# Snippet Comment Generation Based on Code Context Expansion

HANYANG GUO, School of Software Engineering, Sun Yat-Sen University and Department of Computer Science, Hong Kong Baptist University, China
XIANGPING CHEN, Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-Sen University, China
YUAN HUANG and YANLIN WANG, School of Software Engineering, Sun Yat-Sen University, China
XI DING, School of Computer Science and Engineering, Sun Yat-sen University, China
ZIBIN ZHENG, School of Software Engineering, Sun Yat-Sen University, China
XIAOCONG ZHOU, School of Computer Science and Engineering, Sun Yat-sen University, China
HONG-NING DAI, Department of Computer Science, Hong Kong Baptist University, China

Code commenting plays an important role in program comprehension. Automatic comment generation helps improve software maintenance efficiency. The code comments to annotate a method mainly include header comments and snippet comments. The header comment aims to describe the functionality of the entire method, thereby providing a general comment at the beginning of the method. The snippet comment appears at multiple code segments in the body of a method, where a code segment is called a code snippet. Both of them help developers quickly understand code semantics, thereby improving code readability and code maintainability. However, existing automatic comment generation models mainly focus more on header comments, because there are public datasets to validate the performance. By contrast, it is challenging to collect datasets for snippet comments, because it is difficult to determine their scope. Even worse, code snippets are often too short to capture complete syntax and semantic information. To address this challenge, we propose a novel Snippet Comment Generation approach called *SCGen*. First, we utilize the context of the code snippet to expand the syntax and semantic information. Specifically, 600,243 snippet code-comment pairs are collected from 959 Java projects. Then, we capture variables from code snippets and extract variable-related statements from the context. After that, we devise an algorithm to parse and traverse abstract syntax tree (AST) information of code snippets and corresponding context. Finally, *SCGen* generates snippet comments after inputting the source code snippet and corresponding AST information into a

The work described in this article is supported by the Key-Area Research and Development Program of Guangdong Province (2020B010164002), the National Natural Science Foundation of China (62032025, 61976061), and the Guangdong Basic and Applied Basic Research Foundation (2023A1515010746).
Authors' addresses: H. Guo, School of Software Engineering, Sun Yat-Sen University and Department of Computer Science, Hong Kong Baptist University, China, 519000; email: guohy36@mail2.sysu.edu.cn; X. Chen, Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-Sen University, Guangzhou, China; email: chenxp8@mail.sysu.edu.cn; Y. Huang, Y. Wang, and Z. Zheng (corresponding author), School of Software Engineering, Sun Yat-Sen University, Zhuhai, China; X. Ding and X. Zhou, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; emails: 769019734@qq.com, isszxc@mail.sysu.edu.cn; H.-N. Dai, Department of Computer Science, Hong Kong Baptist University, Hong Kong, China.

1049-331X/2023/11-ART24 $15.00
https://doi.org/10.1145/3611664

sequence-to-sequence-based model. We conducted extensive experiments on the dataset we collected to evaluate our *SCGen*. Our approach obtains 18.23 in BLEU-4 metrics, 18.83 in METEOR, and 23.65 in ROUGE-L, which outperforms state-of-the-art comment generation models.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools*; • **Computing methodologies** → **Artificial intelligence;**

Additional Key Words and Phrases: Snippet comment generation, code summarization, neural machine translation, contextual information

**ACM Reference format:**
Hanyang Guo, Xiangping Chen, Yuan Huang, Yanlin Wang, Xi Ding, Zibin Zheng, Xiaocong Zhou, and Hong-Ning Dai. 2023. Snippet Comment Generation Based on Code Context Expansion. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 24 (November 2023), 30 pages.
https://doi.org/10.1145/3611664

## 1 INTRODUCTION

Code comment refers to the explanation and description of code functionality. Its purpose is to make software developers easily understand code semantics [24, 27]. Therefore, it plays an important role in program comprehension and code readability [10, 18]. High-quality code comments can help software maintainers have a detailed understanding of software source code and thereby improve the efficiency of discovering and fixing bugs [7, 34]. Although code comments are important in software maintenance, writing code comments manually costs lots of time and effort [30]. To this end, many studies focus on automatic comment generation [40, 45].

Automatic comment generation is also important to *methods* in **object-oriented programming (OOP)**, where a method is a programmed procedure defined in a class and instantiated by an object. There are two kinds of comments to annotate a method: (1) header comments [46] and (2) snippet comments [6]. Header comments are used to summarize the whole method (or called function) information [1, 51]. They are typically located at the head of the method. Snippet comments are fine-grained comments, which are typically embedded inside the method. They are generally used to explain some specific intentions of the code and form a complementary relationship with header comments [17, 47]. Both snippet comments and header comments are important in program comprehension for developers [19]. Generally, a header comment has a fixed scope, i.e., the whole method. By contrast, a snippet comment has a variable scope, which can be one line of code or several continuous lines of code.

In recent years, many studies have proposed automatic comment generation approaches based on **deep neural networks (DNNs)** due to their outstanding comment generation performance [15, 29, 54]. These approaches utilize **neural machine translation (NMT)** algorithms to "translate" source code into comments [53], though they have mainly been used in header comment generation by inputting semantics and structure information of method code (code covered by header comments) to the deep learning model. However, the performance of generating snippet comments is not as good as generating header comments when code snippets are input to the NMT approach [16]. DNN-based approaches essentially aim at a translation task, which needs an independent and complete unit as the input. As mentioned above, the header comments cover the entire method or the function, which has complete syntax and semantic information. By contrast, snippet comments only cover a part of the code snippet of the method in most cases. As a result, their semantic and syntactic information is fragmented and incomplete. Therefore, those previous NWT methods cannot achieve a good performance on snippet comment generation [19].

Compared with header comments, snippet comments contain not only relations with covered code snippets but also expanded information from the code context. Take Figure 1(a) as an example,[1] where line 13 shows a snippet comment written by the developer with the comment scope at line 14. It can be found that `data` and `written` are the keywords to represent the key meaning of comments, though they do not exist in the code snippet. If the scope of the code snippet is broadened to the whole method where the code snippet is located, then two key tokens, i.e., the `DatagramChannel` and `write` tokens can be located in lines 7 and 12, respectively. These two contextual statements (i.e., variable definition and usage statements) can be obtained through the data flow analysis of variable `dc`. Moreover, it is also necessary to determine the scope of the variable to determine the scope of the context. Take Figure 1(b) as another example, where line 54 shows the snippet comment and the comment scope is line 55. The code snippet does not contain the `seed` token from the comment. Through data flow analysis of variable `i`, the related statements in lines 15, 24, and 53 can be obtained although variable `i` in line 55 only works in the `for` loop structure from line 53 to line 56. Therefore, the context statement actually only includes line 53. The `seed` token in the comment can be obtained in this line. Considering that snippet comments are related to the context of code snippets, it is feasible to extract information from the context of code snippets by data flow analysis as the input to improve snippet comment generation.

It is another important issue to extract and capture **abstract syntax trees (ASTs)** from code snippets with context. AST is a tree representation of the source code structure, which includes semantic, lexical, and syntax information of code. Existing DNN-based comment generation approaches utilize ASTs as one of the inputs to capture this information to improve comment generation performance. ASTs of codes covered by header comments can be easily parsed, because the entire method forms a tree with complete structure information. However, code snippets and the corresponding context may have fragmented syntax and semantics information, which can be scattered in different subtrees or nodes of the AST. Therefore, how to parse and capture ASTs of code snippets with expanded context is a challenge.

In this article, we propose an approach, namely, **S**nippet **C**omment **Gen**eration **(*SCGen*)**. Specifically, we search the variable in the code snippet and utilize data flow analysis and AST analysis to find statements related to the variable definition and variable usage [50]. We parse the AST of a method (or a function) to extract the AST subtrees related to the code snippet and variable-related statements. After that, we utilize AST analysis information to recover the structure node with subtrees, thereby obtaining the snippet AST with expanded information from the context. Then, we traverse the type and value information of the AST separately to ensure that syntactic and semantic information can be captured at the same time. Next, we employ a model based on the encoder-decoder model taking AST sequences with context and source code snippet sequences as the input and then the model outputs comments. We obtain 600,243 snippet code-comment pairs from 959 Java projects to construct the dataset. We then conduct experiments to evaluate our *SCGen* on the dataset. In particular, We find that using data flow and AST analysis information from the context can improve the generation of snippet comments. The BLEU-4 result of the proposed model is 18.23, which is better than other state-of-the-art approaches. *SCGen* can play an important role in the lifecycle of fully automated comment generation, providing automatic snippet comment generation after the adoption of the automatic commenting location determination approach [17]. To facilitate research and application, we make *SCGen* and the dataset available at https://github.com/Anonymous123xx/SCGen.

[1]Codes of Figure 1 are from https://github.com/google/j2objc and https://github.com/google/guava

```java
public void testReadWrite_Block_WriterNotBound() throws Exception {
    byte[] sourceArray = new byte[CAPACITY_NORMAL];
    byte[] targetArray = new byte[CAPACITY_NORMAL];
    for (int i = 0; i < sourceArray.length; i++) {
        sourceArray[i] = (byte) i;
    }
    DatagramChannel dc = DatagramChannel.open();
    // The writer isn't bound, but is connected.
    dc.connect(channel1Address);
    // write
    ByteBuffer sourceBuf = ByteBuffer.wrap(sourceArray);
    assertEquals(CAPACITY_NORMAL, dc.write(sourceBuf));
    // Connect channel2 after data has been written.
    channel2.connect(dc.socket().getLocalSocketAddress());
    // read
    ByteBuffer targetBuf = ByteBuffer.wrap(targetArray);
    closeBlockedReaderChannel2(targetBuf);
    dc.close();
}
```

(a) DatagramChannelTest.java (project: j2objc)

```java
public void testRemovalNotification_clear_basher() throws InterruptedException {
    AtomicBoolean computationShouldWait = new AtomicBoolean();
    CountDownLatch computationLatch = new CountDownLatch(1);
    QueuingRemovalListener<String, String> listener = queuingRemovalListener();
    final LoadingCache<String, String> cache =
        CacheBuilder.newBuilder()
            .removalListener(listener)
            .concurrencyLevel(20)
            .build(new DelayingIdentityLoader<String>(computationShouldWait,
            computationLatch));
    int nThreads = 100;
    int nTasks = 1000;
    int nSeededEntries = 100;
    Set<String> expectedKeys = Sets.newHashSetWithExpectedSize(nTasks + nSeededEntries
    );
    // seed the map, so its segments have a count>0; otherwise, clear() won't visit
    the in-progress entries
    for (int i = 0; i < nSeededEntries; i++) {
      String s = "b" + i;
      cache.getUnchecked(s);
      expectedKeys.add(s);
    }
    computationShouldWait.set(true);
    final AtomicInteger computedCount = new AtomicInteger();
    ExecutorService threadPool = Executors.newFixedThreadPool(nThreads);
    final CountDownLatch tasksFinished = new CountDownLatch(nTasks);
    for (int i = 0; i < nTasks; i++) {
      final String s = "a" + i;
      @SuppressWarnings("unused") // go/futurereturn-lsc
      Future<?> possiblyIgnoredError =
          threadPool.submit(
              new Runnable() {
                @Override
                public void run() {
                  cache.getUnchecked(s);
                  computedCount.incrementAndGet();
                  tasksFinished.countDown();
                }
              });
      expectedKeys.add(s);
    }
    computationLatch.countDown();
    while (computedCount.get() < nThreads) {
      Thread.yield();
    }
    cache.invalidateAll();
    tasksFinished.await();
    Map<String, String> removalNotifications = Maps.newHashMap();
    for (RemovalNotification<String, String> notification : listener) {
      removalNotifications.put(notification.getKey(), notification.getValue());
      assertEquals(
          "Unexpected key/value pair passed to removalListener",
          notification.getKey(),
          notification.getValue());
    }
    for (int i = 0; i < nSeededEntries; i++) {
      // Get removal notification for the seed value.
      assertEquals("b" + i, removalNotifications.get("b" + i));
    }
    assertEquals(expectedKeys, Sets.union(cache.asMap().keySet(), removalNotifications
    .keySet()));
    assertTrue(Sets.intersection(cache.asMap().keySet(), removalNotifications.keySet
    ()).isEmpty());
  }
```

(b) CacheBuilderTest.java (project: guava)

Fig. 1. Examples showing context helpful in generating snippet comments.

The contributions of this work can be summarized as:

— We propose a snippet comment generation model called *SCGen* to generate snippet comments automatically. This approach utilizes not only the code snippet but also the context to expand code information and improve the comment generation performance.

— We present an AST construction approach for code snippets and the context. This approach utilizes data flow and AST analysis to extract variable-related information (i.e., variable definition and usage-related information) from the context and utilize AST structure information to construct snippet ASTs. We propose a unique way to traverse the type and value information of ASTs to capture syntax and semantics information.
— We construct a public snippet code-comment dataset that aims to train and test snippet comment generation models. It consists of 600,243 code-comment pairs from 959 Java projects.
— We compare our approach with other state-of-the-art approaches based on different evaluation metrics. The results show that the proposed approach outperforms other approaches. We find that the code data flow expansion contributes to performance improvement.

The rest of the article is shown as follows: Related work is introduced in Section 2. The details of the approach are provided in Section 3. Section 4 presents the experiment implementation, evaluation, and discussion. Section 5 points out the threats to validity, and Section 6 concludes the article and outlines future work.

## 2 RELATED WORK

### 2.1 Automatic Comment Generation

Most of the research has focused on the header comment generation. We first discuss the works related to the header comment generation and then survey the works related to snippet comment generation.

Algorithms of automatic header comment generation can mainly be classified into three types: template filling-based algorithm [38], information retrieval-based algorithm [55], and deep learning-based algorithm [26]. In the template filling-based algorithm, researchers [38, 39] often analyzed the signature, method body, and context information of the method to identify the behavior or role played by the method and then utilized approaches to generate comments in natural language formats based on the predefined template. For example, Sridhara et al. [46] utilized **Software Word Usage Model (SWUM)** to predefine some heuristic rules to identify keywords from code text and generate the template comments for Java methods. This approach could generate comments with a good format and sometimes could represent the function of method codes accurately. Besides, McBurney et al. [35, 36] proposed to collect method invocation information as the context of the target source code of Java methods. Then they used keywords to describe the context and utilized SWUM to identify the different parts of speech. At last, they adopted PageRank to select keywords as the template to generate Java method comments. However, in general, the current template filling-based algorithms have limited ability to summarize the code. On the one hand, designing the model to generate templates and rules costs a lot of manpower. On the other hand, the types of comments that can be only generated depend on the types of predefined templates.

The main idea of information retrieval is to search codes similar to the target code in the dataset or corpus to match, extract the comments corresponding to the matching code, and use these comments to generate the comments of the target code [33, 58]. Haiduc et al. [11] utilized **Latent Semantic Indexing (LSI)** to construct Java class comments. Based on this research, they proposed to use LSI and **Vector Space Model (VSM)** to improve summary generation performance [12]. Ying et al. [60] also utilized LSI to achieve code fragment summarization on the Web. They exploited syntactic features of the source code and whether a line is related to the given query. Fowkes et al. [9] proposed the TASSAL approach, which automatically generated the corresponding comment when folding non-core code. The core of the approach is to identify and retrieve the most relevant tokens for the code content based on VSM and the topic model. However, the

success rate of generating code comments by information retrieval is limited, because the retrieved similar codes that can be reused are very limited in the dataset.

With the rapid development of deep learning algorithms, many researchers have begun to use deep learning techniques to directly translate codes into comments. Iyer et al. [21] designed an approach to automatically generate comments for C# code fragments based on LSTM. Hu et al. [15] proposed an NMT-based Java method comment generation model that takes AST as the input to generate the comments. Based on this model, they [16] proposed an improved NMT model with two encoders and a decoder for source code and AST inputs. This approach also aims to achieve Java method comment generation. LeClair et al. [25] proposed a GRU-module-based encoder-decoder model that combines source codes and ASTs to generate code summaries. Besides, some deep learning-based approaches utilized information retrieval methods as an assistant. For example, Zhang et al. [62] retrieved the most similar code and similar AST from the training set and then input them and the target code snippet into the encoder-decoder model to improve comment generation performance. Wei et al. [55] proposed to retrieve the similar code and extract the comment of the similar code, the target code, and the AST as the model input to improve the performance of header comment generation. In addition, some researchers have proposed to expand the context of code for improving code summarization. Haque et al. [13] proposed to utilize other subroutine codes from the same file as the target subroutine combined with the target subroutine code and the AST to improve subroutine comment generation. This approach captures only the semantic information of the coarse-grained context.

Although the above methods achieve some excellent results in header annotation generation and some of them use contextual information, they are generally coarse-grained to take the context of other functions within the same file (e.g., directly take the source code information and API information). This will cause context redundancy and is not suitable for snippet comment generation. Different from the approaches above, we take variable-related AST nodes outside the code snippet as one of the inputs of the deep learning network to attain not only semantics but also syntax information of the fine-grained context. We also control the structure size to prevent contextual redundancy. This approach is suitable to achieve snippet comment generation.

There is a small amount of research focusing on snippet comment generation. Feng et al. [8] proposed a pre-trained model called *CodeBERT* for programming and natural language transfer based on the transformer. This model can be fine-tuned to achieve the source code-to-natural language task including snippet comment generation. Huang et al. [17] proposed an automatic approach to identify the location of adding the block (i.e., snippet) comment by adopting context information. They utilized machine learning techniques to predict the comment locations. The result showed the feasibility and effectiveness of the proposed approach. Sridhara et al. [47] proposed a heuristics approach to achieve specific snippet comment generation to describe high-level abstract algorithmic actions. Huang et al. [19] proposed a reinforcement learning-based approach to generate snippet comments. This approach utilized AST as the input. Existing snippet comment generation methods only considered the target snippet code segment itself as the research object. They lack the mining of snippet code structure information from the context. Therefore, we put forward to use the snippet AST and the context information of the code snippet to improve the performance of snippet comment generation.

### 2.2 Empirical Study of Code Comments

There was some research working on the analysis of code comments. For example, Huang et al. [20] proposed a statistical analysis of header comments and snippet comments and proposed an automatic approach to detect whether a method code needed a header comment automatically. Developers could add comments to method codes with the help of this approach. Wen et al. [56]

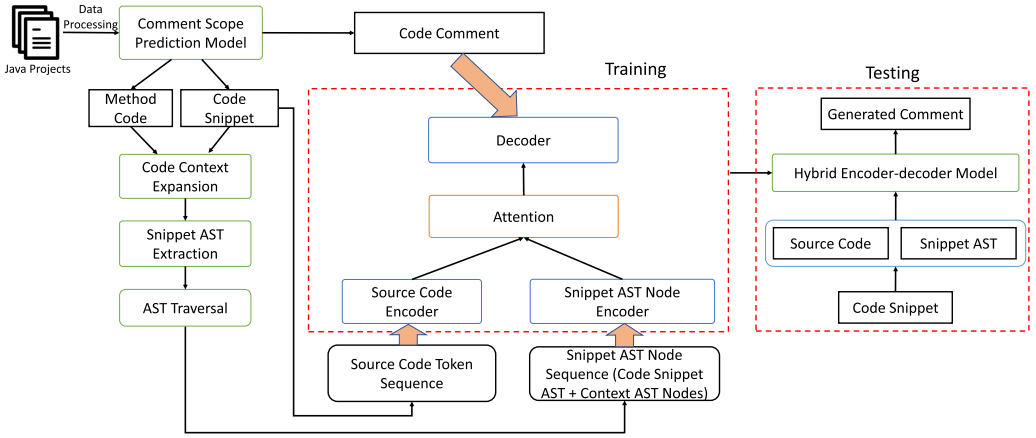



Fig. 2. The overview of *SCGen*.

analyzed inconsistencies in code-comment. They investigated which commit type might cause comment updates and indicated that these findings can guide fixing code-comment inconsistencies. McBurney et al. [37] conducted an empirical study to evaluate method comments written by developers, readers, and automatic comment generation approaches. Their result showed that the quality of human-written comments could be evaluated by the textual similarity of source codes. It indicated that good comments should have a high semantic similarity to the source code. Steidl et al. [49] proposed a comment quality evaluation approach based on machine learning. They proposed four criteria to evaluate the comment quality including consistency between codes and comments.

Some researchers worked on the analysis of the header comment and snippet comment density. Oman et al. [41] evaluated the comment quality by calculating the proportion of code comments. Arafat et al. [2, 3] counted the comment density for the open-source projects that are well maintained and the result was 18.67%. They also concluded that commenting on source code is a consistently followed practice of successful open-source projects.

Some researchers worked on evaluating code comment generation performance. For instance, Stapleton et al. [48] made a comparison between human-written comments and model-generated comments. The findings indicated that developer participants were able to complete the programming better with the help of human-written comments, although they did not realize the difference between human-written comments and model-generated comments.

## 3 APPROACH

This section elaborates on the detailed design of *SCGen*. Figure 2 depicts the overview of *SCGen*. First, we extract method codes and filter out invalid data (e.g., template comments). We utilize a comment scope prediction approach [6] to extract pairs of code snippets and comments. The method code where the code snippet is located and the code snippet are used to generate comments. We parse the AST of the method where the code snippet is located and adopt data flow analysis based on the control flow graph to extract variable-related information and ensure the context scope with AST analysis information. This variable-related information and context information consists of scattered AST subtrees and nodes. We also utilize AST structure analysis information to obtain structure nodes to connect scattered subtrees and nodes and then construct the snippet AST. After that, we traverse the type and value information of the snippet AST to create the AST type

sequence and the AST value sequence. The source code snippet is also traversed into the sequence. In the model training phase, the snippet AST sequence (i.e., type sequence and value sequence) and the source code sequence are input to an NMT model called the hybrid encoder-decoder model. By exploiting snippet comments in the dataset as ground truth, we train a model to generate snippet comments automatically. In the testing phase, the code snippet to be commented is parsed into two kinds of token sequences (i.e., source code token sequences and snippet AST token sequences containing type sequences and value sequences) and both of them are fed into the trained model and then the comment is generated.

### 3.1 Data Processing

We download Java projects from the open-source code repository. Then, we utilize regular expression matching to extract method codes in the project. After that, we filter out those comments generated according to templates in the dataset, such as comments in *setter* and *getter* methods or comments generated by the template predefined in the IDE comment plugin. Comments on these types of code are trivial for training the model. We filter out these codes and comments to eliminate the impact of these template comments on model training.

Next, we extract snippet comments from the method codes. We employ a comment scope detection approach proposed by Chen et al. [6] to extract the code snippet and comment pairs. This approach adopts code features (e.g., statement types, No. of sub-statements, No. of layers of nested statements, No. of lines in the statement and same method calls) and comment features (e.g., length of the comment, No. of verbs, No. of nouns) for training a supervised machine learning model to determine the code range covered by each snippet comment. We utilize this approach to detect the comment scope automatically and then randomly sample data to validate the accuracy of this approach. Specifically, we utilize the trained comment scope detection model proposed by the reference to detect the comment scope and then we take a random sample of 1,500 entries in the validation dataset and 500 entries in the test dataset to manually validate the accuracy of the comment scope detection. The accuracy is 80.20% and 80.67%, respectively, which are similar to the result proposed in the reference (81.45%). We also correct the samples that are detected incorrectly by the comment scope detection model and utilize corrected data as the input of the comment generation model to generate snippet comments. We find that comment generation results have no significant difference from *SCGen*'s result. The detailed result is shown in Section 4.5.1. Therefore, by using this comment scope detection approach, snippet code-comment pairs are attained. Moreover, we also retain the method code where the code snippet is located to capture its context.

### 3.2 Code Context Expansion and Snippet AST Construction

Figure 1 showcases the importance of extracting the context to supplement the missing tokens for comment generation. To improve the performance of snippet comment generation, we propose to introduce the context of the code snippets. The source code of the whole project constructs the context of the code snippet. However, not all the contexts are related to the code snippets. In object-oriented programming, methods are the main part of basements for expressing algorithmic intentions. In fact, it is the method that is the foundation of program behavior. Therefore, the whole method where the code snippet is located is viewed as the context scope. Considering that not all code in the context method is related to the code snippet for comment generation, we also use the information from AST analysis and data flow to extract the context code related to the target code snippet. The unrelated context code is deleted, because it may not contribute to the comment generation.

Recently, several approaches adopt control flow and data flow information to split the method AST, thereby obtaining code structure and hierarchy information to generate header comments

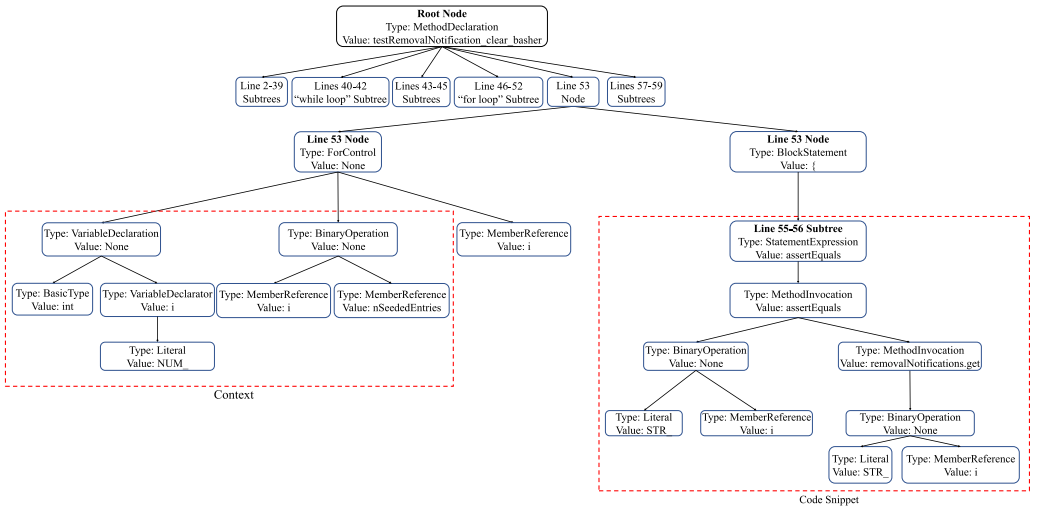



Fig. 3. The abstract AST of the source code example in Figure 1(b).

[31]. In our study, we use data flow and AST analysis information to extract contextual nodes related to the code snippet from the AST and construct the snippet AST. We first extract the ASTs from method code using a Python module javalang.[2] Each node in the AST of a method has features information including type, value, and children nodes. The corresponding line number in the source code is also recorded for each node.

We extract the AST of the code snippet and its corresponding context. The details are shown in Algorithm 1. First, we input the method AST and the start line and end line of the code snippet to find all the code snippet-related AST subtrees. The AST subtrees consist of AST nodes corresponding to the code snippet. For each AST node in code snippet-related subtrees, if it is a variable-related node, then we utilize *find_context* method to get the variable information and adopt data flow information to find the variable definition and variable usage-related AST nodes in the method AST. The data flow information is attained by data flow analysis based on control flow information (e.g., control flow graph). In particular, for some variables defined in a loop structure or selective structure, we use the nearest relevant structure nodes in the AST (e.g., *if*, *for*, and blocks enclosed with "{"and "}") as the working scope for these variables before adding the variable-related nodes to the context. That is the working process of method *find_variable_scope*. We utilize this method before finding context. For example, in Figure 1(b), we find the variable *i* only works in the loop structure from line 53 to line 56. So, we only search for variable-related information in this scope. We search variable-related information in line 53.

For code snippets and the corresponding context, their semantics and structure information are fragmented, which may consist of several subtrees or nodes in method ASTs. As shown in Figure 3, in the abstract AST of the source code example in Figure 1(b), the subtree and nodes corresponding to the code snippet and the context are marked with the red box. The code snippet consists of one subtree and the context consists of two subtrees. How to integrate the scattered subtrees and nodes into a tree while keeping the information of subtrees complete and concise is a challenge. We utilize the method AST structure information to construct the tree (i.e., snippet AST). Specifically, after we get the subtrees and AST nodes of the code snippet and the context, we search

[2]https://github.com/c2nes/javalang

**Algorithm 1: Extract Snippet AST**

**Input**: $M$: The set of node statements of the method AST;
$s$: Start line number of the code snippet;
$e$: End line number of the code snippet;
$l$: AST length threshold;
**Output**: $B$: The set of node statements of the snippet AST;
**Begin**

```
function Extract_snippet_AST(M,s,e)
  context ← ∅
  B ← ∅
  // Search subtrees of the code snippet;
  subtree_set = find_subtree(M,s,e)
  For st in subtree_set do:
    For node in st do:
      If (node.type == variable) then:
        variable_scope = find_variable_scope(node.value)
        context ← find_context(method_ast,node.value,
                  variable_scope)
      End If
    End For
  End For
  B ← subtree_set
  // Find the least common ancestor node without context;
  root_source = LCA(M,B)
  If (len(B) >l) then:
    B ← root_source
    Return B;
  End If
  B ← context
  // Find the least common ancestor with context;
  root = LCA(M,B)
  If (len(B) <l) then:
    B ← root
    Return B;
  Else:
    While (len(B) >l):
      max_distance = 0
      remove_node = null
      For node in context do:
        If (distance(node,subtree_set) >max_distance) then:
          max_distance = distance(node,subtree_set)
          remove_node = node
        End If
      End For
      B = B−remove_node
    End While
    B ← root
    Return B;
  End If
end function
End
```

the **least common ancestor (LCA)** nodes for subtrees and nodes in the method AST according to the tree structure. The purpose is to obtain a structural node in the method AST to connect the scattered subtrees and nodes corresponding to the code snippet and contextual information and then construct a snippet AST.

In particular, adding context nodes and structural nodes will extend the scale of snippet AST. If the scale is too large, then the traversed result of the AST will be so long that it has to be truncated by the comment generation model. This may cause AST key information to be lost. To add relevant contextual information while ensuring that as much information as possible related to the code snippet itself is not lost in the AST, we design a set of rules to control the scale of extending context. Specifically, we set 100 AST nodes as a threshold first. The reason why we set 100 nodes is that we set the truncation length of the model as 400. This length can cover 99.55% of the dataset, that is, 99.55% of the samples in the dataset have a sequence length of less than 400. If the length of the AST subtree set without expanded nodes (i.e., the node number of the AST subtree) of the code snippet exceeds this threshold, then we directly adopt the AST subtree set of the code snippet and utilize the structure node directly to construct snippet ASTs without adding any context node for expansion. For the expanded AST with more than the length threshold, we will remove the farthest node according to the distance between the expanded node and the AST subtree set of code snippets, until the AST length is less than the threshold. The distance of expanded node *EN* and AST subtree set of code snippet is calculated as follows:

$$\text{distance}(EN, \text{subtreesset}) = \sum_{i=1}^{N} \text{path}(EN, \text{node}_i), \tag{1}$$

where $\text{node}_i$ refers to the node in the AST subtree set, $N$ is the amount of AST subtree nodes, and path is the length of the AST path between the expanded node and the AST subtree node, which is calculated as follows:

$$\text{path}(EN, \text{node}) = \text{length}(EN, LCA) + \text{length}(\text{node}, LCA), \tag{2}$$

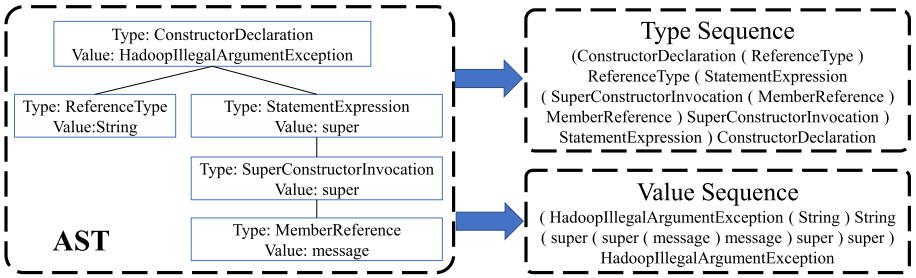
where *LCA* is the least common ancestor node of the expanded node and the AST subtree node. Then, we construct the snippet AST. By constructing the snippet AST, we achieve code context expansion on the AST level based on data flow and AST analysis information.

### 3.3 AST Traversal Algorithm

Because the AST has a tree structure, it can be traversed into a sequence to obtain features and then be input into the comment generation model. Some AST traversal algorithms are proposed to convert the AST to a sequence. For example, Hu et al. [15] proposed an algorithm called **Structure-based Traversal (SBT)** to traverse AST. Sequences obtained by classical traversal methods (e.g., pre-order traversal) are lossy, since the original ASTs cannot unambiguously be reconstructed back from them and the ambiguity may cause different Java methods (each with different comments) to be mapped to the same sequence. SBT utilizes brackets to generate construction information to solve the lossy problem of pre-order traversal. Specifically, as for a root node, a pair of brackets is used to represent the tree structure and put the root node itself behind the right bracket, i.e., `(root node)root node`. Then, the subtrees of the root node are traversed in pre-order and all root nodes of subtrees are put into the brackets. Recursively, each subtree is traversed until all nodes are traversed and the final sequence is obtained. For non-leaf nodes, only type information is kept in the sequence, and for leaf nodes, both type and value information are included in the sequence and linked with underscores. However, this algorithm loses some semantics information, because it discards the value information of non-leaf nodes to control the sequence length of AST. Moreover, many brackets and underscore symbols are added to the sequence. Therefore, the model can only truncate the long sequence, which causes the lost information of the AST sequence. Because the information included in the code snippet is limited, it is important to control the length of the sequence while maintaining the integrity of the AST sequence information (i.e., type and value information of all nodes).

```
1  public HadoopIllegalArgumentException(final String message) {
2      super(message);
3  }
```

(a) Source code




(b) AST Traversal Result

Fig. 4. An example of traversing an AST to two kinds of sequences.

To address this problem, we propose a new traversal algorithm based on the SBT algorithm called **Multi-based Traversal (MBT)**. The proposed MBT works as follows: According to the parsing results of snippet AST, each AST node has features including type and value. Type information declares the statement type of the node, such as *MethodDeclaration* and *FormalParameter*. Value refers to the specific token in the code snippet, which can represent the semantics information of the code snippet. We consider the example as illustrated in Figure 3, where the type of the root node is *MethodDeclaration*, while the value of this node is the method name *testRemovalNotification_clear_basher*. We separate the type feature and the value feature. We then use the same traversal order and the same way of adding parentheses as SBT to convert two kinds of features into two sequences. In this way, our method well retains complete semantic and structural information of AST. At last, these two sequences will be combined into a list after embedding later. Compared with SBT, we transfer AST to two sequences (i.e., type sequence and value sequence) rather than one. We also combine them into one sequence after embedding. Thereby, we control the length of sequences and capture both the type sequences and value sequences of all nodes, so we keep the completeness of the AST information. The traversed result of an illustrative example is presented in Figure 4. In this example, the value *super* in the *StatementExpression* node means that the name of the function called directly in line 2 is called "super." The value *super* in *SuperConstructionInvocation* means that the name of the parent constructor is called "super." So, we can attain inheritance from the value information of AST. By executing MBT, both the type sequence and the value sequence are generated.

### 3.4 Hybrid Encoder-decoder Model

The detailed structure of the hybrid encoder-decoder model is shown in Figure 5. There are two kinds of inputs in this approach: (i) the source code snippet sequence and (ii) the AST token sequence (including type sequence and value sequence) of the code snippet and the expanded context. We employ an NMT model called the hybrid encoder-decoder model. It is a sequence-to-sequence-based model [4], which is composed of two encoders and a decoder. The two encoders can encode these two kinds of sequences and convert inputs into two vector sequences. Then, the decoder decodes these vectors and restores them to another sequence, i.e., the output sequence.

Each sequence can be represented as $x = [x_1, x_2, \ldots x_L]^T$ and $L$ is the sequence length. Each token in these two sequences will be converted into a word vector through the word embedding

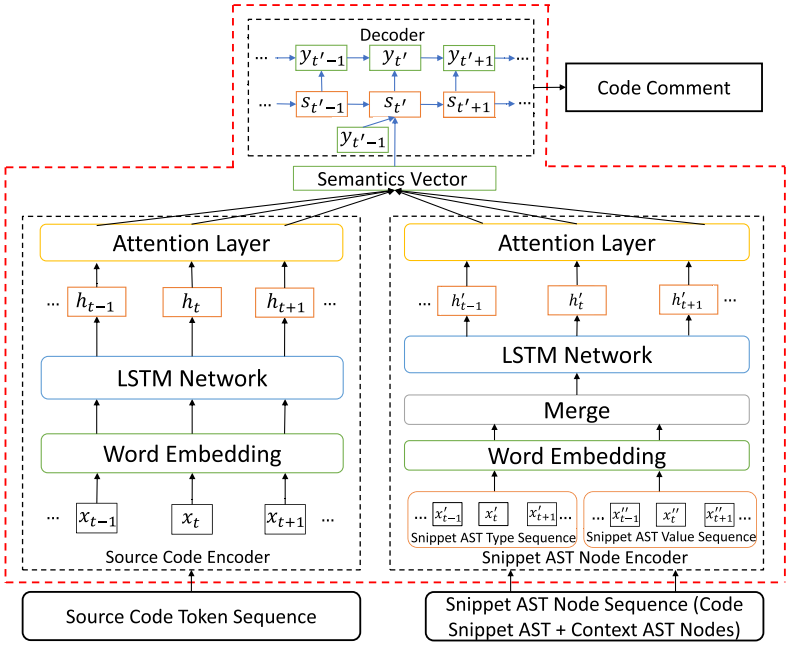



Fig. 5. The structure of hybrid encoder-decoder model.

layer. Specifically, as for the source code snippet, each source code token is embedded in a vector $v_s$. The process is represented as follows:

$$v_s = W_s x_t, \tag{3}$$

where $W_s$ is the embedding matrix, which is a trainable parameter, and $x_t$ is the source code token at each time step $t$. As for AST, there are type and value sequences. We then embed tokens from these two sequences, denoted by $v_t$ and $v_v$ as follows:

$$v_t = W_t x_t^{'}, \tag{4}$$

$$v_v = W_v x_t^{''}, \tag{5}$$

where $v_t$ and $v_v$ are the type vector and value vector, respectively. Terms $W_t$ and $W_v$ are the embedding matrices, and $x_t^{'}$ and $x_t^{''}$ are the AST type token and AST value token, respectively. After that, we merge them to integrate these two features. Finally, the vector of AST tokens is obtained as follows:

$$v_a = v_t + v_v. \tag{6}$$

In the encoders, both of the vector sequences are input to a **Long Short-Term Memory (LSTM)** network [57, 59, 64]. Compared with **recursive neural network (RNN)** [63], the gate structure of LSTM can effectively alleviate the gradient vanishing or explosion problems that may occur in long sequence input. At the same time, compared with transformer [52], LSTM has the advantage of being lightweighted and having less computing cost. By utilizing LSTM, hidden states of the two sequences are obtained. In particular, hidden states at time $t$ denoted by $h_t$ and $h_t^{'}$ are expressed by

$$h_t = \text{lstm}(h_{t-1}, v_s), \tag{7}$$

and

$$h_t^{'} = \text{lstm}(h_{t-1}^{'}, v_a). \tag{8}$$

To improve the performance of the hybrid encoder-decoder model, we utilize the attention mechanism [4, 43] to assign weights to hidden states. There are two sets of hidden state weight distributions, because we employ two attention layers in the two encoders. The semantics vector is the weighted sum of these two sets of hidden states, shown as follows:

$$c_{t'} = \sum_{t=1}^{T} \alpha_{t't} h_t + \sum_{t=1}^{T'} \alpha'_{t't} h'_t, \tag{9}$$

where $T$ and $T'$ are lengths of source code token sequence and snippet AST token sequence, respectively. Terms $\alpha_{t't}$ and $\alpha'_{t't}$ are weight distributions of source code tokens and snippet AST tokens, respectively. They can be calculated as follows:

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^{T} \exp(e_{t'k})}, \tag{10}$$

$$\alpha'_{t't} = \frac{\exp(e'_{t't})}{\sum_{k=1}^{T'} \exp(e'_{t'k})}. \tag{11}$$

In Equations (10) and (11), $e_{t't}$ and $e'_{t't}$ can be calculated by

$$e_{t't} = a(s_{t'-1}, h_t), \tag{12}$$

$$e'_{t't} = a(s_{t'-1}, h'_t), \tag{13}$$

where $a$ is the alignment model aiming to evaluate the correlation between the input word at timestep $t$ and the output word at timestep $t'$.

In the decoder, we aim to generate comment tokens $y_{t'}$ based on context vector $c_{t'}$ and previously generated comment tokens:

$$P(y_{t'}|y_1, y_2, \ldots, y_{t'-1}, c_{t'}) = p(y_{t'-1}, s_{t'}, c_{t'}), \tag{14}$$

where $s_{t'}$ is the hidden state. The object function is shown as follows:

$$H(y) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{t'=1}^{T'} \log p(y^i_{t'}), \tag{15}$$

where $n$ is the total number of training set samples and $y^i_{t'}$ represents the predicted word at timestep $t'$ in the $i$th sample. By minimizing object function by gradient descent [28], we can obtain the model parameters. In addition, the hybrid comment generation model utilizes beam search [23], a heuristic graph search algorithm to find the comments with the minimum value computed by the objective function (15). Compared with the greedy search algorithm, beam search can make the comment generation results closer to the global optimum.

## 4 EXPERIMENTS AND EVALUATION

In this section, we conduct experiments to evaluate the performance of *SCGen* in snippet comment generation and analyze the influencing factors of *SCGen* performance. We are concerned with the following research questions:

**RQ1:** What is the performance of *SCGen* compared with other comment generation approaches?

**RQ2:** What are the effects of different context expansion levels on the performance of snippet comment generation?

**RQ3:** What are the effects of different modules on the performance of snippet comment generation?

**RQ4:** What are the effects of preceding code expansion and following code expansion on the performance?

**RQ5:** How does the length of the code affect the performance of snippet comment generation?

### 4.1 Data Preparation

The goal of *SCGen* is to achieve automatic comment generation for the code snippet. The target programming language of our approach is Java. Although there are some datasets focusing on automatic comment generation, existing datasets are collected and constructed mainly for header comment generation. There is a lack of general datasets for snippet comment generation. Therefore, we construct a dataset of Java from GitHub. We download the top 1,000 Java projects based on the evaluation score (e.g., Star) provided by GitHub and remove the projects without comments. Then, we obtain a dataset with 1,949,290 comments from 959 projects. We extract method codes and execute data cleaning to filter out invalid data (e.g., template comments), as mentioned in Section 3.1. Then, we remove the header comments. Thereafter, we leave 600,243 snippet comments and the method codes, where the snippet comments are located.

After that, we utilize the comment scope prediction model to find out the corresponding code snippets of snippet comments to build code-comment pairs. As for each code plain text and each comment, we also conduct *CamelCase* splitting, *snake_case* splitting to reduce OOV tokens [12]. For example, suppose the vocabulary we set only has words *get* and *index* but does not have the word *getindex*. In that case, we can only use an unknown symbol (e.g., *<UNK>*) to represent the token *getIndex*. But *getIndex* can be represented by using the words in the vocabulary if we use *CamelCase* splitting to split *getIndex* to *get* and *index*. Hence, it is not an OOV token. The process of *snake_case* is similar. We also conduct these two kinds of token-splitting methods in AST value sequences. We provide more details about how the token-splitting method decreases the source code vocabulary size in Section 4.5.2 to demonstrate the effectiveness of the function of token splitting.

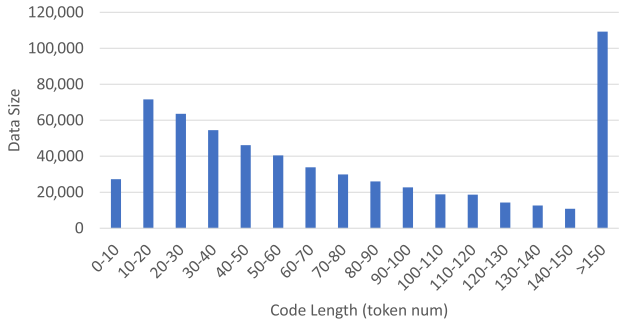
Figure 6 plots the length distributions of the code snippet and comments. The code lengths are mainly scattered in the range of 10–20 tokens and longer than 150 tokens. Except for lengths shorter than 10 tokens and longer than 150 tokens, the data size decreases with the increased code length. Moreover, most comments are less than 5 tokens. It demonstrates that most snippet comments are short. We split the dataset into three partitions based on cross-project: (i) 80% for training data, (ii) 10% for validation data, and (iii) 10% for testing data. All snippet code-comment pairs in one project are grouped into one category. The purpose is to prevent the training data from leaking duplicate information to the validation dataset and the test dataset.

### 4.2 Experiment Configuration

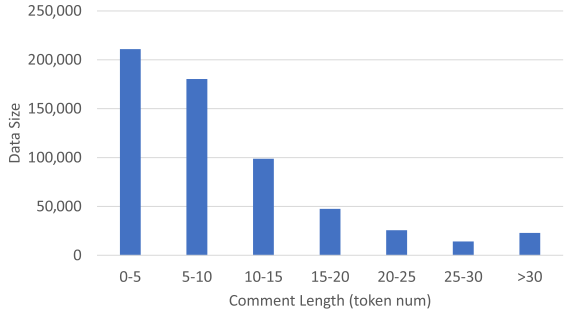
The model hyperparameters are configured as follows: We set the vocabulary sizes as 50,000, 1,000, 50,000, and 50,000 for code, AST type, AST value, and comment, respectively. The batch size is configured as 32, and the maximum number of epochs is set as 120. The model embedding size is 512, and the truncation length is set as 400. As for the optimizer, we utilize Adam [22] with the learning rate $10^{-4}$. The dropout rate is set as 0.25. All experiments are conducted on two servers. One is configured with two GPUs of NVIDIA Tesla V100 and the other is configured with two GPUs of NVIDIA GeForce RTX 2080 Ti.

### 4.3 Effectiveness Evaluation Metrics

In this article, we utilize **BLEU (Bilingual Evaluation Understudy)** [42], METEOR [5], and ROUGE-L [32] to evaluate the performance of our approach.




(a) Code snippet length distribution




(b) Snippet comment length distribution

Fig. 6. Length distribution of the data.

*4.3.1 BLEU.* BLEU is a popular evaluation metric that is implemented in neural machine translation and conversation systems [42]. Its purpose is to evaluate the difference between the sentences generated by the model (i.e., candidate sentence) and a set of ground truth sentences (i.e., reference sentence). BLEU uses $n$-gram for matching and calculates the ratio of $n$ groups of word similarity between generated sentences and reference sentences. The utilization of BLEU has some advantages such as low computational cost and it is easy to understand. The specific calculation process of BLEU is given as follows:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{m} \omega_n \log p_n\right), \tag{16}$$

where $\omega_n$ is the weight of $n$-gram and $p_n$ is the precision of $n$-gram. Usually, the maximum value of $n$ is 4, which is represented by BLEU-4. BP is the brevity penalty factor for generated comment length, which is shown as follows:

$$\text{BP} = \begin{cases} 1 & , \quad l_c > l_r, \\ \exp\left(1 - \dfrac{l_r}{l_c}\right), & l_c \leq l_r, \end{cases} \tag{17}$$

where $l_c$ is the length of generated comment and $l_r$ represents the reference comment. In this article, we employ **BLEU from 1-gram (BLEU-1)** to **4-grams (BLEU-4)**.

*4.3.2 Meteor.* METEOR was proposed by Banerjee and Lavir [5] after discovering the significance of recall in the evaluation metrics. Based on single-precision weighted harmonic mean and single-word recall rate, METEOR aims to solve some inherent defects in the BLEU standard.

METEOR is calculated as the harmonic average of the accuracy and recall between the candidate sentence and the reference sentence:

$$\text{METEOR} = (1 - \text{Pen})F_{\text{mean}}, \tag{18}$$

where Pen is the penalty parameter and $F_{\text{mean}}$ is the harmonic average of precision and recall. Pen is calculated as follows:

$$\text{Pen} = \gamma \left(\frac{\text{ch}}{m}\right)^{\theta}. \tag{19}$$

$F_{\text{mean}}$ is calculated as follows:

$$F_{\text{mean}} = \frac{P_m R_m}{\alpha P_m + (1-\alpha)R_m}, \tag{20}$$

where $\alpha$, $\gamma$, and $\theta$ are the default parameters for evaluation. The term ch is the number of tokens and $m$ represents match tokens. $P_m$ represents unigram precision, i.e., the ratio of match tokens and candidate sentence tokens. $R_m$ represents unigram recall, i.e., the ratio of match tokens and reference sentence tokens.

*4.3.3 ROUGE-L.* ROUGE-L [32] is a similarity measurement method based on recall rate. It mainly examines the adequacy and authenticity of generated sentences. It calculates the co-occurrence probability of the **longest common subsequence (LCS)** in the reference sentence and the candidate sentence. ROUGE-L can be calculated as follows:

$$R_{lcs} = \frac{LCS(X,Y)}{m}, \tag{21}$$

$$P_{lcs} = \frac{LCS(X,Y)}{n}, \tag{22}$$

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}, \tag{23}$$

where $LCS(X,Y)$ is the length of the longest common subsequence of reference sentence $X$ and candidate sentence $Y$. $R_{lcs}$ and $P_{lcs}$ represent recall rate and precise rate, respectively. The term $F_{lcs}$ is the ROUGE-L value. Usually, $\beta$ is set to a very large number, so ROUGE-L almost only considers $R_{lcs}$, i.e., the recall rate.

### 4.4 Results

*RQ1: What is the performance of SCGen compared with other comment generation approaches?*

To verify the effectiveness of *SCGen*, we compare it with seven state-of-art approaches, namely, *Rencos* [62], *CodeBERT* [8], *Code-NN* [21], *DeepCom* [15], *Hybrid DeepCom* [16], *ast-attendgru* [25], and *RLcom* [19]. *Rencos* is a code summarization generation model based on information retrieval and the encoder-decoder model. The model takes a similar source code and retrieved similar AST as input to assist the comment generation of the target code. *CodeBERT* is a pre-trained model for programming and natural language based on the transformer. This model can be fine-tuned to achieve the source code-to-natural language task. *Code-NN* is an LSTM-based comment generation approach for C# language. The source code sequences and the comment token sequences are the input and the output, respectively. We also employ it as one of the comparative approaches in Java comment generation. *DeepCom* was a sequence-to-sequence-based approach that aims to generate header comments. It utilizes SBT to traverse AST to generate the model input. The sequence-to-sequence approach translates the AST sequences to the code comments. *Hybrid DeepCom* is a deformation based on *DeepCom*. It combines method source codes and ASTs as the input and then generates the corresponding comments. Similarly, *ast-attendgru* also combines source codes

Table 1. Evaluation Results for the Competing Approaches

| Approaches | BLEU | | | | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | | |
| ***Rencos*** | 17.88 | 14.28 | 13.26 | 12.80 | 10.70 | 11.81 |
| ***CodeBERT*** | 16.09 | 12.47 | 11.29 | 10.94 | 15.33 | 25.59 |
| ***Code-NN*** | 15.72 | 12.56 | 11.70 | 11.36 | 11.90 | 17.13 |
| ***DeepCom*** | 17.02 | 13.99 | 13.10 | 12.73 | 13.26 | 18.18 |
| ***Hybrid DeepCom*** | 18.16 | 15.00 | 14.08 | 13.67 | 14.43 | 19.57 |
| ***ast-attendgru*** | 17.16 | 13.98 | 13.04 | 12.64 | 13.41 | 18.48 |
| ***RLcom*** | 17.24 | 12.24 | 11.15 | 11.19 | 10.35 | 16.38 |
| ***Our SCGen*** | **22.36** (23.13%) | **19.50** (30.00%) | **18.65** (32.46%) | **18.23** (33.36%) | **18.83** (22.83%) | **23.65** (−7.58%) |

The percentage in parentheses is the growth rate of *SCGen* compared to the best baseline.

and ASTs to generate code summaries, though it is based on GRU module-based encoder-decoder model. Moreover, we also adopt *RLcom*, a snippet comment generation model based on reinforcement learning and the encoder-decoder model.

To make a fair model training, we train these seven baseline approaches with the same dataset as *SCGen*, which is proposed in Section 3.1. Table 1 shows the model performance of all the approaches. It can be found that our approach achieves superior performance than other methods in most of the metrics. In particular, the BLEU-4 of *SCGen* is 42.42% higher than that of *Rencos*. As for METEOR and ROUGE-L, *SCGen* outperforms 75.98% and 100.25% than *Rencos*. The results of *SCGen* are all statistically different from *Rencos* (all $p$-values are less than 0.01 in Wilcoxon signed-rank test). The input of *Rencos* is the source code and its similar AST and source code in the training set. As for the comment generation of methods, it is easy to search for method codes with similar semantics and syntax. But it is difficult to search similar codes for code snippets because of the short length of code snippets. It is difficult to catch complete semantics and syntax to achieve information retrieval. Therefore, information retrieval-based approaches like *Rencos* do not perform well on snippet comment generation.

*SCGen* also outperforms *CodeBERT*, *Code-NN*, *DeepCom,* and *RLcom* in BLEU-4, METEOR, and ROUGE-L. The input of *CodeBERT* and *Code-NN* is the source code only, while the input of *DeepCom* and *RLcom* is the AST. *SCGen* achieves 66.64%, 60.48%, 43.21%, and 62.91% higher BLEU-4 scores than *CodeBERT*, *Code-NN*, *DeepCom,* and *RLcom*, respectively, with statistical significance (i.e., $p$-values are less than 0.01). Moreover, *SCGen* has 22.83%, 58.24%, 42.01%, and 81.93% higher METEOR scores than these four approaches with statistical significance ($p$-values are less than 0.01). *SCGen* also performs 38.06%, 30.09%, and 44.38% better than *Code-NN*, *DeepCom,* and *RLcom*, respectively, with statistical significance ($p$-values are less than 0.01), though it is 7.58% slightly worse than *CodeBERT* in ROUGE-L. It is because many generated comments from *CodeBERT* are short. One piece of evidence is that the average number of tokens for comments generated by *SCGen* is 9.88, while the average number of tokens for comments generated by *CodeBERT* is 3.15. Therefore, it has a high recall rate and a better performance in ROUGE-L. In conclusion, it indicates that combining code and AST information contributes to comment generation.

*SCGen* also performs better than *Hybrid DeepCom*, *ast-attendgru* by 33.36% and 44.22% on BLEU-4. Moreover, it outperforms these two approaches by 30.49% and 40.42% in terms of METEOR, and 20.85% and 27.98% in terms of ROUGE-L. All the results have statistical significance, too ($p$-values are less than 0.01). All these three approaches take code and AST as input, though *Hybrid DeepCom* and *ast-attendgru* do not use context information. Thus, the results indicate that adding context (i.e., expanding code information) can improve the performance of snippet comment generation. Furthermore, because *Hybrid DeepCom* only combines the type information of the AST with the code snippet, utilizing both type and value information can improve the performance of snippet comment generation. In particular, the BLEU-4 result of *Hybrid DeepCom* in this article is 13.67, which is much lower than the reference [16]. The reason is that both these two approaches are

Table 2. The Performance of Different Context Expansion Levels

| Context Expansion Level | BLEU | | | | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | | |
| 1st **Level** | **22.36** | **19.50** | **18.65** | **18.23** | **18.83** | **23.65** |
| 2nd **Level** | 21.54 | 18.57 | 17.66 | 17.24 (**$p < 0.01$) | 17.89 | 23.09 |
| 3rd **Level** | 21.85 | 18.86 | 17.95 | 17.50 (**$p < 0.01$) | 18.17 | 23.12 |
| 4th **Level** | 21.60 | 18.66 | 17.76 | 17.32 (**$p < 0.01$) | 18.12 | 22.94 |
| ***Infinite Level*** | 21.60 | 18.63 | 17.74 | 17.31 (**$p < 0.01$) | 17.91 | 22.98 |

aimed at header comment generation but not snippet comment generation. In addition, the datasets in references are not split based on cross-project, thereby achieving better results. This is also the reason why the BLEU-4 result of *RLcom* is lower than the reference [19].

*RQ2: What are the effects of different context expansion levels on the performance of snippet comment generation?*

To answer RQ2, we investigate the impact of the level of context expansion on comment generation. In other words, we aim to answer whether the more level of context expansion, the better the performance of comment generation, or whether there is a threshold for context expansion. As mentioned above, the snippet comment is related to the statements in the context, which involve variables in the code snippet. We call it the 1st level context expansion. In these variable-related statements, there may exist some usage of other variables except those related to code snippets. According to data flow, it may cause another layer of context expansion, which we call the 2nd level context expansion. For example, referring to Case 1 to be illustrated in Table 6, line 12 is the code to be commented, and there exists usage of variables `numItemsLeft`, `contiguousCount`, and `numToRemove`. Lines 4, 5, 8, 10, 14, 17, and 20 are the 1st level context. In line 14, there is a variable `floorOfEachRun` that does not exist in the code snippet. The context using `floorOfEachRun` is included in the 2nd level context (i.e., lines 21). Similarly, if there exists the usage of new variables in 2nd level context expansion, it may cause the 3rd level context expansion. The 4th level will also exist if there are usages of new variables in 3rd level context expansion. Besides, we treat the whole method where the code snippet is located as the infinite-level context expansion. We want to analyze whether different kinds of context expansion levels have an impact on the result of comment generation. Table 2 shows the performance of different context expansion levels. It can be shown that the BLEU score, METEOR score, and ROUGE-L of the 1st level are all the highest. The results are 4.17%~5.74%, 3.63%~5.25%, and 2.29%~2.43% better than the 2nd level, the 3rd level, and the 4th level on BLEU-4, METEOR, and ROUGE-L. A Wilcoxon signed-rank test also confirms that the 1st level indeed has higher BLEU-4 scores than the 2nd level, 3rd level, and the 4th level with statistical significance (the p-values are equal to $1.16 \cdot 10^{-14}$, $8.80 \cdot 10^{-9}$, and $2.26 \cdot 10^{-12}$, respectively). It indicates that increasing context expansion levels does not improve comment generation performance. When we utilize the method as the context, the result is not better than the 1st level, either ($p$-value = $1.30 \cdot 10^{-12}$). It means the information from the whole method indeed contains redundancy, which impacts the performance negatively. The context information needs to be filtered. Therefore, we regard 1st level context expansion as the context expansion layer, which ensures a good comment generation performance.

*RQ3: What are the effects of different modules on the performance of snippet comment generation?*

As mentioned in Section 3.2, we utilize AST analysis information to define the fine-grained scope of the context and use data flow information to find variable information in the context to construct snippet ASTs. In Section 3.3, we utilize MBT to traverse the AST type and value

Table 3. Ablation Study

| | BLEU | | | | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | | |
| **SBT** | 18.16 | 15.00 | 14.08 | 13.67 | 14.43 | 19.57 |
| **Traverse Type & Value Information in One Sequence** | 18.19 | 15.09 | 14.18 | 13.77 | 14.46 | 19.48 |
| **MBT** | 19.69 | 16.67 | 15.74 | 15.28 | 16.16 | 21.10 |
| **MBT & AST Analysis** | 19.95 | 16.96 | 16.04 | 15.58 | 16.47 | 21.34 |
| **MBT & AST Analysis & Data Flow (Our approach)** | **22.36** | **19.50** | **18.65** | **18.23** | **18.83** | **23.65** |

information to keep the AST information complete. To analyze the influence of different kinds of information on snippet comment generation, we conduct an ablation study. We evaluate the performance by comparing our approach with three models: (1) the model with SBT traversal approach and without any AST analysis and data flow information; (2) the model with traversal approach that traversal type and value information in one sequence; (3) the model with MBT traversal approach; (4) the model with MBT traversal approach and AST analysis information. The result is reported in Table 3. It indicates that adding type information of non-leaf nodes can improve the comment generation performance by 0.73% on BLEU-4. If the type and value information is encoded separately (i.e., MBT), then it can improve the comment generation performance by 11.78% on BLEU-4 compared with SBT. MBT traversal approach and AST analysis information can have an improvement by 13.97% over SBT on BLEU-4. Adding the MBT traversal approach, AST analysis information, and data flow information can improve by 33.36% over SBT on BLEU-4. The result shows that adding value information, AST analysis information, and data flow information all improve snippet comment generation performance.

*RQ4: What are the effects of preceding code expansion and following code expansion on the performance?*

The expanded information in the context includes the preceding context and the following context. We analyze what are the effects of this information on snippet comment generation. The results are reported in Table 4. We process the dataset in three ways: (i) retain all the extracted context with code snippets, (ii) retain only the preceding context with code snippets, and (iii) retain only the following context with code snippets. We also utilize code snippets removing all the context as the baseline. We train the model with these four kinds of data and utilize BLEU-4 to measure their comment generation performances. It can be found that the generation performance is improved when adding the preceding information or adding the following information. Among them, adding both the preceding and following information has the best improvement, which is 6.17% better than that with no context. It indicates that programmers can utilize *SCGen* to generate snippet comments when the complete method codes have been written. Moreover, only adding the preceding information has an improvement of 3.96%, indicating that programmers can utilize *SCGen* to generate snippet comments during development when they have not completely implemented the whole method. Adding the preceding information only also performs better than adding the following information only (17.85 and 17.41). This is because we add both data flow information and AST analysis information when adding the preceding information. However, AST analysis information such as *if*, *for* loop information is not included in the following information, thereby limiting the improvement brought by only adding the following information.

*RQ5: How does the length of the code affect the performance of snippet comment generation?*

Compared with the header comment, the length of the source code covered by the snippet comment is shorter. Thus, it is difficult to have a good comment generation performance. To analyze

Table 4. Effects of Context on Snippet Comment Generation

| | No Context | Only Preceding Code | Only Following Code | Preceding and Following Code |
|---|---|---|---|---|
| **BLEU-4** | 17.17 | 17.85 | 17.41 | 18.23 |
| **Improvements Compared to No Context** | - | 3.96% | 1.40% | 6.17% |

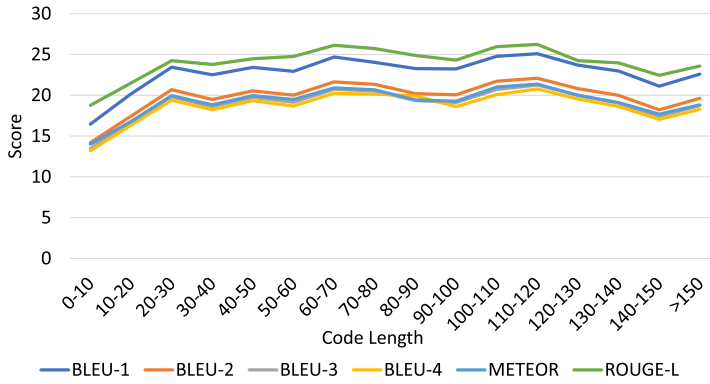



Fig. 7. The performance of *SCGen* over the code snippets with different lengths.

the impact of input code length on the effect of snippet comment generation, we compare the BLEU, METEOR, and ROUGE-L scores of the code snippets of different lengths in the test set. Figure 7 plots the results. The results show that codes with a length between 60–70 tokens have the best generation performance based on different metrics. The average number of lines of codes covered by 60–70 tokens is 4.94 lines. The corresponding average BLEU-4 score is 20.25. The average METEOR score is 20.90 and the ROUGE-L score is 26.14. The code length of 0–10 tokens has the worst performance. They are samples, for which no variable-related information could be found in the context, and these samples represent about 0.48% of the dataset. The average code line number is 2.49. The average BLEU-4 score is 13.20. The average METEOR score is 14.01 and the ROUGE-L score is 18.77. As the code length increases, the code comment generation performance has been maintained in a range between 13 to 21 for BLEU-4. The METEOR and ROUGE-L scores are located in ranges of 14~22 and 18~27. Moreover, with the increased code length, the comment generation performance does not show a clear upward or downward trend. Therefore, code length has a limited influence on the comment generation performance.

### 4.5 Discussion

*4.5.1 The Influence of Comment Scope Detection Approach.* As mentioned in Section 3.1, we utilize a comment scope detection approach proposed by Chen et al. [6] to attain snippet code-comment pairs. We utilize the trained model in the reference and validate the accuracy of our dataset. We randomly sample 1,500 entries in the validation dataset and 500 entries in the test dataset, consequently obtaining accuracy rates of 80.20% and 80.67%, respectively. Those results are quite close to the result proposed in the reference (81.45%). To evaluate the impact of this approach's inaccuracy on snippet comment generation, we correct the samples that are detected incorrectly by the comment scope detection model. Then, we utilize the data samples before the correction and those after the correction as the input of the model to generate snippet comments,

Table 5. The Influence of Comment Scope Detection Approach

| Samples | BLEU | | | | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | | |
| 1,500 Validation Entries before Manual Comment Scope Correction | 24.04 | 21.68 | 20.90 | 19.35 | 21.13 | 26.33 |
| 1,500 Validation Entries after Manual Comment Scope Correction | 23.74 | 21.53 | 20.81 | 19.30 | 21.09 | 26.01 |
| 500 Test Entries before Manual Comment Scope Correction | 22.41 | 19.40 | 18.61 | 18.22 | 18.53 | 23.82 |
| 500 Test Entries after Manual Comment Scope Correction | 22.30 | 19.38 | 18.34 | 18.15 | 18.76 | 23.55 |

respectively. The results are shown in Table 5. It can be found the performance of entries before and after comment scope correction has no significant difference in all the metrics (i.e., the $p$-values are greater than 0.05). Therefore, the comment scope detection approach's inaccuracy has limited influence on snippet comment generation.

*4.5.2 The Function of Token Splitting.* As mentioned in Section 4.1, we use *CamelCase* splitting, *snake_case* splitting to address OOV issue. In this section, we discuss more details. Since we set the vocabulary size as 50,000 for the source code, the total number of unique tokens in our source code dataset is 1,440,654, and the OOV rate is 96.53% if we do not conduct token splitting. But the number of unique tokens is decreased to 58,261, and the OOV rate is decreased to 14.18% if we conduct token splitting. Therefore, using token splitting can effectively alleviate the OOV problem.

*4.5.3 Qualitative Analysis.* To illustrate the validity of *SCGen*, we show a qualitative analysis on two real snippet code-comment pairs in Table 6. We collect these two cases from GitHub. One comes from project *Voldemort*,[3] the other is from *hindex*.[4] In Table 6, we highlight the code snippet to be commented and its corresponding context, which is utilized in the generated comment in the first row. The commented code snippet is in the red box and its context is in the green box. The original comment represents the comment written by programmers. The reference comment indicates the original comment after lemmatization, which is the ground truth. The comments generated by several approaches are also proposed. We compare detailed generation results of seven kinds of comment generation approaches. The words that are the closest to the ground truth are bolded. It can be found that the results of *SCGen* in both two cases are closest to the ground truth. In Case 1, *SCGen* can generate *run* token, which exists in the context in line 10. Although other approaches, such as *Hybrid DeepCom* and *ast-attendgru*, can generate some comment tokens, which match with the ground truth (e.g., *num*, *break*), they are not able to generate tokens that do not exist in code snippet but in the context. Similarly, in Case 2, based on line 7 in the context, *SCGen* can generate *configuration* tokens, existing in the reference comment. Moreover, our approach generates *value* token in the comment. We conjecture it is because there is a combination of tokens, such as *configuration value* in the snippet comments of the training set, which may generate noise and lead to the generation of such tokens in the test set. However, our approach based on semantic expansion can still capture context semantics information to generate snippet comments, which are semantically very similar to the reference comment.

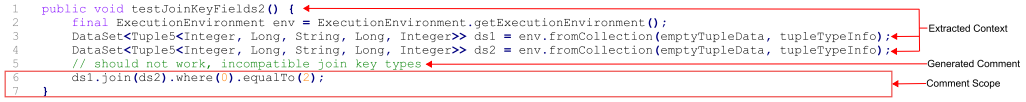
*4.5.4 Context Contribution.* To demonstrate how the context feature contributes to snippet comment generation, we provide an example to illustrate that the attention mechanism acts on the

[3]https://github.com/voldemort/voldemort
[4]https://github.com/Huawei-Hadoop/hindex

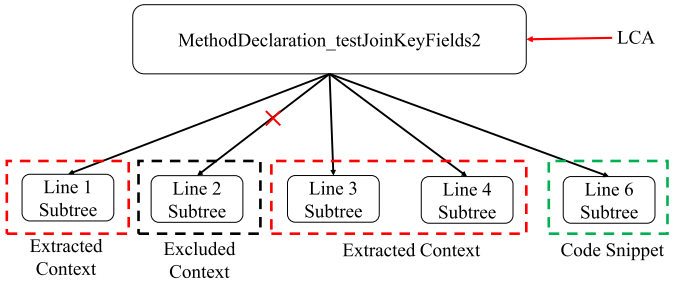
Table 6. Comments Generated by Different Approaches

| | |
|---|---|
| | **Case 1** |
| **Code Snippet and the Context** | `public static List<Integer> removeItemsToSplitListEvenly(final List<Integer> inputList, int maxContiguous) {`<br>`  List<Integer> itemsToRemove = new ArrayList<Integer>();`<br>`  int contiguousCount = inputList.size();`<br>`  if(contiguousCount > maxContiguous) {`<br>`  // Determine how many items must be removed to ensure no contig run`<br>`  // longer than maxContiguous`<br>`  int numToRemove = contiguousCount / (maxContiguous + 1);`<br>`  // Breaking in numToRemove places results in numToRemove+1 runs.`<br>`  int numRuns = numToRemove + 1;`<br>`  //`<br>`  int numItemsLeft = contiguousCount - numToRemove;`<br>`  // Determine minimum length of each run after items are removed.`<br>`  int floorOfEachRun = numItemsLeft / numRuns;`<br>`  // Determine how many runs need one extra element to evenly`<br>`  // distribute numItemsLeft among all numRuns`<br>`  int numOfRunsWithExtra = numItemsLeft - (floorOfEachRun * numRuns);`<br><br>`  int offset = 0;`<br>`  for(int i = 0; i < numToRemove; ++i) {`<br>`  offset += floorOfEachRun;`<br>`  if(i < numOfRunsWithExtra)`<br>`  offset++;`<br>`  itemsToRemove.add(inputList.get(offset));`<br>`  offset++;`<br>`  }`<br>`  }`<br>`  return itemsToRemove;`<br>`}` |
| **Original Comment** | Num items left to break into numRuns |
| **Reference Comment** | num item leave to break into num run |
| ***Rencos*** | set up the topology client |
| ***CodeBERT*** | number of item leave to remove |
| ***Code-NN*** | remove the element at the end of the number of files added to the end of the list |
| ***DeepCom*** | num row with current one: // if one, make it at least one entry in the same filter |
| ***Hybrid DeepCom*** | num item to remove bit of file to avoid grow |
| ***ast-attendgru*** | num item leave to break into remove space |
| ***RLcom*** | remove the number of item to be a , |
| ***SCGen*** | **num item leave to break into num run** |
| | **Case 2** |
| **Code Snippet and the Context** | `public void testCreateFamilyCompressionMap() throws IOException {`<br>`  for (int numCfs = 0; numCfs <= 3; numCfs++) {`<br>`  Configuration conf = new Configuration(this.util.getConfiguration());`<br>`  Map<String, Compression.Algorithm> familyToCompression = getMockColumnFamilies(numCfs);`<br>`  HTable table = Mockito.mock(HTable.class);`<br>`  setupMockColumnFamilies(table, familyToCompression);`<br>`  HFileOutputFormat.configureCompression(table, conf);`<br>`  //`<br>`  Map<byte[], String> retrievedFamilyToCompressionMap = HFileOutputFormat.createFamilyCompressionMap(conf);`<br>`  for (Entry<String, Algorithm> entry : familyToCompression.entrySet()) {`<br>`  assertEquals(\"Compression configuration incorrect for column family:\" + entry.getKey(), entry.getValue()`<br>`  .getName(), retrievedFamilyToCompressionMap.get(entry.getKey().getBytes()));`<br>`  }`<br>`  }`<br>`}` |
| **Original Comment** | read back family specific compression setting from the configuration |
| **Reference Comment** | read back family specific compression setting from the configuration |
| ***Rencos*** | family compression map |
| ***CodeBERT*** | create a map of family to retrieve the family |
| ***Code-NN*** | compression |
| ***DeepCom*** | convert to a map base on hfile map value |
| ***Hybrid DeepCom*** | read all family into the table |
| ***ast-attendgru*** | read through family map of family compression |
| ***RLcom*** | for the compression file to be a |
| ***SCGen*** | **read back family specific compression from configuration value** |

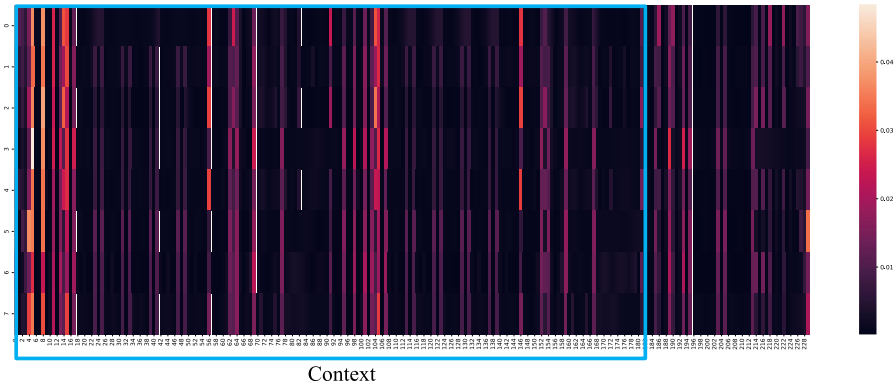
context and has an impact on snippet comment generation, which is shown in Figure 8. Figure 8(a) shows the source code snippet, the generated comment, and the method where the comment is located. The corresponding snippet AST is shown in Figure 8(b), and the LCA, the extracted context subtree, and the code snippet subtree are highlighted. Line 2 subtree is excluded by the snippet AST. We traverse the AST tokens and input them into the model. We utilize the attention heatmap to visualize the weight of each AST token when generating each comment token, which is shown in Figure 8(c). In the attention heatmap, the horizontal axis represents the AST token sequence number, and the vertical axis represents the generated comment token sequence number. It can be




(a) Source code




(b) Snippet AST




(c) Attention heatmap

Fig. 8. The case of context contribution.

found from the heatmap that No. 4, 5, 8, 104, and 105 AST tokens play important roles when generating each comment token. These tokens are *key*, *fields2*, *set*, *type*, and *tuple5* in the AST token sequence. They all belong to variable-related tokens and the LCA in the context (i.e., extracted context). Their corresponding statements are located in lines 1, 3, and 4 in Figure 8(a). Therefore, data flow information and AST analysis information in the context contribute to generating snippet comments.

*4.5.5 Human Evaluation.* Besides utilizing automatic NMT metrics, we also conduct a human evaluation to compare the performance of different comment generation approaches. Specifically, we randomly select 30 samples from the test set. Then, we make a questionnaire by using a questionnaire production website called Wenjuanxing[5] and invite 10 volunteers with more than three years of software development experience and good English ability to rate the samples on a scale between 1 and 5 (the higher, the better). Seven of them are graduate students and three of them

[5]https://www.wjx.cn/

| Reference comment | give time for data cleanup to run |
|---|---|
| Model 1 | sleep for 3 second |
| Model 2 | wait a bit |
| Model 3 | wait for 3 second request on the second time |
| Model 4 | wait for some second for interval to terminate |
| Model 5 | give time for data loss |
| Model 6 | give time for the time the second time |
| Model 7 | wait for the sleep to be a , |
| Model 8 | give time for cleanup everything to run |

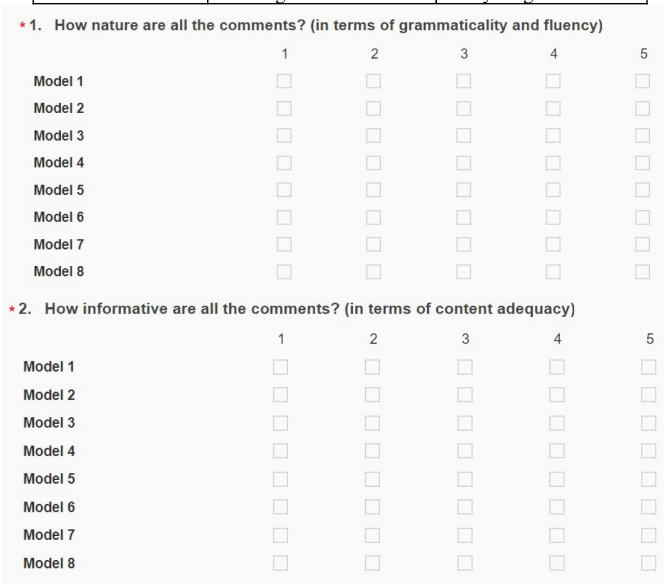



Fig. 9. Example questions of our questionnaire.

are Java software development practitioners. We send the questionnaire link to them by email or social media. Similar to previous work [44], those volunteers are asked to rate from three aspects: **similarity** of the generated comments and the reference comments, **naturalness** (i.e., grammaticality and fluency), and **informativeness** (i.e., content adequacy). To ensure the fairness of the human evaluation, we also mask the model names and make participants evaluate only on the basis of source codes and generated comments. Figure 9 shows example questions of our questionnaire. Each comment is evaluated by 10 volunteers, and we average their ratings to get the score of the comment being evaluated. Table 7 shows the evaluation result. It can be found that our approach outperforms other approaches in three aspects. Our approach gets 3.45 in informativeness, 3.68 in naturalness, and 3.33 in similarity, while other approaches get 2.49~3.01 in informativeness, 2.75~3.55 in naturalness, and 2.27~2.81 in similarity. It means that our approach can generate comprehensive snippet comments. In addition, we confirm the superiority of *SCGen* using Wilcoxon signed-rank tests for human evaluation. And the results reflect that the improvement of *SCGen* over other approaches is statistically significant with most $p$-values smaller than 0.05 at 95% confidence level (except for *Rencos*, *CodeBERT*, *Code-NN* on Naturalness).

*4.5.6 Motivating Scenario.* *SCGen* is an automatic snippet comment generation approach that essentially performs a conversion task from programming language to natural language (PL-NL task). Therefore, it is possible that the approach can be applied to other similar PL-NL tasks. For

Table 7. Human Evaluation Results

| Approaches | Informativeness | Naturalness | Similarity |
|---|---|---|---|
| ***Rencos*** | 2.98 | 3.55 | 2.67 |
| ***CodeBERT*** | 2.84 | 3.51 | 2.52 |
| ***Code-NN*** | 2.98 | 3.49 | 2.68 |
| ***Deepcom*** | 2.94 | 3.43 | 2.66 |
| ***Hybrid Deepcom*** | 2.98 | 3.37 | 2.73 |
| ***ast-attendgru*** | 3.01 | 3.40 | 2.81 |
| ***RLcom*** | 2.49 | 2.75 | 2.27 |
| *Our* ***SCGen*** | **3.45** | **3.68** | **3.33** |

example, using function code to user notice [14] or using program code to achieve App Privacy Policy generation automatically [61]. The prerequisite for these tasks is the availability of a suitable dataset and a reasonable code feature extraction method for these tasks.

## 5 THREATS TO VALIDITY

In this section, we introduce the threats that can affect the results of our case studies.

**Threats to internal validity** refer to the scale and quality of the dataset for training the model. In this article, we collect 959 projects containing 600,243 snippet comments from GitHub. The size is limited and increasing the data size can increase the diversity of samples and improve the scalability of the model. Moreover, we adopt a comment scope detection approach based on machine learning. The non-perfect accuracy of this approach may allow the dataset to contain noise. We have verified its accuracy by random sampling parts of data in both the validation dataset and test dataset to prove the limited effects of noise. Due to the data size, we cannot manually verify all the data. But we can find that the comment scope detection approach's inaccuracy has limited influence on snippet comment generation based on the results. In addition, the soundness and completeness of the data-flow analysis may influence the result of capturing the context, since we utilize static analysis methods (data flow information) to extract the context. For example, some incorrect or unrelated context will be captured if the analysis result is sound. If the analysis result is complete, then some related contexts are missed. Furthermore, some Java language features such as reflection and native code may also influence the accuracy of static analysis methods, thereby influencing the accuracy of extract context and the comment generation performance. The extent of the impact needs further investigation.

**Threats to external validity** refer to the scalability of the proposed model. In this article, *SCGen* utilizes code snippets written in Java language. When the proposed approach is applied in other languages, such as C, C++, and Python, they may have slightly different code features. These kinds of features should be carefully processed when inputting the AST or source code. Therefore, more investigation by analyzing codes written by other languages should be executed.

**Threats to construct validity** mean the suitability of our evaluation approach. In this article, we utilize BLEU, METEOR, and ROUGE-L, which are all machine translation evaluation metrics to calculate the performance of comment generation models. *SCGen* is also compared with the baseline models in terms of these metrics. Thus, we believe that there is little threat to the suitability of our evaluation method.

## 6 CONCLUSION AND FUTURE WORK

Code comments including header comments and snippet comments play a vital part in program comprehension and software maintenance, and automatic comment generation helps improve software development efficiency. In this article, we propose a snippet comment generation approach

called *SCGen*. We first download Java projects from GitHub. After that, we extract snippet comments and utilize a comment scope prediction model to collect snippet code-comment pairs data and utilize data flow and AST analysis information of code snippets to expand the context information. The context information is variable-related information and structural information for constructing snippet ASTs. We input the source code and ASTs with expanded information into a sequence-to-sequence-based model to generate snippet comments. The experimental results demonstrate that *SCGen* has a better performance compared with other state-of-the-art comment generation approaches. Moreover, the context information contributes to improving the effectiveness of snippet comment generation. In the future, we will improve the model so it can also play a role in the comment generation of other programming languages.

## REFERENCES


[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33rd International Conference on Machine Learning (IMCL'16)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, NY, 2091–2100. Retrieved from http://proceedings.mlr.press/v48/allamanis16.html

[2] Oliver Arafat and Dirk Riehle. 2009. The comment density of open source software code. In *Proceedings of the 31st International Conference on Software Engineering-Companion Volume (ICSE-Companion'09)*. IEEE, 195–198. DOI: https://doi.org/10.1109/ICSE-COMPANION.2009.5070980

[3] Oliver Arafat and Dirk Riehle. 2009. The commenting practice of open source. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 857–864. DOI: https://doi.org/10.1145/1639950.1640047

[4] Dzmitry Bahdanau, KyungHyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*. DOI: https://doi.org/abs/1409.0473

[5] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. 65–72. Retrieved from https://aclanthology.info/papers/W05-0909/w05-0909

[6] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. 2019. Automatically detecting the scopes of source code comments. *J. Syst. Softw.* 153 (2019), 45–63. DOI: https://doi.org/10.1016/j.jss.2019.03.010

[7] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 25 (2021), 29 pages. DOI: https://doi.org/10.1145/3434280

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'20)*. ACL. Retrieved from https://arxiv.org/abs/2002.08155

[9] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2017. Autofolding for source code summarization. *IEEE Trans. Softw. Eng.* 43, 12 (2017), 1095–1109. DOI: https://doi.org/10.1109/TSE.2017.2664836

[10] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to comment "Translation": Data, metrics, baselining & Evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 746–757. DOI: https://doi.org/10.48550/arXiv.2010.01410

[11] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE'10)*. IEEE, New York, NY, 223–226. DOI: https://doi.org/10.1145/1810295.1810335

[12] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*. IEEE, New York, NY, 35–44. DOI: https://doi.org/10.1109/WCRE.2010.13

[13] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, 300–310. Retrieved from https://doi.org/10.1145/3379597.3387449

[14] Xing Hu, Zhipeng Gao, Xin Xia, David Lo, and Xiaohu Yang. 2021. Automating user notice generation for smart contract functions. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. 5–17. DOI: https://doi.org/10.1109/ASE51524.2021.9678552

[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *IEEE/ACM 26th International Conference on Program Comprehension (ICPC'18)*. ACM, New York, NY, 200–210. DOI: https://doi.org/10.1145/3196321.3196334

[16] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* 25, 3 (2020), 2179–2217.

[17] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Yingfei Xiong, and Zibin Zheng. 2020. Learning code context information to predict comment locations. *IEEE Trans. Reliab.* 69, 1 (2020), 88–105. DOI: https://doi.org/10.1109/TR.2019.2931725

[18] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Zibin Zheng, and Xiapu Luo. 2020. CommtPst: Deep learning source code for commenting positions prediction. *J. Syst. Softw.* 170 (2020), 110754. DOI: https://doi.org/10.1016/j.jss.2020.110754

[19] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020. Towards automatically generating block comments for code snippets. *Inf. Softw. Technol.* 127 (2020), 106373. DOI: https://doi.org/10.1016/j.infsof.2020.106373

[20] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. 2020. Does your code need comment? *Softw.: Pract. Exper.* 50, 3 (2020), 227–245.

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16)*. ACL, 2073–2083. DOI: https://doi.org/10.18653/v1/P16-1195

[22] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, Yoshua Bengio and Yann LeCun (Eds.). Retrieved from http://arxiv.org/abs/1412.6980

[23] Philipp Koehn. 2004. Pharaoh: A beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the Conference of the Association for Machine Translation in the Americas (AMTA'04)*, Robert E. Frederking and Kathryn B. Taylor (Eds.). Springer Berlin, 115–124. DOI: https://doi.org/10.1007/978-3-540-30194-3_13

[24] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC'20)*. ACM, New York, NY, 184–195. DOI: https://doi.org/10.1145/3387904.3389268

[25] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, New York, NY, 795–806. DOI: https://doi.org/10.1109/ICSE.2019.00087

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[27] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: A deep code comment generation tool with hybrid lexical and syntactical information. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, New York, NY, 1571–1575. DOI: https://doi.org/10.1145/3368089.3417926

[28] Yuzheng Li, Chuan Chen, Nan Liu, Huawei Huang, Zibin Zheng, and Qiang Yan. 2021. A blockchain-based decentralized federated learning framework with committee consensus. *IEEE Netw.* 35, 1 (2021), 234–241. DOI: https://doi.org/10.1109/MNET.011.2000263

[29] Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Deli Yu. 2021. SeCNN: A semantic CNN parser for code comment generation. *J. Syst. Softw.* 181 (2021), 111036. DOI: https://doi.org/10.1016/j.jss.2021.111036

[30] Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. *Proc. AAAI Conf. Artif. Intell.* 32, 1 (2018). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/11963

[31] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. 184–195. DOI: https://doi.org/10.1109/ICPC52881.2021.00026

[32] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Proceedings of Workshop on Text Summarization Branches Out, Post-conference Workshop of ACL*. ACL, 74–81.

[33] Peng-fei Liu and Xiao-meng Wang. 2020. Utilizing keywords in source code to improve code summarization. In *Proceedings of the IEEE 6th International Conference on Computer and Communications (ICCC'20)*. IEEE, New York, NY, 664–668. DOI: https://doi.org/10.1109/ICCC51575.2020.9345066

[34] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. ACM, New York, NY, 585–597. DOI: https://doi.org/10.1145/3324884.3416581

[35] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. Association for Computing Machinery, New York, NY, 279–290. DOI: https://doi.org/10.1145/2597008.2597149

[36] Paul W. McBurney and Collin McMillan. 2016. Automatic source code summarization of context for Java methods. *IEEE Trans. Softw. Eng.* 42, 2 (2016), 103–119. DOI: https://doi.org/10.1109/TSE.2015.2465386

[37] Paul W. McBurney and Collin McMillan. 2016. An empirical study of the textual similarity between source code and source code summaries. *Empir. Softw. Eng.* 21, 1 (2016), 17–42.

[38] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. IEEE, New York, NY, 23–32. DOI: https://doi.org/10.1109/ICPC.2013.6613830

[39] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA: An approach for the automated generation of release notes. *IEEE Trans. Softw. Eng.* 43, 2 (2017), 106–127. DOI: https://doi.org/10.1109/TSE.2016.2591536

[40] Laura Moreno and Andrian Marcus. 2017. Automatic software summarization: The state of the art. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C'17)*. 511–512. DOI: https://doi.org/10.1109/ICSE-C.2017.169

[41] Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings of the Conference on Software Maintenance*. IEEE, 337–344. DOI: https://doi.org/10.1109/ICSM.1992.242525

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL'02)*. ACL, 311–318. Retrieved from http://www.aclweb.org/anthology/P02-1040.pdf

[43] Rohit Prabhavalkar, Tara N. Sainath, Yonghui Wu, Patrick Nguyen, Zhifeng Chen, Chung-Cheng Chiu, and Anjuli Kannan. 2018. Minimum word error rate training for attention-based sequence-to-sequence models. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'18)*. IEEE, New York, NY, 4839–4843. DOI: https://doi.org/10.1109/ICASSP.2018.8461809

[44] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. EMNLP.

[45] Xiaotao Song, Sakib Haque Sun, Xu Wang, and Jiafei Yan. 2019. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access* 7 (2019), 111411–111428. DOI: https://doi.org/10.1109/ACCESS.2019.2931579

[46] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, 43–52. DOI: https://doi.org/10.1145/1858996.1859006

[47] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 101–110. DOI: https://doi.org/10.1145/1985793.1985808

[48] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, New York, NY, 2–13. Retrieved from https://doi.org/10.1145/3387904.3389258

[49] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC'13)*. IEEE, New York, NY, 83–92. DOI: https://doi.org/10.1109/ICPC.2013.6613836

[50] Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Recovering variable names for minified code with usage contexts. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, New York, NY, 1165–1175.

[51] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. 2014. CODES: Mining source code descriptions from developers discussions. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. ACM, New York, NY, 106–109. DOI: https://doi.org/10.1145/2597008.2597799

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'17)*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[53] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY, 397–407. DOI: https://doi.org/10.1145/3238147.3238206

[54] Deze Wang, Yong Guo, Wei Dong, Zhiming Wang, Haoran Liu, and Shanshan Li. 2019. Deep code-comment understanding and assessment. *IEEE Access* 7 (2019), 174200–174209. DOI: https://doi.org/10.1109/ACCESS.2019.2957424

[55] Bolin Wei. 2019. Retrieve and refine: Exemplar-based neural comment generation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, New York, NY, 1250–1252. DOI: https://doi.org/10.1109/ASE.2019.00152

[56] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC'19)*. IEEE, New York, NY, 53–64.

[57] Yingce Xia, Tianyu He, Xu Tan, Fei Tian, Di He, and Tao Qin. 2019. Tied transformers: Neural machine translation with shared encoder and decoder. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'19)*, Vol. 33. AAAI, 5466–5473. DOI: https://doi.org/10.1609/aaai.v33i01.33015466

[58] Guang Yang, Ke Liu, Xiang Chen, Yanlin Zhou, Chi Yu, and Hao Lin. 2022. CCGIR: Information retrieval-based code comment generation method for smart contracts. *Knowl.-based Syst.* 237 (2022), 107858. DOI: https://doi.org/10.1016/j.knosys.2021.107858

[59] Yatao Yang, Zibin Zheng, Xiangdong Niu, Mingdong Tang, Yutong Lu, and Xiangke Liao. 2021. A location-based factorization machine model for web service QoS prediction. *IEEE Trans. Serv. Comput.* 14, 5 (2021), 1264–1277. DOI: https://doi.org/10.1109/TSC.2018.2876532

[60] Annie T. T. Ying and Martin P. Robillard. 2013. Code fragment summarization. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 655–658. DOI: https://doi.org/10.1145/2491411.2494587

[61] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. 2017. Toward automatically generating privacy policy for Android apps. *IEEE Trans. Inf. Forens. Secur.* 12, 4 (2017), 865–880. DOI: https://doi.org/10.1109/TIFS.2016.2639339

[62] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, New York, NY, 1385–1397.

[63] Xiaoqin Zhang, Runhua Jiang, Tao Wang, and Jinxin Wang. 2021. Recursive neural network for video deblurring. *IEEE Trans. Circ. Syst. Vid. Technol.* 31, 8 (2021), 3025–3036. DOI: https://doi.org/10.1109/TCSVT.2020.3035722

[64] Zibin Zheng, Xiaoli Li, Mingdong Tang, Fenfang Xie, and Michael R. Lyu. 2022. Web service QoS prediction via collaborative filtering: A survey. *IEEE Trans. Serv. Comput.* 15, 4 (2022), 2455–2472. DOI: https://doi.org/10.1109/TSC.2020.2995571

Received 22 August 2022; revised 22 May 2023; accepted 3 July 2023