In Practice

Augmenting Java method comments generation with context information based on neural networks

Yu Zhou\textsuperscript{a,}\textsuperscript{*}, Xin Yan\textsuperscript{a}, Wenhua Yang\textsuperscript{a}, Taolue Chen\textsuperscript{b,c}, Zhiqiu Huang\textsuperscript{a}

\textsuperscript{a}College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
\textsuperscript{b}Department of Computer Science and Information Systems, Birkbeck, University of London, UK
\textsuperscript{c}State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

\textbf{A R T I C L E I N F O}

Article history:
Received 22 February 2019
Revised 20 July 2019
Accepted 23 July 2019
Available online 24 July 2019

Keywords:
Comment generation
Neural networks
Natural language processing

\textbf{A B S T R A C T}

Code comments are crucial to program comprehension. In this paper, we propose a novel approach ContextCC to automatically generate concise comments for Java methods based on neural networks, leveraging techniques of program analysis and natural language processing. Firstly, ContextCC employs program analysis techniques, especially abstract syntax tree parsing, to extract context information including methods and their dependency. Secondly, it filters code and comments out of the context information to build up a high-quality data set based on a set of pre-defined templates and rules. Finally, ContextCC trains a code comment generation model based on recurrent neural networks. Experiments are conducted on Java projects crawled from GitHub. We show empirically that the performance of ContextCC is superior to state-of-the-art baseline methods.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

Program comprehension represents an expensive and time-consuming task in software development and maintenance (Xia et al., 2018). Comments written in natural languages can greatly facilitate programmers to understand the meaning of a code snippet (Takang et al., 1996; Tenny, 1988; Forward and Lethbridge, 2002), since they provide useful insights into code functionalities and the intention underpinning the design choices. Unfortunately, comments in software artifacts are often incomplete, outdated, incorrect or otherwise missing, partially because commenting code is generally time-consuming and laborious, and thus programmers often ignore writing and updating the comments either intentionally or unconsciously. Automatic generation of code comments can be a valuable alternative to alleviate programmers’ burden in software maintenance and to enhance program comprehension.

An intuitive way to generate comments is via pre-defined templates and rules. Templates were designed to generate comments for the conditions under which exceptions may be thrown (Buse and Weimer, 2008). However, the work Buse and Weimer (2008) only focused on statements related to exception triggering conditions, which limits its generality and applicability. In Sridhara et al. (2010), selected statements from Java methods were extracted for which natural language phrases were then generated. Follow-up studies identified and described code segments that implement high-level actions within methods (Sridhara et al., 2011b), and generated comments for parameters to be part of the method comments (Sridhara et al., 2011a). Aside from the need to manually define templates and rules, another prerequisite of their approaches is that source code must contain meaningful and descriptive identifiers. Beyond the method level, Moreno et al. (2013) proposed an approach to generate comments for Java classes. Similarly, a set of templates was designed to conduct content selection and text generation. Although the performance is promising, the approach shares the same limitations of the template-based methods. Unavoidably, these limitations would jeopardize the applicability of these approaches for generating code comments.

There are also approaches which are based on information retrieval (IR) techniques to generate code comments. Generally, these approaches utilize tokens in documents to calculate the similarity between documents and queries. For example, Wong et al. (2013) proposed to mine the code and description mappings from Stack Overflow and then to harness them to generate description comments automatically for similar code segments matched in open-source projects. Allamanis et al. (2015b) created a probabilistic model over the code which was used to retrieve natural language snippets. Haiduc et al. (2010) examined a number of methods, e.g., term frequency-inverse document frequency

\textsuperscript{*} Corresponding author.
E-mail address: zhouyu@nuaa.edu.cn (Y. Zhou).

https://doi.org/10.1016/j.jss.2019.07.087
0164-1212/© 2019 Elsevier Inc. All rights reserved.
or even no information. Apparently, these comments should have been removed from the training set.

To address the above limitations, we propose ContextCC, an approach to automatically generate method comments based on neural networks. Different from the approaches mentioned above, it takes context information into account. In general, the context inform may be from Java method themselves or their dependency. First, instead of treating methods as independent entities, ContextCC takes the entire Java project as a unit to analyze so as to extract more complete contextual information. For example, as illustrated in Example 1(a), the type information of val and the literal information of M03, M13 and M23 are part of context information and can be extracted via the abstract syntax tree (AST) of the Java method. Second, with the extracted context information, we reconstruct the code to complement with the contextual information. For example, as illustrated in Example 1(b), the identifiers M03, M13 and M23 are replaced with their literals (i.e., 12, 13 and 14) respectively, and the type information of the identifier var is supplemented in the formula of FIELDDECLARATION [ float [ ] val ; ].

Comments have many types and can be used to communicate a variety of information. In our work, we focus on a particular type of comments which mainly describes the intention of the Java method. In literature, some other terms, such as summaries and summarization, are used interchangeably as “comments” here. Observe that the first sentence of a Javadoc comment usually expresses the meaning of the whole method whereby we take and extract it as the standard comment of the method. However, as mentioned before, in many cases the first sentence of Javadoc comments suffers from a low quality. To cope with this issue, we manually design a set of pre-defined templates and rules combined with Part-Of-Speech (POS) tagging technique to filter comments, so as to build a high-quality data set.

In summary, our proposed approach ContextCC augments neural network based comments generation with context information, aiming to automatically generate concise comments for Java methods. We have conducted experiments on a large-scale of Java projects crawled from Github and compared with existing baselines. The experimental results show that our approach is considerably more effective. Concretely, for code comment generation task for Java methods, our approach improves the results of BLEU-4 from 38.08% to 40.52% and that of METEOR scores from 26.83% to 28.51% on the benchmark data sets, respectively.

The main contributions of this paper can be summarized as follows.

- We highlight the importance of contextual information during the comments generation process, and propose a novel approach incorporating such information to automatically generate code comments based on neural networks. Our approach outperforms strong baseline work in terms of multiple metrics such as BLEU-4 and METEOR.
- A large-scale empirical study has been conducted to demonstrate the feasibility of our approach. The accompanying data set contains around 540,000 Java methods extracted from over 6700 projects from Github. Remarkably, our data set also includes the contextual information which could be reused and extended in other similar or related studies.

The remainder of this paper is structured as follows. Section 2 introduces the related work. Section 3 presents our proposed approach. Section 4 describes our experimental setup and provides experimental results. Section 5 discusses the threats to validity. Finally, Section 6 concludes our work.
2. Related work

There has been a large body of research on generating comments from source code. Generally, the related work can be classified into three categories: templates based, IR based, and neural networks based approaches.

A common and intuitive way to generate code comments is through pre-defined templates and rules (Abid et al., 2015; Moreno et al., 2013; Zhang et al., 2011; Kamimura and Murphy, 2013; Cortés-Coy et al., 2014; Buse and Weimer, 2008; 2010; Zhou et al., 2017; 2018). Hill et al. (2009) introduce Software Word Usage Model (SWUM) model which can translate Java method invocations into descriptive statements. Sridhara et al. (2010) use SWUM combined with a group of pre-defined templates and rules to generate comments for Java methods. Sridhara et al. (2011b) extend their prior work to automatically detect and describe the high level actions in Java methods. As the complementary of prior tasks, Sridhara et al. (2011a) introduce a new technique to generate comments for parameters and treat the descriptions as part of method summary. Beyond method level, Moreno et al. (2013) develop an approach that generates summaries for Java classes. They design a set of stereotypes and templates to make operations of content selection and text generation. Although the performance is convincing and promising, the two methods highly depend on high quality of names of identifiers and methods. Once they are named poorly, the methods may finally fail.

There exist some IR technique based approaches for code summarization. Such techniques utilize tokens in documents to calculate the similarity between documents and queries. Wong et al. (2013) propose a method to retrieve comments that mines <code, text > pairs from Stack Overflow, and then tries to match the code in question with an example in stack overflow. Allamanis et al. (2015b) create a probabilistic model over code, but use it in the opposite direction to also retrieve full natural language snippets. Haiduc et al. (2010) examine a number of methods for selecting which keywords to use in a summary, including the lead method, TF-IDF, or LSI.

With great promise in many natural language preprocessing tasks, deep learning based approaches are gaining more and more attention. Spontaneously, researchers attempt to utilize the advantages of neural networks, especially RNNs, to extract the features of codes and comments and then automatically create code comments. These deep learning based approaches could be applied to both Domain-Specific Languages (DSLs) such as SQL and General-Purpose Languages (GPLs) such as Java. Iyer et al. (2016) present CODE-NN, an end-to-end neural attention model using LSTM to generate summaries of C# and SQL statements by learning from noisy online programming websites. Yao et al. (2019) propose an effective framework based on reinforcement learning, which explicitly employs a deep learning based code annotation model to generate annotations that can be used for the retrieval task of SQL statements. Allamanis et al. (2015a) introduce an attentional neural network that employs convolution on the input tokens to convert source code snippets into short, descriptive function name-like summaries. Hu et al. (2018) propose an algorithm named SBT combined with Seq2Seq model to generate descriptive comments for Java methods. There exists other approaches for generating comments for source codes by leveraging deep neural networks. Liang and Zhu (2018) make use of a new recursive neural network called Code-RNN to convert source code into one vector and then introduce a new recurrent neural network, Code-GRU, to generate text descriptions for the code. However, what the most existing approaches ignore is the quality of data set itself. With only simple filter operations, the data set is used to train deep neural network models, which may result that the performance of final models are barely satisfactory, since deep neural networks are driven by data.

3. The ContextCC approach

In this section, we present our approach, ContextCC, which follows the process illustrated in Fig. 1. Our approach generates code comments for Java methods in the following steps: (1) data preparation; (2) context information extraction; (3) code filtering and reconstruction; (4) comments filtering by pre-defined templates combined with POS tagging technique; (5) comment generation model training.

3.1. Data preparation

To build up a high-quality data set, we crawl over 6700 Java projects from Github to extract their methods and the corresponding Javadoc comments. We employ AST parsing to analyze each Java project. Particularly, we resort to Eclipse JDT (Fuhrer et al., 2007) to conduct the AST analysis where MethodDeclaration nodes represent a Java method declaration. We traverse the generated ASTs of the Java files and locate all the MethodDeclaration nodes. We then extract Java methods and their corresponding Javadoc comments from the nodes of this type. Note that not all Java methods have associated comments, and only those with comments are considered. For each Java method and its comment, we record their mapping relation and then apply the filtering operations separately (cf. Sections 3.2.2–3.4) to prepare a data set for the training of the neural network model for code comment generation (cf. Section 3.5).

3.2. Context information extraction

As mentioned before, context information consists of two parts: the methods and their dependency. In this step, we still leverage the AST tree and utilize numerous APIs provided by Eclipse JDT.

3.2.1. Information from Java methods

We categorize the identifiers as: method names (M), literals (L), variable names (V), names of variable types (P), method invocation names (I) and names of inner method declarations (D). To achieve this, methods are first transformed to ASTs. For each AST, we traverse it and focus on specific types of its nodes to extract the name sets.

Formally, given an AST tree T of a method, we use depth-first search to traverse T. Specifically, we focus on MethodDeclaration nodes for M and D, NumberLiteral, StringLiteral and CharacterLiteral nodes for L, SimpleType, PrimitiveType, QualifiedName, QualifiedType nodes for P, method invocation related nodes (e.g., MethodInvocation, SuperMethodInvocation) for I and SimpleName nodes for V, which means we aim to build the set $S = \{ M, L, V, P, I, D \}$ of information from methods themselves. By invoking related APIs, the related AST nodes in T can be located precisely and the set S can be easily built up.

3.2.2. Dependency information

Java methods could be very difficult to interpret without dependency information. Concretely, dependency information mainly includes (1) field declaration information (F), (2) method declaration information for method invocations (C), and (3) qualified names information (Q). To obtain the dependency information effectively, we need to regard the project as a unit and then parse the whole project.

By employing the JDT toolkit, we can successfully find the AST nodes we are interested in and then extract the corresponding dependency information via the following three stages.

---

1 http://help.eclipse.org/.
• First, we concentrate on field related nodes (e.g., FieldAccess, SuperFieldAccess). However, it is incomplete to only take information from these nodes. We also recognize fields from the set of variable names V. By merging the two parts of information, we realize the extraction of field declaration information (F) which consists of the types, names and initializers of related fields.

• Second, we focus on method invocation related nodes (e.g., MethodInvocation, SuperMethodInvocation) to obtain C for each method invocation. Here, we extract the information of corresponding qualified classes (e.g., CCTMXTiledMap in Example 2) and parameter types (e.g., CGPoint in Example 2) as C.

• Third, we traverse QualifiedName nodes. Specifically, if a QualifiedName node refers to a literal (cf. Section 3.3), we build a map from the qualified name of the QualifiedName node to the homologous literal. Therefore, the qualified names information (Q) is extracted as a map set. Ultimately, with F, C, Q extracted, we build the set Y = {F, C, Q}.

During these steps, we only traverse the corresponding ASTs of Java methods and extract useful information without changing the structures of the ASTs. By combining the method information (S) and dependency information (Y), we obtain and store the context information for each method which provides the input for the next step.

3.3. Code filtering and reconstruction

3.3.1. Field Replacement

We divide literals in methods into the following three categories: number literals (e.g., int, float, double, Integer), string literals (e.g., String, CharSequence), and character literals (e.g., char, Character). For a method without its contextual information, it is virtually impossible to interpret field literals. As illustrated in Example 1(a), fields M03, M13 and M23 in fact represent 12, 13 and 14, respectively (with the int type in the specific Java class). However, the method per se does not inform the related field declaration for M03, M13 and M23. To address this issue, we apply field replacement operations, i.e., to recognize and replace fields with their initializers by leveraging the extracted field declaration information F (cf. Section 3.2). As shown in Example 1(b), M03, M12 and M23 are replaced by 12, 13 and 14 respectively. Considering that these fields have been replaced with certain kinds of literals, the replaced literals are taken as complement of literal information (L).

3.3.2. Field Declaration Supplement

Similar to the field replacement, field declaration supplement aims to complement the related field declaration information for methods. The difference is that we focus on the fields whose initializers do not refer to literals. As shown in Example 1(a), although the exact type of the variable val is unavailable by inspecting the source code of the method alone, one can analyze the class level files to get the related field declaration statements and then fill the missing type information (i.e., float []) val. To avoid introducing too much redundant information, we focus on the type information without supplementing the initializer of the field declaration. To distinguish related field declarations from variable declarations in the method, we define specific templates to represent related field declaration information (cf. the template of FieldAccess in Table 1). As shown in Example 1, the field declaration information of the variable, val, is supplemented in the form of FIELDDECLARATION [float [] val].

3.3.3. Qualified name replacement

Due to the dependency among source files in a project, qualified names are often introduced. As shown in Example 2, the
qualified name, CCTMXTiledMap.CCTMXOrientationOrtho, is actually 0 of int type. Such information can be obtained from the QualifiedName information Q which is part of the dependency information. In the above mentioned example, we collect the dependency information and replace CCTMX-TiledMap.CCTMXOrientationOrtho with 0. It is notable that the replacement occurs only when the corresponding qualified name refers to a literal (e.g., a string literal).

3.3.4. Method invocation reconstruction

A method invocation node in AST can be simply expressed by the following BNF:

\[
[ \text{Expression.} \mid \text{Identifier} \{ [ \text{Expression} \{, \text{Expression} \} ] \} ]
\]

Here Expression may represent different types of AST nodes (e.g., Name). The format is as below:

\[
[ \text{VarName/QualifiedClass.} \mid \text{Identifier} \{ [ \text{ParamName} \{, \text{ParamName} \} ] \} ]
\]

The first line could be either a variable name or a qualified class; the second line Identifier refers to a method name; the third line refers to a list of parameter names. Based on the representation, we define two templates to reconstruct method invocations (cf. MethodInvocation in Table 1). The first (resp. second) template is used when a method invocation occurs without (resp. with) the VarName part.

Method invocations exist and play an important role in most of the cases. However, with only the source code in methods we may not be able to obtain the qualified class or parameter types of the method invocation. As illustrated in Example 2(a), we cannot understand the qualified class of the method invocation positionForOrthoAt with class source code alone. In our approach, we extract and leverage the method declaration information for method invocations (C) to complement the lost information of qualified class and parameter types by analyzing the whole project. The reconstructed code is represented as line 5 in Example 2(b), which is matched with the first template of MethodInvocation in Table 1.

3.3.5. Try-catch filter

A try block in a method can help capture exceptions which is usually followed by a catch block as the exception handler. In light of the fact that the catch clauses are usually not addressed in code comments, we only consider the try block. Namely, when traversing the AST of a Java method, we concentrate on all TryStatement nodes. This could decrease the length of code sequences, compress the size of vocabulary and cut off redundant information, contributing to a high-quality dataset.

(a):

```java
if (hit)
    return false;
else {
    if (intersection != null)
        intersection.set(best);
    return true;
}
```

(b):

```java
IF0 ( hit == false )
return false ;
ENDIF0
ELSE0 IF1 ( intersection != null )
    ( Vector3 ) intersection . set ( Vector3 ) ( best ) ;
ENDIF1
return true ;
ENDELSE0
```

Example 3. (a) Represents the original code sequence of a Java method; (b) represents the filtered code sequence without making identifier replacement operation.

3.3.6. Loop and condition reconstruction

Some loop and condition statements account for a vital role in Java methods. To emphasize the importance of these statements, we choose and reconstruct if-else and for statements by the templates in Table 1. Thereinto, if-else statements are set to match with the template of IfStatement, while for statements containing two styles are set to match with the template of ForStatement and EnhancedForStatement respectively. We illustrate this process with an example for IfStatement category as follows (Example 3).

From the above example, we can see that each matched if-else is replaced with the IfStatement template in Table 1. To be specific, the first if is replaced with the specific token IF0, and the first else is matched with the first if and then replaced with ELSE0. At the end of if section, we add a specific token ENDIF0 for end. Similarly, we add ENDELSE0. The second if is replaced with IF1 since it is a nested if structure. Correspondingly, a token ENDIF1 is added. It is also noteworthy that the punctuation is unnecessary and removed since the token ENDELSE0 has expressed the meaning of termination. To explain the way to replace keywords with specific tokens (e.g., FOR0), we represent the algorithm of ForStatement reconstruction in Algorithm 1. Other cases of loop and condition reconstruction algorithms are similar.

3.3.7. Identifier replacement

There exist numerous unique tokens whose size far exceeds that of a reasonable vocabulary. Therefore, we need to compress the vocabulary size.
To this aim, we introduce a replacement algorithm to limit the vocabulary size in a reasonable range. Specifically, we replace identifiers in Java methods with some specific tokens by leveraging the context information, especially information from Java methods (S). Firstly, we integrate and sort all unique tokens by occurrence frequency and choose the top 30,000 tokens as the origin code vocabulary. Then, for those tokens from the origin code vocabulary, we make corresponding replacement operations. As for replacement operations, they can be divided into six categories, i.e., method name replacement, method invocation replacement, literal replacement, variable type replacement, variable name replacement and method declaration replacement. Correspondingly, we introduce some special tokens as substitutions (see Table 2). Considering that there exists only one name for each method, we add only one fixed token `<METHODNAME>` as a substitution. Since it is meaningless to distinguish string literals from each other, we replace them with the fixed token `<STRINGLITERAL>`. Similarly, we replace character literals with token `<CHARACTERLITERAL>`.

As for other special tokens, they all contain a variable i, a nonnegative integer, which aims to give a unique identifier. After such replacement operation, the tokens out of vocabulary will be replaced with the above mentioned tokens. Therefore, the size of unique tokens for code can be reduced to a limited range. For example, Example 1(b) shows the code sequence of the Java method before replacement, while Example 1(c) represents the final code sequence of the Java method with all code filtering and reconstruction operations. In our data set, the final vocabulary size for code is 30,351. However, the prerequisite is that we can extract necessary information from these methods (cf. Section 3.2). By leveraging this extracted context information, we can compress the code vocabulary considerably and build the final code vocabulary for training models.

### Algorithm 1 ForStatement Reconstruction.

**Input**: A nonnegative integer for counting the nested layers of `for` structure, `count`: An AST node, ForStatement, for representing the `for` structure; `r`: A token sequence of for structure, seq.

**Output**: A reconstructed token sequence of for structure, seq.

```plaintext
1: initial count ← 0
2: function RECONSTRUCTOR(r, count, seq)
3:   if r.isForStatement then
4:     for ← FORI ![replace current keyword for with specific token FORI] and i is a nonnegative integer and is equal to count
5:     count++
6:   end if
7:   if r.hasChild then
8:     for c in r.children do
9:       RECONSTRUCTOR(c, count, seq)
10:   end for
11: end if
12: add token ENDFORi ![ii is same to the i in token FORI]
13: return seq
14: end function
```

<table>
<thead>
<tr>
<th>Algorithm 1 ForStatement Reconstruction.</th>
</tr>
</thead>
</table>

3.3.8. Method elimination

We hypothesize that not all methods are worth considering. For instance the purpose of a constructor is to create an instance of a class, which is straightforward for developers to read and understand, and thus makes little sense to be included into the dataset. As a result, we leave out all constructors, getter, setter and test methods. Besides, we restrict the lengths of token sequences for Java methods to be in the range of 10 to 400. (Otherwise they are deemed to be overly simple or complicated.)

3.4. Comment filtering

Our training set originates from numerous open source projects from GitHub which feature comments of varying quality. However, a high-quality dataset is crucial to train a neural network. For this purpose, we need to distill these comments. The first sentence of a Javadoc comment usually expresses the meaning of the whole method. Thus, we choose to take it as the standard comment of the method. (Methods with no comments are simply ignored.) Moreover, quantities of comments contain useless, little or even no information, which should also be eliminated.

3.4.1. Filter by templates

Considering a comment like "never ever save this reference", which cannot directly provide any useful information for program comprehension. To remove these comments, we define a set of templates which are listed in Table 3. They are used to address a significant amount of comments which either indicate the current method is for testing, debugging, not implemented, useless and so on, or are automatically generated by some IDE tools (like ‘todo’ messages), or otherwise contain warning information not to use these comments or methods.

### Table 3

<table>
<thead>
<tr>
<th>Defined templates/rules for filtering comments with three categories of Automation, Indications and Uninformative.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Automation</td>
</tr>
<tr>
<td>Indications</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Uninformative</td>
</tr>
<tr>
<td>prepare - e.g., get Parameters</td>
</tr>
<tr>
<td>do not use</td>
</tr>
</tbody>
</table>

3.4.2. Filter by POS tagging

To further improve the quality of the dataset, we adopt the part-of-speech (POS) tagging technique. Particularly, we posit that a qualified comment should at least contain a verb in order to describe its main motivation. Therefore, comments without a verb are to be removed. We make use of the Stanford Tagger, which is the most commonly used tagger for English and has been applied to software artifacts (Binkley et al., 2011; Tian and Lo, 2014) widely. In particular, we set two thresholds to constrain the size of comments to be in the range of 3 to 30. (The comments which are out of the range are not considered.)

3.4.3. Identifier replacement

Observe that many identifiers in code occur in the corresponding comments as well. We therefore apply the similar replacement described in Section 3.3 to the comments. We sort all unique tokens ordered by frequency and choose the top 30,000 tokens as the comment vocabulary. For those tokens not in the comment vocabulary we apply the four kinds of replacement, i.e., method name replacement, method invocation replacement, variable type replacement and variable name replacement. Note that the replacement must respect the constraint that the replaced tokens in comments should be consistent with the corresponding replaced tokens in code. For example, if one token occurring in code and comment is out of the two vocabularies and replaced with `< SIMPLENAME_1 >`, the token in comment will be replaced with `< SIMPLENAME_1 >` likewise. In this case, even if one token is replaced with a special token, it can be converted to the original form by recording and leveraging extracted information. In this way, our approach can not only decrease the size of comment vocabulary, but also store the original forms of replaced tokens to help recover the replaced tokens.

3.5. Seq2Seq model

In this step, we train our code comment generation neural network by applying a Seq2Seq model which has been widely used for Neural Machine Translation (NMT) tasks (Sutskever et al., 2014; Rush et al., 2015).

Generally speaking, a Seq2Seq model can be simply divided into two recurrent neural networks (RNNs): the encoder which maps the input sequence into a fixed-dimensional vector, and the decoder, which maps the vector to the target sequence. In this paper, we choose LSTM, a variant of RNN, as the encoder and the decoder.

Fig. 2 illustrates the general architecture of a Seq2Seq model. Given the input sequence $X = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$, the target of the Seq2Seq model is to learn to generate the output sequence $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(l)})$. The aim of the model training is to estimate the conditional probability:

$$p(y^{(1)}, y^{(2)}, \ldots, y^{(l)} | x^{(1)}, x^{(2)}, \ldots, x^{(n)})$$

The model computes the conditional probability by first obtaining a fixed-dimensional representation $v$ of the input sequence $X = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$ (e.g., the last hidden state of the Encoder), and then computing the probability of $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(l)})$ with a standard LSTM-Language Model (LSTM-LM) formulation (Sutskever et al., 2014) whose initial hidden state is set to the representation $v$:

$$p(y^{(1)}, y^{(2)}, \ldots, y^{(l)} | x^{(1)}, x^{(2)}, \ldots, x^{(n)}) = \prod_{i=1}^{l} p(y^{(i)} | v, y^{(1)}, y^{(2)}, \ldots, y^{(i-1)})$$

(1)

In (1), $p(y^{(i)} | v, y^{(1)}, y^{(2)}, \ldots, y^{(i-1)})$ is represented as a softmax over all the words in the vocabulary.

Unfortunately, when the input sequence is too long, it is difficult for $v$ to store enough information which motivates the introduction of the attention mechanism (Bahdanau et al., 2014). It allows the Encoder to look up tokens in the input sequence whenever necessary. More concretely, the attention mechanism introduces a context vector $c_i$ which is usually computed as a weighted sum of the hidden states $h_j$ from the Encoder, i.e.,

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j$$

(2)

where the weight $\alpha_{ij}$ of each $h_j$ is computed by the following formula:

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

where $e_{ij} = a(s_{i-1}, h_j)$. Here, $a$ is an alignment model which is parameterized as a feedforward neural network and is jointly trained with all the other components; $s_i$ is the hidden state at time step $i$ in the Decoder, computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

(3)

4. Evaluation

4.1. Experimental setup

We train a neural network to automatically generate comments for Java methods. To this end, we extract (code, comment) pairs and build up a data set from 6705 Java projects downloaded from GitHub. By adopting the AST parser and the filter operations mentioned in Section 3, we obtain 542,429 (code, comment) pairs. We then divide the pairs into the training set, the validation set and the test set with the ratio of 8:1:1.
Fig. 3. Distribution of filtered code length.

Fig. 4. Distribution of filtered comment length.

Table 4
Statistics of all Java methods.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Java methods</th>
<th>Category</th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>6,705</td>
<td>542,429</td>
<td>Code</td>
<td>38,136,348</td>
<td>1,002,976</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comment</td>
<td>6,351,337</td>
<td>117,536</td>
</tr>
</tbody>
</table>

Table 5
Statistics for filtered Java methods on training set.

<table>
<thead>
<tr>
<th>Category</th>
<th>Unique tokens</th>
<th>Occurrence number &gt; 2</th>
<th>Average length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>30,351</td>
<td>-</td>
<td>79.36</td>
</tr>
<tr>
<td>Comment</td>
<td>96,241</td>
<td>29,433</td>
<td>10.40</td>
</tr>
</tbody>
</table>

Tables 4 and 5 give an overview of the collected data before and after processing respectively. Table 4 shows that there are more than 1,000,000 unique tokens in code which is prohibitively large. The unique tokens in comments are over 110,000 among which only about 30,000 unique tokens occur at least two times. Therefore, we set the size of the vocabulary for comments to 30,000. Similarly, the size of the vocabulary for codes is 30,351 (cf. Section 3.2).

Figs. 3 and 4 illustrate the distributions of the filtered code length and comment length respectively. Based on our statistics, the methods with less than 190 tokens account for more than 90% of the collected data, and the comments with less than 22 tokens account for more than 95% of the whole.

4.1.1. Training

We add special tokens <sos> and <eos> to our training sequences as the start flag and the end flag respectively. After the processing described in Section 3.3, there exists no <unk> token and the final size of the vocabulary for code is 30,351. The model is implemented in Python by leveraging the Tensorflow framework and extending the encoder-decoder model. Our hyper-parameters are determined based on the performance on the validation set. We use minibatch stochastic gradient descent to train and update parameters. The minibatch size is set to 100 and the dimensionality of the LSTM hidden states and word embedding is set to be 512. The learning rate is set to 0.99 initially and is decreased by a factor of 0.8. The parameter gradient is capped at 5. To avoid overfitting, we use a dropout rate of 0.3.

We train the models on GPUs. The training runs for about 50 epochs. We compute BLEU score on the validation set to select the
Table 6: Precision, Recall, and F1-score of different approaches.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>57.19%</td>
<td>41.42%</td>
<td>48.04%</td>
</tr>
<tr>
<td>Seq2Seq with Attention</td>
<td>60.17%</td>
<td>45.39%</td>
<td>51.74%</td>
</tr>
<tr>
<td>DeepCom</td>
<td>60.12%</td>
<td>48.19%</td>
<td>53.49%</td>
</tr>
<tr>
<td>ContextCC</td>
<td>62.03%</td>
<td>50.54%</td>
<td>55.70%</td>
</tr>
</tbody>
</table>

best model. During decoding, we set the beam size to 5, and the maximum comment length to 30 tokens.

4.1.2. Evaluation metrics

We first employ IR metrics to evaluate our proposed approach. Specifically, we calculate the precision, recall and F1-score of our approach. Precision represents the fraction of generated comments tokens that are relevant, while recall represents the fraction of relevant tokens that are generated. F1-score is the comprehensive measure combining both precision and recall. Let \( r \) be the total number of unigrams in the translation, \( t \) be the total number of unigrams in the reference, and \( m \) be the number of mapped unigrams between translation and reference. The unigram precision and the unigram recall are then \( P = m/t \) and \( R = m/r \) respectively.

We also evaluate the code comment generation model using two kinds of automatic machine translation metrics, i.e., BLEU-4 (Papineni et al., 2002) and METEOR (Denkowski and Lavie, 2014). BLEU-4 has been widely used for accuracy measure in multiple machine translation tasks. Given a set of reference sentences, BLEU-4 score depends on the geometric average of modified n-gram precision multiplying an exponential brevity penalty factor which is related to the lengths of translation and reference sentences. METEOR emphasizes the importance of recall to improve the correlation with human judgments. It extracts all uni-gram matches between translations and references. Based on these matches, METEOR is calculated by uni-gram recall, uni-gram precision and a fragmentation penalty which measures how well the matches correlate with each other. The higher BLEU-4 or METEOR score is, the closer the generated comment is to the reference one (ground truth), which suggests a better quality of the generated comment. The two metrics have also been used in other code comment generation tasks to measure accuracy (Erk and Smith, 2016; Hu et al., 2018).

4.2. Experimental results

4.2.1. Baseline

We compare ContextCC with DeepCom (Hu et al., 2018) which represents a state-of-the-art code comment generation approach. DeepCom introduces an algorithm called SBT to represent the code sequence and adopts the advantages of Seq2Seq model with attention mechanisms to generate comments for Java methods. DeepCom turns to outperform IR-based approaches and the model CODE-NN (Iyer et al., 2016), so they are not part of our baselines. We also compare ContextCC with a basic Seq2Seq model and a Seq2Seq model with attention mechanism.

4.2.2. Results

Table 6 presents the experimental results on the IR metrics for different approaches mentioned above. Precision denotes the proportion of matching words in the generated comments for Java methods. Results show that ContextCC outperforms other approaches.

We also evaluate the gap among different approaches on the two neural machine translation metrics, BLEU-4 and METEOR. Table 7 illustrates corpus level BLEU-4 scores and METEOR scores of different approaches to Java methods summarization. We show empirically that ContextCC outperforms the other approaches on both metrics of corpus level BLEU-4 and METEOR. The basic Seq2Seq model and the Seq2Seq model with attention adopt the advantages of LSTMs to explore and learn the semantics of Java source code with acceptable results. DeepCom attempts to use SBT algorithm to advance the representation of code sequences for Java methods, which performs better than the basic Seq2Seq model and the Seq2Seq model with attention. Compared with DeepCom, the corpus level BLEU-4 and METEOR scores of our approach, ContextCC, are both higher than those of DeepCom. Specifically, the BLEU-4 score of ContextCC is 40.52%, increasing by 6.41% by contrast with that of DeepCom, 38.08%, while the METEOR score of ContextCC improves by about 6.26%. We also compare the average BLEU-4 scores for ContextCC and DeepCom of different code and comment lengths. Figs. 5 and 6 show the comparison respectively. As illustrated in Fig. 5, the average BLEU-4 scores of ContextCC and DeepCom tend to decrease as code length increases. However, ContextCC is always superior to DeepCom despite of different code lengths. As shown in Fig. 6, ContextCC is also superior to DeepCom despite of different comment lengths. Through the experiments and evaluation, we can demonstrate the effectiveness of ContextCC, as well as the superiority over other baseline works.

4.2.3. Qualitative analysis

We focus the qualitative analysis on the ground truths and comments generated by ContextCC for Java methods. Table 8 presents examples of the generated comments by ContextCC in the testing set. Most generated comments are brief, natural and informative. For further analysis and research, we mainly clarify the relationship between ground truths and Java method summaries generated by ContextCC as follows:

4.3. Absolutely matched comments.

ContextCC can generate absolutely matched comments for Java methods regardless of the lengths of token sequences of Java methods (Example 1, Example 2 and Example 3 in Table 8). According to our statistics, many generated comments in the testing set are perfectly matched comments, which verifies the capability of ContextCC.

4.4. Unknown tokens

We take 30,000 as the size of the vocabulary for comments, while the unique tokens occurring in comments are far more than 30,000. Therefore, it is unavoidable that there exist unknown tokens in generated comments. As the fourth example shown in Table 8, ContextCC predicts the special token `<unk>` instead of `javax.servlet.http.HttpServletResponse` which is the actually correct token. Arguably it is acceptable for ContextCC to fail to predict some identifiers; in this case, `javax.servlet.http.HttpServletResponse` is an identifier defined by the developer which rarely occurs and identifier prediction is generally challenging.

Table 7: Performance on the testing set.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU-4 (%)</th>
<th>METEOR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>31.21</td>
<td>23.40</td>
</tr>
<tr>
<td>Seq2Seq with Attention</td>
<td>35.25</td>
<td>25.95</td>
</tr>
<tr>
<td>DeepCom</td>
<td>38.08</td>
<td>26.83</td>
</tr>
<tr>
<td>ContextCC</td>
<td>40.52</td>
<td>28.51</td>
</tr>
</tbody>
</table>
4.5. Replaced tokens

Some tokens may be replaced by their corresponding synonyms, antonyms or some other tokens in the same domain in the generated comments by ContextCC. For the fifth example illustrated in Table 8, ContextCC fails to predict the token “Stops”, but to predict the token “starts”, an antonym of the token “Stops”, which results in the opposite of the ground truth.

5. Threats to validity

5.1. Internal validity

We extract the first sentence of Javadoc comments as the standard comment of the method which is also adopted by previous work (Hu et al., 2018). As discussed earlier, the quality of Javadoc comments varies and low-quality comments could potentially hamper the performance of the trained model. To mitigate the threat, we design rules and patterns to filter the comments. However, noise may still exist in the final dataset. We also set the length constraints to further improve the comments quality. In general, comments with too few tokens are difficult to clarify the intention of the methods clearly. For example, methods’ comments which are simply “return false”, “return true” or “return this” are not informative. To improve comments’ quality, we set 3 as the minimum length of comments and filter those less than 3 out. Moreover, we analyzed the distribution of such kinds of comments empirically: the percentages of the comments with less than 3 tokens or more than 30 tokens are around 0.97% and 0.72% respectively.
Another threat originates from the source codes of the selected projects. To prepare the training data, we extract the code related context information from Java projects. However, this depends on the completeness and correctness of the analyzed Java projects. (If some Java or jar files are missing, the extracted context information may be incomplete.) To address this concern, we scale up the studied projects and extract context information from 6705 Java projects. Moreover, we only consider those projects from Github with high stars. We only consider the methods whose lengths are within a certain range, and omit those which are either too long or too short. Empirically, we found that these eliminated methods only account for a quite small proportion of the whole dataset. Concretely, there are 7018 methods with less than 10 tokens, taking up 0.64%, most of which are empty methods (some are with a single return statement). There are 24,115 methods with more than 400 tokens, taking up 2.20%. Our findings are consistent with those reported in the baseline work (Hu et al., 2018) and we set the same threshold in our experiment.

5.2. External validity

External validity is concerned with the generalizability of results on the datasets other than the ones used in the experiments (Feldt and Magazinius, 2010). Indeed, in our approach, we only focus on the comments generation for Java methods. However, we believe that the approach is essentially independent of specific (object-oriented) programming languages, since the context information harnessed in our approach generally exists and could be extracted from the projects developed by other general-purpose object-oriented programming languages. However, for some DSLs, they are usually declarative, and offering only a restricted suite of notations and abstractions for a particular application domain (Deursen et al., 2000). Their grammar usually differs widely with GPLs, and thus our approach could not be directly applied.

6. Conclusion

In this paper, we presented ContextCC, an automated approach to generate comments for Java methods. ContextCC harnesses a Seq2Seq Neural Network model with an attention mechanism. It takes code sequences of Java methods and the corresponding comments as inputs. The code sequences are obtained by leveraging context information extraction in conjunction with method filtering and reconstruction operations. Complemented by the context information, the code sequences contain more complete and richer information. Experiments confirm that ContextCC outperforms most state-of-the-art approaches.

For future work, we plan to further generalize our approach, integrate more structural information and provide better usability to end users for comment generation related tasks.

Acknowledgements

This work was partially supported by the National Key R&D Program of China (No. 2018YFB1003900), the Fundamental Research Funds for the Central Universities (No. NS2019055), the National Natural Science Foundation of China (No. 61802179), the Collaborative Innovation Center of Novel Software Technology in China, and Qing Lan Project. T. Chen is partially supported by UK Engineering and Physical Sciences Research Council grant (EP/P00430X/1), National Natural Science Foundation of China grant (No. 61662035), and Guangdong Science and Technology Department Grant (No. 2018B010107004).

References


Yu Zhou is a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his BSc degree in 2004 and PhD degree in 2009, both in computer science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politecnico di Milano, Italy. From 2015–2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis. He has been supported by several national research programs in China.

Xin Yan received his BSc degree in software engineering, from Jiangsu University, China, in 2017. He is currently a MSc student in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. His research interests include software evolution analysis, artificial intelligence, and mining software repositories.

Wenhua Yang is an assistant professor in the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. He received his PhD degree in the computer science and technology in Nanjing University. His research interests include software engineering, self-adaptive software systems and IoT systems.
Taolue Chen received the Bachelor’s and Master’s degrees from the Nanjing University, China, both in computer science. He was a junior researcher (OiO) at the CWI and acquired the PhD degree from the Free University Amsterdam, The Netherlands. He is currently a lecturer at the Department of Computer Science and Information Systems, Birkbeck, University of London. Prior to this post, he was a [senior] lecturer at Middlesex University London, a research assistant at the University of Oxford, and a postdoctoral researcher at the University of Twente, The Netherlands. His research interests include formal verification and synthesis, program analysis, logic in computer science, and software engineering.

Zhiqiu Huang is a full professor of Nanjing University of Aeronautics and Astronautics. He received his BSc. and MSc degrees in computer science from National University of Defense Technology of China. He received his PhD degree in computer science from Nanjing University of Aeronautics and Astronautics of China. His research interests include big data analysis, cloud computing, and web services.