Front. Comput. Sci. China, 2018, 2(3)
DOI 10.1007/s11704-009-0000-0

RESEARCH ARTICLE

CodeAttention: Translating Source Code to Comments by Exploiting the Code Constructs

Wenhao Zheng, Hong-Yu Zhou, Ming Li (✉) , Jianxin Wu

National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Abstract Appropriate comments of code snippets provide insight for code functionality, which are helpful for program comprehension. However, due to the great cost of authoring with the comments, many code projects do not contain adequate comments. Automatic comment generation techniques have been proposed to generate comments from pieces of code in order to alleviate the human efforts in annotating the code. Most existing approaches attempt to exploit certain correlations (usually manually given) between code and generated comments, which could be easily violated if coding patterns change and hence the performance of comment generation declines. In addition, recent approaches ignore exploiting the code constructs and leveraging the code snippets like plain text. Furthermore, previous datasets are also too small to validate the methods and show their advantage. In this paper, we propose a new attention mechanism called CodeAttention to translate code to comments, which is able to utilize the code constructs, such as critical statements, symbols and keywords. By focusing on these specific points, CodeAttention could understand the semantic meanings of code better than previous methods. To verify our approach in wider coding patterns, we build a large dataset from open projects in GitHub. Experimental results in this large dataset demonstrate that the proposed method has better performance over existing approaches in both objective and subjective evaluation. We also perform ablation studies to determine effects of different parts in CodeAttention.

Keywords software mining; machine learning; code comment generation; recurrent neural network; attention mechanism

1 Introduction

Program comments usually provide insight for code functionality, which are important for program comprehension, maintenance and reusability. For example, comments are helpful for working efficiently in a group or integrating and modifying open-source software. However, since it is time-consuming to create and update comments constantly, plenty of source code, especially the code from open-source software, lack adequate comments [1]. Source code without comments would reduce the maintainability and usability of software.

To mitigate the impact, automatic program annotation techniques have been proposed to automatically supplement the missing comments by analyzing source code. Sridhara et al. [2] generated summary comments by using variable names in code. Rastkar et al. [3] managed to give a summary by reading software bug reports. Mcburney and McMillan [4] leveraged the documentation of API to generate comments of code snippets. Sulir et al. [5] traced the program being executed and generated its method documentation.

Source code is usually structured while the comments in natural language are organized in a relatively free form. Therefore, the key in automatic program annotation is to identify the relationship between the functional semantics of code and its corresponding textual descriptions. Since identifying such relationships from the raw data is rather challenging due to the heterogeneity nature between programming language and natural language, most aforementioned techniques usually rely on certain assumptions on the correlation between the code and their corresponding comments (e.g., providing paired code and comment templates to be filled in), based on which the code is converted to comments in natural language. However, the assumptions may highly be coupled with certain projects while invalid on other projects. Consequently, these approaches may contain large variance in performance on real-world applications.

In order to improve the applicability of automatic code commenting, machine learning has been introduced to learn how to generate comments in natural language for source code of various programming languages. Srinivasan et al. [6] and Allamanis et al. [7] treated source code as natural language texts, and learned a neural network to summarize the words in source code into brief phrases or sentences. However, as pointed out by [8], source code usually carries non-negligible semantics on the program functionality and should not be simply treated as natural language texts. Therefore, the comments generated by [6] may not well capture the functionality semantics embedded in the program structure. For example, as shown in Fig 1, if we only consider the lexical information in this code snippet, the comment would be "*swap two elements in the array*".

Received December 29, 2017; accepted June 7, 2018

E-mail: *lim@nju.edu.cn*

However, if we think about both the structure and the lexical information, the correct comment should be "*shift the first element in the array to the end*".

```
int i = 0;
while(i<n){
  swap(array[i],array[i+1]);
  i++;}
```

Figure 1: An example of code snippet. If the structural semantics provided by the `while` is not considered, comments indicating wrong semantics may be generated.

One question arises: Can we *directly learn a mapping* between two heterogeneous languages? Based on current sequence to sequence framework in machine translation, we propose a novel attention mechanism called CodeAttention to directly *translate* the source code in programming language into comments in natural language. Our approach is able to leverage rich information from code constructs, e.g. the semantic embedding of critical statements such as loops and branches, emphasizing the functional operations of source code like symbols and keywords. By utilizing code constructs, our approach can generate more readable and meaningful comments. To verify the effectiveness of CodeAttention, we build a large dataset collected from open source projects in GitHub. The whole framework of our proposed method is shown in Fig 3. Empirical studies indicate that CodeAttention can generate better comments than previous works, and the comments we generate would conform to the functional semantics in the program, by explicitly modeling the code constructs.

The rest of this paper is organized as follows. After briefly introducing the related work and background, we describe the process of collecting and preprocessing data in Section 4. In Section 5, we introduce the CodeAttention module, which is able to leverage the constructs of the source code. In Section 6, we report the experimental results by comparing it with five popular approaches in both objective and subjective evaluation. On BLEU and METEOR scores, our approach outperforms all other approaches and achieves new state-of-the-art performance in large dataset. The subjective evaluation also shows the same result.

Our contribution can be summarized as: i) we propose a novel model named CodeAttention, which leverages the code constructs information to extract functional semantics from source code. The constructs information can improve the performance of generated comments. Compared with traditional methods, our approach achieves the best results on BLEU and METEOR beating all other methods in different experiments. ii) A new large dataset for code to comments translation, which contains over 1k projects from GitHub, making it more practical and 20× larger than previous datasets [6].

2 Related Work

Previously, there already exist some works on producing code descriptions based on source code. These works mainly focused on how to extract key information from source code through rule-based matching, information retrieval, or probabilistic methods. Sridhara et al. [2] generated conclusive comments of specific source code by using variable names in code. Sridhara et al. [9] used several templates to fit the source code. If one piece of source code matches the template, the corresponding comment would be generated automatically. Movshovitz and Cohen [10] predicted class-level comments by utilizing open source Java projects to learn n-gram and topic models, and they tested their models using a character-saving metric on existing comments. There are also retrieval methods which generate summaries for source code based on automatic text summarization [11], topic modeling [12], or integrating with the physical actions of expert engineers [13].

There are different datasets describing the relation between code and comments. Most datasets are from Stack Overflow [6,14,15] and GitHub [16]. Stack Overflow based datasets usually contain lots of pairs in the form of Q&A, which assume that real world code and comments are also in such pattern. However, this assumption may not hold all the time, since those questions are carefully designed. On the contrary, we argue that current datasets from GitHub are more practical but usually small, for example, Wong et al. [16] only contains 359 comments. In this paper, our proposed dataset is much larger and also has the ability to keep the accuracy.

In most cases, generating comments from source code is the sub-task of sequence to sequence translation. There have been many research works about it in this community. Brown et al. [17] described a series of five statistical models of the translation process and developed an algorithm for estimating the parameters of these models given a set of pairs of sentences that each pair contains mutual translations, and they also defined a concept of word-by-word alignment between such pairs of sentences. Koehn et al. [18] proposed a new phrase-based translation model and decoding algorithm that enabled us to evaluate and compare several previously proposed phrase-based translation models. However, the system itself consists of many small subcomponents and they are designed to be tuned separately. Although these approaches achieved good performance on NLP tasks, few of them have been applied on code to comments translation. Recently, deep neural networks achieve excellent performance on difficult problems such as speech recognition [19], visual object recognition [20] and machine translation [21]. For example, the neural translator proposed in [21] is a newly emerging approach which attempted to build and train a single, large neural network which takes a sentence as an input and outputs a corresponding translated sentence.

Two most relevant works are [6] and [7]. Allamanis et al. [7] mainly focused on extreme summarization of source code snippets into short, descriptive functional summaries but our goal is to generate human-readable comments of code snippets. Srinivasan et al. [6] presented the first completely data driven approach for generating high level summaries of source code by using Long Short Term Memory (LSTM) networks to produce sentences. However, they considered the code snippets as natural language texts and employed roughly the same method in NLP without considering the code constructs.

Although translating source code to comments is similar to language translation, there does exist some differences. For instance, the structure of code snippets is usually much

more complex than that of natural language and usually has some specific features, such as various identifiers and symbols. The length of source code is usually much larger than its comment and some comments are very simple while the code snippets are very complex. All approaches we have mentioned above do not make any optimization for source code translation. In contrast, we design a new attentional mechanism called CodeAttention which is specially optimized for code constructs to help make the translation process more specific. By separating the identifiers and symbols from natural code segments, CodeAttention is able to understand the code snippets in a more structural way.

3 Background

In this section, we introduce the recurrent neural networks (RNNs), a family of neural networks designed for processing sequential data. Some traditional types of neural networks (e.g., convolution neural networks, recursive networks) make an assumption that all elements are independent of each other, while RNNs perform the same task with the output being depended on the previous computations. For instance, in natural language processing, if you want to predict the next word in a sentence you would better know which words come before it.

The seq2seq model

RNN is a kind of neural network that consists of a hidden state **h** and an optional output **y** which operates on a variable length sequence. RNN is able to predict the next symbol in a sequence by modeling a probability distribution over the sequence $\mathbf{x}= (x_1, \ldots, x_T)$. At each timestep t, the hidden state $\mathbf{h}_t$ is updated by

$$\mathbf{h}_t = f_{encoder}(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{1}$$

where $f_{encoder}$ is a non-linear activation function (e.g., sigmoid function [22], LSTM [23], GRU [24]). One usual way of defining the recurrent unit $f_{encoder}$ is an affine transformation plus a nonlinear activation, e.g.,

$$\mathbf{h}_t = tanh(W[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}) \tag{2}$$

where we parameterized the relation between $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ into matrix W, and **b** is the bias term. Each element of its input is activated by the function $tanh$. A simple RNN aims to learn the parameters W and **b**. In this case, we can get the final joint distribution,

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(\mathbf{x}_t|\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) \tag{3}$$

The basic cell unit in RNN is important to decide the final performance. A gated recurrent unit is proposed by Cho et al. [25] to make each recurrent unit to adaptively capture dependencies of different time scales. GRU has gating units but no separate memory cells when compared with LSTM.

GRU contains two gates: an update gate **z** and a reset gate **r** which correspond to forget and input gates in LSTM, respectively. We show the update rules of GRU in the Equations (4) to (7),

$$\mathbf{z}_t = \sigma(W_z[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) \tag{4}$$

$$\mathbf{r}_t = \sigma(W_r[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) \tag{5}$$

$$\widetilde{\mathbf{h}}_t = tanh(W_h[\mathbf{r}_t \odot \mathbf{t}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h) \tag{6}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \widetilde{\mathbf{h}}_t \tag{7}$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$, $\circ$ is the component-wise product between two vectors. There are two reasons which make us choose GRU: the first one is that Chung et al. [26] found that GRU can outperform LSTM both in terms of convergence in CPU time and in terms of parameter updates and generalization; the second is that GRU is much easier to implement and train when compared with LSTM.

In order to learn a better phrase representation, a classical recurrent neural network architecture learns to encode a variable-length input into a fixed-length vector representation and then decode the vector into a variable-length output. To be simple, this architecture bridges the gap between two variable-length vectors. While if we look inside the architecture from a more probabilistic perspective, we can rewrite Eq. (3) into a more general form, e.g., $p(y_1, \ldots, y_K \mid x_1, \ldots, x_T)$, where it is worth noting that the length of input and output may differ in this case.

The above model contains two RNNs. The first one is the encoder, while the other one is used as a decoder. The encoder reads each symbol of an input sequence **x** sequentially. As it reads each symbol, the hidden state of the encoder updates according to Eq. (1). At the end of the input sequence, there is always a symbol telling the end, and after reading this symbol, the last hidden state is a summary **c** of the whole input sequence.

As we have discussed above, the decoder is another RNN which is trained to generate the output sequence by predicting the next symbol $\mathbf{y}_t$ given the hidden state $\mathbf{h}_t$.

$$p(\mathbf{y}_t \mid \mathbf{y}_{t-1}, \ldots, \mathbf{y}_1, \mathbf{c}) = f_{decoder}(\mathbf{h}_t, \mathbf{y}_{t-1}, \mathbf{c}), \tag{8}$$

where $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1}, \mathbf{c})$ and $f_{decoder}$ is usually a softmax function to produce valid probabilities. Note that there are several differences between this one and the original RNN. The first is that the hidden state at timestep t is no longer based on $\mathbf{x}_{t-1}$ but on the $\mathbf{y}_{t-1}$ and the summary **c**, and the second is that we model $\mathbf{y}_t$ and $\mathbf{x}_t$ jointly which may result in a better representation.

The Attention Mechanism

A potential issue with the above encoder-decoder approach is that a recurrent neural network has to compress all the necessary information $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ into a context vector **c** for all time, which means the length of vector **c** is fixed. There are several disadvantages here. This solution may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus, and Cho [27] showed that the performance of a basic encoder-decoder deteriorates rapidly as the length of an input sentence increases. Specifically, when backing to code-to-comment case, every word in the code may have different effects on each word in the comment. For instance, some *keywords* in the source code can have direct influences on the comment while others do nothing to affect the result.

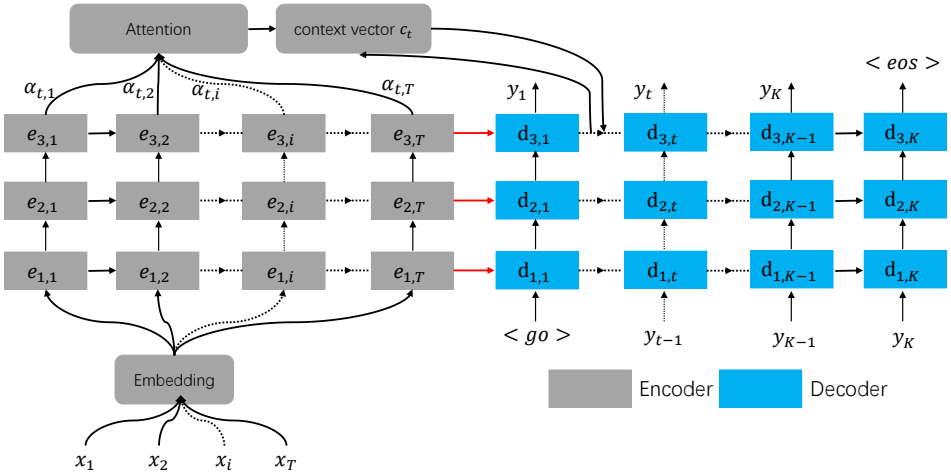

Figure 2: An overview of the classical sequence to sequence model with attention mechanism

Considering all factors we have talked above, a global attention mechanism should exist in a translation system. An overview of the model is provided in Fig. 2. $\mathbf{h}_{i,j}$ is the hidden state located at the ith ($i = 1, 2$) layer and jth ($j = 1, \ldots, T$) position in the encoder. $\mathbf{s}_{i,k}$ is the hidden state located at the ith ($i = 1, 2$) layer and jth ($k = 1, \ldots, K$) position in the decoder. Instead of LSTM, GRU [25] could be used as the cell of both $f_{encoder}$ and $f_{decoder}$. Unlike the fixed vector $\mathbf{c}$ in the traditional encoder-decoder approach, current context vector $\mathbf{c}_t$ varies with the step t,

$$\mathbf{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \mathbf{h}_{2,j} \tag{9}$$

and then we can get a new form of $\mathbf{y}_t$,

$$\mathbf{y}_t = f_{decoder}(\mathbf{c}_t, \mathbf{s}_{2,t-1}, \mathbf{s}_{1,t}) \tag{10}$$

where $\alpha_{t,j}$ is the weight term of jth location at step t in the input sequence. Note that the weight term $\alpha_{t,j}$ is normalized to $[0, 1]$ using a softmax function,

$$\alpha_{t,j} = \frac{\exp(\mathbf{e}_{t,j})}{\sum_{i=1}^{T} \exp(\mathbf{e}_{t,i})}, \tag{11}$$

where $\mathbf{e}_{t,j} = a(\mathbf{s}_{2,t-1}, \mathbf{h}_{2,j})$ scores how well the inputs around position j and the output at position t match, which is a learnable parameter of the model.

4 Data preprocessing

In order to evaluate the proposed method effectively, we first build a large dataset. We collected data from GitHub, a web-based Git repository hosting service. We crawled over 1,600 open source projects from GitHub, and got 1,006,584 Java code snippets. After data cleaning, we finally got 879,994 Java code snippets and the same number of comment segments. Although these comments are written by different developers with different styles, there exist common characteristics under these styles. For example, the same code could have totally different comments but they all explain the same meaning. In natural language, same source sentence may have more than one reference translations, which is similar to our setups. We name our dataset as C2CGit.

To the best of our knowledge, there does not exist such a large public dataset with paired code snippets and comments. One choice is using human annotation [28]. By this way, the comments could have high accuracy and reliability. However, it needs many experienced programmers and consumes a lot of time if we want to get big data. Another choice is to use recent CODE-NN [6] which mainly collected data from Stack Overflow which contains some code snippets in answers. For the code snippet from accepted answer of one question, the title of this question is regarded as a comment. Compared with CODE-NN (C#), our C2CGit (Java) possesses two obvious advantages:

- Code snippets in C2CGit are more practical. In many real projects from C2CGit, several lines of comments often correspond to a much larger code snippet, for example, a 2-line comment is annotated above 50-line code. However, this seldom appears in Stack Overflow.
- C2CGit is much larger and more diversified than CODE-NN. We make a detailed comparison in Fig 4 and Table 1. We can see that C2CGit is about 20× larger than CODE-NN no matter in statements, loops or conditionals. Also, C2CGit holds more tokens and words which demonstrate its diversity.

Extraction. We downloaded projects from the GitHub website by using web crawler. Then, the Java files can be easily extracted from these projects. Source code and comments should be split into segments. If we use the whole code from a Java file as the input and the whole comments as the output, we would get many long sentences and it is hard to handle them even in NLP tasks. Through analyzing the abstract syntax tree (AST) [29] of code, we got code snippets from the complete Java file. By utilizing the method raised by [16], the comment extraction is much eas-

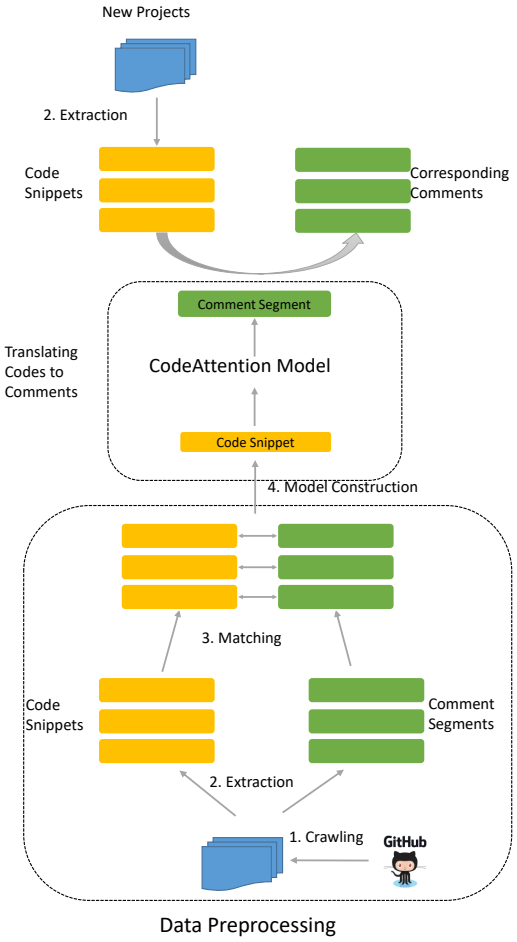

Figure 3: The whole framework of our proposed method. The main skeleton includes two parts: building dataset (named C2CGit) and code to comments translation.

Table 1: Average code and comments together with vocabulary sizes for C2CGit, compared with CODE-NN. The best one is marked as boldface.

	Avg. code length	Avg. title length	tokens	words
CODE-NN	38 tokens	12 words	91k	25k
C2CGit	**128 tokens**	**22 words**	**129,340k**	**22,299k**

ier, as it only needs to detect different comment styles in Java.

Matching. Through the above extraction process, one project would generate many code snippets and comment segments. The next step is to find a match between code snippets and comment segments. We extracted all identifiers other than keyword nodes from the AST of code snippets. Besides, the Java code prefers the camel case convention (e.g., StringBuilder can be divided into two terms, String and Builder). Each term from code snippets is then broken down based on the camel case convention. Note that if a term uses underline to connect two words, it can also be broken down. After these operations, a code snippet is broken down into many terms. Since comments are natural

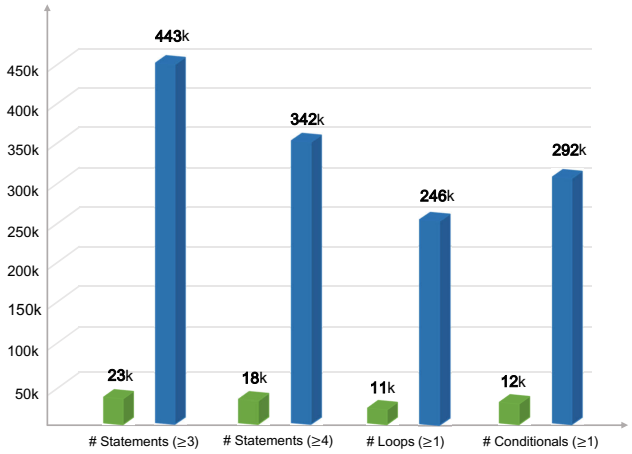

Figure 4: A comparison between C2CGit (blue) and CODE-NN (green). **#Statement(>=3)** means the number of code snippets containing more than 2 statements.

language, we use a tokenization tool, widely used in natural language processing, to handle the comment segments. If one code snippet shares the most terms with another comment segment, the comment segment can be regarded as a translation matching to this code snippet.

Cleaning. We use some prior knowledge to remove noise in the dataset. The noise is mainly from two aspects. One is that we have various natural languages, the other is the shared words between code snippets and comment segments are often too few. Programmers coming from all around the world can upload projects to GitHub, and their comments usually contain non-English words. These comments would make the task more difficult but only occupy a small portion. Therefore, we deleted instances containing non-ASCII characters if they appear in either code snippets or comment segments for reducing noises. Some code snippets only share one or two words with comment segments, which suggests the comment segment can't express the meaning of code. These code and comment pairs also should be deleted.

5 The CodeAttention Approach

In this section, we mainly talk about the CodeAttention. For the encoder-decoder structure, we employ a 3-layer translation model, whose basic element is Gated Recurrent Unit (GRU). On this basis, we propose a novel attention mechanism to leverage the code constructs. For convenience, we provide an overview of the entire model in Fig. 5.

Unlike traditional statistical language translation, code snippets have some different characteristics, such as critical statements, symbols and keywords. However, previous works usually ignore these differences and employ the common encoding methods in NLP. In order to emphasize these features of code constructs, we import two effective strategies: functional region alignment and token encoding, after which we develop a global attention module to learn their specific weights in code snippets. We will introduce details of functional region alignment, token encoding and global attention in the following.

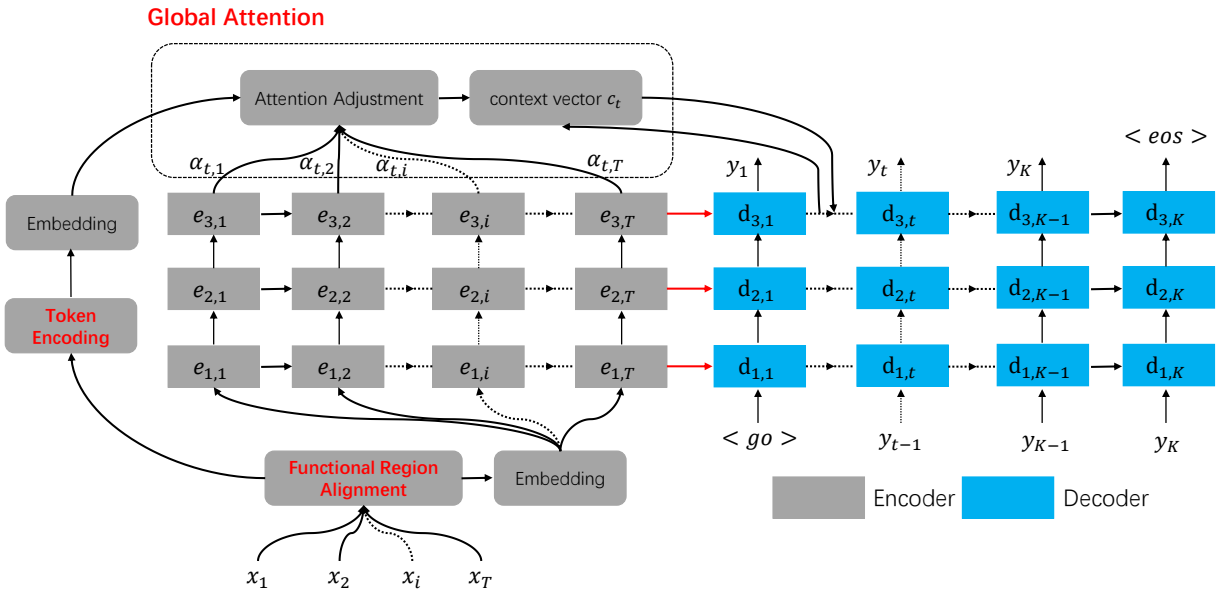

Figure 5: The framework of CodeAttention.

Functional Region Alignment. The loop or branch statement blocks are very important in the code constructs and may indicate the meaning of code. We call these blocks as functional regions. To distinguishing different functional regions, we directly sort *for* and *if* in code snippets based on the order they appear. The *while* and *switch* could be handled in the same way. After sorting,

$$for/if \longrightarrow for/if + N$$

where N is decided by the order of each *if* or *for* in its upper nest. For example, when we have multiple *if* and *for*, after functional region alignment, we have such forms as shown in Fig. 6. After that, each *if* or *for* would be projected into different embedding vector as the input of RNN.

```
1 FOR1(i=0; i<len - 1; i++)
2   FOR2(j=0; j<len - 1 - i; j++)
3     IF1(arr[j] > arr[j + 1])
4       temp = arr[j]
5       arr[j] = arr[j+1]
6       arr[j+1] = temp
7     ENDIF1
8   ENDFOR2
9 ENDFOR1
```

Figure 6: An example of collected code snippet after identifier sorting.

The above strategy is able to keep the original order of critical statements. It is worth noting that functional region alignment makes a difference among *fors* or *ifs* appeared in different loop levels.

Token Encoding. Tokens often contain symbols, keywords and identifiers in code snippets. In order to stress their specificity, these tokens should be encoded in a way which helps to make them more conspicuous than naive encoded code snippets. To be specific, we first build a dictionary including all symbols, like $\times$, $\div$, ;, $\{$, $\}$ and keywords, such as $int, float, public, ...$ in code snippets. The tokens are not contained in this dictionary are regarded as identifiers, such as method names, class names, constructor calls etc. Furthermore, immediate values in code snippets would be mapped into all ones vector. Next, we construct an independent token vocabulary which has the same size as the vocabulary of all input snippets, and encode these tokens using an extra embedding matrix. The embedded tokens can be treated as learnable weights in global attention.

Global Attention. In order to emphasize the importance of tokens in code, we propose a novel attention mechanism called global attention. We represent $\mathbf{x}$ as a set of inputs. Let $Ord(\cdot)$ and $Enc(\cdot)$ stand for our *functional region alignment* and *token encoding*, respectively. $E(\cdot)$ is used to represent the embedding method. The whole global attention operation can be summarized as,

$$E(Enc\,(Ord\,(\mathbf{x}))) \otimes f_e(\mathbf{x}) \tag{12}$$

where $f_e(\cdot)$ is the encoder, $\otimes$ represents dot product to stress the effects of encoded tokens.

After token encoding, we now have another token embedding matrix: $\mathbf{F}$ for symbols, keywords and variables. We set $\mathbf{m}$ as a set of one-hot vectors $\{\mathbf{m}_1, ..., \mathbf{m}_T\} \in \{0,1\}^{|F|}$ for different tokens. We represent the results of $E(Enc\,(Ord\,(\mathbf{x})))$ as a set of vectors $\{\mathbf{w}_1, ..., \mathbf{w}_T\}$, which can be regarded as a learnable parameter for each token,

$$\mathbf{w}_i = \mathbf{m}_i\mathbf{F} \tag{13}$$

Since the context vector $\mathbf{c}_t$ varies over time, the formulation of context vector c_t is as follows,

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{t,i}(\mathbf{w}_i \otimes \mathbf{e}_{3,i}) \tag{14}$$

where $\mathbf{e}_{3,i}$ is the hidden state located at the 3rd layer and ith position ($i = 1, \ldots, T$) in the encoder, T is the input size.

$\alpha_{t,i}$ is the weight term of ith location at step t in the input sequence, which is used to tackle the situation when input piece is overlength. Then we can get a new form of $\mathbf{y}_t$,

$$\mathbf{y}_t = f_d(\mathbf{c}_t, \mathbf{d}_{3,t-1}, \mathbf{d}_{2,t}, \mathbf{y}_{t-1}) \tag{15}$$

$f_d(\cdot)$ is the decoder function. $\mathbf{d}_{3,t}$ is the hidden state located at the 3rd layer and tth step ($t = 1, \ldots, K$) in the decoder. Here, we assume that the length of output is K. Instead of LSTM in [30], we take GRU [25] as basic unit in both $f_e(\cdot)$ and $f_d(\cdot)$. Note that the weight term $\alpha_{t,i}$ is normalized to $[0, 1]$ using a softmax function,

$$\alpha_{t,i} = \frac{\exp(\mathbf{s}_{t,i})}{\sum_{i=1}^{T} \exp(\mathbf{s}_{t,i})}, \tag{16}$$

where $\mathbf{s}_{t,i} = score(\mathbf{d}_{3,t-1}, \mathbf{e}_{3,i})$ scores how well the inputs around position i and the output at position t match. As in [31], we parametrize the score function $score(\cdot)$ as a feed-forward neural network which is jointly trained with all the other components of the proposed architecture.

6 Experiments

We compare the CodeAttention with several baseline methods on C2CGit dataset. The metrics contain both objective and subjective evaluation.

Baseline methods

To evaluate the effectiveness of CodeAttention, we compare it with four popular approaches from natural language and code translation.

- **CloCom** [16]: A state-of-the-art code comment generation method which leverages code clone detection to match code snippets with comment segments. This "matching" nature makes this method incapable of generating comments for the code snippets that are not similar to any code snippets in the code base.
- **MOSES** [32]: A widely-used phase-based method in traditional statistical machine translation. It is usually used as a competitive baseline method in sequence to sequence translation models. In our experiments, MOSES is equipped with a 4-gram language model using KenLM [33].
- **LSTM-NN** [6]: This is a method leveraging RNN to generate texts from source code. The parameters of LSTM-NN are selected according to the suggestions in [6].
- **GRU-NN** [34]: A 3-layer RNN model with GRU cells. For better comparison, we improve the RNN structure in [35] to make it deeper and use GRU [25] units instead of LSTM proposed in the original paper, both of which help it become a strong baseline approach.
- **Self-Attention** [30]: A recently proposed state-of-the-art method based solely on the attention mechanisms, dispensing with recurrence and convolutions.

Experimental Settings

We employ a 3-layer encoder-decoder architecture with a CodeAttention mechanism to model the joint conditional probability of the input and output sequences.

The initial value of learning rate is 0.5. When step loss doesn't decrease after 3k iterations, the learning rate multiplies decay coefficient which is usually 0.99. Reducing the learning rate during the training helps avoid missing the lowest point. Meanwhile while large initial value can speed up the learning process.

We use buckets to deal with code snippets with various lengths. To get a good efficiency, we put every code snippet and its comment to a specific bucket, e.g., for a bucket sized $(40, 15)$, the code snippet in it should be at most 40 words in length and its comment should be at most 15 words in length. In our experiments, we found that bucket size has a great effect on the final result, and we employed a 10-fold cross-validation method to choose a good bucket size. After cross-validation, we choose the following buckets, $(40, 15)$, $(55, 20)$, $(70, 40)$, $(220, 60)$.

We use stochastic gradient descent to optimize the network. In this network, the embedding size is 512 and the hidden unit size is 1024. Also, we have tried different sets of parameters. For example, a 3-layer RNN is better than 2-layer and 4-layer RNNs, the the 2-layer model has low scores while the 4-layer model's score is only slightly higher than that of the 3-layer one but with much longer running time. Finally, it takes three days and about 90k iterations to finish the training stage of our model on one NVIDIA K80 GPU. Note that we employ beam search in the inference.

All experiments are performed on C2CGit dataset. The whole dataset was divided into 90% training set and 10% test set. All methods run 10-fold cross-validation to tune the parameters.

Objective Evaluation

We use BLEU [36] and METEOR [37] as objective evaluation metrics, which is popular in evaluating the quality of generated comments.

BLEU measures the average n-gram precision on a set of reference sentences. Most sequence to sequence algorithms are evaluated by BLEU scores, which is a popular evaluation metric. METEOR is recall-oriented and measures how well the model captures content from the references in the output. The higher the METEOR score is, the better it is. Denkowski et al. [38] argued that METEOR can be applied in any target language, and the translation of code snippets could be regarded as a kind of minority language. We report all factors affecting the METEOR score, e.g., precision, recall, f1, fMean and final score. Precision is the proportion of the matched n-grams out of the total number of n-grams in the evaluated translation; Recall is the proportion of the matched n-grams out of the total number of n-grams in the reference translation; fMean is a weighted combination of Precision and Recall; Final Score is the fMean with penalty on short matches.

The average results for running all methods by 5 times are shown in Table 2 and Table 3.

Since BLEU is calculated on *n*-grams, we report the BLEU scores when *n* takes different values. From Table 2, we can see that the BLEU scores of our approach are relatively high when compared with previous algorithms, which suggests that CodeAttention is suitable for translating source code into comment. Equipped with our CodeAttention module, RNN achieves the best results on BLEU-1

Table 2: BLEU of all methods. The best one is marked as boldface.

Methods	BLEU-1	BLEU-2	BLEU-3	BLEU-4
CloCom	25.31	18.67	16.06	14.13
MOSES	45.20	21.70	13.78	9.54
LSTM-NN	50.26	25.34	17.85	13.48
GRU-NN	58.69	30.93	21.42	16.72
Attention	25.00	5.58	2.4	1.67
CodeAttention	**61.19**	**36.51**	**28.20**	**24.62**

Table 3: METEOR of different comments generation methods. The best one is marked as boldface.

Methods	Precision	Recall	fMean	Final Score
CloCom	0.4068	0.2910	0.3571	0.1896
MOSES	0.3446	**0.3532**	0.3476	0.1618
LSTM-NN	0.4592	0.2090	0.3236	0.1532
GRU-NN	0.5393	0.2397	0.3751	0.1785
Self-Attention	0.1369	0.0986	0.1205	0.0513
CodeAttention	**0.5626**	0.2808	**0.4164**	**0.2051**

to BLEU-4 and outperform the original GRU-NN by a large margin, e.g., about 50% on BLEU-4.

Table 3 shows the METEOR scores of each comment generation methods. The results are similar to those in Table 2. Our approach already outperforms other methods, and it significantly improves the performance when compared with GRU-NN in all evaluation metrics. Our approach exceeds GRU-NN by 0.027 (over 15%) in METEOR Score. It suggests that the CodeAttention module has an effect in both BLEU and METEOR scores. In METEOR scores, MOSES gets the highest recall compared with other methods, since it always generates long sentences and the words in references would have a high probability to appear in the generated comments. In terms of METEOR scores, CloCom beats MOSES and LSTM-NN, which is different from Table 2. The average length of comments generated by CloCom is very short but high quality, since CloCom only generate comments from few certain code templates. The METEOR scores stress quality, and shorter sentences get less plenty. Therefore, CloCom gets a higher score.

Unexpectedly, Self-Attention achieves the worst performance among different methods in both BLEU and METEOR, which implies that Self-Attention might not have the ability to capture specific features of code snippets. We argue that the typical structure of RNN can be necessary to capture the long-term dependency in code which are not fully reflected in the position encoding method from Self-Attention [30].

For a better demonstration of the effect of CodeAttention, we make a naive ablation study about it. The study reports the BLEU-4 results of different combinations. From Table 4, we can get two interesting observations. First, comparing line 1 with line 2, we can see that even single *functional region alignment* has some effects. RNN with *functional region alignment* exceeds normal GRU-NN by 1.63, which reflects that stressing the order of different critical statements could be useful. The improvement can be reached when applying it on other methods based on neural machine translation. Second, $\otimes$ (line 4) exceed $\oplus$ (line 4) by 2.24, which precisely follows our intuition, that $\otimes$ amplifies the effects of tokens more efficiently than $\oplus$ does.

All above results indicate that the performance can be

Table 4: Ablation study about effects of different parts in CodeAttention.

	BLEU-4	Ident	Token	Global Attention
1	16.72	w/o	w/o	w/o
2	18.35	w/	w/o	w/o
3	22.38	w/	w/	$\oplus$
4	24.62	w/	w/	$\otimes$

Table 5: Criterion of understandability.

Level	Semantics
5	Fluent, and grammatically correct
4	Not fluent, and grammatically correct
3	Grammatically incorrect, but easy to understand
2	Grammatically incorrect, and hard to understand
1	Meaningless

improved by leveraging the structural information of code.

Subjective Evaluation

Objective evaluation do not always agree with actual quality of the results [39], and the generated comments must be understood by programmers. Hence, we perform subjective evaluation. We employ 5 programmers with 5+ years Java experience to finish this task. Each programmer rates the comments independently for eliminating prejudice. The criteria would be shown in the following:

- Understandability. We consider the fluency and grammar of generated comments. The programmers would score these comments according to the understandability criterion shown by Table 5. If programmers catch the meaning of code snippets in a short time, the scores of understandability would be high.
- Similarity. We should compare the generated comments with human written ones, which suggests what the models learn from the training set. The similarity between generated comments and human written is shown in Table 6. This criterion measures the similarity between generated comments and human written.
- Interpertability. The connection between code and generated comments also should be considered, which is referred to as interpertability. The detailed criterion is shown in Table 7, which means the generated comments convey the meaning of code snippets.

We randomly choose 220 pairs of code snippets and comment segments from the test set, and let programmers rate them according to above three evaluations. The generated comments from different methods would be shuffled before rating. The comments generated by Self-Attention were incomplete and most of them are meaningless, therefore, it's no necessary to let programmers rate them.

Table 8 shows the subjective evaluation of all auto-generated comments methods from three aspects. The three aspects are understandability, similarity and interpretability. Our method gets the best performance in all aspects. We can tell that our proposed method has an improvement than other methods in subjective evaluation. For details, we show the each subjective evaluation scores in the following.

In terms of understandability, we are able to draw several conclusions from Fig. 7. Firstly, our method, with maxi-

Table 6: Criterion of similarity between generated comments and human writing.

Level	Semantics
5	Generated comments are easier to understand than the human writing
4	The meanings of generated and human writing comments are same, so as the expression
3	The meanings of generated and human writing comments are same, but the expressions are different
2	The meanings of generated and human writing comments are different, but the generated comments express some information of code
1	The generated comments are meaningless.

Table 7: Criterion of interpertability.

Level	Semantics
4	The generated comments show the high level meaning in code snippets
3	The generated comments only show partial meaning in code snippets.
2	The generated comments only shows some keywords in code snippets
1	There doesn't exist connection between code snippets and generated comments.

mum ratios of *good* comments (4 and 5 points), achieves the best results over other four approaches. Secondly, LSTM-NN and GRU-NN obtain the most comments in the "middle zones" (3 and 4 points). It suggests that sequence to sequence model can generate readable comments. However, they still can not generate high quality comments without leveraging code constructs. The last phenomenon that draws much attention is that ColCom has the worst performance in general, although it has more 5 points than GRU-NN and LSTM-NN. The reason might be that the ColCom chooses the comments of similar code snippets as generated comments and these comments often have high quality. However, when facing many other code snippets, ColCom cannot generate enough appropriate comments.

Regarding with similarity, the results in Fig. 8 are nearly the same as those in Fig. 7. We can claim that the ColCom has the least similar comments with ground-truth ones, which suggests that two code snippets might share many common words (ColCom usually chooses the comments of similar code snippets) but the meaning of each could be different from the other.

In terms of the interpretability, we can see that our method performs much better than other ones. The methods based on RNN architecture, e.g. LSTM-NN, GRU-NN and our method, are much better than other methods. It's suggested that RNN architecture could capture both deep semantics and literal meaning in code snippets.

In summary, the experimental results from subjective evaluation indicate that leveraging the code constructs can generate the more readable comments, and it can help the programmers understand code snippets easily.

Table 8: The scores of all methods in subjective evaluations. The best one is marked as boldface.

Methods	Understandability	Similarity	Interpretability
CloCom	2.55	2.00	1.77
MOSES	3.08	2.84	2.60
LSTM-NN	3.70	2.96	2.39
GRU-NN	3.60	3.27	2.76
CodeAttention	**4.08**	**3.36**	**2.98**

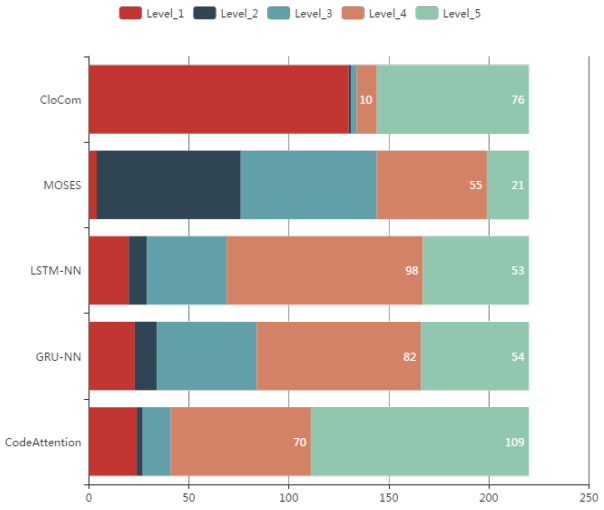

Figure 7: Understandability distribution of all methods

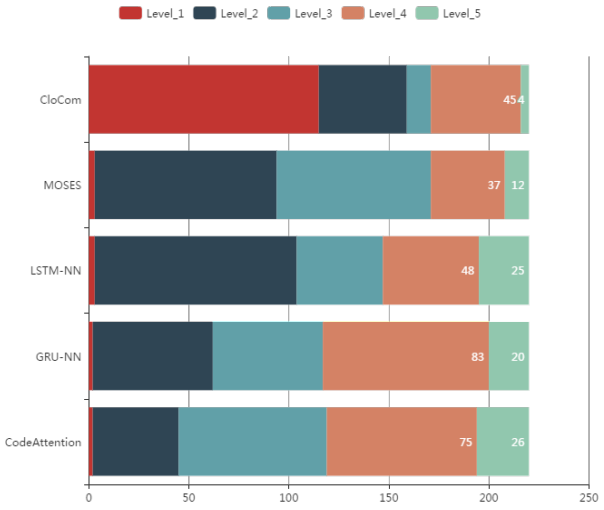

Figure 8: Similarity distribution of all methods

Case Study

Table 9 shows examples from the outputs generated by our models and other methods in the test set. In fact, not all methods can generate meaningful sentences, suggesting the task is difficult and traditional methods can hardly achieve this goal. For these two examples, the comments translated by neural networks are shorter and more meaningful than others. Our CodeAttention can generate higher quality comments than other neural models. The generated comments by CodeAttention are almost the same as ground truth. It suggests that our proposed method can make the translation better by leveraging the constructs of code. MOSES generates longer comments than other meth-

Table 9: Two examples of code comments generated by different methods.

code	1 private void createResolutionEditor(Composite control, 2 IUpdatableControl updatable) { 3 screenSizeGroup = new Group(control, SWT.NONE); 4 screenSizeGroup.setText("Screen Size"); 5 screenSizeGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
GroundTruth	the property key for horizontal screen size
ColCom	None
Moses	create a new resolution control param control the control segment the segment size group specified screen size group for the current screen size the size of the data is available
LSTM-NN	creates a new instance of a size
GRU-NN	the default button for the control
Attention	param the viewer to select the tree param the total number of elements to select
CodeAttention	create the control with the given size
code	1 while (it.hasNext()) { 2 EnsembleLibraryModel currentModel (EnsembleLibraryModel) it.next(); 3 m_ListModelsPanel.addModel(currentModel); 4 }
GroundTruth	gets the model list file that holds the list of models in the ensemble library
ColCom	the library of models from which we can select our ensemble usually loaded from a model list file mlf or model xml using the l command line option
Moses	adds a library model from the ensemble library that the list of models in the model
LSTM-NN	get the current model
GRU-NN	this is the list of models from the list in the gui
Attention	the predicted value as a number regression object for every class attribute
CodeAttention	gets the list file that holds the list of models in the ensemble library

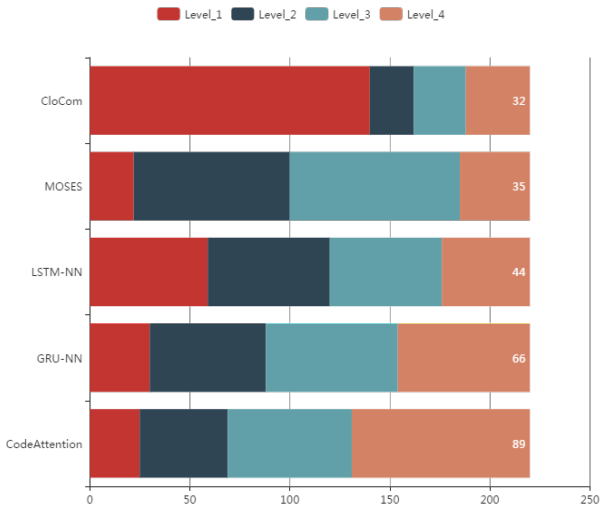

Figure 9: Interpretability distribution of all methods

ods, since it translates source code by using word alignment. A snippet of code would align many comment segments and MOSES could not find out the core meanings. LSTM-NN generates fluent sentences, which are shorter but information is less comparable with our method, which suggested that LSTM-NN can't capture the whole semantics of code snippets.

In Summary, case study indicates that CodeAttention generates more accurate and more easy-to-understand comments compared with other baseline methods by leveraging the code constructs.

7 Conclusion

In this paper, we propose an attention module named CodeAttention to utilize the code constructs, like statement blocks, symbols and identifiers. CodeAttention contains 3 steps: Functional Region Alignment, Token Encoding and Global Attention. Equipped with RNNs, CodeAttention outperforms competitive baselines and gets the best performance in both objective and subjective evaluations. Our results suggest that generated comments would conform to the functional semantics of program by explicitly modeling the code constructs.

In the future, we will extend the CodeAttention by further exploiting the AST tree, which is expected to provide richer information of code for better comments generation.

References

1. Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *WCRE*, pages 70–79. IEEE, 2007.
2. Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52, 2010.
3. Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *ICSE*, pages 505–514. ACM, 2010.
4. Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *ICPC*, pages 279–290. ACM, 2014.
5. Matúš Sulír and Jaroslav Porubän. Generating method documentation using concrete values from executions. In *OASIcs-OpenAccess Series in Informatics*, volume 56. Schloss

Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
6. Iyer Srinivasan, Konstas Ioannis, Cheung Alvin, and Zettlemoyer Luke. Summarizing source code using a neural attention model. In *ACL*, 2016.
7. Miltiadis Allamanis, Hao Peng, and Charles Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML*, 2016.
8. Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source codes. In *IJCAI*, pages 1606–1612. IEEE, 2016.
9. Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, pages 101–110, 2011.
10. Dana Movshovitz-Attias and William W. Cohen. Natural language models for predicting programming comments. In *ACL*, pages 35–40, 2013.
11. Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, pages 35–44. IEEE, 2010.
12. Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *ICPC*, pages 13–22. IEEE, 2013.
13. Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney DMello. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE*, pages 390–401. ACM, 2014.
14. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE 2013, pages 422–431, May 2013.
15. Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE*, pages 562–567, 2013.
16. Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *ICSA*, pages 380–389. IEEE, 2015.
17. Peter E Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.
18. Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *ACL*, pages 48–54. ACL, 2003.
19. Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
20. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2014.
21. Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.
22. Xinyou Yin, JAN Goudriaan, Egbert A Lantinga, JAN Vos, and Huub J Spiertz. A flexible sigmoid function of determinate growth. *Annals of botany*, 91(3):361–371, 2003.
23. Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. In *Neural Computation*, volume 9, pages 1735 – 1780, 1997.
24. Kyunghyun Cho, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoderâĂŞdecoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.
25. Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In *arXiv preprint arXiv:1409.1259*, 2014.
26. Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In *arXiv preprint arXiv:1412.3555*, 2014.
27. Kyunghyun Cho, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.
28. Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *ASE*, pages 574–584. IEEE, 2015.
29. Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
30. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *arXiv preprint arXiv:1706.03762*, 2017.
31. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
32. Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *ACL*, pages 177–180, 2007.
33. Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the 6th Workshop on Statistical Machine Translation*, pages 187–197. Association for Computational Linguistics, 2011.
34. Kyunghyun Cho, Bart Van Merriënboer, 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.
35. Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *NIPS*, pages 2773–2781, 2015.
36. Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318, 2002.
37. Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. *Proceedings of the Association for Computational Linguistics workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 29:65–72, 2005.
38. Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the 9th Workshop on Statistical Machine Translation*. Citeseer, 2014.
39. Amanda Stent, Matthew Marge, and Mohit Singhai. Evaluating evaluation methods for generation in the presence of variation. In *Proceedings of the 6th International Conference on Intelligent Text Processing and Computational Linguistics*, pages 341–351. Springer, 2005.

Wenhao Zheng received the BSc degree in Computer Science and Technology from Nanjing University, Nanjing, China, in 2015. In the same year, he was admitted to study for a MSc degree in Nanjing University without entrance examination. He is currently a member of the LAMDA Group. His research interests mainly include machine learning and software mining. Mr. Zheng received National Scholarship in 2014.

Hong-Yu Zhou received his B.E. degree from Wuhan University, China, in 2015. He is currently a postgraduate student in the Department of Computer Science and Technology, Nanjing University, China. His research interests include computer vision and machine learning.

Ming Li received the BSc and PhD degrees in computer science from Nanjing University, China, in 2003 and 2008 respectively. He is currently an assistant professor with LAMDA Group, the Department of Computer Sciences and Technology, Nanjing University. His major research interests include machine learning, data mining and information retrieval, especially on learning with labeled and unlabeled data. He has been granted various awards including the CCF Outstanding Doctoral Dissertation Award (2009), Microsoft Fellowship Award (2005), etc. He has served on the program committee of a number of important international conferences including KDD'10, ACML'10, ACML'09, ACM CKIM'09, IEEE ICME'10, AI'10, etc, and served as reviewers for a number of journals including IEEE Trans. KDE, IEEE Trans. NN, IEEE Trans. SMCC, ACM Trans. IST, Pattern Recognition, Knowledge and Information Systems, Journal of Computer Science and Technology, etc. He is a committee member of CAAI machine learning society, member of ACM, IEEE, IEEE computer society, CCF and CAAI.

Jianxin Wu received his BS and MS degrees in computer science from Nanjing University, and his PhD degree in computer science from the Georgia Institute of Technology. He is currently a professor in the Department of Computer Science and Technology at Nanjing University, China, and is associated with the National Key Laboratory for Novel Software Technology, China.