Deep Code Comment Generation*

Xing Hu[1,2] , Ge Li[1,2], Xin Xia[3], David Lo[4], Zhi Jin[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
[2] Institute of Software, EECS, Peking University, Beijing, China
[3]Faculty of Information Technology, Monash University, Australia
[4]School of Information Systems, Singapore Management University, Singapore
[1,2]{huxing0101,lige,zhijin}@pku.edu.cn, [3]xin.xia@monash.edu, [4] davidlo@smu.edu.sg

ABSTRACT

During software maintenance, code comments help developers comprehend programs and reduce additional time spent on reading and navigating source code. Unfortunately, these comments are often mismatched, missing or outdated in the software projects. Developers have to infer the functionality from the source code. This paper proposes a new approach named DeepCom to automatically generate code comments for Java methods. The generated comments aim to help developers understand the functionality of Java methods. DeepCom applies Natural Language Processing (NLP) techniques to learn from a large code corpus and generates comments from learned features. We use a deep neural network that analyzes structural information of Java methods for better comments generation. We conduct experiments on a large-scale Java corpus built from 9,714 open source projects from GitHub. We evaluate the experimental results on a machine translation metric. Experimental results demonstrate that our method DeepCom outperforms the state-of-the-art by a substantial margin.

CCS CONCEPTS

• **Software and its engineering → Documentation**; • **Computing methodologies** → *Neural networks*;

KEYWORDS

program comprehension, comment generation, deep learning

ACM Reference Format:
Xing Hu[1,2] , Ge Li[1,2], Xin Xia[3], David Lo[4], Zhi Jin[1,2]. 2018. Deep Code Comment Generation. In *ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196334

*This research is supported by the National Basic Research Program of China (the 973 Program) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant Nos. 61232015 and 61620106007. Zhi Jin and Ge Li are corresponding authors.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196334

1 INTRODUCTION

In software development and maintenance, developers spend around 59% of their time on program comprehension activities [45]. Previous studies have shown that good comments are important to program comprehension, since developers can understand the meaning of a piece of code by using the natural language description of the comments [35]. Unfortunately, due to tight project schedule and other reasons, code comments are often mismatched, missing or outdated in many projects. Automatic generation of code comments can not only save developers' time in writing comments, but also help in source code understanding.

Many approaches have been proposed to generate comments for methods [24, 35] and classes [25] of Java, which is the most popular programming language in the past 10 years[1]. Their techniques vary from the use of manually-crafted [25] to Information Retrieval (IR) [14, 15]. Moreno et al. [25] defined heuristics and stereotypes to synthesize comments for Java classes. These heuristics and stereotypes are used to select information that will be included in the comment. Haiduc et al. [14, 15] applied IR approaches to generate summaries for classes and methods. IR approaches such as Vector Space Model (VSM) and Latent Semantic Indexing (LSI) usually search comments from similar code snippets. Although promising, these techniques have two main limitations: First, they fail to extract accurate keywords used for identifying similar code snippets when identifiers and methods are poorly named. Second, they rely on whether similar code snippets can be retrieved and how similar the snippets are.

Recent years have seen an emerging interest in building probabilistic models for large-scale source code. Hindle et al. [17] have addressed the naturalness of software and demonstrated that code can be modeled by probabilistic models. Several subsequent studies have developed various probabilistic models for different software tasks [12, 23, 40, 41]. When applied to code summarization, different from IR-based approaches, existing probabilistic-model-based approaches usually generate comments directly from code instead of synthesizing them from keywords. One of such probabilistic-model-based approaches is by Iyer et al. [19] who propose an attention-based Recurrent Neural Network (RNN) model called CODE-NN. It builds a language model for natural language comments and aligns the words in comments with individual code tokens directly by attention component. CODE-NN recommends code comments given source code snippets extracted from Stack Overflow. Experimental results demonstrate the effectiveness of probabilistic models on code summarization. These studies provide principled methods

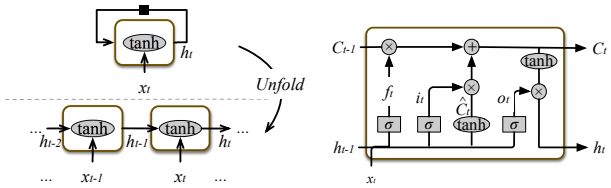
[1]https://www.tiobe.com/tiobe-index/

for probabilistically modeling and resolving ambiguities both in natural language descriptions and in the source code.

In this paper, to utilize the advantage of deep learning techniques, we propose a novel approach DeepCom to generate descriptive comments for Java methods which are functional units of Java language. DeepCom builds upon advances in Neural Machine Translation (NMT). NMT aims to automatically translate from one language (e.g., Chinese) to another language (e.g., English) and it has been shown to achieve great success for natural language corpora [6, 37]. Intuitively, generating comments can be considered as a variant of the NMT problem, where source code written in a programming language needs to be translated to text in natural language. Compared to CODE-NN which only builds a language model for comments, the NMT model builds language models for both source code and comments. The words in comments align with the RNN hidden states which involve the semantics of code tokens. DeepCom generates comments by automatically learning from features (e.g., identifier names, formatting, semantics, and syntax features) extracted from a large-scale Java corpus. Different from traditional machine translation, our task is challenging since:

(1) **Source code is structured:** In contrast to natural language text which is weakly structured, programming languages are formal languages and source code written in them are unambiguous and structured [3]. Many probabilistic models used in NMT are sequence-based models that need to be adapted to structured code analysis. The main challenge and opportunity is how to take advantage of rich and unambiguous structure information of source code to boost effectiveness of existing NMT techniques.
(2) **Vocabulary:** In natural language (NL) corpora normally used for NMT, the vocabulary is usually limited to the most common words, e.g., 30,000 words, and words outside the vocabulary are treated as unknown words – often marked as ⟨UNK⟩. It is effective for such NL corpora because words outside the dominant vocabulary are so rare. In code corpora, the vocabulary consists of keywords, operators, and identifiers. It is common for developers to define various new identifiers, and thus they tend to proliferate. In our dataset, we get 794,711 unique tokens after replacing numerals and strings with generic tokens ⟨NUM⟩ and ⟨STR⟩. In a codebase used to build probabilistic models, there are likely to be many out-of-vocabulary identifiers. As Table 1 illustrates, there are 794,621 unique identifiers in our dataset. If we use most common 30,000 tokens as the code vocabulary, about 95 % identifiers will be regarded as ⟨UNK⟩. Hellendoorn and Devanbu [16] have demonstrated that it is unreasonable for source code to use such a vocabulary.

To address these issues, DeepCom customizes a sequence-based language model to analyze Abstract Syntax Trees (AST) which capture structures and semantics of Java methods. The ASTs are converted into sequences before they are fed into DeepCom. It is generally accepted that a tree cannot be restored from a sequence generated by a classical traversal method such as pre-order traversal and post-order traversal. To better present the structure of ASTs, and keep the sequences unambiguous, we propose a new structure-based traversal (SBT) method to traverse ASTs. Using



(a) Standard RNN model and its unfolded architecture through time step

(b) The LSTM unit

Figure 1: An illustration of basic RNN and LSTM

SBT, a subtree under a given node is included into a pair of brackets. The brackets represent the structure of the AST and we can restore a tree unambiguously from a sequence generated using SBT.

Moreover, to address the vocabulary challenge, we propose a new method to represent unknown tokens. The tokens in AST sequences include terminal nodes, non-terminal nodes, and brackets in our work. The unknown tokens come from the terminal tokens of ASTs. We replace the unknown tokens with their types instead of a universal special ⟨UNK⟩ token.

DeepCom generates comments word-by-word from AST sequences. We train and evaluate DeepCom on the Java dataset that consists of 9,714 Java projects from GitHub. The experimental results show that DeepCom can generate informative comments. Additionally, the results show that DeepCom achieves the best performance when compared with a number of baselines including the state-of-the-art approach by Iyer et al. [19].

The main contributions of this paper are as follows:

- We formulate code comments generation task as a machine translation task.
- We customize a sequence-based model to process structural information extracted from source code to generate comments for Java methods. In particular, we propose a new AST traversal method (namely structure-based traversal) and a domain-specific method to deal with out-of-vocabulary tokens better.

Paper organization. The remainder of this paper is organized as follows. Section II presents background materials on language models and NMT. Section III elaborates on the details of DeepCom. Section IV and Section V present the experiment setup and results. Section VI discusses strengths of DeepCom, and threats to validity. Section VII surveys the related work. Finally, Section VIII concludes the paper and points out potential future directions.

2 BACKGROUND

2.1 Language Models

Our work is inspired by the machine translation problem in the NLP field. We exploit the language models learning from a large-scale source code corpus. The models generate code comments from the learned features. The language models learn the probabilistic distribution over sequences of words. They work tremendously well on a large variety of problem (e.g., machine translation [6], speech recognition [9], and question answering [46]).

For a sequence $x = (x_1, x_2, ..., x_n)$ (e.g., a statement), the language model aims to estimate the probability of it. The probability

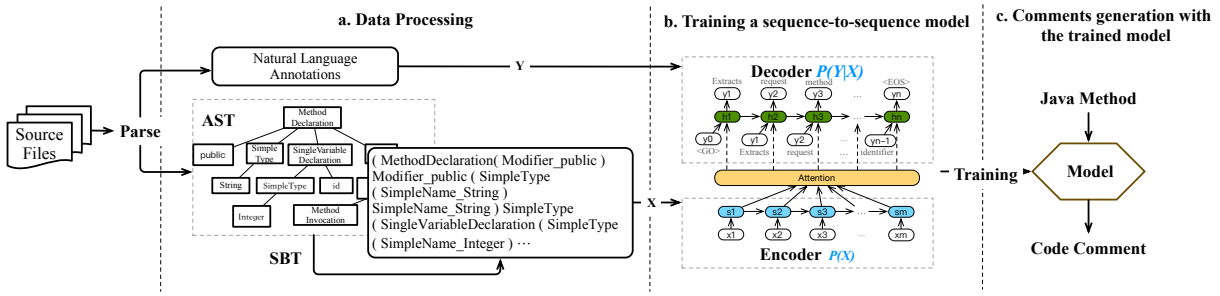


Figure 2: Overall framework of DeepCom.

of a sequence is computed via each of its tokens. That is,

$$P(x) = P(x_1)P(x2|x_1)...P(x_n|x_1...x_{n-1}) \quad (1)$$

In this paper, we adopt a language model based on the deep neural network called Long Short-Term Memory (LSTM) [18]. LSTM is one of the state-of-the-art RNNs. LSTM outperforms general RNN because it is capable of learning long-term dependencies. It is a natural model to use for source code which has long dependencies (e.g., a class is used far away from its import statement). The details of RNN and LSTM are shown in Figure 1.

2.1.1 Recurrent Neural Networks. RNNs are intimately related to sequences and lists because of their chain-like natures. It can in principle map from the entire history of previous inputs to each output. At each time step t, the unit in the RNN takes not only the input of the current step but also the hidden state outputted by its previous time step $t-1$. As Figure 1(a) illustrates, the hidden state of time step t is updated according to the input vector x_t and its previous hidden state h_{t-1}, namely, $h_t = tanh(Wx_t + Uh_{t-1} + b)$ where W, U, and b are the trainable parameters which are updated while training, and $tanh$ is the activation function: $tanh(z) = (e^z - e^{-z})/(e^z + e^{(-z)})$.

A prominent drawback of the standard RNN model is that gradients may explode or vanish during the back-propagation. These phenomena often appear when long dependencies exist in the sequences. To address these problems, some researchers have proposed several variants to preserve long-term dependencies. These variants include LSTM and Gated Recurrent Unit (GRU). In this paper, we adopt the LSTM which has achieved success on many NLP tasks [6, 37].

2.1.2 Long Short-Term Memory. LSTM introduces a structure called the *memory cell* to solve the problem that ordinary RNN is difficult to learn long-term dependencies in the data. The LSTM is trained to selectively "forget" information from the hidden states, thus allowing room to take in more important information [18]. LSTM introduces a gating mechanism to control when and how to read previous information from the memory cell and write new information. The memory cell vector in the recurrent unit preserves long-term dependencies. In this way, LSTM handles long-term dependencies more effectively than vanilla RNN. LSTM has been widely used to solve semantically related tasks and has achieved convincing performance. These advantages motivate us to exploit LSTM for building models for source code and comments. Figure 1(b) illustrates a typical LSTM unit and for more details of LSTM, please refer to [10, 18].

2.2 Neural Machine Translation

NMT [44] is an end-to-end learning approach for automated translation. It is a deep learning based approach and has made rapid progress in recent years. NMT has shown impressive results surpassing those of phrase-based systems while addressing shortcomings such as the need for hand engineered features. Its architecture typically consists of two RNNs, one to consume the input text sequences and the other one to generate the translated output sequences. It is often accompanied by an attention mechanism that aligns target with source tokens [6].

NMT bridges the gap between different natural languages. Generating comments from the source code is a variant of machine translation problem between the source code and the natural language. We explore whether the NMT approach can be applied to comments generation. In this paper, we follow the common Sequence-to-Sequence (Seq2Seq) [37] learning framework with attention [6] which helps cope effectively with the long source code.

3 PROPOSED APPROACH

The transition process between source code and comments is similar to the translation process between different natural languages. Existing research has applied machine translation methods translating code from one source language (e.g., Java) to another (e.g., C#) [13]. A few studies adopt machine translation method for generating natural language descriptions from the source code. Oda et al. [30] present a machine translation approach to generate natural language Pseudo-code of the source code at the statement level. In this paper, DeepCom translates the source code to a high-level description at the method level.

The overall framework of DeepCom is illustrated in Figure 2. DeepCom mainly consists of three stages: data processing, model training, and online testing. The source code we obtained from GitHub is parsed and preprocessed into a parallel corpus of Java methods and their corresponding comments. In order to learn the structural information, the Java methods are converted into AST sequences by a special traversal approach before input into the model. With the parallel corpus of AST sequences and comments, we build and train generative neural models based on the idea of NMT. There are two challenges during training process:

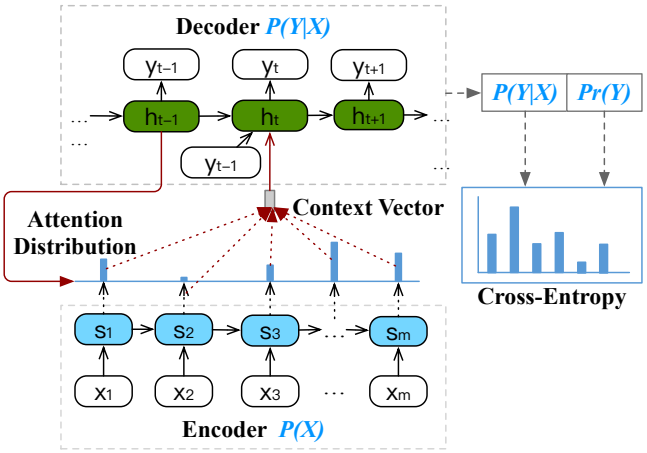


Figure 3: Sequence-to-Sequence model.

- How to represent ASTs to store the structural information and keep the representation unambiguous while traversing the ASTs?
- How to deal with out-of-vocabulary tokens in source code?

In the following paragraphs, we will introduce the details of the model and the approaches we propose to resolve the above-mentioned challenges.

3.1 Sequence-to-Sequence Model

In this paper, we apply a Sequence-to-Sequence (Seq2Seq) model to learn source code and generate comments. Seq2Seq model is widely used for machine translation [37], text summarization [34], dialogue system [39], etc. The model consists of three components, an Encoder, a Decoder, and an Attention component, in which the Encoder and Decoder are both LSTMs. Figure 3 illustrates the detailed Seq2Seq model.

3.1.1 Encoder. The encoder is an LSTM we describe in Section 2 and responsible for learning the source code. At each time step t, it reads one token x_t of the sequence, then updates and records the current hidden state s_t, namely,

$$s_t = f(x_t, s_{t-1}) \tag{2}$$

where f is an LSTM unit that maps a word of source language x_t into a hidden state s_t. The encoder learns latent features from source code, and the features are encoded into the context vector c. These latent features include the identifiers naming conventions, control structures, and etc. In this paper, DeepCom adopts the attention mechanism to compute the context vector c.

3.1.2 Attention. Attention mechanism is a recent model that selects the important parts from the input sequence for each target word. For example, the token "whether" in comments usually aligns with the "if" statements in the source code. The generation of each word is guided by a classic attention method proposed by Bahdanau et al. [6].

It defines individual c_i for predicting each target word y_i as a weighted sum of all hidden states $s_1, .., s_m$ in encoder and computed as

$$c_i = \sum_{j=1}^{m} \alpha_{ij} s_j \tag{3}$$

The weight α_{ij} of each hidden state s_j is computed as

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{m} exp(e_{ik})} \tag{4}$$

and

$$e_{ij} = a(h_{i-1}, s_j) \tag{5}$$

is an alignment model which scores how well the inputs around position j and the output at position i match.

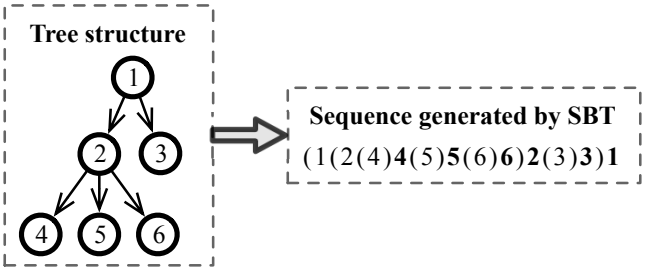


Figure 4: An example of sequencing an AST to a sequence by SBT. (For a number, the bold font number after bracket indicates node itself and the number in brackets denotes the tree structure by taking it as the root node.)

3.1.3 Decoder. The Decoder aims to generate the target sequence y by sequentially predicting the probability of a word y_i conditioned on the context vector c_i and its previous generated words $y_1, ..., y_{i-1}$, i.e.,

$$p(y_i|y_1, ..., y_{i-1}, x) = g(y_{i-1}, h_i, c_i) \tag{6}$$

where g is used to estimate the probability of the word y_i. The goal of the model is to minimize the cross-entropy, i.e., minimize the following objective function:

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

where N is the total number of training instances, and n is the length of each target sequence. $y_j^{(i)}$ means the jth word in the ith instance. Through optimizing the objective function using optimization algorithms such as gradient descendant, the parameters can be estimated.

3.2 Abstract Syntax Tree with SBT traversal

Translation between source code and NL is challenging due to the structure of source code. One simple way to model source code is to just view it as plain text. However, in such way, the structure information will be omitted, which will cause inaccuracies in the generated comments. To learn the semantic and syntactic information at the same time, we convert the ASTs into specially formatted sequences by traversing the ASTs. Sequences obtained by classical traversal methods (e.g., pre-order traversal) are lossy since the original ASTs cannot unambiguously be reconstructed back from them. This ambiguity may cause different Java methods (each with different comments) to be mapped to the same sequence

representation. It is confusing for the neural network if there are multiple labels (in our setting, comments) given to a specific input. For addressing this problem, we propose a Structure-based Traversal (SBT) method to traverse the AST. The details are presented in Algorithm 1. Figure 4 illustrates a simple example of SBT to traverse a tree and the detailed procedure is as follows:

- From the root node, we first use a pair of brackets to represent the tree structure and put the root node itself behind the right bracket, that is (1)1, shown in Figure 4.
- Next, we traverse the subtrees of the root node and put all root nodes of subtrees into the brackets, i.e., (1(2)2(3)3)1.
- Recursively, we traverse each subtree until all nodes are traversed and the final sequence (1(2(4)4(5)5(6)6)2(3)3)1 is obtained.

Algorithm 1 Structure-based Traversal

```
1:  procedure SBT(r)                          ▷ Traverse a tree from root r
2:      seq ← ∅                               ▷ seq is the sequence of a tree after traversal
3:      if !r.hasChild then
4:          seq ← (r)r                        ▷ Add brackets for terminal nodes
5:      else
6:          seq ← (r                          ▷ Add left bracket for non-terminal nodes
7:          for c in childs do
8:              seq ← seq + SBT(c)
9:          seq ← seq+)r                      ▷ Add right bracket for non-terminal
    nodes after traversing all their children
10:     return seq
```

DeepCom processes each AST into a sequence following the SBT algorithm. For example, the AST sequence of the following Java method extracted from project Eclipse Che[2] is shown in Figure 5:

```
public String extractFor(Integer id){
  LOG.debug("Extracting method with ID:{}", id);
  return requests.remove(id);
}
```

The left part of Figure 5 is the AST of the method. The non-terminal nodes (those without boxes) illustrate the structural information of source code. They have the feature "type" which is a fixed set (e.g., IfStatement, Block, and ReturnStatement). The terminal nodes (those within boxes) not only have "type" but also have "value" (token within brackets). The "value" is the concrete token occurring in the source code and "type" indicates the type of the token. The right part of the figure is the sequence constructed by traversing the AST. The terminal nodes are represented by their "type" and "value" (connected by "_"), such as "log" is represented by "SimpleName_Log". The non-terminal nodes are represented by their "type". A subtree is included in a pair of brackets and we can restore the AST from the given sequence easily. In this way, we can keep the structural information and make the representation lossless – the original AST can be unambiguously reconstructed from the sequence.

3.3 Out-of-vocabulary tokens

Vocabulary is another challenge to model source code [16]. In NL, studies usually limit vocabulary to the most common words (e.g., top 30,000) during data processing. The out-of-vocabulary tokens are replaced by a special unknown token, e.g., ⟨UNK⟩. It is effective for NLP because words outside vocabulary are so rare. However, this method is arguably inappropriate when it comes to source code. In addition to fixed operators and keywords, there are user-defined identifiers which take up the majority of code tokens [7]. These identifiers have a substantial influence on the vocabulary of language models. If we keep a regular vocabulary size for source code, there will be many unknown tokens. If we want the occurrences of ⟨UNK⟩ tokens to be as few as possible, the vocabulary size will increase a lot. A large vocabulary size will make it difficult to train a deep learning model since it requires more training data, time, and memory. To achieve optimal and stable results, models need to run a larger number of iterations to tune the parameters for each word in the vocabulary.

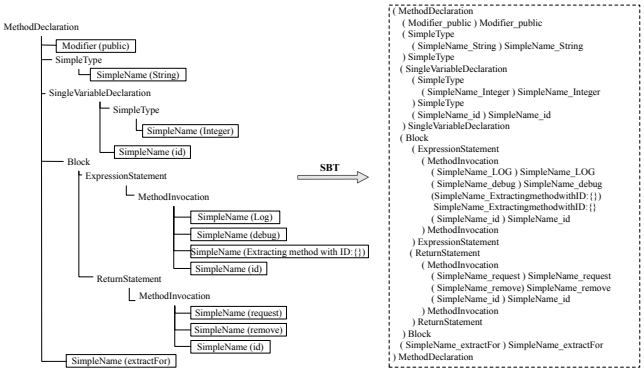


Figure 5: AST of the Java method named *extractFor*.

Hence, we propose a new method to represent the out-of-vocabulary tokens for source code. In AST, the non-terminal nodes have "type" feature, and terminal nodes not only have "type" feature, but also have "value" feature. DeepCom takes the AST sequences as inputs, the vocabulary consists of brackets, all "type" of nodes (including non-terminal nodes T_{non} and terminal nodes T_{term}), and partial *type-value* pairs of terminal tokens. We keep the tokens which appear in the most frequent 30,000 tokens as the AST sequences vocabulary. For the *type-value* pairs outside the vocabulary, DeepCom uses their "type" T_{term} instead of the ⟨UNK⟩ token to replace them. For example, for the terminal nodes "extractFor" and "id" in the code presented above, their types are both "SimpleName" as shown in Figure 5. The tokens input into the model should be "SimpleName_extractFor" and "SimpleName_id" respectively. However, since the token "SimpleName_extractFor" is out of the vocabulary, we use its type "SimpleName" representing it instead. In this way, the out-of-vocabulary tokens are represented by their related type information instead of the meaningless word.

4 EXPERIMENT SETUP

Then we use the Eclipse's JDT compiler[3] to parse the Java methods into ASTs and extract corresponding Javadoc comments which are standard comments for Java methods. The methods without Javadoc are omitted in this paper. For each method with a comment, we use the first sentence appeared in its Javadoc description as

[2]https://github.com/eclipse/che

[3]http://www.eclipse.org/jdt/

Table 1: Statistics for code snippets in our dataset

#Methods	#All Tokens	# All Identifiers	# Unique Tokens	#Unique Identifiers
588,108	44,378,497	13,779,297	794,711	794,621

Table 2: Statistics for code lengths and comments lengths

Code Lengths					
Avg	Mode	Median	<100	<150	<200
99.94	16	65	68.63%	82.06%	89.00%
Comments Lengths					
Avg	Mode	Median	<20	<30	<50
8.86	8	13	75.50%	86.79%	95.45%

the comment since it typically describes the functionalities of Java methods according to Javadoc guidance[4]. Empty or just one-word descriptions are filtered out in this work because these comments have no ability to express the Java methods functionalities. We also exclude the setter, getter, constructor and test methods, since they are easy for a model to generate the comments.

Finally, we get 588,108 ⟨ Java method, comment⟩ pairs[5]. Similar to Jiang et al. [20]'s work, we randomly select 80% of the pairs for training, 10% of the pairs for validation, and rest 10% for testing.

Table 1 and Table 2 illustrate statistics of the corpus. We also give the details of methods lengths and comments lengths. The average lengths of Java methods and comments are 99.94 and 8.86 tokens in this corpus. We find that more than 95% code comments have no more than 50 words and about 90% Java methods no longer than 200 tokens.

During the training, the numerals and strings are replaced with generic tokens ⟨NUM⟩ and ⟨STR⟩ respectively. The maximum length of AST sequences is set to 400. We use a special symbol ⟨PAD⟩ to pad the shorter sequences and the longer sequences will be cut into sequences with 400 tokens. We add special tokens ⟨START⟩ and ⟨EOS⟩ to the decoder sequences during training. ⟨START⟩ is the start of the decoding sequence and the ⟨EOS⟩ means the end of it. The maximum comment length is set to 30. The vocabulary sizes for AST sequences and comments are both 30,000 in this paper. While there is no ⟨UNK⟩ in ASTs sequences, there are a few out-of-vocabulary tokens in comments that are replaced by ⟨UNK⟩.

4.1 Training Details

The model is validated every 2,000 minibatches on the validation set by BLEU [31] which is a commonly used automatic metric for NMT. Training runs for about 50 epochs and we select the best model that has best results on the validation set as the final model. The model is then evaluated on the test set by computing average BLEU scores and the results will be discussed in Section 5. All models are implemented using the Tensorflow framework[6] and extended based on the Seq2Seq model in Tensorflow tutorials[7]. The parameters are shown as follows:

- The SGD (with minibatch size 100 randomly chosen from training instances) is used to train the parameters.
- DeepCom uses two-layered LSTMs with 512 dimensions of the hidden states and 512-dimensional word embeddings.
- The learning rate is set to 0.5 and we clip the gradients norm by 5. The learning rate is decayed using the rate 0.99.
- To prevent over-fitting, we use dropout with 0.5.

4.2 Evaluation Measure: BLEU-4

DeepCom uses machine translation evaluation metrics BLEU-4 score[31] to measure the quality of generated comments. BLEU score is a widely-used accuracy measure for NMT [22] and has been used in software tasks evaluation [12, 20]. It calculates the similarity between the generated sequence and reference sequence (usually a human-written sequence). The BLEU score ranges from 1 to 100 as a percentage value. The higher the BLEU, the closer the candidate is to the reference. If the candidate is completely equal to the reference, the BLEU becomes 100%. Jiang et al. [20] exploit it to evaluate the generated summaries for commit messages. Gu et al. [12] use BLEU to evaluate the accuracy of generated API sequences from natural language queries. Their experiments show that BLEU score is reasonable to measure the accuracy of generated sequences.

It computes the n-gram precision of a candidate sequence to the reference. The score is computed as:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n log p_n) \tag{8}$$

where p_n is the ratio of length n subsequences in the candidate that are also in the reference. In this paper, we set N to 4, which is the maximum number of grams. BP is brevity penalty,

$$BP = \begin{cases} 1 & if c > r \\ e^{(1-r/c)} & if c \leq r \end{cases} \tag{9}$$

where c is the length of the candidate translation and r is the effective reference sequence length.

In this paper, we regard a generated comment as a candidate and a programmer-written comment (extracted from Javadoc) as a reference.

5 RESULTS

In this section, we evaluate different approaches by measuring their accuracy on generating Java methods' comments. Specifically, we mainly focus on the following research questions:

- RQ1: How effective is DeepCom compared with the state-of-the-art baseline?
- RQ2: How effective is DeepCom to source code and comments of varying lengths?

5.1 RQ1: DeepCom vs. Baseline

5.1.1 Baseline. We compare DeepCom with CODE-NN [19] which is a state-of-the-art code summarization approach and also a deep learning based method. CODE-NN is an end-to-end generation

[4]http://www.oracle.com/technetwork/articles/java/index- 137868.html
[5]Data is available at https://github.com/huxingfree/DeepCom
[6]https://www.tensorflow.org/
[7]https://github.com/tensorflow/nmt

Table 3: Evaluation results on Java methods

Approaches	BLEU-4 score (%)
CODE-NN	25.30
Seq2Seq	34.87
Attention-based Seq2Seq	35.50
DeepCom (Pre-order)	36.01
DeepCom (SBT)	**38.17**

system to generate summaries for code snippets. It exploits an RNN with attention to generate summaries by integrating the token embeddings of source code instead of building language models for source code. We do not use IR approaches as baselines, because the results in CODE-NN has shown that CODE-NN outperforms the IR based approaches.

We also compare DeepCom with its variants, that are, the basic Seq2Seq model, the attention based Seq2Seq model, and DeepCom with a classical traversal method (i.e., pre-order traversal). The Seq2Seq model and the attention based Seq2Seq model take the source code as inputs. They aim to evaluate the effectiveness of NMT approaches for comments generation. To evaluate the effectiveness of SBT, we compare SBT with one of the most ordinary traversal methods – pre-order traversal. In addition, we also compare DeepCom with CODE-NN on the dataset that CODE-NN uses.

5.1.2 Results. We measure the gap between automatically generated comments and human-written comments. The difference is evaluated by a machine translation metric, i.e., BLEU-4 score. Table 3 illustrates the average BLEU-4 scores of different approaches to generating comments for Java methods. The accuracy of machine translation model Seq2Seq substantially outperforms CODE-NN. CODE-NN fails to learn the semantic of the source code when it generates comments from token embeddings of source code directly. Seq2Seq model exploits RNN to build a language model for the source code and effectively learns the semantics of Java methods. The BLEU-4 score increases further while integrating the structural information. Compared to DeepCom with the pre-order traversal, the SBT based model is much more capable of learning semantic and syntactic information within Java methods. In a word, the improvement of our proposed DeepCom (SBT) over CODE-NN is large. The average BLEU-4 score of DeepCom improves about 13% compared to CODE-NN. The results of DeepCom are comparable to the BLEU scores of state-of-the-art NMT models on natural language translation which are about 40%[21, 44].

We further conduct experiments on the same datasets CODE-NN used, which includes C# and SQL snippets collected from Stack Overflow. The results are shown in Table 4. Since many of the code snippets in their provided dataset are incomplete and hard to parse them into ASTs, we compare the Seq2Seq model with CODE-NN. It highlights that the Seq2Seq outperforms the state-of-the-art method CODE-NN in different languages. The average BLEU scores of Seq2Seq improve more than 10% on various program languages compared to CODE-NN.

Through the evaluation, we have verified that comments generation task is very similar to machine translation except that the

Table 4: Evaluation results on CODE-NN datasets including C# and SQL programming languages.

Language	Approaches	BLEU-4 score(%)
C#	CODE-NN	20.4
	Seq2Seq	**30.00**
SQL	CODE-NN	17.0
	Seq2Seq	**30.94**

structural information in source code needs to be taken into account. DeepCom can generate more informative comments than the state-of-the-art method. Compared to the model without AST, the BLEU score of DeepCom increases to 38.17% and the BLEU-4 scores of about 38% of the instances are greater than 50%. We evaluate two traversal methods SBT and pre-order traversal. DeepCom with SBT performs better than traditional pre-order traversal. This is the case because SBT better preserves the structure of ASTs. Experimental results indicate that the structural information is important for translating text in structured languages to unstructured ones.

5.2 RQ2: BLEU-4 scores for source code and comments of different lengths

We further analyze the prediction accuracy for Java methods and comments of different lengths. Figure 6 presents the average BLEU-4 scores of DeepCom and CODE-NN for source code and ground truth comments of varying lengths. As Figure 6(a) illustrates, the average BLEU-4 scores tend to be lower when we increase source code length. For most code lengths, the average BLEU-4 scores of DeepCom improve those of CODE-NN by about 10%. For DeepCom, AST lengths grow rapidly as the source code lengths increase and as a result, some features are lost when cutting the long AST sequences into a fixed length sequence during training.

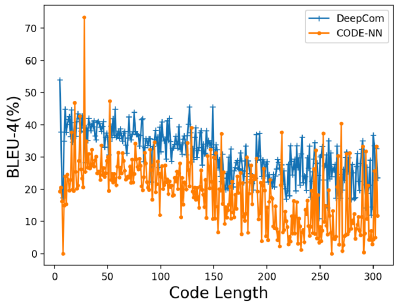
For comments of different lengths, DeepCom maintains similar accuracy as shown in Figure 6(b). However, the accuracy of CODE-NN decreases sharply while code comment length increases. When the code comment lengths are greater than 25 tokens, the accuracy of CODE-NN decreases to less than 10%. DeepCom still performs better when we need to generate comments consisting of 25-28 words.

6 DISCUSSION

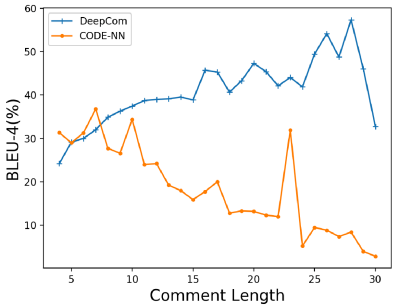
6.1 Qualitative analysis

Here, we perform qualitative analysis on the human-written comments and comments which are automatically generated by our approach. Table 5 shows some examples of Java methods, the comments generated by DeepCom and human-written comments. By analyzing cases of generated results, we find the cases can be divided into the following situations.

6.1.1 Exactly correct comments. DeepCom can generate exactly correct comments from the source code of different lengths (Case 1 and Case 2), which validate the capability of our approach to encode Java methods and decode comments. Generally, DeepCom



(a) BLEU-4 scores for different code lengths



(b) BLEU-4 scores for different comment lengths

Figure 6: The average BLEU-4 scores of different lengths of code and comment in Java language. (We compare two methods DeepCom with SBT and CODE-NN)

performs well when the business logic of these Java methods is clear and code conventions are universal.

6.1.2 Algorithm implementations. For the Java methods which are more concerned about algorithm than business logic, DeepCom can generate accurate comments. The algorithm concerned Java methods usually use similar structures to implement the same algorithm function. As Case 5 shows, the method "sort" aims to sort an array using Binary Sort, DeepCom captures the correct functionality and generates the correct comments.

6.1.3 Cases when generated comments are better than human-written ones. By analyzing the generated comments and the source code, we find that DeepCom performs better than human written comments when the Java methods aim to determine something true or not. Developers write interrogative sentences as comments sometimes (shown in Case 6 and Case 7). These comments are nonstandard even though they can express the functionalities of Java methods. DeepCom can not only generate accurate comments but also more standard comments.

6.1.4 API invocations intensive Java methods. Developers usually invoke APIs to implement a specific function. These APIs include platform standard APIs and customized APIs defined by third parties or developers themselves. We find that DeepCom can generate accurate comments when most API invocations are platform standard APIs (shown in Case 1). However, when the majority API invocations in a Java method are customized APIs, DeepCom does not perform as good as human-written comments (shown in Case 4 and Case 9). The influence of API invocations explains that DeepCom can learn the platform standard APIs usage patterns from a large-scale dataset. However, it can not learn customized APIs well because the customized APIs with the same name have different usage patterns in different programs.

6.1.5 Low BLEU score cases. The results with lower BLEU scores are mainly divided into two types, meaningless sentences, and sentences with clear semantics. The former mainly contains empty sentences and results with too many repetitive words. We conjecture the problems come from out-of-vocabulary words in original comments or mismatch between the Java methods and comments in the original dataset.

In the latter ones, most of them are irrelevant to original comments in their semantics. There are also some interesting results that hold relevant semantics but gain low BLEU scores (shown in Case 4). The automatically generated and manual comments may describe similar functionalities but with different words or order.

6.1.6 Unknown words in generated comments. There are unknown words in the generated comments sometimes. As Case 3 shows, DeepCom fails to predict the token "FactoryConfigurationError" which is the method name defined by developers. DeepCom is not good at learning the method or identifiers names occurred in comments. Developers define various names while programming and most of these tokens appearing at most once in the comments. During the training process, we replaced all unknown identifier tokens in AST sequences with their types, but we do not replace the unknown identifiers occur in comments. It is hard for DeepCom to learn these user-defined tokens in comments that have been replaced by the unknown token ⟨UNK⟩.

6.2 Strengths of DeepCom

A major challenge for generating comments from code is the semantic gap between code and natural language descriptions. Existing approaches are based on manually crafted templates or information retrieval and lack a model to capture the semantic relationship between source code and natural language. DeepCom, a machine translation model, has the ability to bridge the gap between two languages, i.e., programming language and natural language.

6.2.1 Probabilistic model connecting semantics of code and comments. One advantage of DeepCom is generating comments directly by learning source code instead of synthesizing comments from keywords or searching similar code snippets' comments.

Synthesizing comments from keywords usually uses some manually crafted templates. The procedure of templates definition is time-consuming and the quality of keywords depends on the quality of a given Java method. They fail to extract accurate keywords when the identifiers and methods are poorly named. The IR based approaches usually search the similar code snippets and take their comments as the final results. These IR based approaches rely on whether similar code snippets can be retrieved and how similar the snippets are.

DeepCom builds language models for code and natural language descriptions. The language models are able to handle the uncertainty in the correspondence between code and text. DeepCom

Table 5: Examples of generated comments by DeepCom. These samples are necessarily limited to short methods because of space limitations. AST structure is not shown in the table, because AST is much longer than source code.

Case ID	Java method	Comments
1	`public static byte[] bitmapToByte(Bitmap b){ ByteArrayOutputStream o = new ByteArrayOutputStream(); b.compress(Bitmap.CompressFormat.PNG,100,o); return o.toByteArray(); }`	Automatically generated: convert Bitmap to byte array Human-written: convert Bitmap to byte array
2	`private static void addDefaultProfile(SpringApplication app, SimpleCommandLinePropertySource source){ if(!source.containsProperty("spring.profiles.active") &&!System.getenv().containsKey("SPRING_PROFILES_ACTIVE")){ app.setAdditionalProfiles(Constants.SPRING_PROFILE_DEVELOPMENT); } }`	Automatically generated: If no profile has been configured , set by default the "dev" profile. Human-written: If no profile has been configured , set by default the "dev" profile.
3	`public FactoryConfigurationError(Exception e){ super(e.toString()); this.exception=e; }`	Automatically generated: Create a new ⟨UNK⟩ with a given Exception base cause of the error. Human-written: Create a new FactoryConfigurationError with a given Exception base cause of the error.
4	`protected void createItemsLayout(){ if (mItemsLayout == null){ mItemsLayout=new LinearLayout(getContext()); mItemsLayout.setOrientation(LinearLayout.VERTICAL); } }`	Automatically generated: Creates item layouts if any parameters Human-written: Creates item layout if necessary
5	`public static void sort(Comparable[] a){ int n=a.length; for (int i=1; i < n; i++){ Comparable v=a[i]; int lo=0, hi=i; while (lo < hi) { ... } ... } assert isSorted(a); }`	Automatically generated: Sorts the array in ascending order,using the natural order. Human-written: Rearranges the array in ascending order,using the natural order.
6	`public boolean isEmpty(){ return root == null; }`	Automatically generated: Returns true if the symbol is empty. Human-written: Is this symbol table empty?
7	`public boolean contains(int key){ return rank(key) != -1; }`	Automatically generated: Checks whether the given object is contained within the given set. Human-written: Is the key in this set of integers?
8	`public void tag(String inputFileName,String outputFileName, OutputFormat outputFormat){ List<String> sentences=jsc.textFile(inputFileName).collect(); tag(sentences,outputFileName,outputFormat); }`	Automatically generated: Replaces the message with a given tag Human-written: Tags a text file , each sentence in a line and writes the result to an output file with a desired output format.
9	`public void unlisten(String pattern){ UtilListener listener=listeners.get(pattern); if(listener!=null){ listener.destroy(); listeners.remove(pattern); }else{ client.onError(Topic.RECORD,Event.NOT_LISTENING,pattern); } }`	Automatically generated: It can be called when the product only or refresh has ended. Human-written: Removes a listener that was previously registered with listenForSubscriptions.

learns common patterns from a large-scale source code and the encoder itself is a language model which remembers the likelihood of different Java methods. The decoder of DeepCom learns the context of source code which bridges the gap between natural language and code. Furthermore, the attention mechanism helps align code tokens and natural language words.

6.2.2 Generation assisted by structural information. Programming languages are formal languages which are more structure dense than text and have formal syntax and semantics. It is difficult for models to learn semantic and syntax information at the same time just given code sequences. Existing approaches usually analyze source code directly and omit its syntax representation.

In contrast to traditional NMT models, DeepCom takes advantage of rich and unambiguous code structures. In this way, DeepCom bridges the gap between code and natural language with the assistance of structure information within the source code. From

the evaluation results, we find that the structural information improves the quality of comments. The improvements for methods implementing standard algorithms are much more obvious. Java methods realizing the same algorithm may define different variables while their ASTs are much more similar.

6.3 Threats to Validity

We have identified the following threats to validity:

Automatic evaluation metrics: We evaluate the gap between generated comments and human-written comments by machine translation metric BLEU which is gradually used in generative-based software issues [12, 20]. The reason for this setting is that we want to reduce the impact of the subjectivity of manual evaluation.

Quality of collected comments: We collected the comments for Java methods from the first sentence of Javadoc as other work does [12]. Although we define heuristic rules to decrease the noise in comments, there are some mismatched comments in the dataset. In the future, we will investigate a better technique to build a better parallel corpus.

Comparisons on Java dataset: Another threat to validity is that our approach is experimented on Java dataset. Although we fail to evaluate DeepCom directly on CODE-NN' dataset which is difficult to parse into ASTs, the results on Java have proved the effectiveness of DeepCom. In the future, we will extend our approach to other programming languages (e.g., Python).

7 RELATED WORK

7.1 Code Summarization

As a critical task in software engineering, code summarization aims to generate brief natural language descriptions for source code. Automatic code summarization approaches vary from manually-crafted template [24, 35, 36], IR [14, 15, 43] to learning-based approaches [4, 19, 28].

Creating manually-crafted templates to generate code comments is one of the most common code summarization approaches. Sridhara et al. [35] use the Software Word Usage Model (SWUM) to create a rule-based model that generates natural language descriptions for Java methods. Moreno et al. [25] predefine heuristic rules to select information and generate comments for Java classes by combining the information. These rule-based approaches have been expanded to cover special types of code artifacts such as test cases [48] and code changes [8]. Human templates usually synthesize comments by extracting keywords from the given source code.

IR approaches are widely used in summary generation and usually search comments from similar code snippets. Haiduc et al. [15] apply the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) to generate term-based comments for classes and methods. Their works are replicated and expanded by Eddy et al. [11] which exploit a hierarchical topic model. Wong et al. [42] apply code clone detection techniques to find similar code snippets and use the comments from similar code snippets. The work is similar to their previous work AutoComment [43] which mines human-written descriptions for automatic comment generation from Stack Overflow.

Recently, some studies try giving natural language summaries by deep learning approaches. Iyer et al. [19] present RNN networks with attention to produce summaries that describe C# code snippets and SQL queries. It takes source code as plain text and models the conditional distribution of the summary. Allamanis et al. [4] apply a neural convolutional attentional model to the problem that extremely summarizes the source code snippets into short, name-like summaries. These learning-based approaches mainly learn the latent features from source code, such as semantics, formatting, and etc. The comments are generated according to these learned features. The experimental results of them have proved the effectiveness of deep learning methods on code summarization. In this paper, DeepCom integrates the structure information which is verified important for comments generation.

7.2 Language models for source code

Recently, thanks to the insight of Hindle et al. [17], there is an emerging interest in building language models of source code. These language models vary from n-gram model [1, 29], bimodal model [5], and RNNs [12, 19]. Hindle et al. [17] first propose to explore N-gram to model the source code and demonstrate that most software is also natural and find regularities in natural code. Some studies build the models to bridge the gap between the programming language and natural language descriptions. Allamanis et al. [1] develop a framework to learn the code conventions of a codebase and the framework exploits N-gram model to name Java identifiers. Allamanis et al. [2] and Raychev et al. [33] suggest names for variables, methods, and classes. Mou et al. [26] present a tree-based convolutional neural networks to model the source code and classify programs. Gu et al. [12] present a classic encoder-decoder model to bridge the gap between the Java API sequences and natural language. Yin and Neubig [47] build a data-driven syntax-based neural network model for generating code from natural language.

Learning from source code is applied to various software engineering tasks, e.g., fault detection [32], code completion [27, 29], code clone [38] and code summarization [19]. In this paper, we explore the combination of deep learning methods and source code features to generate code comments. Compared to the previous works, DeepCom explains the code summarization procedure from a machine translation perspective. The experimental results also prove the ability of DeepCom.

8 CONCLUSION

This paper formulates code summarization task as a machine translation problem which translates source code written in a programming language to comments in natural language. We propose DeepCom, an attention-based Seq2Seq model, to generate comments for Java methods. DeepCom takes ASTs sequences as input. These ASTs are converted to specially formatted sequences using a new structure-based traversal (SBT) method. SBT can express the structural information and keep the representation lossless at the same time. DeepCom outperforms the state-of-the-art approaches and achieves better results on the machine translation metric. In future work, we plan to improve the effectiveness of our proposed approach by introducing more domain-specific customizations. We also plan to apply our proposed approach to other software engineering tasks that can be mapped to a machine translation problem (e.g., code migration, etc.).

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).

[4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[5] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *Computer Science* (2014).

[7] Manfred Broy, Florian Deißenböck, and Markus Pizka. 2005. A holistic approach to software quality at work. In *Proc. 3rd World Congress for Software Quality (3WCSQ)*.

[8] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 33–42.

[9] Ciprian Chelba, Dan Bikel, Maria Shugrina, Patrick Nguyen, and Shankar Kumar. 2012. Large scale language modeling in automatic speech recognition. *arXiv preprint arXiv:1210.8440* (2012).

[10] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[11] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 13–22.

[12] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[13] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. *arXiv preprint arXiv:1704.07734* (2017).

[14] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 223–226.

[15] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 35–44.

[16] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[17] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model.. In *ACL (1)*.

[20] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 135–146.

[21] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, et al. 2016. Google's multilingual neural machine translation system: enabling zero-shot translation. *arXiv preprint arXiv:1611.04558* (2016).

[22] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[23] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. *arXiv preprint arXiv:1704.04856* (2017).

[24] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 279–290.

[25] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 23–32.

[26] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.. In *AAAI*, Vol. 2. 4.

[27] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211* (2015).

[28] Dana Movshovitz-Attias and William W Cohen. 2013. Natural language models for predicting programming comments. (2013).

[29] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 532–542.

[30] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 574–584.

[31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[32] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 428–439.

[33] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.

[34] Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* (2015).

[35] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[36] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[37] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[38] Jeffrey Svajlenko and Chanchal K Roy. 2016. A Machine Learning Based Approach for Evaluating Clone Detection Tools for a Generalized and Accurate Precision. *International Journal of Software Engineering and Knowledge Engineering* 26, 09n10 (2016), 1399–1429.

[39] Oriol Vinyals and Quoc Le. 2015. A neural conversational model. *arXiv preprint arXiv:1506.05869* (2015).

[40] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 297–308.

[41] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[42] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 380–389.

[43] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 562–567.

[44] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[45] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* (2017).

[46] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. 2015. Neural generative question answering. *arXiv preprint arXiv:1512.01337* (2015).

[47] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *arXiv preprint arXiv:1704.01696* (2017).

[48] Sai Zhang, Cheng Zhang, and Michael D Ernst. 2011. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 63–72.