Code Attention: Translating Code to Comments by Exploiting Domain Features

Anonymous

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 the coding patterns change and hence the performance of comment generation declines. In this paper, we first build C2CGit, a large dataset from open projects in GitHub, which is more than $20\times$ larger than existing datasets. Then we propose a new attention module called Code Attention to translate code to comments, which is able to utilize the domain features of code snippets, such as symbols and identifiers. We make ablation studies to determine effects of different parts in Code Attention. Experimental results demonstrate that the proposed module has better performance over existing approaches in both BLEU and METEOR.

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, because it is time-consuming to create and update comments constantly, plenty of source code, especially the code from open-source software, lack adequate comments (Fluri, Wursch, and Gall 2007). 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. 2010) generated summary comments by using variable names in code. (Rastkar, Murphy, and Murray 2010) managed to give a summary by reading software bug reports. (McBurney and McMillan 2014) leveraged the documentation of API to generate comments of code snippets.

As is well known, source code are 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 the code and its corresponding textual descriptions. Since identifying such relationships from the raw data is rather challenging due to the heterogeneity nature of programming language and natural language, most of the 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 are 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 have large variance in performances 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 from source code in programming languages. (Srinivasan et al. 2016) and (Allamanis, Peng, and Sutton 2016) treated source code as natural language texts, and learned a neural network to summarize the words in source code into briefer phrases or sentences. However, as pointed out by (Huo, Li, and Zhou 2016), source code carry non-negligible semantics on the program functionality and should not be simply treated as natural language texts. Therefore, the comments generated by (Srinivasan et al. 2016) may not well capture the functionality semantics embedded in the program structure. For example, as shown in Figure 1, if only considering the lexical information in this code snippet, the comment would be "*swap two elements in the array*". However, if considering both the structure and the lexical information, the correct comment should be "*shift the first element in the array to the end*".

One question arises: Can we *directly learn a mapping* between two heterogeneous languages? Inspired by the recent advances in neural machine translation (NMT), we propose a novel attention mechanism called Code Attention to directly *translate* the source code in programming language into comments in natural language. Our approach is able to explore domain features in code, e.g. explicitly modeling the semantics embedded in program structures such as loops and

```
int i = 0;
while(i<n){
// swap is a build-in function in Java
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.

symbols, based on which the functional operations of source code are mapped into words. To verify the effectiveness of Code Attention, we build C2CGit, a large dataset collected from open source projects in Github. Empirical studies indicate that our proposed method can generate better comments than previous work, and the comments we generate would conform to the functional semantics in the program, by explicitly modeling the structure of the code.

The rest of this paper is organized as follows. After briefly introducing the related work, we describe the process of collecting and preprocessing data in Section 3, in which we build a new benchmark dataset called C2CGit. In Section 4, we introduce the Code Attention module, which is able to leverage the structure of the source code. In Section 5, we report the experimental results by comparing it with five popular approaches against different evaluation metrics. On BLEU and METEOR, our approach outperforms all other approaches and achieves new state-of-the-art performance in C2CGit.

Our contribution can be summarized as:

i) A new benchmark dataset for code to comments translation. C2CGit contains over 1k projects from GitHub, which makes it more real and 20× larger than previous dataset (Srinivasan et al. 2016).

ii) We explore the possibility of whether recent pure attention model (Vaswani et al. 2017) can be applied to this translation task. Experimental results show that the attention model is inferior to traditional RNN, which is the opposite to the performance in NLP tasks.

iii) To utilize domain features of code snippets, we propose a Code Attention module which contains three steps to exploit the structure in code. Combined with RNN, our approach achieves the best results on BLEU and METEOR over all other methods in different experiments.

Related Work

Previously, there already existed some work on producing code descriptions based on source code. These work mainly focused on how to extract key information from source code, through rule-based matching, information retrieval, or probabilistic methods. (Sridhara et al. 2010) generated conclusive comments of specific source code by using variable names in code. (Sridhara, Pollock, and Vijay-Shanker 2011) 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-Attias and Cohen 2013) 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 to generate summaries for source code based on automatic text summarization (Haiduc et al. 2010) or topic modeling (Eddy et al. 2013), possibly combining with the physical actions of expert engineers (Rodeghero et al. 2014).

Datasets. There are different datasets describing the relation between code and comments. Most of datasets are from Stack Overflow (Dyer et al. 2013; Wong, Yang, and Tan 2013; Srinivasan et al. 2016) and GitHub (Wong, Liu, and Tan 2015). 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 Q&A pattern. However, this assumption may not hold all the time because those questions are carefully designed. On the contrary, we argue that current datasets from GitHub are more real but small, for example, (Wong, Liu, and Tan 2015) only contains 359 comments. In this paper, our C2CGit is much larger and also has the ability to keep the accuracy.

Machine Translation. In most cases, generating comments from source code is similar to the sub-task named machine translation in natural language processing (NLP). There have been many research work about machine translation in this community. (Brown et al. 1993) 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 define a concept of word-by-word alignment between such pairs of sentences. (Koehn, Och, and Marcu 2003) 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 sub-components 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 (Hinton et al. 2012), visual object recognition (Krizhevsky, Sutskever, and Hinton 2014) and machine translation (Sutskever, Vinyals, and Le 2014). For example, the neural translator proposed in (Sutskever, Vinyals, and Le 2014) 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 translation.

Two most relevant works are (Srinivasan et al. 2016) and (Allamanis, Peng, and Sutton 2016). (Allamanis, Peng, and Sutton 2016) mainly focused on extreme summarization of source code snippets into short, descriptive function name-like summaries but our goal is to generate human-readable comments of code snippets. (Srinivasan et al. 2016) 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 structure of code.

Although translating source code to comments is similar to language translation, there does exist some differences. For instance, the structure of code snippets is 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 longer than the comment; 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 unit called Code Attention specially optimized for code structure to help make the translation process more specific.

C2CGit: A New Benchmark for Code to Comment Translation

We collected data from GitHub, a web-based Git repository hosting service.[1] 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 exactly same code could have totally different comments but they all explain the same meaning of the code. 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 for code and comment pairs. One choice is using human annotation (Oda et al. 2015). 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 (Srinivasan et al. 2016) which mainly collected data from Stack Overflow[2] 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) holds two obvious advantages:

- **Code snippets in C2CGit are more real**. 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 Figure 3 and Table 1. We can see that C2CGit is about 20$\times$ 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.[3] Then, the Java file can be

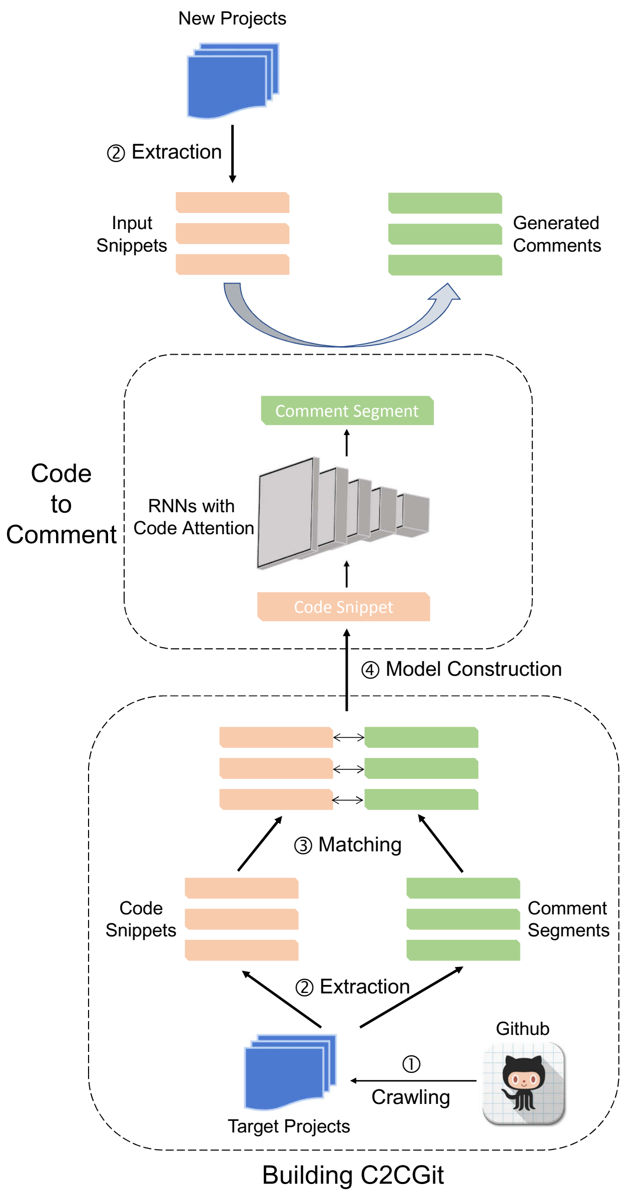

Figure 2: The whole framework of our proposed method. The main skeleton includes two parts: building C2CGit and code to comments translation.

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 both in statistical machine translation and neural machine translation. Through analyzing the abstract syntax tree (AST) (Neamtiu, Foster, and Hicks 2005) of code, we got code snippets from the complete Java file. By leveraging the method raised by (Wong, Liu, and Tan 2015), the comment extraction is much easier, since it only needs to detect different comment styles in Java.

[1]`https://github.com`

[2]`https://stackoverflow.com`

[3]The crawler uses the Scrapy framework. Its documentation can be found in `http://scrapy.org`

Table 1: Average code and comments together with vocabulary sizes for C2CGit, compared with CODE-NN.

	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**

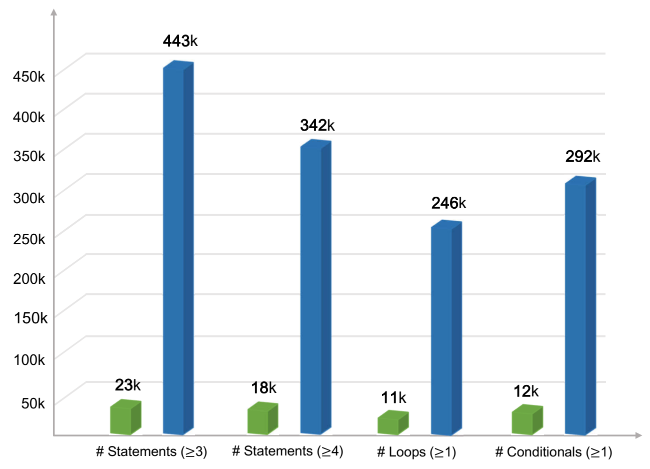

Figure 3: We make a comparison between C2CGit (blue) and CODE-NN (green). We can see that C2CGit is larger and more diversified.

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 prefer 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. Otherwise, if a term uses underline to connect two words, it can also be broken down. After these operations, a code snippet is broken down to many terms. Because comments are natural language, we use a tokenization tool[4], 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 from two aspects. One is that we have various natural languages, the other is that the shared words between code snippets and comment segments are 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-English words (non-ASCII characters) if they appear in either code snippets or comment segments. Some code snippets only share one word or two with comment segments, which suggests the comment segment can't express the meaning of code. These code and comment pairs also should be deleted.

[4]http://www.nltk.org/

Code Attention Mechanism

In this section, we mainly talk about the Code Attention mechanism in the model. For the encoder-decoder structure, we first build a 3-layer translation model, whose basic element is Gated Recurrent Unit (GRU). Then, we modify the classical attention module in encoder. To be specific, we consider the embedding of symbols in code snippets as learnable prior weights to evaluate the importance of different parts of input sequences. For convenience, we provide an overview of the entire model in Figure 4.

Unlike traditional statistical language translation, code snippets have some different characteristics, such as some identifiers (*for* and *if*) and different symbols (e.g., $\times, \div, =$). However, former works usually ignore these differences and employ the common encoding methods in NLP. In order to underline these identifiers and symbols, we simply import two strategies: Identifier Sorting and Symbol Encoding, after which we then develop a Global Attention module to learn their weights in input code snippets. We will first introduce details of Identifier Sorting and Symbol Encoding in the following.

Identifier Sorting. As the name suggests, we directly sort *for* and *if* in code snippets based on the order they appear. After sorting,

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

where N is decided by the order of each identifier in its upper nest. For example, when we have multiple *if* and *for*, after identifier sorting, we have such forms,

```
FOR1(i=0; i<len - 1; i++)
    FOR2(j=0; j<len - 1 - i; j++)
        IF1(arr[j] > arr[j + 1])
            temp = arr[j]
            arr[j] = arr[j+1]
            arr[j+1] = temp
        ENDIF1
    ENDFOR2
ENDFOR1
```

Figure 5: An example of collected code snippet after identifier sorting.

We can see that replaced identifiers are able to convey the original order of each of them. It is worth noting that Identifier Sorting can be regarded as a preprocessing method before input embedding.

Symbol Encoding. In order to stress the importance of these symbols, in this module, we decide to encode these signals in a way which helps make them more conspicuous than naive encoded inputs. To be specific, we first build a vocabulary including only two tokens. One token represents all symbols in code snippets, like $\times$, $\div$, ;, \{, \}, while the

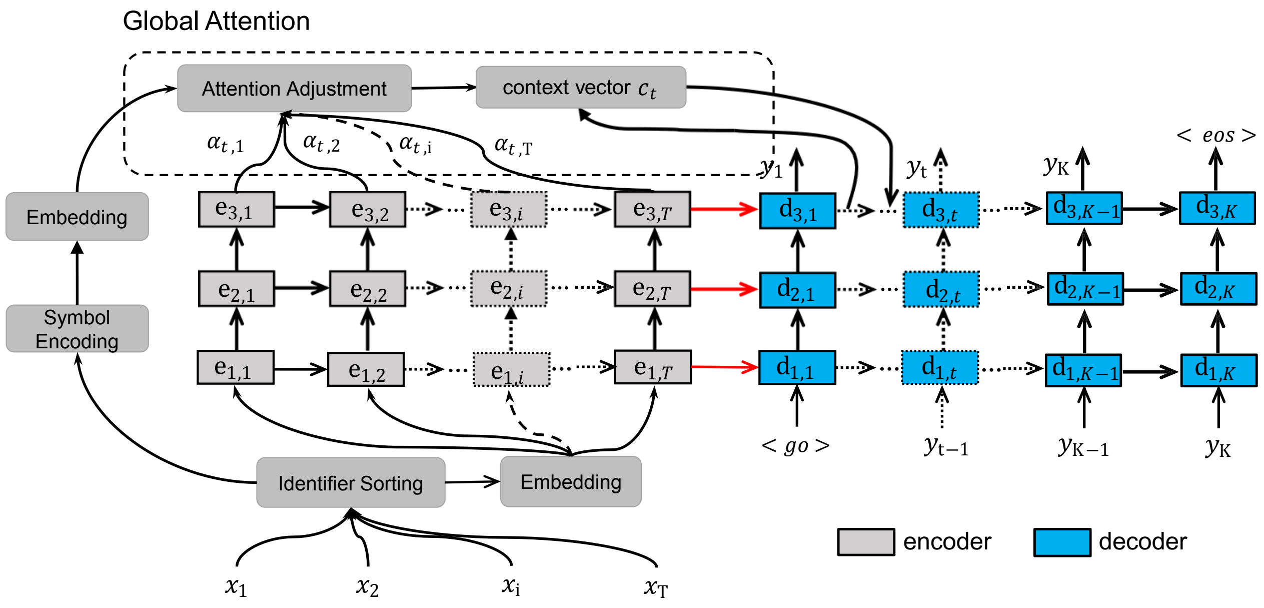
Global Attention
Attention Adjustment
context vector c_t
$\alpha_{t,1}$
$\alpha_{t,2}$
$\alpha_{t,i}$
$\alpha_{t,T}$
Embedding
Symbol Encoding
Identifier Sorting
Embedding
$e_{3,1}$ $e_{3,2}$ $e_{3,i}$ $e_{3,T}$
$e_{2,1}$ $e_{2,2}$ $e_{2,i}$ $e_{2,T}$
$e_{1,1}$ $e_{1,2}$ $e_{1,i}$ $e_{1,T}$
$d_{3,1}$ $d_{3,t}$ $d_{3,K-1}$ $d_{3,K}$
$d_{2,1}$ $d_{2,t}$ $d_{2,K-1}$ $d_{2,K}$
$d_{1,1}$ $d_{1,t}$ $d_{1,K-1}$ $d_{1,K}$
y_1 y_t y_K $< eos >$
$< go >$ y_{t-1} y_{K-1} y_K
x_1 x_2 x_i x_T
encoder
decoder

Figure 4: The whole model architecture. Note that Code Attention mainly contains 3 steps: Identifier Sorting, Symbol Encoding and Global Attention. The first two module are followed by two independent embedding layers as shown in the flow diagram above.

other one is mapped to all variables and keywords. Next, we encode these two tokens using an embedding matrix. The embedded symbols can be treated as learnable weights in Global Attention.

Global Attention

In order to underline the importance of symbols in code, we import a simple attention mechanism called Global Attention. We represent $\mathbf{x}$ as a set of inputs. Let $Ident(\cdot)$ and $Sym(\cdot)$ stand for our Identifier sorting and Symbol Encoding, respectively. We use $E(\cdot)$ to represent the embedding method. The whole Global Attention operation can be summarized as,

$$E(Sym\left(Ident\left(\mathbf{x}\right)\right)) \otimes f_e(\mathbf{x}) \tag{1}$$

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

After Symbol Encoding, we now have another token embedding matrix: $\mathbf{F}$ for symbols. We set m as a set of 1-hot vectors $\mathbf{m}_1, ..., \mathbf{m}_T \in \{0,1\}^{|F|}$ for each source code token. We represent the results of $E(Sym\left(Ident\left(CS\right)\right))$ as a set of vectors $\{\mathbf{w}_1, ..., \mathbf{w}_T\}$, which can be regarded as a learnable parameter for each token,

$$\mathbf{w}_i = \mathbf{m}_i\mathbf{F} \tag{2}$$

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

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

where $\mathbf{e}_{3,i}$ is the hidden state located at the 3rd layer and ith position ($i = 1, \dots, T$) 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 $\mathbf{y}_t$,

$$\mathbf{y}_t = f_d(\mathbf{c}_t, \mathbf{d}_{3,t-1}, \mathbf{d}_{2,t}, \mathbf{y}_{t-1}) \tag{4}$$

$f_d(\cdot)$ is the decoder function. $\mathbf{d}_{3,t}$ is the hidden state located at the 3rd layer and tth step ($t = 1, \dots, K$) in the decoder. Here, we assume that the length of output is K. Instead of LSTM in (Vaswani et al. 2017), we take GRU (Cho et al. 2014a) 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,

$$\alpha_{t,i} = \frac{\exp(\mathbf{s}_{t,i})}{\sum_{i=1}^{T}\exp(\mathbf{s}_{t,i})}, \tag{5}$$

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 (Bahdanau, Cho, and Bengio 2014), 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.

Ablation Study

For a better demonstration of the effect of Code Attention, we make a naive ablation study about it. For Table 2, we can get two interesting observations. First, comparing line 1 with line 2, we can see that even single *Identifier Sorting* have some effects. RNN with Identifier Sorting surpasses normal GRU-NN by 1.63, which reflects that stressing the order of different identifiers can be useful. Second, $\otimes$ (line 3) surpasses $\oplus$ (line 3) by 2.24, which precisely follows our intuition, that $\otimes$ amplifies the effects of symbols more efficiently than $\oplus$ does.

Table 2: Ablation study about effects of different parts in Code Attention. This table reports the BLEU-4 results of different combinations.

	BLEU-4	Ident	Sym	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/	$\otimes$

Experiments

Baselines

To evaluate the effectiveness of Code Attention, we compare different popular approaches from natural language and code translation, including CloCom, MOSES, LSTM-NN (Srinivasan et al. 2016), GRU-NN and Attention Model (Vaswani et al. 2017). All experiments are performed on C2CGit. It is worth noting that, for a better comparison, we improve the RNN structure in (Vinyals et al. 2015) to make it deeper and use GRU (Cho et al. 2014a) units instead of LSTM proposed in the original paper, both of which help it become a strong baseline approach.

- **CloCom:** This method raised by (Wong, Liu, and Tan 2015) leverages code clone detection to match code snippets with comment segments, which can't generate comment segments from any new code snippets. The code snippets must have similar ones in the database, then it can be annotated by existing comment segments. Hence, most code segments would fail to generate comments. CloCom also can be regarded as an information retrieval baseline.
- **MOSES:** This phase-based method (Koehn et al. 2007) is popular in traditional statistical machine translation. It is usually used as a competitive baseline method in machine translation. We train a 4-gram language model using KenLM (Heafield 2011) to use MOSES.
- **LSTM-NN:** This method raised by (Srinivasan et al. 2016) uses RNN networks to generate texts from source code. The parameters of LSTM-NN are set up according to (Srinivasan et al. 2016).
- **GRU-NN:** GRU-NN is a 3-layer RNN structure with GRU cells (Cho et al. 2014b). Because this model has a contextual attention, it can be regarded as a strong baseline.
- **Attention Model:** (Vaswani et al. 2017) proposed a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. The simple model achieves state-of-the-art results on various benchmarks in natural language processing.

Evaluation Metrics

We use BLEU (Papineni et al. 2002) and METEOR (Banerjee and Lavie 2005) as our automatic evaluation index. BLEU measures the average n-gram precision on a set of reference sentences. Most machine translation algorithms are evaluated by BLEU scores, which is a popular evaluation index.

METEOR is recall-oriented and measures how well the model captures content from the references in the output. (Denkowski and Lavie 2014) 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. In Table 4, we report the factors impacting the METEOR score, e.g., precision, recall, f1, fMean and final score.

Table 3: BLEU of Each Auto-Generated Comments Methods

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
Ours	**61.19**	**36.51**	**28.20**	**24.62**

Table 4: METEOR of different comments generation models. **Precision**: the proportion of the matched n-grams out of the total number of n-grams in the evaluated translation; **Recall**: the proportion of the matched n-grams out of the total number of n-grams in the reference translation; **fMean**: a weighted combination of Precision and Recall; **Final Score**: Fmean with penalty on short matches.

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
Attention	0.1369	0.0986	0.1205	0.0513
Ours	**0.5626**	0.2808	**0.4164**	**0.2051**

Experimental Results Analysis

In Table 3, BLEU scores for each of the methods for translating code snippets into comment segments in C2CGit, and since BLEU is calculated on n-grams, we report the BLEU scores when n takes different values. From Table 3, we can see that the BLEU scores of our approach are relatively high when compared with previous algorithms, which suggests Code Attention is suitable for translating source code into comment. Equipped with our Code Attention module, RNN gets the best results on BLEU-1 to BLEU-4 and surpass the original GRU-NN by a large margin, e.g., about 50% on BLEU-4.

Table 4 shows the METEOR scores of each comments generation methods. The results are similar to those in Table 3. Our approach already outperforms other methods and it significantly improves the performance compared with GRU-NN in all evaluation indexes. Our approach surpasses GRU-NN by 0.027 (over 15%) in Final Score. It suggests that our Code Attention module has an effect in both BLEU and METEOR scores. In METEOR score, MOSES gets the highest recall compared with other methods, because it al-

Table 5: Two examples of code comments generated by different translation models.

code	`private void createResolutionEditor(Composite control,`
	`                    IUpdatableControl updatable) {`
	`    screenSizeGroup = new Group(control, SWT.NONE);`
	`    screenSizeGroup.setText("Screen Size");`
	`    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
Ours	create the control with the given size

code	`while (it.hasNext()) {`
	`    EnsembleLibraryModel currentModel = (EnsembleLibraryModel) it.next();`
	`    m_ListModelsPanel.addModel(currentModel); }`

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
Ours	gets the list file that holds the list of models in the ensemble library

ways generates long sentences and the words in references would have a high probability to appear in the generated comments. In addition, in METEOR, the Final Score of CloCom is higher than MOSES and LSTM-NN, which is different from Table 3 because CloCom can't generate comments for most code snippets, the length of comments generated by CloCom is very short. The final score of METEOR would consider penalty of length, so CloCom gets a higher score.

Unexpectedly, Attention model achieves the worst performance among different models in both BLEU and METEOR, which implies that Attention Model 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 Attention model (Vaswani et al. 2017).

We give examples of two different input code snippets and corresponding generated comments from different models in Table 5. We can see that comments from our method are not only the most readable ones but also the most similar to ground truth annotations (e.g., the 2nd example).

Implementation Details

For RNN architecture, as we have discussed above, we employed a 3-layer encoder-decoder architecture with a Code Attention module to model the joint conditional probability of the input and output sequences.

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

Choose the right buckets. 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, $(40, 15)$, $(55, 20)$, $(70, 40)$, $(220, 60)$. It takes three days and about 90,000 iterations to finish the training stage of our model on one NVIDIA K80 GPU.

Conclusion

In this paper, we propose a novel attention module named Code Attention to utilize the specific features of code snippets, like identifiers and symbols. Code Attention contains 3 steps: Identifier Sorting, Symbol Encoding and Global Attention. Equipped with RNN, our model outperforms competitive baselines and gets the best performance on automatic metrics, such as BLEU and METEOR. Our results suggest that the generated comments would conform to the functional semantics of the program, by explicitly modeling the structure of the code. In future work, we plan to try applying AST tree to Code Attention and explore the effectiveness of it in other programming language such as C# and C++.

References

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML*.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Banerjee, S., and Lavie, A. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. volume 29, 65–72.

Brown, P. E.; Pietra, S. A. D.; Pietra, V. J. D.; and Mercer, R. L. 1993. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics* 19(2):263–311.

Cho, K.; van Merrienboer, B.; Bahdanau, D.; and Bengio, Y. 2014a. On the properties of neural machine translation: Encoder-decoder approaches. In *arXiv preprint arXiv:1409.1259*.

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014b. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Denkowski, M., and Lavie, A. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the 9th Workshop on Statistical Machine Translation*. Citeseer.

Dyer, R.; Nguyen, H. A.; Rajan, H.; and Nguyen, T. N. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE 2013, 422–431.

Eddy, B. P.; Robinson, J. A.; Kraft, N. A.; and Carver, J. C. 2013. Evaluating source code summarization techniques: Replication and expansion. In *ICPC*, 13–22. IEEE.

Fluri, B.; Wursch, M.; and Gall, H. C. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *WCRE*, 70–79. IEEE.

Haiduc, S.; Aponte, J.; Moreno, L.; and Marcus, A. 2010. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, 35–44. IEEE.

Heafield, K. 2011. Kenlm: Faster and smaller language model queries. In *Proceedings of the 6th Workshop on Statistical Machine Translation*, 187–197. Association for Computational Linguistics.

Hinton, G.; Deng, L.; Yu, D.; E. Dahl, G.; Mohamed, A.-r.; Jaitly, N.; Senior, A.; Vanhoucke, V.; Nguyen, P.; N. Sainath, T.; and Kingsbury, B. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine* 29(6):82–97.

Huo, X.; Li, M.; and Zhou, Z.-H. 2016. Learning unified features from natural and programming languages for locating buggy source codes. In *IJCAI*, 1606–1612. IEEE.

Koehn, P.; Hoang, H.; Birch, A.; Callison-Burch, C.; Federico, M.; Bertoldi, N.; Cowan, B.; Shen, W.; Moran, C.; Zens, R.; et al. 2007. Moses: Open source toolkit for statistical machine translation. In *ACL*, 177–180.

Koehn, P.; Och, F. J.; and Marcu, D. 2003. Statistical phrase-based translation. In *ACL*, 48–54. ACL.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. 2014. ImageNet classification with deep convolutional neural networks. In *NIPS*.

McBurney, P. W., and McMillan, C. 2014. Automatic documentation generation via source code summarization of method context. In *ICPC*, 279–290. ACM.

Movshovitz-Attias, D., and Cohen, W. W. 2013. Natural language models for predicting programming comments. In *ACL*, 35–40.

Neamtiu, I.; Foster, J. S.; and Hicks, M. 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 30(4):1–5.

Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; and Nakamura, S. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *ASE*, 574–584. IEEE.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*, 311–318.

Rastkar, S.; Murphy, G. C.; and Murray, G. 2010. Summarizing software artifacts: a case study of bug reports. In *ICSE*, 505–514. ACM.

Rodeghero, P.; McMillan, C.; McBurney, P. W.; Bosch, N.; and D'Mello, S. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE*, 390–401. ACM.

Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; and Vijay-Shanker, K. 2010. Towards automatically generating summary comments for java methods. In *ASE*, 43–52.

Sridhara, G.; Pollock, L.; and Vijay-Shanker, K. 2011. Automatically detecting and describing high level actions within methods. In *ICSE*, 101–110.

Srinivasan, I.; Ioannis, K.; Alvin, C.; and Luke, Z. 2016. Summarizing source code using a neural attention model. In *ACL*.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; N. Gomez, A.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All You Need. In *arXiv preprint arXiv:1706.03762*.

Vinyals, O.; Kaiser, Ł.; Koo, T.; Petrov, S.; Sutskever, I.; and Hinton, G. 2015. Grammar as a foreign language. In *NIPS*, 2773–2781.

Wong, E.; Liu, T.; and Tan, L. 2015. Clocom: Mining existing source code for automatic comment generation. In *ICSA*, 380–389. IEEE.

Wong, E.; Yang, J.; and Tan, L. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE*, 562–567.