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

Yu Zhou[a,*], Xin Yan[a], Wenhua Yang[a], Taolue Chen[b,c], Zhiqiu Huang[a]

[a]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[b]*Department of Computer Science and Information Systems, Birkbeck, University of London, UK*
[c]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*

Abstract

Code comments are crucial to program comprehension. In this paper, we propose a novel approach ContexCC to automatically generate concise comments for Java methods based on neural networks, leveraging techniques of program analysis and natural language processing. Firstly, ContexCC 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, ContexCC 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 ContexCC is superior to state-of-the-art baseline methods.

Keywords: Comment generation, Neural networks, Natural language processing

1. Introduction

Program comprehension represents an expensive and time-consuming task in software development and maintenance [1]. Comments written in natural languages can greatly facilitate programmers to understand the meaning of a code snippet [2, 3, 4], 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 Unfortunately, comments in software artifacts are often incomplete, outdated, incorrect or otherwise missing, partially because commenting code is generally time-consuming and laborious, and thus programmers often ignore writing and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

*Corresponding author
Email address: `zhouyu@nuaa.edu.cn` (Yu Zhou)

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 [5]. However, the work [5] only focused on statements related to exception triggering conditions, which limits its generality and applicability. In [6], 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 [7], and generated comments for parameters to be part of the method comments [8]. 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.* [9] 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.* [10] 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.* [11] created a probabilistic model over the code which was used to retrieve natural language snippets. Haiduc *et al.* [12] examined a number of methods, e.g., term frequency-inverse document frequency (TF-IDF) and latent semantic indexing (LSI), to select keywords for code summary. In order to generate comments for the code, these IR-based approaches rely on assumptions that similar code snippets exist in the repository which could later be retrieved.

Neural Networks have demonstrated great promise in a variety of natural language processing (NLP) tasks. In particular, Recurrent Neural Networks (RNNs) are widely exploited to learn from the training code and comments to generate new comments. For instance, Iyer *et al.* [13] presented CODE-NN, an end-to-end neural attention model using long short-term memory (LSTM, a variant of RNN) units, to generate comments of C# and SQL code by learning from online programming websites. Allamanis *et al.* [14] introduced an attentional neural network that employs convolution on the input tokens to convert source code snippets into short and descriptive function name-like comments. Hu *et al.* [15] proposed an algorithm, SBT, to generate descriptive comments for Java methods based on RNNs.

These neural network based approaches are effective, yet have limitations. First, the code snippets are simply treated as independent symbols and are directly fed to the neural network, in which the context information (e.g., the type information of variables) is largely ignored. As a result, the information in the code snippet is incomplete, which inevitably degrades the quality of the collected data. For example, as illustrated in Example 1(a), the identifiers *val* and *M03* are used directly in the method body whereas the declarations of these variables are in the enclosed class. This suggests that examining the method solely may not be able to capture valuable context information such as the type and literal information. Second, these approaches pay little attention to the quality of comments in the collected data set. Usually they merely take the first sentence of Javadoc comments as the default comments of code snippets without further processing. However, we have found that a considerable portion of these comments contain useless, little or even no information. Apparently, these comments should have been removed from the training set.

Example 1. (a) represents the original code sequence of the Java method; (b) represents the filtered code sequence without making identifier replacement operation; (c) represents the final code sequence.

```
(a):
public Matrix4 trn (Vector3 vector) {
     val[M03] += vector.x;
     val[M13] += vector.y;
     val[M23] += vector.z;
     return this;
  }

(b):
Matrix3 SEGMENTATION trn [ Vector3 vector ] {
     FIELDDECLARATION [ float [ ] val ; ]
       val [ 12 ] += vector . x ;
       val [ 13 ] += vector . y ;
       val [ 14 ] += vector . z ;
       return this ;
  }

(c):
Matrix4 SEGMENTATION <METHODNAME> [ Vector3 vector ] {
     FIELDDECLARATION [ float [ ] val ; ]
       val [ 12 ] += vector . x ;
       val [ 13 ] += vector . y ;
       val [ 14 ] += vector . z ;
       return this ;
  }
```

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, ContexCC 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 ContexCC 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 6,700 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 predefined templates and rules [16, 9, 17, 18, 19, 5, 20, 21, 22]. Hill et al. [23] introduce Software Word Usage Model (SWUM) model which can translate Java method invocations into descriptive statements. Sridhara et al. [6] use SWUM combined with a group of pre-defined templates and rules to generate comments for Java methods. Sridhara et al. [7] 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. [8] introduce a new technique to generate comments for parameters and treat the descriptions as part of method summary. Beyond method level, Moreno et al. [9] 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. [10] 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. [11] create a probabilistic model over code, but use it in the opposite direction to also retrieve full natural language snippets. Haiduc et al. [12] 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 (DLSs) such as SQL, and General-Purpose Languages (GPLs) such as Java. Iyer et al. [13] present CODE-NN, an end-to-end neural attention model using LSTMs to generate summaries of C# and SQL statements by learning from noisy online programming websites. Yao et al. [24] 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. [14] 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. [15] 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 leverag-

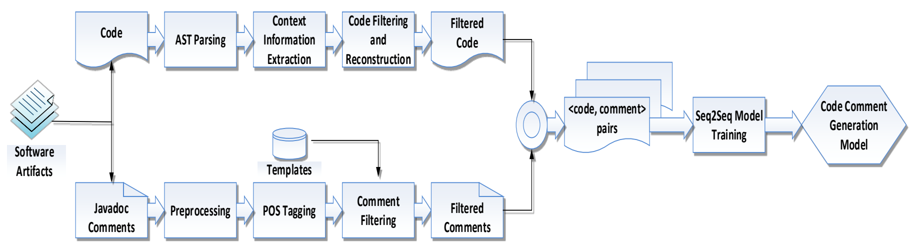

Figure 1: Overview of our approach

ing deep neural networks. Liang et al. [25] 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 ContexCC Approach

In this section, we present our approach, ContexCC, 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 6,700 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 [26] to conduct the AST analysis where *MethodDeclaration*[1] 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. Section 3.2—3.4) to prepare a data set for the training of the neural network model for code comment generation (cf. Section 3.5).

[1]http://help.eclipse.org/

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.

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.

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.

- 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

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*, *M13* 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).

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

Table 1: Templates for dependency information. FIELDDECLARATION refers to a special token. *VarDeclarationList* refers to a list of variable declarations. *QualifiedClass* refers to a qualified class name, *VarName* refers to a variable name, Identifier refers to a method name, *(ParamType* { *, ParamType* } *])* refers to a list of parameter types and *([ParamName* { *, ParamName* } *])* refers to a list of parameter names. i represents a nonnegative integer and can increase with nested layers of some specific statements (e.g., *IfStatement*).

Category	Templates/Rules
MethodInvocation	*QualifiedClass . Identifier (ParamType* { *, ParamType* } *]) ([ParamName* { *, ParamName* } *])* *(QualifiedClass) VarName . Identifier (ParamType* { *, ParamType* } *]) ([ParamName* { *, ParamName* } *])*
FieldAccess	FIELDDECLARATION [VarDeclarationList]
IfStatement	IF_i ENDIF_i ELSE_i ENDELSE_i
ForStatement	FOR_i ENDFOR_i
EnhancedForStatement	ENHANCEDFOR_i ENDENHANCEDFOR_i

field declaration information of the variable, *val*, is supplemented in the form of `FIELDDECLARATION [float [ ] val]`.

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 *CCTMXTiledMap.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).

Example 2. (a) represents the original code sequence of the Java method; (b) represents the filtered code sequence without making identifier replacement operation.

```
(a):
public CGPoint positionAt(CGPoint pos) {
     CGPoint ret = CGPoint.zero();
     switch( layerOrientation_ ) {
        case CCTMXTiledMap.CCTMXOrientationOrtho:
           ret = positionForOrthoAt(pos);
           break;
        case CCTMXTiledMap.CCTMXOrientationIso:
           ret = positionForIsoAt(pos);
           break;
        case CCTMXTiledMap.CCTMXOrientationHex:
           ret = positionForHexAt(pos);
           break;
     }
     return ret;
  }

(b):
CGPoint SEGMENTATION positionAt [ CGPoint pos ] {
       FIELDDECLARATION [ int layerOrientation_ ; ]
       CGPoint ret = CGPoint . zero ( ) ;
       switch ( layerOrientation_ ) {
           case 0 :
               ret = CCTMXLayer . positionForOrthoAt ( CGPoint ) ( pos ) ;
               break ;
           case 2 :
               ret = CCTMXLayer . positionForIsoAt ( CGPoint ) ( pos ) ;
                break ;
           case 1 :
               ret = CCTMXLayer . positionForHexAt ( CGPoint ) ( pos ) ;
               break ;
        }
        return ret ;
  }
```

Method Invocation Reconstruction. A method invocation node in AST can be simply expressed by the following BNF:

[Expression .] Identifier ([Expression { , Expression }])

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

[VarName/QualifiedClass .]
Identifier
([ParamName { , 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 methods. 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.

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.

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

```
(a):
```

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

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

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.

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

Algorithm 1 ForStatement Reconstruction

```
Input: A nonnegative integer for counting the nested layers of *for* structure,
    $count$; An AST node, ForStatement, for representing *for* structure, $r$; A
    token sequence of *for* structure, $seq$.
Output: A reconstructed token sequence of *for* structure, $seq$.
 1: initial $count \leftarrow 0$
 2: function RECONSTRCTOR($r, count, seq$)
 3:     if $r.isForStatement$ then
 4:         $for \leftarrow FORi$ //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 $c$ $in$ $r.childs$ do
 9:             RECONSTRCTOR($c$,$count$,$seq$)
10:         end for
11:     end if
12:     $add\ token\ ENDFORi$ //$i$ is same to the $i$ in token $FORi$
13:     return $seq$
14: end function
```

<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.

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 distil these comments.

Table 2: Identifier Replacement where i represents a nonnegative integer.

Category	**Special Tokens**
Method Name Replacement	<METHODNAME>
Method Invocation Replacement	<METHODINVOCATION_i>
Literal Replacement	STRINGLITERAL <NUMLITERAL_i> <CHARACTERLITERAL>
Variable Type Replacement	<SIMPLETYPE_i>
Variable Name Replacement	<SIMPLENAME_i>
Method Declaration Replacement	<METHODDECLARATIONNAME_i>

Table 3: Defined templates/rules for filtering comments with three categories of **Automation**, **Indications** and **Uninformative**. A template ending with the punctuation $*$ represents that its lower case is another kind of template. <hyperlink> starts with "http(s)://" or "<a?href".

Category	Templates/Rules
Automation	//TODO: TODO
Indications	Tests for* For test* bug/Bug/BUG For debug* WARNING : note/Note/NOTE : Not implement* not been implemented NOT IMPLEMENTED* TEST METHOD* NEVER EVER SAVE THIS REFERENCE*
Uninformative	<hyperlink> prepare - e.g., get Parameters do not use

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.

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.

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 [27, 28] widely. In particular, we set two thresholds to constraint the size of comments to be in the range of 3 to 30. (The comments which are our of the range are not considered.)

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 origin 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 [29, 30].

Generally speaking, a Seq2Seq model can be simply divided into two recurrent neural networks (RNNs): the *encoder* which maps the input sequence into a fix-dimensional vector, and the *decoder*, which maps the vector to the target

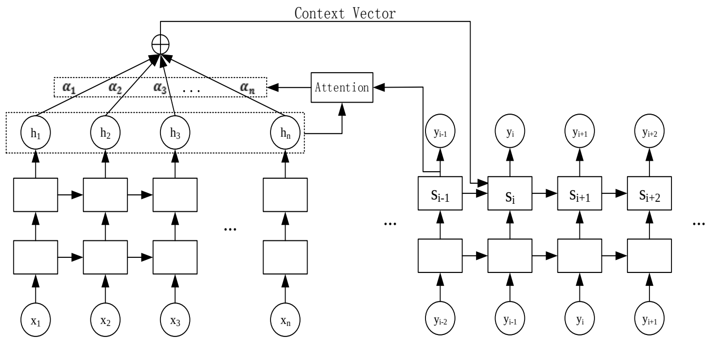

Figure 2: Seq2Seq Model with Attention Mechanism

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:

$$p\left(y^{(1)}, y^{(2)}, ..., y^{(l)} \mid x^{(1)}, x^{(2)}, ..., x^{(n)}\right)$$

The model computes the conditional probability by first obtaining a fixed-dimensional representation v of the input sequence $X = (x^{(1)}, x^{(2)}, ..., x^{(n)})$ (e.g., the last hidden state of the Encoder), and then computing the probability of $Y = (y^{(1)}, y^{(2)}, ..., y^{(l)})$ with a standard LSTM-Language Model (LSTM-LM) formulation [29] whose initial hidden state is set to the representation v:

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

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 [31]. 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.,

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

where the weight α_{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 parametrized 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

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{3}$$

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 6,705 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.

Table 4 and Table 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 (see Section 3.2).

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

Table 4: Statistics of all Java methods

Projects	Java Methods	Category	Total	Unique
6,705	542,429	Code	38,136,348	1,002,976
		Comment	6,351,337	117,536

Table 5: Statistics for filtered Java methods on training set

Category	Unique tokens	Occurrence Number>2	Average Length
Code	30,351	-	79.36
Comment	96,241	29,433	10.40

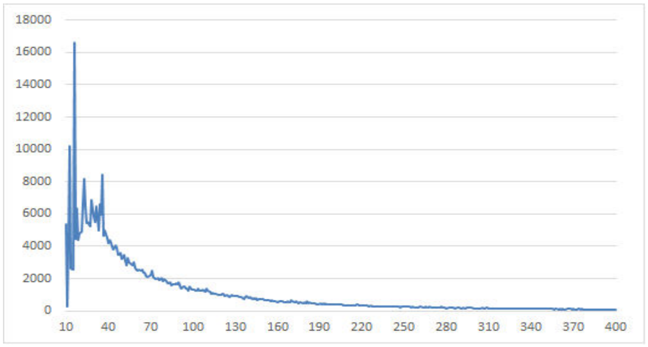

Figure 3: Distribution of filtered code length

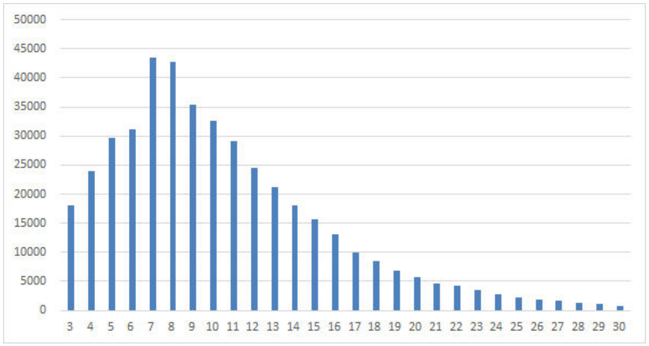

Figure 4: Distribution of filtered comment length

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 over-fitting, 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 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 t be the total number of unigrams in the translation, r 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 [32] and METEOR [33]. BLEU-4 has been widely used for accuracy measure in multiple machine translation tasks, which aims to measure the geometric average of modified n-gram precisions on a set of reference sentences with multiplying an exponential brevity penalty factor. METEOR is a recall-oriented metrics and evaluates translation hypotheses by aligning them to reference translations and calculating sentence-level similarity scores. The higher BLEU 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 [34, 15].

4.2. Experimental Results

4.2.1. Baseline

We compare ContexCC with DeepCom [15] 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 [13], so they are not part of our baselines. We also compare ContexCC with a basic Seq2Seq model and a Seq2Seq model with attention mechanism.

Table 6: Precision, Recall, and F1-score of different approaches.

Approaches	**Precision**	**Recall**	**F1-score**
Seq2Seq	57.19%	41.42%	48.04%
Seq2Seq with Attention	60.17%	45.39%	51.74%
DeepCom	60.12%	48.19%	53.49%
ContextCC	**62.03%**	**50.54%**	**55.70%**

Table 7: Performance on the testing set.

Model	**BLEU-4 (%)**	**METEOR (%)**
Seq2Seq	31.21	23.40
Seq2Seq with Attention	35.25	25.95
DeepCom	38.08	26.83
ContextCC	**40.52**	**28.51**

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 ContexCC 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 ContexCC 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, ContexCC, are both higher than those of DeepCom. Specifically, the BLEU-4 score of ContexCC is 40.52%, increasing by 6.41% by contrast with that of DeepCom, 38.08%, while the METEOR score of ContexCC improves by about 6.26%. Through the experiments and evaluation, we can demonstrate the effectiveness of ContexCC, 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 ContexCC for Java methods. Table 8 presents examples of the generated comments by ContexCC 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 ContexCC as follows:

Absolutely matched comments. ContexCC 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 ContexCC.

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, ContexCC predicts the special token <unk> instead of *javax.sip.TimeoutEvent* which is the actually correct token. Arguably it is acceptable for ContexCC to fail to predict some identifiers; in this case, *javax.sip.TimeoutEvent* is an identifier defined by the developer which rarely occurs and identifier prediction is generally challenging.

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 ContexCC. For the fifth example illustrated in Table 8, ContexCC 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.

Table 8: Examples of comments generated by models for Java methods in testing set

Number	Java Method	Comments

1	public static String UTF8toUTF16 (byte [ ] utf8 , int offset , int len) {
	char [ ] out = new char [len] ;
	int n = UTF8toUTF16 (utf8 , offset , len , out , 0) ;
	return new String (out , 0 , n) ;
	}
		Gold: Convert UTF8 bytes into a String
		ContextCC: convert UTF8 bytes into a String

2	public void addChangingListener (OnWheelChangedListener listener) {
	changingListeners . add (listener) ;
	}
		Gold: Adds wheel changing listener
		ContexCC: adds wheel changing listener

3	public void removeConnection (Connection connection) {
	connections . remove (connection . handle ( )) ;
	persistence . deleteConnection (connection) ;
	}
		Gold: Removes a connection from the map of connections
		ContexCC: removes a connection from the map of connections

4	public ServerHeader createServerHeader (List product) throws ParseException {
	if (product == null)
	throw new NullPointerException ("null productList arg") ;
	Server server = new Server ( ) ;
	server . setProduct (product) ;
	return server ;
	}
		Gold: Returns the collected javax.sip.TimeoutEvent or null if no event has been collected
		ContexCC: returns the collected <unk> or null if no event has been collected

5	private void stopPolling ( ) {
	String notifier = Preference . getString (
	context , Constants . PreferenceFlag . NOTIFIER_TYPE) ;
	if (Constants . NOTIFIER_LOCAL . equals (notifier)) {
	LocalNotification . stopPolling (context) ; }
	}
		Gold: Stops server polling task
		ContextCC: starts server polling task

5. Threats to Validity

Internal Validity. We extract the first sentence of Javadoc comments as the standard comment of the method which is also adopted by previous work [15]. As discussed earlier, the quality of Javadoc comments varies and low-quality comments could potentially hamper the performance of the trained model. To

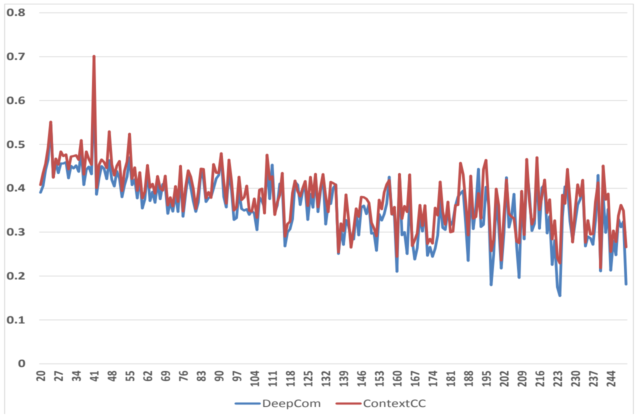

Figure 5: The average BLEU-4 scores of different code lengths for ContexCC and DeepCom

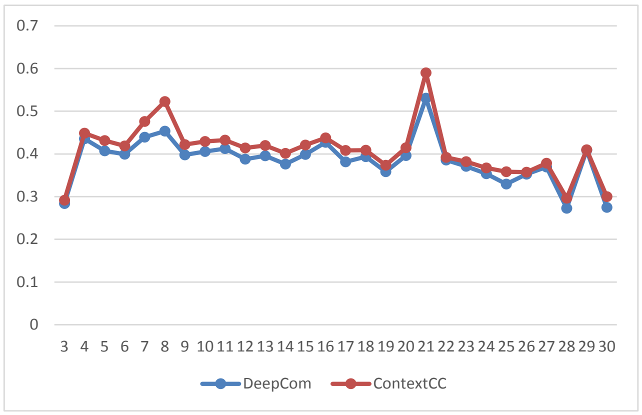

Figure 6: The average BLEU-4 scores of different comment lengths for ContexCC and DeepCom

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 6,705 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 7,018 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 [15] and we set the same threshold in our experiment.

External Validity. External validity is concerned with the generalizability of results on the datasets other than the ones used in the experiments [35]. 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.

6. Conclusion

In this paper, we presented ContexCC, an automated approach to generate comments for Java methods. ContexCC 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 ContexCC 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 ender users for comment generation related tasks.

Acknowledgements

This work was partially supported by the National Key R&D Program of China (No. 2018YFB1003902), the Fundamental Research Funds for the Central Universities (No. NS2019055), the Collaborative Innovation Center of Novel Software Technology in China and Qing Lan Project. T. Chen is partially supported by UK EPSRC grant (EP/P00430X/1), NSFC grant (No. 61662035), and Guangdong Science and Technology Department Grant (No. 2018B010107004).

References

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li, Measuring program comprehension: A large-scale field study with professionals, IEEE Transactions on Software Engineering 44 (10) (2018) 951–976.

[2] A. A. Takang, P. A. Grubb, R. D. Macredie, The effects of comments and identifier names on program comprehensibility: an experimental investigation, J. Prog. Lang. 4 (3) (1996) 143–167.

[3] T. Tenny, Program readability: Procedures versus comments, IEEE Transactions on Software Engineering 14 (9) (1988) 1271–1279.

[4] A. Forward, T. Lethbridge, The relevance of software documentation, tools and technologies: a survey., in: ACM Symposium on Document Engineering, 2002, pp. 26–33.

[5] R. P. Buse, W. R. Weimer, Automatic documentation inference for exceptions, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ACM, 2008, pp. 273–282.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, K. Vijay-Shanker, Towards automatically generating summary comments for java methods, in: 2010 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2010, pp. 43–52.

[7] G. Sridhara, L. L. Pollock, K. Vijay-Shanker, Automatically detecting and describing high level actions within methods, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 101–110.

[8] G. Sridhara, L. Pollock, K. Vijay-Shanker, Generating parameter comments and integrating with method summaries, in: 2011 IEEE 19th International Conference on Program Comprehension (ICPC), 2011, pp. 71–80.

[9] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, K. Vijay-Shanker, Automatic generation of natural language summaries for java classes, in: 2013 IEEE 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 23–32.

[10] E. Wong, J. Yang, L. Tan, Autocomment: Mining question and answer sites for automatic comment generation, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 562–567.

[11] M. Allamanis, D. Tarlow, A. Gordon, Y. Wei, Bimodal modelling of source code and natural language, in: International Conference on Machine Learning, 2015, pp. 2123–2132.

[12] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, ACM, 2010, pp. 223–226.

[13] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer, Summarizing source code using a neural attention model, in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vol. 1, 2016, pp. 2073–2083.

[14] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Suggesting accurate method and class names, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 38–49.

[15] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: Proceedings of the 26th Conference on Program Comprehension, ACM, 2018, pp. 200–210.

[16] N. J. Abid, N. Dragan, M. L. Collard, J. I. Maletic, Using stereotypes in the automatic generation of natural language summaries for c++ methods, in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE, 2015, pp. 561–565.

[17] S. Zhang, C. Zhang, M. D. Ernst, Automated documentation inference to explain failed tests, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2011, pp. 63–72.

[18] M. Kamimura, G. C. Murphy, Towards generating human-oriented summaries of unit test cases, in: 2013 IEEE 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 215–218.

[19] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, D. Poshyvanyk, On automatically generating commit messages via summarization of source code changes, in: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on, IEEE, 2014, pp. 275–284.

[20] R. P. Buse, W. R. Weimer, Automatically documenting program changes, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM, 2010, pp. 33–42.

[21] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, H. Gall, Analyzing apis documentation and code to detect directive defects, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, 2017, pp. 27–37.

[22] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, H. C. Gall, Automatic detection and repair recommendation of directive defects in java api documentation, IEEE Transactions on Software Engineering.

[23] E. Hill, L. Pollock, K. Vijay-Shanker, Automatically capturing source code context of nl-queries for software maintenance and reuse, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 232–242.

[24] Z. Yao, J. R. Peddamail, H. Sun, Coacor: Code annotation for code retrieval with reinforcement learning, CoRR abs/1904.00720.

[25] Y. Liang, K. Q. Zhu, Automatic generation of text descriptive comments for code blocks, arXiv preprint arXiv:1808.06880.

[26] R. Fuhrer, M. Keller, A. Kiezun, Advanced refactoring in the eclipse jdt: Past, present, and future, in: Proc. ECOOP Workshop on Refactoring Tools (WRT), 2007, pp. 31–32.

[27] D. Binkley, M. Hearn, D. Lawrie, Improving identifier informativeness using part of speech information, in: Working Conference on Mining Software Repositories, 2011, pp. 203–206.

[28] Y. Tian, D. Lo, A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, 2014, pp. 570–574.

[29] I. Sutskever, O. Vinyals, Q. V. Le, Sequence to sequence learning with neural networks, in: Advances in neural information processing systems, 2014, pp. 3104–3112.

[30] A. M. Rush, S. Chopra, J. Weston, A neural attention model for abstractive sentence summarization, arXiv preprint arXiv:1509.00685.

[31] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv preprint arXiv:1409.0473.

[32] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th annual meeting on association for computational linguistics, Association for Computational Linguistics, 2002, pp. 311–318.

[33] M. Denkowski, A. Lavie, Meteor universal: Language specific translation evaluation for any target language, in: Proceedings of the ninth workshop on statistical machine translation, 2014, pp. 376–380.

[34] K. Erk, N. A. Smith, Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers), in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vol. 1, 2016.

[35] R. Feldt, A. Magazinius, Validity threats in empirical software engineering research-an initial survey., in: Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering, 2010, pp. 374–379.