parse('http://www.reddit.com/r/Python/.rss')
for entry in d.entries:
  date = entry.published_parsed
  if date.tm_year == 2015 and date.tm_mon == 4 and date.tm_mday >= 15 and date.tm_mday <= 16:
    print entry.title
<END>
```

Figure 1: Example of comment and code in our data

Since we are basically dealing with a language translation model, the length of a sentence matters. In Fig. 2 and Fig. 3 we show the distribution of the lengths of comments and codes in our training set. The comment of a median length contains about 50 words, which is not a short sentence in general. In contrast, the code snippet of a median length contains about 160 words. As a reference, the length of the code snippet shown in Fig. 1 is 250. It is quite common that a code snippet in our data set is only composed of one or two lines, but the corresponding comment is much verbose.

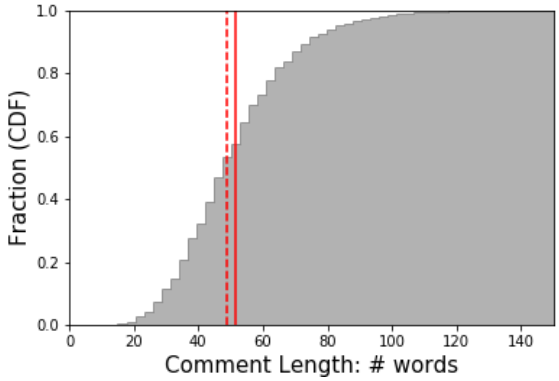

Figure 2: Cumulative fractions of comment lengths in the training set. The mean and median comment length are denoted by a solid and a dashed line respectively.

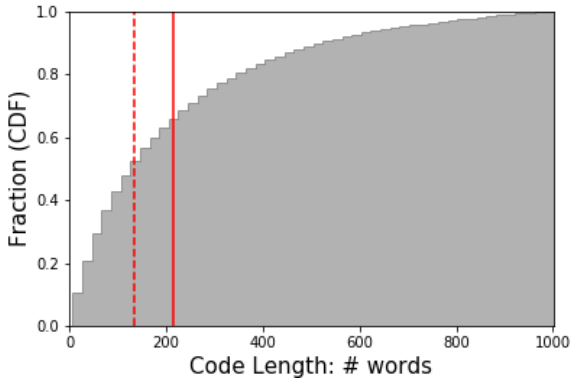

Figure 3: Cumulative fractions of code lengths in the training set. The mean and median code length are denoted by a solid and a dashed line respectively.

This paper is structured as follows: Section 2 discusses the related literature. Section 3 describes the sequence to sequence model. Section 4 discusses the results obtained. Section 5 discusses the results from different models we experimented.

2 Related Work

SUM-NN (Rush et al. 2015) [2] is the neural attention-based abstractive summarization model, which is the first model to propose a method of text summarization using attention mechanism. It uses encoder-decoder architecture with an attention mechanism based on a fixed context window of previously generated words.

CODE-NN (Iyer et al. 2016) [3] is the first model to make use of neural network to generate comments for source code. The model uses LSTM and attention mechanism to generate summaries that describe C# code snippets and SQL queries. It has strong performance on code summarization and code retrieval. This model can achieve a Rouge [4] 2 (overlapping bigrams between target and generated sequences) score of about 0.1.

CODE-RNN (Liang et al. 2018) [5] is very similar to CODE-NN, but it uses GRU as Recurrent NN to extract features from the source code and embed them into one vector. The performance of this model is much better than that of other learning based approaches such as sequence-to-sequence model. This model can achieve a Rouge 2 score of about 0.2. CODE-RNN is the state of the art for such a task.

3 Method

We model the conversion from Python source code to comments as translation between different natural languages. Therefore we build a sequence to sequence model that is commonly used in machine translation [6]. A simple sequence to sequence model consists of two components: an encoder and a decoder. We use the encoder to learn the latent features of the source code and use the decoder to generate the comment sequence. The power of this model is that they can map sequences of different lengths to each other. Moreover, we also build a sequence to sequence model that uses attention mechanism, as shown in Figure 4. Attention allows the decoder network to focus on a different part of the encoder's outputs for every step of the decoder's own outputs.

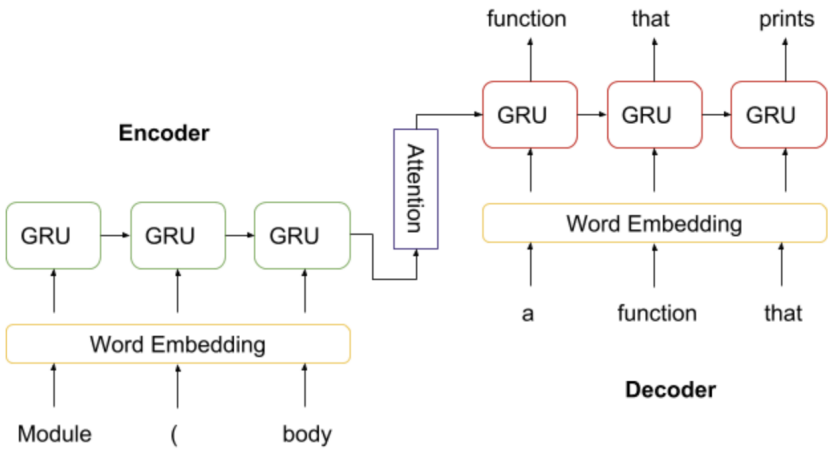

Figure 4: Sequence to Sequence model for comment generation with attention

3.1 Code Embedding

Before feeding source code into our model, we must figure out how to represent the code. The simplest way is to just view it as plain text. In the prototype model we have tried this, but it results in weird loss curve and comments that are hardly recognizable. The reason may lie in that, unlike human language, the semantics and syntactics of code strongly depend on its structure. Without critical relations like inclusion and juxtaposition, plain text of code is impossible to be interpreted.

Therefore, in this work we introduce the Abstract Syntax Tree (AST) [7], which is the tree representation of the syntactic structure of source code, and is typically built by a parser during compiling. AST best retains the structure of code, and add necessary contextual information as well, which is obviously beneficial to learning. Nevertheless, we still need to convert the tree representation of source code to a sequence, such that it can be assessed by the recurrent network. A simple Depth-first Search (DFS) can concatenate all the tokens in a tree, though omit the structure information that we desire to retain. Here, to address this problem, we adopt a Structure-based traversal (SBT)[7], and render some variants. The detailed algorithm is presented in Algorithm 1. There are only two tokens in SBT representation that will serve as

structural components, i.e. '(' and ')'. The children of a node will be wrapped in brackets, lead by the token in the node. Hence we can exclusively reconstruct a tree from the SBT representation.

Algorithm 1: Structure-based Traversal

Input: The root r of a tree L
Output: A sequence seq representing the structures and semantics of the tree

$seq \leftarrow \phi$
if *!r.hasChild* **then**
  $seq \leftarrow (+r+)$
else
  $seq \leftarrow (+r$
  for c *in* $childs$ **do**
    $seq \leftarrow seq + SBT(c)$
  end
  $seq \leftarrow seq+)$
end
return seq

Below we show a piece of code in our dataset as example, followed by its AST (Fig. 5[1]) and SBT representation. The corresponding comment is '`Trouble With Lists of Lists in Python`'.

```
class Obj1(object):
    def __init__(self):
        self.list_of_obj2 = []
```

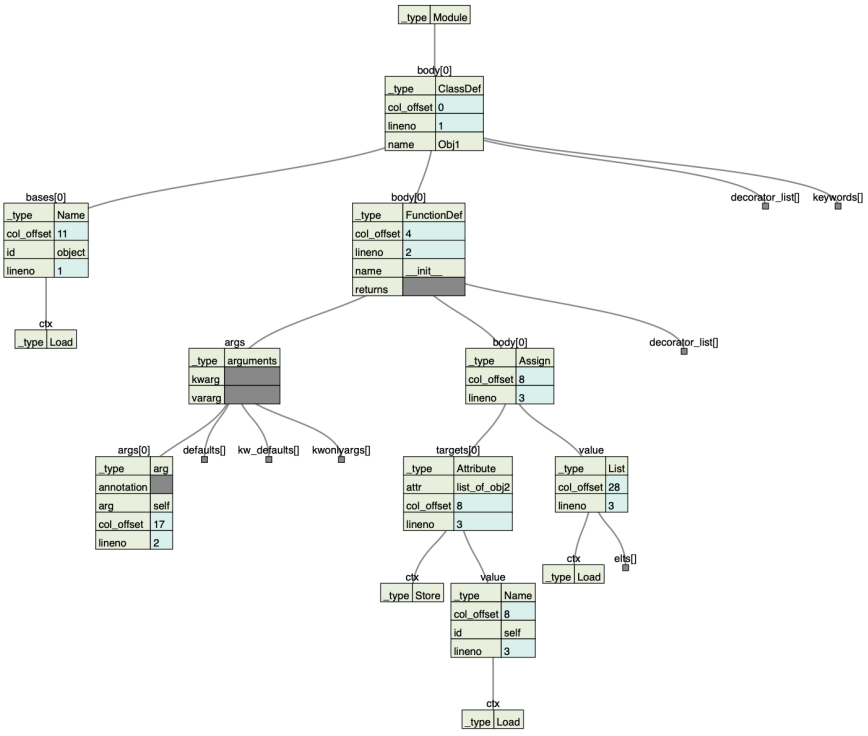

Figure 5: The Abstract Syntax Tree of the sample code

```
Module ( body ( ClassDef ( bases ( Name( ctx ( Load) id ) )
```

[1] `https://vanya.jp.net/vtree/index.html`

```
body ( FunctionDef ( args ( arguments ( args ( arg ( annotation_None arg_self ) )
defaults kw_defaults kwarg_None kwonlyargs vararg_None ) )
body ( Assign ( targets ( Attribute ( attr ctx ( Store )
value ( Name ( ctx ( Load ) id_self ) ) ) ) value ( List ( ctx ( Load ) elts ) ) ) )
decorator_list name returns_None ) ) decorator_list keywords name ) ) )
```

Note that we concatenate the token and its corresponding value with '_' in a leaf, i.e. terminal node. In AST, a node itself contains a value, which is the key to interpret the source code. Specifically, besides its own value, a leaf also has a child value that record the attribute of its key, which is often defined by user, such as the name of the variable, function, class or module. These names could be arbitrary and we will retain a part of such information in our representation, that is, if the name defined by user is in a vocabulary with a fixed length, then we will keep it and concatenate it with its leading key, as mentioned above. On the other hand, if a name is not in the vocabulary, then we leave it thus only its leading key will be retained. In this way, we preserve the information in AST to the most.

3.2 Encoder

The encoder is a GRU model and responsible for learning features from source code. The encoder reads one token x_t from the input sequence each time step and gets hidden state h_t with the information of h_{t-1}. This process can be described by equation 1. The output of encoder is the last hidden states of the encoder and will be fed into the attention mechanism.

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1}) \tag{1}$$

3.3 Attention

Attention[8] is a mechanism recently suggested to extract important parts from the input sequence for each output unit. The insight behind it is for each output unit, the importance of each input unit is different. For example, if we would like to translate a piece of Python code with the purpose of calculating the sum of two integers to its code comments, the importance of "+" to word "sum" is obviously more important than some other characters.

Different from simple Seq2seq models, input to this attention mechanism is all hidden states of the encoder. It defines a c_i for each output unit in the decoder as:

$$c_i = \sum_{j=1}^{n} a_{ij}h_j \tag{2}$$

Where n is the sequence size of the encoder, h_j is the hidden state calculated in the encoder and a_{ij} is the weights for each hidden state h_j. a_{ij} is calculated as below:

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1} nexp(e_{ik})} \tag{3}$$

Where

$$e_{ij} = a(s_{i-1}, h_j) \tag{4}$$

is an alignment model which scores how well the inputs around position j and the output at position i match.4. Here s_{i-1} is the hidden state of the decoder.

3.4 Decoder

The decoder is tasked to generate the target sequence y by predicting the probabilities of a word y_t at each timestamp t. In this case, the target would be the comments that summarize the source code. We use teacher-forcing method during training, meaning at each timestamp, we feed in the target word no matter the output of the decoder. The output of a hidden state at time step t is shown in equation 5, where x_t is the input of the encoder at timestamp t, h_{t-1} is the previous hidden state, c_t is the weighted hidden states of the encoder.

$$h_t = f(x_t, c_t, h_{t-1}) \tag{5}$$

The output of the decoder at timestamp t is computed using a Softmax function as shown in equation 6, which gives a probability vector that helps us determine the output word.

$$y_t = Softmax(g(x_t, c_t, h_{t-1})) \tag{6}$$

The goal of our model is to minimize the cross-entropy, which is:

$$H(y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{n} t_j^{(i)} log(y_j^{(i)}) \tag{7}$$

Where N is the total number of training instances, $t_j^{(i)}$ is the target output of the jth word in the ith instance.

We build two different decoders. One is the above Seq2seq model which contains the attention mechanism. The other is a simple decoder that takes the last hidden state of the encoder directly. We compare the performance of both decoders in the following section.

4 Result

We use Adam Optimizer and a learning rate of 0.001 for our training. Early stopping is performed to prevent overfitting. We trained for about 10 epochs on the training set and evaluate on the validation set. The first experiment we used a sequence to sequence model without attention.

4.1 Vanilla Sequence to Sequence Model

The simple sequence to sequence is similar to Figure 4. Instead of passing the last hidden state through an attention mechanism, we pass it directly as the first hidden state of the decoder. Figure 6 shows the loss curves for this model.

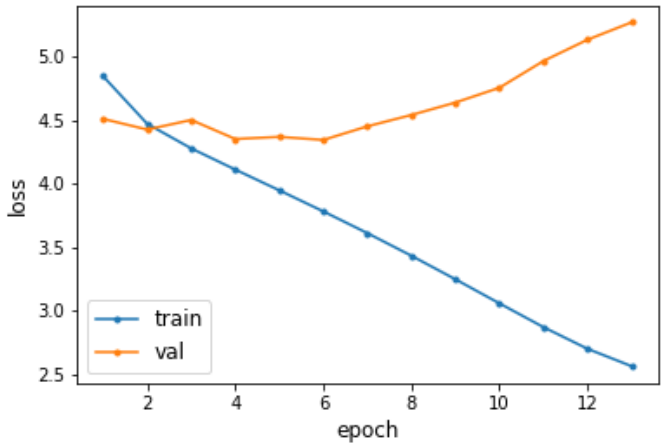

Figure 6: Training and validation cross entropy loss vs number of epochs for simple decoder model

We can see that the sequence to sequence model without attention does not perform well on the dataset. In particular, the validation loss does not show significant decrease. Therefore, we applied attention mechanism for our next experiment.

4.2 Attention Sequence to Sequence Model

The sequence to sequence model with attention is the same as Figure 4. The last hidden state of the encoder is passed through an attention mechanism before feeding into the decoder. This allows the decoder to focus on different part of the encoder's hidden state at each timestamp. Figure 7 shows the loss curves for our attention sequence to sequence model.

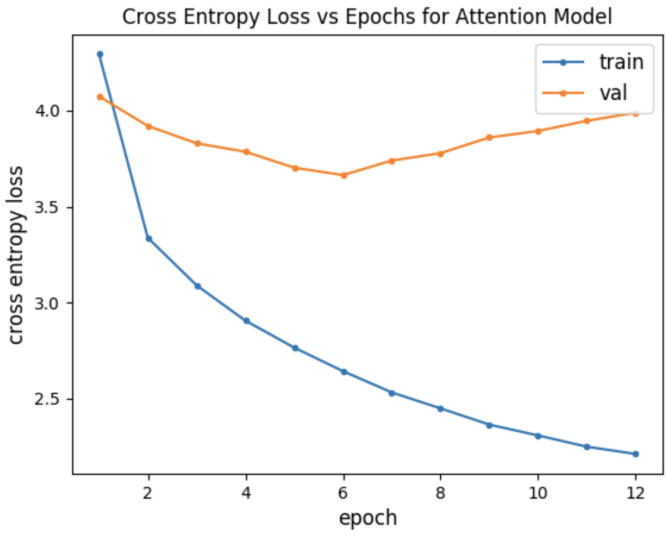

Figure 7: Training and validation cross entropy loss vs number of epochs for attention model

We can see that the validation error stops decreasing after about 6 epochs. We keep the model state with the lowest validation error. Compared with the validation loss for the simple sequence to sequence model above, we can see that attention helps improve the performance by lowering the loss.

The result of the training is shown in Table 1, the cross-entropy loss on the validation set is reported for each model. We can see from the table that the sequence to sequence model with attention outperforms the other model.

Model	Cross Entropy Loss
Seq2seq Vanilla	4.48
Seq2seq Attention	3.67

Table 1: Cross entropy loss on validation set for three different models

4.3 Generation

We run our best model, the sequence to sequence model with attention, to generate comments from Python code. Table 2 shows two of the best generated results from our test dataset. We can see that our model turns to perform better on short Python code segments. The best Rouge 2 score obtained from a sequence is 0.25, but our average Rouge 2 score is lower compared to CODE-NN and CODE-RNN in the related work.

Python Code	Human Written Comment	Model Output Comment
django.template.loader .get_template(template_name)	check if a template exists in Django	get a template in Django
list(the_iterator)	Standard method for returning all elements generated by a generator or iterator	get the list of the iterator

Table 2: Table for comparison between generated result and human written result

Our model does not perform well on long Python code. One reason might be that the longer Python code contains more information and it is harder for the model to understand. We believe that different encoding method for source code will increase the model complexity and might help with this issue.

Another reason might be that the length of code in our training data is very short, which is clearly shown in Fig. 3. As a result, the model is more familiar with shorter length code compared to longer length, causing it to perform better on short code segment.

5 Discussion & Conclusion

We model the conversion from Python code into comment as translating between different languages. Therefore we use a sequence to sequence model that is a common choice for such task. In order to maintain the structural information presented in the code, we use a Structural Based Traversal (SBT) technique on the abstract syntax tree of Python. In the sequence-to-sequence model we implement two different types of decoder. One is a simple decoder that takes the last hidden state of the encoder directly. The other uses attention mechanism to improve the performance.

From our result, we can see that a sequence to sequence model that uses attention mechanism outperforms the other model. With the help of SBT to maintain the structural information, we are able to generate some comments that closely resemble human written comments. It shows the great importance of SBT and attention mechanism in translating Python code into comment.

As pointed out above, our model behaves poor on a longer code snippet, which may be due to the imbalance of code lengths. We may need to seek some variants in data preprocessing and training to handle the imbalance problem. In addition, we can try curriculum learning, that is train the model on short and easy code at the beginning, and gradually feed longer code to the model so that the model can utilize its knowledge on short snippets to ravel the meaning of complex ones.

For future work, we are looking to experiment other preprocessing method to encode structural information of Python code. We also would like to increase the depth of our sequence to sequence model to help our model learn better on longer Python code.

6 Individual Contribution

Weifeng Hu worked on building the decoder architecture and developed the code for training. Weifeng worked on the decoder part, result and the discussion parts of the report. Weifeng presented the project during presentation period.

Rui Yang worked on building the attention architecture and worked on the introduction and methods parts of the report.

Shang Gao worked on training the model by sequence to sequence model without attention. Shang also worked on the related part of the report.

Chun Ni worked on building the encoder architecture and orgnizing the complete model. Chun also worked on encoder part of the report.

Ye Fan worked on reading and concluding related work and implementing SBT algorithm. Ye also worked on the related part of the report.

Chengyu Dong worked on building the baseline, implementing SBT algorithm as well as source code data preprocessing. Chengyu also worked on the related part of the report.

References

[1] 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. *arXiv preprint arXiv:1406.1078*, 2014.

[2] Sumit Chopra AlexanderM.Rush and Jason Weston. Aneural attention model for sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389. EMNLP, 2015.

[3] Alvin Cheung Srinivasan Iyer, Ioannis Konstas and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083. ACL, 2016.

[4] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out: Proceedings of the ACL-04 workshop, volume 8*, 2004.

[5] Yuding Liang and Kenny Q. Zhu. Automatic generation of text descriptive comments for code blocks. In *AAAI*, 2018.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceeding of the 27th International Conference on Neural Information Processing Systems - Volume 2*, pages 3104–3112. NIPS, 2014.

[7] Xin Xia David Lo Xing Hu, Ge Li and Zhi Jin. Deep code comment generation. In *Proceedings of the 2018 26th IEEE/ACM International Confernece on Program Comprehension*. ACM, 2018.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.