Neural Computing and Applications
https://doi.org/10.1007/s00521-022-07877-z

ORIGINAL ARTICLE

ClassSum: a deep learning model for class-level code summarization

Mingchen Li[1] · Huiqun Yu[1,2] · Guisheng Fan[1,3] · Ziyi Zhou[1] · Jiawen Huang[1]

Received: 18 March 2022 / Accepted: 21 September 2022

Abstract
Code summaries are clear and concise natural language descriptions of program entities. Meaningful code summaries assist developers in better understanding. Code summarization refers to the task of generating a natural language summary from a code snippet. Most researches on code summarization focus on automatically generating summaries for methods or functions. However, in an object-oriented language such as Java, class is the basic programming unit rather than method. To fill this gap, in this paper, we investigate how to generate summaries for Java classes utilizing deep learning-based approaches. We propose a novel encoder–decoder model called ClassSum to generate functionality descriptions for Java classes and build a dataset containing 172,639 <class, summary> pairs from 3185 repositories hosted on Github. Since the code of class is much longer and more complicated, encoding a whole class via neural network is more challenging than encoding a method. On the other hand, the content within a class may be incomplete. To overcome this difficulty, we reduce the code of a class by only keeping its key elements, namely class signatures, method signatures and attribute names. To utilize both lexical and structural information of code, our model takes token sequence and abstract syntax tree of the reduced class content as inputs. ClassSum and five baselines (designed for method-level code summarization) are evaluated on our dataset. Experiment results show that summaries generated by ClassSum are more accurate and readable than those generated by baselines. Our dataset is available at https://github.com/classsum/ClassSum.

Keywords Program comprehension · Code summarization · Class documentation · Deep learning

1 Introduction

Code summaries are clear and concise natural language descriptions of program entities, e.g., comment for a statement, document for a method or a class. During software development and maintenance, developers spend a considerable amount of time on program comprehension activities [1]. High-quality summaries not only save energy for the developers on understanding programs, but also benefit many important code-related tasks like code search [2]. Unfortunately, code summaries are often missing or mismatching in practical. The task of code summarization aims to generate a corresponding summary written in natural language (e.g., English) for a given code snippet. Most of the early studies on automatic code summarization are based on template and information retrieval techniques. In recent years, with the development and significant progress of neural machine translation (NMT), data-driven and deep learning-based automatically code summarization models have also received a great deal of research interest. But almost all of these approaches focus on generating

✉ Huiqun Yu
yhq@ecust.edu.cn

✉ Guisheng Fan
gsfan@ecust.edu.cn

Mingchen Li
lmc@mail.ecust.edu.cn

Ziyi Zhou
zhouziyi@mail.ecust.edu.cn

Jiawen Huang
jwhuang@mail.ecust.edu.cn

1 Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China

2 Shanghai Engineering Research Center of Smart Energy, Shanghai, China

3 Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai Computer Software Technique Development Center, Shanghai, China

Published online: 10 October 2022

summaries for methods or functions [3]. However, class is the programming component unit in Object-Oriented programming languages such as Java, rather than method [4]. Existing efforts on automatic summarization for class are mainly based on code retrieval and heuristics, which have difficulties on generalizing to arbitrary program. What lacks from the literature is how to produce class summaries with the help of deep learning. What's more, there is a lack of large-scale parallel corpus for class-level code summarization. This paper aims to solve the above issues by introducing new dataset and novel model. Compared to the method-level code summarization, the main challenges of class-level code summarization are as follows:

1. The code within a class is much longer and complex than the method. Statistical results indicate that the average number of tokens within a method is about 100 or less [5, 6], while that within a class is 826 (will be detailed in Sect. 4). The long inputs challenge neural model to handle long-range dependencies and thus make it difficult to extract useful features in the source code.
2. Class structure is different from method structure. For example, in Java programming language, a method often contains a signature, variable declarations, and statements, while a class contains a signature, inter methods and some data attributes. In method-level code summarization, abstract syntax trees (ASTs) are usually used for representing structure information. Unfortunately, we find that a widely used input form of AST called SBT [6] in method-level code summarization is not suitable for class structural information representation.
3. We lack a large-scale dataset for class-level summarization. Many public code repositories host on the Internet, which can be used to extract parallel corpus of classes and summaries. However, not all of the repositories are about software development, such as blogs and notebooks. We find that many low-quality and informationless comments exist everywhere, such as authors' personal information and copyright declarations. If these comments occur in training dataset, it will prevent the generality of summarization models.

Our solutions to overcome these challenges are as follows:

1. We propose a method to extract key elements of class to obtain reduced class content as model input. By utilizing the reduced source code as the input of model rather than original source code, the average length of input decreases from 826 to 311. Another advantage of using reduced source code is that the developers do not need to fully implement the class when applying ClassSum. This situation is very common in software development, because the source code within methods of a Java class from other organizations or companies is not always visible.
2. We develop a traversal method for the class called CSBT on the basis of SBT. CSBT is generated from class AST of the reduced class content. Compared with the original source code, the average numbers of nodes within reduced AST decreases from 305 to 80.
3. We build a parallel corpus with 172,639 <class, summary> pairs from 3185 Java projects hosted on Github [7]. To the best of our knowledge, the corpus is the first large-scale dataset for class-level code summarization.

Moreover, we propose an encoder–decoder framework called ClassSum for class-level code summarization. It has two encoders: one for encoding the lexical information from reduced class content, one for encoding the structural information from CSBT. Then we split our dataset to a training set, a validation set, a hybrid-projects test set and a cross-projects test set. We evaluate ClassSum and compare it to various state-of-the-art baselines for method-level code summarization on the datasets. Experimental results show that ClassSum achieves the best performance.

Paper organization The rest of this paper is organized as follows. Section 2 provides the background of automatic code summarization and neural networks are involved in this paper. Section 3 elaborates on the details of ClassSum. Section 4 describes the details of the dataset and experiment setup. Section 5 shows the experimental results and analysis. Section 6 presents the threats to validity. Section 7 surveys the related studies. Finally, Sect. 8 concludes the paper and proposes potential future directions.

2 Background

In this section, we introduce the background of automatic code summarization and neural networks involved in this paper.

2.1 Automatic code summarization

Automatic code summarization is a task focusing on generating summaries for program entities. The current paradigm of this filed is to treat the task as a supervised learning task [3]. Assuming that $\boldsymbol{C}$ represents the source code and $\boldsymbol{N}$ represents the corresponding human-written summary. The training dataset can be represented as $(\boldsymbol{C}, \boldsymbol{N})$ pairs and is used to train the code summarization model. Note that the source code can be different granularities, such as methods [8], code blocks [9], APIs [10] or classes. Programming language model is the basis of automatic code summarization. Hindle et al. [11] proves that the programming language is often with well-predictable statistical properties which can be captured by statistical language models and leveraged for software engineering tasks. More formally, given a program token sequence (including source code and documents) $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, the joint distribution probability of the sequence can be computed via each of its previous items by:

$$p(\boldsymbol{x}) = p(x_1)p(x_2|x_1)\cdots p(x_n|x_{n-1}\cdots x_1) \tag{1}$$

Suppose $\boldsymbol{c} = (c_1, \ldots, c_n)$ is the code sequence, $\boldsymbol{w} = (w_0, w_1, \ldots, w_m)$ is the corresponding summary, w_0 and w_m are two special words, *BOS* and *EOS*, which represent the begin of a sentence and the end of a sentence. A code summarization model is a parametric nonlinear function and formularized utilizing a deep neural network. The network is trained to fit the following conditional distribution:

$$p(w_i|\boldsymbol{c}, \boldsymbol{w_{i<t}}) \quad i = 1\cdots m \tag{2}$$

which represents the probability of the next word w_i conditioned on previously generated words $(w_0, w_1, \ldots, w_{i-1})$ and the source code sequence $\boldsymbol{c}$. From the conditional probability formula, we get:

$$p(w_i|\boldsymbol{c}, \boldsymbol{w_{i<t}}) = \frac{p(\boldsymbol{c}, \boldsymbol{w_{i\le t}})}{p(\boldsymbol{c}, \boldsymbol{w_{i<t}})} \tag{3}$$

According to Eq. (1), $p(\boldsymbol{c}, \boldsymbol{w_{i\le t}})$ and $p(\boldsymbol{c}, \boldsymbol{w_{i<t}})$ are in well-statistical properties. Therefore $p(w_i|\boldsymbol{c}, \boldsymbol{w_{i<t}})$ can also be fitted utilizing data-driven methods.

2.2 Encoder–decoder framework

Encoder–decoder [12] framework is the most popular framework in NMT. The encoder–decoder framework typically consists of an encoder and a decoder.

The encoder usually works as a dimension reduction component. Given a sequence of input $\boldsymbol{x} = (x_1, \ldots, x_n)$, where n is the length of $\boldsymbol{x}$. The encoder maps $\boldsymbol{x}$ to a continuous vector $\boldsymbol{h}$:

$$\boldsymbol{h} = encoder(\boldsymbol{x}) \quad x \in R^{n\times m}, h \in R^d \tag{4}$$

The decoder is used to generate target sequence. It often contains an attention mechanism. Attention mechanism is an effective model that selects different parts from the input sequence for each output token. It's always used to boost encoder–decoder models [13]. For example, in an attentional RNN-based encoder–decoder framework, at each decoding time step t, the attention mechanism computes a context vector $\boldsymbol{c_t}$ as a weighted sum of the outputs of encoder:

$$\boldsymbol{c_t} = \sum_{i=1}^{n} \alpha_{ti}\boldsymbol{o_i} \tag{5}$$

The weight a_{ti} of current state $\boldsymbol{s_t}$ on the i_{th} outputs of encoder is computed as:

$$\alpha_{ti} = \frac{exp(e_{ti})}{\sum_{k=1}^{n} exp(e_{tk})} \tag{6}$$

and

$$e_{tk} = a(\boldsymbol{s_t}, \boldsymbol{o_k}) \tag{7}$$

where a is a score function (such as dot function), $\boldsymbol{o_k}$ is the output of encoder at the k_{th} encoding time step.

The decoder is trained to fit the conditional probability distribution of next word prediction. The distribution is computed as:

$$p(w_t|\boldsymbol{w_{i<t}}, \boldsymbol{x}) = softmax(g(\boldsymbol{s_t}, \boldsymbol{c_t})) \tag{8}$$

where g is the decoder, $\boldsymbol{w} = (w_1, \ldots, w_m)$ is the target sequence, $\boldsymbol{s_t}$ is the current state and $\boldsymbol{c_t}$ is the context vector at decoding time step t.

2.3 Recurrent neural network

Recurrent neural network (RNN) [14] has been widely used in code summarization [6, 8, 15]. In this section, we will introduce a variant of RNN—the long short-term memory (LSTM) [16].

During the back-propagation process of standard RNN, the gradient may vanish or explode, especially when the input sequence is too long. Long short-term memory (LSTM) is designed to reduce the problem. LSTM is normally augmented by recurrent gates called "forget gates" which can prevent back-propagated errors from vanishing or exploding. The forget gates decide whether the information should be discarded or retained. Figure 1 shows a typical cell of LSTM.

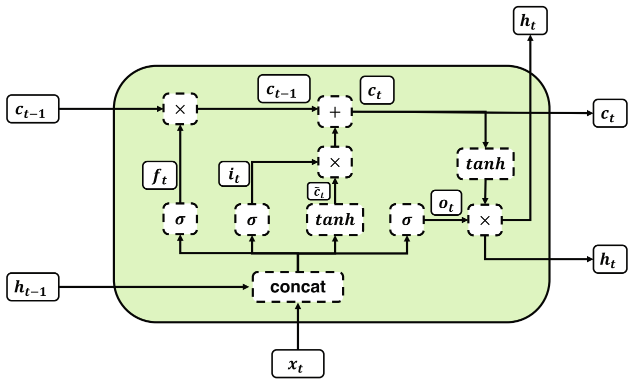


Fig. 1 The structure of a LSTM cell

The hidden state $\boldsymbol{h_t}$, cell state $\boldsymbol{c_t}$ and output $\boldsymbol{o_t}$ of the LSTM cell at each time step t is computed as follows:

$$\begin{aligned}
\boldsymbol{f_t} &= tanh(\boldsymbol{W_f x_t} + \boldsymbol{U_f h_{t-1}} + \boldsymbol{b_f})\\
\boldsymbol{i_t} &= tanh(\boldsymbol{W_i x_t} + \boldsymbol{U_i h_{t-1}} + \boldsymbol{b_i})\\
\boldsymbol{o_t} &= tanh(\boldsymbol{W_o x_t} + \boldsymbol{U_o h_{t-1}} + \boldsymbol{b_o}) \qquad (9)\\
\widetilde{\boldsymbol{c}}_{\boldsymbol{t}} &= tanh(\boldsymbol{W_f x_t} + \boldsymbol{U_f h_{t-1}} + \boldsymbol{b_f})\\
\boldsymbol{c_t} &= \boldsymbol{f_t} \cdot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \cdot \widetilde{\boldsymbol{c}}_{\boldsymbol{t}} \boldsymbol{h_t} = \boldsymbol{o_t} \cdot tanh(\boldsymbol{c_t})
\end{aligned}$$

where $\boldsymbol{h_{t-1}}$ is the previous hidden state, $\boldsymbol{c_{t-1}}$ is the previous cell state, $\boldsymbol{x_t}$ is the input at current time step t, $\boldsymbol{W_f}$, $\boldsymbol{W_i}$, $\boldsymbol{W_o}$, $\boldsymbol{U_f}$, $\boldsymbol{U_i}$, $\boldsymbol{U_o}$, $\boldsymbol{b_f}$, $\boldsymbol{b_i}$ and $\boldsymbol{b_o}$ are the trainable parameters.

3 Proposed framework

In this section, we introduce the overall framework and details of proposed model.

3.1 Overall framework of ClassSum

ClassSum is a class-level code summarization model for Java programming language. Figure 2 shows the overall framework of ClassSum. As seen, there are four components in ClassSum: a data preprocessor, a lexical encoder, a syntax encoder and a joint decoder. The two encoders map discrete inputs to continuous vectors in Euclidean space, which will be fed to the joint decoder for decoding and generating summaries. After testing the performance of different RNN types including Simple RNN [14], GRU [17], and LSTM [16], we finally decide to utilize LSTM as the encoder. Both the lexical encoder and the syntax encoder are bidirectional LSTM with two layers. We use two encoders because the lexical and structural information of the source code are both important [15]. Later, we will show that the performance will be worse if we remove either encoder of the model. To better illustrate ClassSum, we split the model to three steps: data preprocessing, encoding, and decoding. The details of each step of ClassSum are introduced in the following subsections.<Figure ID="

3.2 Data preprocessing

As for data preprocessing, we firstly reduce the original source code within a Java class and parse it to an AST. Next, we use class structure-based traversal to convert the AST to a CSBT sequence. Then, we tokenize the reduced source code to token sequence. Finally, the CSBT sequence is fed to the syntax encoder and the token sequence is fed to the lexical encoder. The subsections indicate the details of code reducing and CSBT generating.

3.2.1 Source code reducing

As we've mentioned in Sect. 1, the source code of methods within a class is not always fully implemented before writing the comments, and the average number of tokens in code with a class is eight times greater than method. However, many tokens are redundant for summary generation. Such long inputs challenge summarization model to handle long-range dependencies and thus make it difficult to extract useful features of token sequence. To solve this problem, we propose to reduce the class content as follows. The bodies of method and nested class were removed. We only keep the class signature, attribute types and names, method signatures of class source code. This is similar to the human's code reading habits, developers mainly

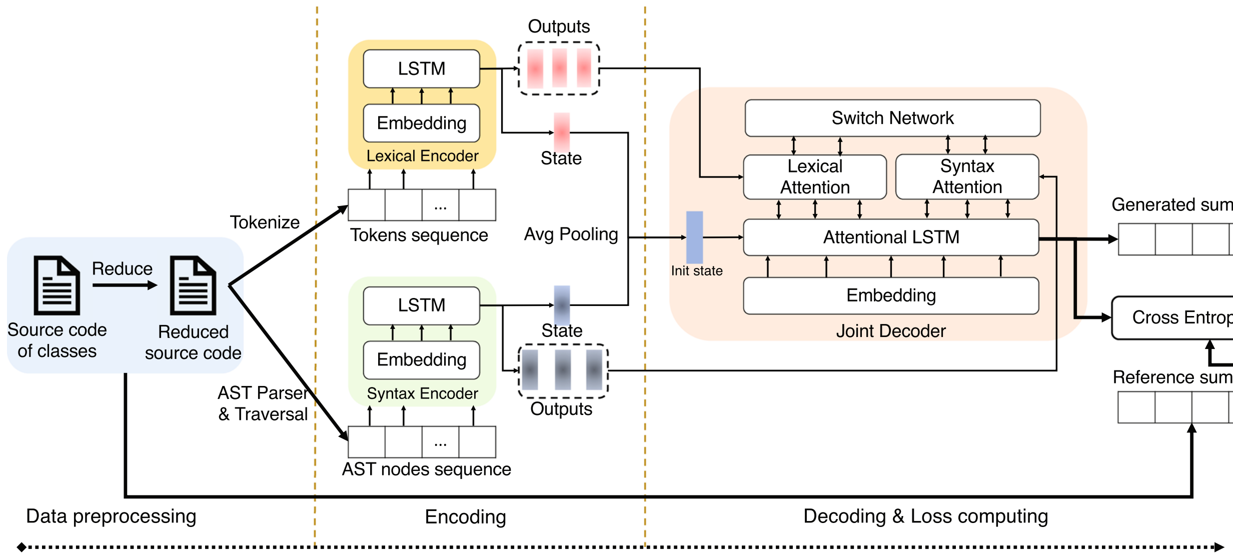


Fig2" Category="Standard" Float="Yes"> **Fig. 2** The overall framework of ClassSum focuses on the class attributes and method signatures when they try to read the source code of a class. More specifically, given the source code a Java class, we adopt the following steps to reduce the class code. First, we use the SrcML tool [18] to convert the Java class code into an XML (eXtensible Markup Language) file. Then, we traverse the XML in depth-first search. Once there is a method or a nested class, we will remove its internal element with a "body" tag. After that, we use the modified XML file to rebuild the Java class. The script we used to reduce the Java class code can be found in our code repository. Figure 3 shows an example of reduced class content.

3.2.2 Structure-based traversal of AST for Java class

Hu et al. [6] propose a structure-based traversal (SBT) which flattens the AST to a sequence and splice the nodes type and name with underline. Though SBT has achieved excellent performance on the code summarization task, it has some limitations. Firstly, SBT contains the vocabulary explosion problem. Secondly, for many AST nodes, their names are often complex tokens under camel case rule, but SBT ignores these words. Thirdly, the average number of nodes in an AST of a Java class can reach 305, it will cause the same problem with directly using original source code—the sequence is too long.

To tackle these limitations, we propose a structure-based traversal method for Java classes (CSBT). Comparing with SBT, the main difference of SBT and CSBT are as follows:

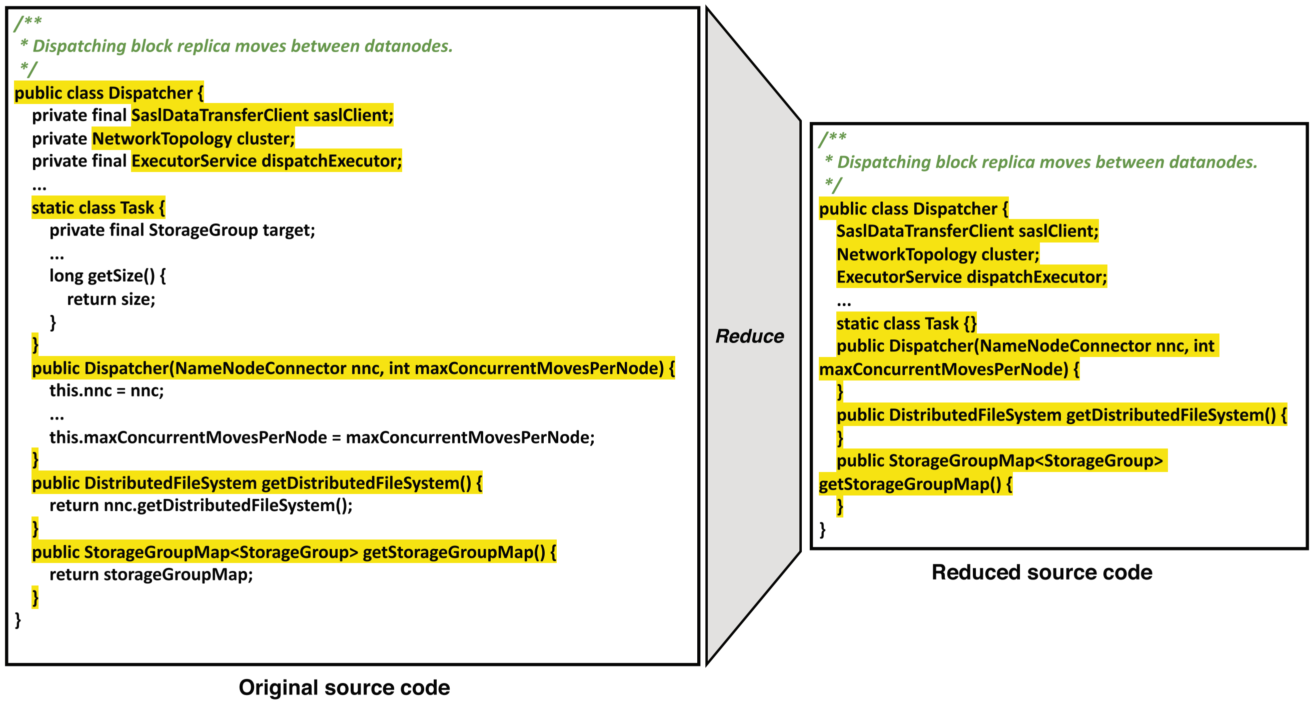
- CSBT is generated from AST of reduced code with in class rather than original source code.
- We split the names of AST nodes to sub tokens. For example, we split "NetworkTopology" to "network" and "topology".

The steps for generating CSBT sequence are shown in Fig. 4. We use javalang to parse the reduced source code to an AST. Then the AST is flattened into a sequence includes the names and types of the nodes in the AST. Names of nodes in the AST are split to sub tokens and concatenate them to CSBT sequence.

After data preprocessing, we get the reduced code token sequence $(x_1^{lex}, x_2^{lex}, \ldots, x_n^{lex})$ and the CSBT sequence $(y_1^{csbt}, y_2^{csbt}, \ldots, y_m^{csbt})$. For any i, x_i^{lex} and y_i^{csbt} are the serial numbers in the corresponding vocab. The inputs of ClassSum are shown in Fig. 5. Then we utilize word embedding layers to convert them into continuous vectors $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n})$ and $(\boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_m})$, where $\boldsymbol{x_i}, \boldsymbol{y_i} \in R^d$, where d is the embedding size.

3.3 Lexical encoder and syntax encoder

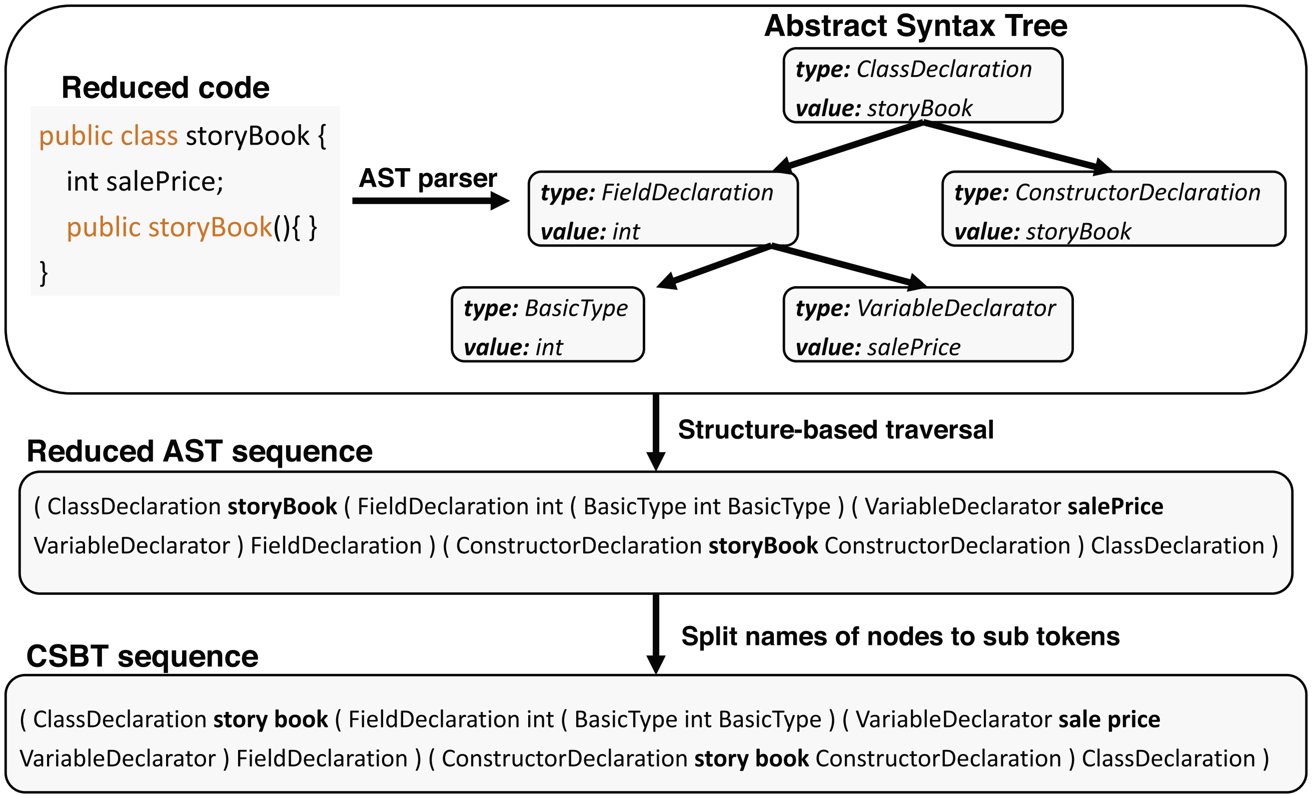
The lexical encoder is a bidirectional LSTM with double layers. Given the embedding vector sequence of reduced code tokens $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n})$, we utilize a bidirectional LSTM to encode it to produce hidden states. To be specific, at each encoding time step t, we compute the hidden states and outputs from Eq. (9):



```
/**
 * Dispatching block replica moves between datanodes.
 */
public class Dispatcher {
  private final SaslDataTransferClient saslClient;
  private NetworkTopology cluster;
  private final ExecutorService dispatchExecutor;
  ...
  static class Task {
    private final StorageGroup target;
    ...
    long getSize() {
      return size;
    }
  }
  public Dispatcher(NameNodeConnector nnc, int maxConcurrentMovesPerNode) {
    this.nnc = nnc;
    ...
    this.maxConcurrentMovesPerNode = maxConcurrentMovesPerNode;
  }
  public DistributedFileSystem getDistributedFileSystem() {
    return nnc.getDistributedFileSystem();
  }
  public StorageGroupMap<StorageGroup> getStorageGroupMap() {
    return storageGroupMap;
  }
}
```

```
/**
 * Dispatching block replica moves between datanodes.
 */
public class Dispatcher {
  SaslDataTransferClient saslClient;
  NetworkTopology cluster;
  ExecutorService dispatchExecutor;
  ...
  static class Task {}
  public Dispatcher(NameNodeConnector nnc, int
maxConcurrentMovesPerNode) {
  }
  public DistributedFileSystem getDistributedFileSystem() {
  }
  public StorageGroupMap<StorageGroup>
getStorageGroupMap() {
  }
}
```

Fig. 3 An example of difference between original source code and reduced source code

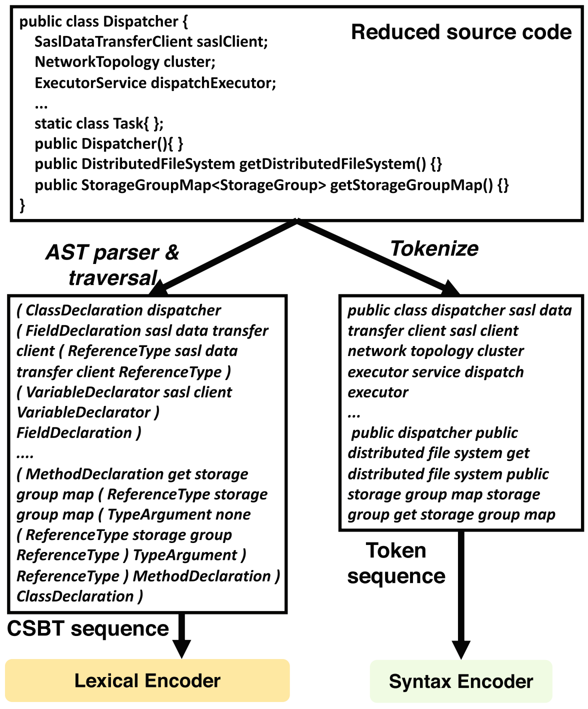


```
public class storyBook {
   int salePrice;
   public storyBook(){ }
}
```

(ClassDeclaration **storyBook** (FieldDeclaration int (BasicType int BasicType) (VariableDeclarator **salePrice** VariableDeclarator) FieldDeclaration) (ConstructorDeclaration **storyBook** ConstructorDeclaration) ClassDeclaration)

(ClassDeclaration **story book** (FieldDeclaration int (BasicType int BasicType) (VariableDeclarator **sale price** VariableDeclarator) FieldDeclaration) (ConstructorDeclaration **story book** ConstructorDeclaration) ClassDeclaration)

Fig. 4 The CSBT sequence generating steps



```
public class Dispatcher {
   SaslDataTransferClient saslClient;
   NetworkTopology cluster;
   ExecutorService dispatchExecutor;
   ...
   static class Task{ };
   public Dispatcher(){ }
   public DistributedFileSystem getDistributedFileSystem() {}
   public StorageGroupMap<StorageGroup> getStorageGroupMap() {}
}
```

(ClassDeclaration dispatcher (FieldDeclaration sasl data transfer client (ReferenceType sasl data transfer client ReferenceType) (VariableDeclarator sasl client VariableDeclarator) FieldDeclaration) .... (MethodDeclaration get storage group map (ReferenceType storage group map (TypeArgument none (ReferenceType storage group ReferenceType) TypeArgument) ReferenceType) MethodDeclaration) ClassDeclaration)

public class dispatcher sasl data transfer client sasl client network topology cluster executor service dispatch executor ... public dispatcher public distributed file system get distributed file system public storage group map storage group get storage group map

Fig. 5 The inputs to ClassSum

$$\overrightarrow{h_t}, \overleftarrow{h_t}, \overrightarrow{o_t}, \overleftarrow{o_t} = LSTM_{lexical}(x_1, x_2, \ldots, x_n) \tag{10}$$

where $\overrightarrow{h_t}$ is the forward hidden state and $\overleftarrow{h_t}$ is the backward hidden state, $\overrightarrow{o_t}$ is the forward output and $\overleftarrow{o_t}$ is the backward output. Then we concatenate the two directions to h_t^{lex} and o_t^{lex}:

$$\begin{aligned} h_t^{lex} &= [\overrightarrow{h_t}; \overleftarrow{h_t}] \\ o_t^{lex} &= [\overrightarrow{o_t}; \overleftarrow{o_t}] \end{aligned} \tag{11}$$

Finally, the lexical encoder outputs the final hidden state $h^{lex} = h_n^{lex}$ and the output sequence $o^{lex} = (o_1^{lex}, o_2^{lex}, \ldots, o_n^{lex})$.

The syntax encoder is similar to the lexical encoder. Given the embedding vector sequence of CSBT $(y_1, y_2, \ldots, y_m)$, it outputs the final hidden state $h^{csbt} = h_m^{csbt}$ and the output sequence $o^{csbt} = (o_1^{csbt}, o_2^{csbt}, \ldots, o_m^{csbt})$.

3.4 Decoding

To capture outputs from lexical encoder and syntax encoder, we use a joint decoder. The joint decoder mainly consists of a uni-direction LSTM, two attention mechanisms and a switch network.

We use an average pooling to compute the initial state of the LSTM from the hidden state $\boldsymbol{h^{lex}}$ produced by the lexical encoder and the hidden state $\boldsymbol{h^{csbt}}$ computed by the syntax encoder:

$$\boldsymbol{s_0} = \frac{\boldsymbol{h^{lex}} + \boldsymbol{h^{csbt}}}{2} \tag{12}$$

At each decoding time step t, we compute the context vector $\boldsymbol{c_t^{lex}}$ and $\boldsymbol{c_t^{csbt}}$ by the lexical attention layer and syntax attention layer:

$$\begin{aligned} \boldsymbol{c_t^{lex}} &= \sum_{j=1}^{n} \alpha_{tj}^{lex} \boldsymbol{h_j^{lex}} \\ \boldsymbol{c_t^{csbt}} &= \sum_{j=1}^{m} \alpha_{tj}^{csbt} \boldsymbol{h_j^{csbt}} \end{aligned} \tag{13}$$

where α_{tj}^{lex} is the weights of current hidden state $\boldsymbol{s_t}$ on output sequence $\boldsymbol{o^{lex}}$ of lexical encoder and α_{tj}^{csbt} is the weights of current hidden state $\boldsymbol{s_t}$ on output sequence $\boldsymbol{o^{csbt}}$ of syntax encoder.

The α_{tj}^{lex} and α_{tj}^{csbt} are computed as :

$$\begin{aligned} \alpha_{tj}^{lex} &= \frac{exp(\boldsymbol{s_t W_a^{lex} o_j^{lex}})}{\sum_{k=1}^{n} exp(\boldsymbol{s_t W_a^{lex} o_k^{lex}})} \\ \alpha_{tj}^{csbt} &= \frac{exp(\boldsymbol{s_t W_a^{csbt} o_j^{csbt}})}{\sum_{k=1}^{m} exp(\boldsymbol{s_t W_a^{csbt} o_k^{csbt}})} \end{aligned} \tag{14}$$

where $\boldsymbol{W_a^{lex}}$ and $\boldsymbol{W_a^{csbt}}$ are the parameters of attention layers.

There are two pairs of encoder and attention in ClassSum, thus we need to combine the output context vectors of attention mechanisms. To achieve this, previous studies use sum pooling [16], simple concatenation [5], pointer network [19], mixture network [20], or switch network [21]. After trying these methods on our dataset, we finally choose switch network to combine the context vectors according to the experimental results. According to the original switch network paper [21], the encoders are pre-trained at first, but we find that switch network can get ideal performance without pre-training.

In ClassSum, switch network is used to dynamically combine the context vectors of different code representations:

$$c_t = tanh(\boldsymbol{p} \times \widetilde{\boldsymbol{c_t^{lex}}} + (1 - \boldsymbol{p}) \times \widetilde{\boldsymbol{c_t^{csbt}}}) \tag{15}$$

where $\boldsymbol{p}$ is the weight vector computed with $\boldsymbol{s_t}$, $\widetilde{\boldsymbol{c_t^{lex}}}$ and $\widetilde{\boldsymbol{c_t^{csbt}}}$ are linear transformations of $\boldsymbol{c_t^{lex}}$ and $\boldsymbol{c_t^{csbt}}$:

$$\begin{aligned} \boldsymbol{p} &= \delta(\boldsymbol{W_p s_t} + \boldsymbol{b_p}) \\ \widetilde{\boldsymbol{c_t^{lex}}} &= \boldsymbol{W_t^{lex} c_t^{lex}} + \boldsymbol{b_t^{lex}} \\ \widetilde{\boldsymbol{c_t^{csbt}}} &= \boldsymbol{W_t^{csbt} c_t^{csbt}} + \boldsymbol{b_t^{csbt}} \end{aligned} \tag{16}$$

δ is the sigmoid function to ensure that the value of each element in $\boldsymbol{p}$ is between 0 and 1. Finally, we use $\boldsymbol{c_t}$ and $\boldsymbol{s_t}$ to compute the next word probability distribution:

$$p(w_t | w_{<t}, \boldsymbol{x}, \boldsymbol{y}) = softmax(\boldsymbol{W_s} tanh([\boldsymbol{c_t}, \boldsymbol{s_t}])) \tag{17}$$

where $\boldsymbol{W_s}$ is the parameter of output layer. In the training mode, the decoder takes the t_{th} word of gold truth at decoding time step t. After decoding, we get the probability distribution sequence $(p_1, p_2, \ldots, p_m)$, and we use cross-entropy to compute the loss:

$$loss = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{l} log\, p(w_j^{(i)}) \tag{18}$$

where N is the number of training samples, l is the length of gold truth sequence, and $w_j^{(i)}$ means the j_{th} word of the i_{th} sample. The loss function describes how much the predicted probability diverges from the gold truth. In the testing mode, we use **BOS** as the first input word to the joint decoder. Then it will generate words step by step until it generates **EOS** or the decoding step reaches the maximum step. Figure 6 shows the difference between training mode and testing mode of decoder.

4 Data preparation and experiment setup

In this section, we will introduce the details of data preparation and the experiment setup.

4.1 Data preparation

As we've mentioned in Sect. 1, there is a lack of large-scale datasets of class-level code summarization. To

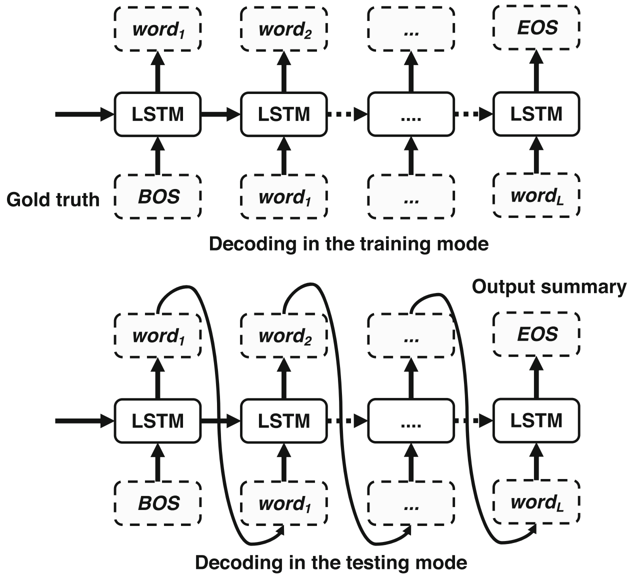


Fig. 6 Difference of training mode and testing mode of decoder

address the issue, we build a dataset from thousands of Java project repositories hosted on Github ranked by the number of stars. However, not all repositories are suitable for code summarization tasks such as low-quality projects or private projects. We filter repositories by following rules.

Firstly, we remove projects with less than 15 stars. Next, we remove the projects that are not public development project. "Public development project" means that the project should be open to public and its content should be about software development. The reason why we take this rule is that many projects hosted on Github are not for software development (e.g., blogs, translation and student homework) or private. We use the words and phrases in projects name and README files that can reflect projects' properties [22] to filter the repositories. Finally, the projects with large amount of summaries duplication are removed. We found that the project with high summaries duplication proportion generally have a default summary sentence. These duplicate summaries will artificially inflate the reported scores. For example, in the *ModularMachinery* [23] project, the default summary of each class is "This class is part of the Modular Machinery Mod.", which is an useless sentence. To overcome the issue, we drop projects whose duplication is bigger than 0.5. After filtering projects with above steps, we get 3185 projects. Figure 7 shows how we filter the download projects. We download source code of projects from Github and use keywords and phrases to filter the projects according to their names and README files, so that we can select the public development projects. Next, we analyze the summaries duplication proportion in each project we've selected, and only adopt those with a relatively low summary duplication proportion. Duplicated and noise samples summaries are also removed. The collected corpus is split to training set, validation set, hybrid-projects test set and cross-projects test set.

After filtering the projects, we utilize SrcML tool [18] to extract the classes and corresponding Javadoc in the projects and utilize javalang [24] to extract summaries sentences from Javadoc. We drop illegal samples according to some rules based on previous literature related to datasets of code summarization. The rules for filtering samples are as follows:

- We remove the classes for testing [6]. The samples that the class signatures or summaries contain the word "test" are removed.
- We remove the automatically generated class [5]. The samples with summaries that contain the words "generated", "created with", etc. are removed.
- We remove samples with too short code or summaries [25]. The samples with code length less than 3 lines or summaries length less than 2 words are removed.
- We remove duplicate samples [20]. This is because that duplicated samples will inflate the reported scores, thus all duplicated samples should be removed to make sure that the results are reliable.

In addition, there are some filtering rules proposed by us for class summaries:

- We remove summaries that begin with "abstract" or "implement", which are typically used to describe inheritance relationships between Java classes. Note that our work focuses on functionality summaries generating, thus these summaries are not within the scope of our research.
- We remove summaries that contain the author's information. Unlike method-level summaries, many developers ignore the Javadoc standard and mark their personal information in the first sentence of documentation. This troubles us a lot. We finally decided to remove all samples that contain the word "author", "copyright", or any timestamps. These samples are useless for the code summarization task.
- We remove summaries that contain self-admitted technical debt [26] such as "TODO:..." and "FIXME:...".

Finally, we get 172,639 <class, summary> pairs. We use javalang to tokenize the source code, and split the variable names according to the camel case naming and snake naming, which alleviates the excessive gap between the source code vocab and the natural language vocab. Non-integer numbers, MD5 and hash values are replaced by a special token <NUM> [20]. All original source code sequences are truncated to keep the number of tokens be less than 900 while 200 of reduced source code. For summaries, dates, numbers, HTML labels, URLs, etc. are removed. We tokenize summaries utilizing NLTK [27]. All summaries are truncated to keep the number of tokens is less than 20. The sizes we specified in the before steps are a trade-off between data distribution and memory size. The statistics of the dataset (before truncation) are shown in Table 1.

The parallel corpus is split by projects. We randomly select 70% of the pairs for training, 10% of the pairs for validation, 10% for hybrid-projects testing, 10% for cross-projects test. Thus, there are four subsets in our dataset: a training set, a validation set, a hybrid-projects test set, and a cross-projects test set. The samples within the cross-projects test set are in different projects from other sets. The validation set and the training set contain samples from same projects. This is because we use early stopping to control the training process according to the performance of model on validation set. A cross-projects validation set

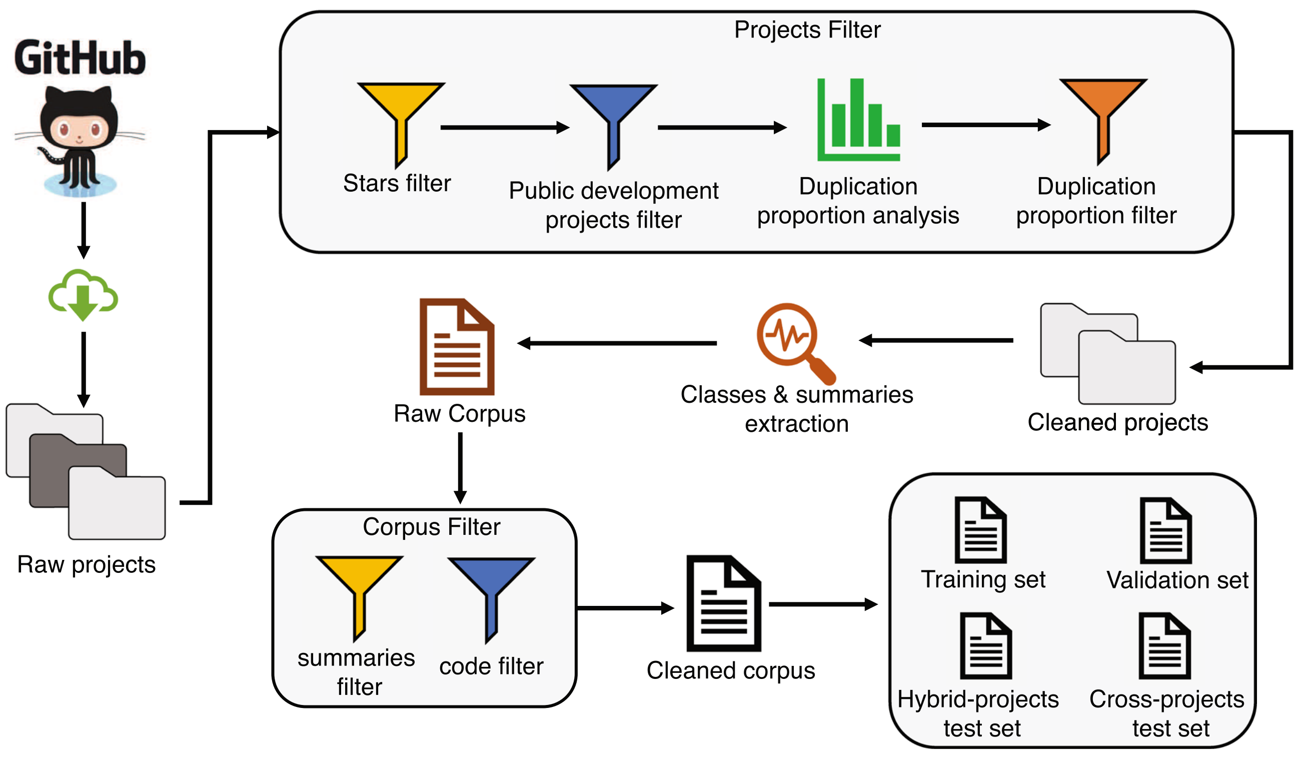


Fig. 7 Dataset preparation

Table 1 Statistics of the dataset (before truncation)

Lengths	Max	Avg	25%	50%	75%
Code	146,503	826	151	361	852
Code (reduced)	68,980	311	49	87	162
Summary	50	11	6	9	14
AST	42,442	305	49	128	316
AST (reduced)	35,592	80	21	41	84

Table 2 Splits of dataset

Dataset	Samples	Projects
Hybrid-projects test set	15,538	2188
Cross-projects test set	17,266	343
Validation set	15,537	2174
Training set	124,298	2831
Total	172,639	3185

may lead to inadequate training. The split results of our dataset are shown in Table 2.

4.2 Research questions

In this section, we evaluate effectiveness and accuracy of ClassSum by comparing it with state-of-the-art baselines in method-level code summarization. To be specific, we focus on the following research questions:

RQ1: How effective is the ClassSum compared with the state-of-the-art baselines introduced in Sect. 4.3?

This RQ is designed to investigate the performance of ClassSum. To answer this RQ, we compare ClassSum with some state-of-the-art models of method-level code summarization.

RQ2: How effective are the different encoders of ClassSum?

This RQ is designed to investigate the effectiveness of the two encoders in ClassSum. To answer this question, we perform some ablation studies on ClassSum by only using one of the lexical encoder and CSBT encoder.

RQ3: What is the difference between using reduced source code and using original source code?

This RQ is designed to investigate the impact of using reduced source code. To answer this RQ, we compared the results of using reduced code or using the original code.

RQ4: How does the dataset split strategy affect the performance of ClassSum?

This RQ is proposed to investigate the impact of dataset splits. LeClair et al. [5] show that, in method-level code summarization, not splitting by project is artificially inflates the reported scores. We believe that this phenomenon still occurs in the class-level code summarization task. To answer this RQ, we test the performance of our model and baselines on both the hybrid-projects test set and the cross-projects test.

RQ5: How do source code and summary length affect the performance of ClassSum?

This RQ is designed to investigate the impact of source code and summary with different length on the performance of ClassSum. To answer this RQ, we collect generated summaries and analyze the experimental results of different summary length and source code length.

4.3 Baselines

In this paper, we take five state-of-the-art method-level code summarization approaches as our baselines.

- *CodeNN* [8]. CodeNN is an end-to-end code summarization approach. It is one of the earliest deep learning-based approach. CodeNN use LSTM decoder to generate summaries given code snippets. The decoder generates words by applying the attention mechanism, which directly computes a weighted sum of tokens embeddings of source code.
- *SBT* [6]. SBT is an approach to convert the AST of source code into flatten sequence in a special format. Then the flatten sequence is fed to a standard Seq2Seq model.
- *AST-attendgru* [5]. AST-attendgru is an approach which uses both original source code and SBT as input. The approach builds a GRU-based-encoder–decoder model with two encoders and one decoder. At each time step of decoding, attention mechanism is applied between the outputs of all the encoders and the decoder.
- *Transformer* [28]. Transformer is an attention-only machine translation model. It has made significant improvement to NMT. With the success of the model, we take it as a baseline. In this paper, we use the model architecture and hyper-parameters that released by Ahmad [29].
- *PLBART* [30]. PLBART is a self-supervised model based on the BART [31] architecture. It was pre-trained on an extensive collection of Java and Python methods and associated NL text via denoising auto-encoding. The pre-trained model can be fine-tuned on code summarization task. The training corpus of PLBART is a method-level corpus.

4.4 Evaluation metrics

We choose automatic metrics (i.e., BLEU [32], METEOR [33] and ROUGE-L [34]) to evaluate the effectiveness of ClassSum. These metrics have been widely used in previous code summarization work.

4.4.1 BLEU

BLEU is a widely used metric for calculating the similarity between the generated sentence and the reference sentence. Given a generated sentence, BLEU calculate the weighted geometric mean of n-gram matching accuracy scores multiplied by the brevity penalty to short generated sentences. It is computed as:

$$BLEU = BP \cdot \left(\sum_{i=0}^{n} w_n log\, p_n \right) \tag{19}$$

where p_n is the geometric average of the modified n-gram precision, w_n is the positive weights, and BP is the brevity penalty. BP is computed as follows:

$$BP = \begin{cases} 1 & c > r \\ e^{1-r/c} & c \leq r \end{cases} \tag{20}$$

where c is the length of the candidate and r represents the length of the reference. In this paper, we also take smoothed BLEU as an evaluation metric [35], and we set $N = 4$ and $w_i = 0.25(i = 1, 2, 3, 4)$. The candidates are generated summaries and the references are the human-written summaries extracted from Javadoc. The implementation of BLEU is from NLTK [27].

4.4.2 METEOR

METEOR is a widely used machine translation-based metric which calculates sentence-level similarity scores between translation hypotheses and translation reference. It can be regarded as an improvement of BLEU. It applies synonym matching to BLEU and takes recall into account. To be specific, METEOR is computed as follows:

$$METEOR = (1 - P_{en}) \cdot F_{mean} \tag{21}$$

where P_{en} is the penalty coefficient, and F_{mean} is a parameterized harmonic mean. P_{en} is computed as follows:

$$P_{en} = \gamma \left(\frac{ch}{m} \right)^{\beta} \tag{22}$$

where m is the number of matching words, ch is the number of chunks. γ determines the maximum penalty $(0 \leq \gamma \leq 1)$. β determines the functional relation between the fragmentation and the penalty. In this paper, we follow the parameter settings for English and set γ and β to 0.20 and 0.60. The implementation of METEOR is also from NLTK.

4.4.3 ROUGE-L

ROUGE-L uses the length of longest common subsequence to compute the similarity between automatically generated summary and human-written summary. Giving a reference summary X of length r and a candidate summary Y of length c, it is computed as:

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

where LCS represents the length of the longest common subsequence of the candidate summary and reference summary. In this paper, we set $\beta = 1$. The reason why using LCS is that it does not require consecutive matches but in-sequence matches that reflect sentence level word order as n-grams. The implementation of ROUGE-L is from py-rouge [36].

4.5 Experiments setup

The parameters of our model settings are described as follows:

- The hidden size of LSTM in encoders is set to 256. The dimensions of embedding layers in encoders and the decoder are set to 128.
- The vocab size is set dynamically. This is because the reduced code and the original code vocab may differ. There are three vocabs in ClassSum, a vocab for reduced source code, a vocab for summaries and a vocab for CSBT. The statistics of vocabs with different minimize frequency are show in Table 3. In our experiments, we use the vocab with $min_{freq} = 2$.
- We use Adam [37] with initial the learning rate of 0.001 to minimize the loss on training set. The batch size in training procedure is set to 64.We clip the gradient norm by 5 and use dropout strategy during the training process and set dropout of 0.1.
- We set maximum number of epochs to 100, the maximum number of steps is 250,000, and early stopping with patience = 4 is used.
- We use beam search [38] with beam width 4 to sample the output probability of decoder.

The parameters of the baselines settings are described as follows:

- The hidden size of AST-attendgru, Seq2seq, CodeNN and DeepCom is set to 256, which is a appropriate value for our dataset.

Table 3 Vocab size with different minimize frequency

vocab suze	$freq = 1$	$freq = 2$	$freq = 3$
Code	67,227	50,511	41,366
Code(reduced)	40,700	28,576	23,259
sbt	55,786	41,622	34,180
csbt (reduced)	37,632	27,894	22,993
Summaries	20,064	13,086	10,505

$freq = 1$ means that the minimize frequency is 1

- The parameter configuration of Transformer keeps the same with the reference [29].
- PLBART is a bart-based pre-trained model, we utilize the released model [30] and fine-tune the model on our dataset.

The values of above parameters are optimized by the validation set. The experiments are implemented on GPU server with an Intel Gold 6248R CPU, 64 GB RAM and a RTX 3090 GPU with 24 GB memory, running Linux 4.15. Software and libraries used includes Python 3.8, PyTorch 1.8, CUDA 11 and CuDNN 7.1.

5 Result analysis

5.1 RQ1: How effective is ClassSum compared with the baselines?

We evaluate ClassSum on our dataset and compare it with the baselines. The evaluation is progressed on both cross-projects test set and hybrid-projects test set. We measure the gap between the automatically generated summaries and human-written summaries through automatic metrics, i.e., BLEU, METEOR and ROUGE-L. Table 4 shows the results on cross-projects test set and Table 5 shows the results on test-projects test set. The rows in bold are the evaluation results of ClassSum, which outperform all baselines and ablation models.

As it is shown in Tables 4 and 5, ClassSum model performs the best both in the hybrid-projects test and cross-projects test. In the hybrid-projects test, it achieves a BLEU score of 15.27, a BLEU(s) score of 22.25, a ROUGE-L score of 39.97, and a METEOR score of 20.42. In the cross-projects test, it achieves a BLEU score of 5.83, a BLEU(s) score of 11.78, a ROUGE-L score of 28.03, and a METEOR score of 13.46. More clearly, comparing with the best baseline, i.e., AST-attendgru, in the hybrid-projects test, the ClassSum outperforms AST-attendgru by 20.90% on BLEU, 13.00% on of BLEU(s), 5.30% on ROUGE-L, 6.78% on METEOR. In the cross-projects test,

Table 4 Overall performance of our model and baselines on hybrid-projects test set, BLEU(s) denotes the smoothed BLEU-4, code(r) denotes the reduced code

Model	Input	BLEU	BLEU(s)	METEOR	ROUGE-L
CodeNN	code	7.99	12.81	12.07	24.82
Transformer	code	9.06	14.95	15.97	31.13
SBT	SBT	11.41	18.82	18.59	37.62
PLBART	code	11.67	18.13	19.78	37.59
AST-attendgru	code + SBT	12.63	19.69	19.09	37.96
ClassSum	**code(r) + CSBT**	**15.27**	**22.25**	**20.42**	**39.97**
ClassSum (no switch)	code(r) + CSBT	14.05	21.31	19.86	39.15
ClassSum (only lexical)	code(r)	13.45	20.57	19.58	38.76
ClassSum (only syntax)	CSBT	12.86	20.00	19.26	38.41

Table 5 Overall performance of our model and baselines on cross-projects test set

Model	Input	BLEU	BLEU(s)	METEOR	ROUGE-L
CodeNN	code	2.81	6.13	6.76	14.56
Transformer	code	3.07	7.49	10.24	20.13
SBT	SBT	3.46	9.92	12.60	26.20
PLBART	code	3.19	9.43	13.28	26.50
AST-attendgru	code + SBT	3.60	10.14	12.81	26.52
ClassSum	**code(r) + CSBT**	**5.83**	**11.78**	**13.46**	**28.03**
ClassSum (no switch)	code(r) + CSBT	4.50	10.82	13.29	27.55
ClassSum (only lexical)	code(r)	4.21	10.60	13.12	26.88
ClassSum (only syntax)	CSBT	3.67	10.23	12.98	26.85

the ClassSum outperforms AST-attendgru by 61.94% on BLEU, by 16.17% on BLEU(s), by 5.69% on ROUGE-L, 5.07% on METEOR. Another interesting finding is that the PLBART doesn't performance better than AST-attendgru. The reason is that PLBART is pre-trained on method-level corpus, while AST-attendgru is trained from random initialization and is trained on our class-level dataset. This indicates that the pre-trained models trained on method-level (or function-level) may not suitable for the task of class summarization.

To verify the competitiveness of ClassSum and other baselines, we conduct Wilcoxon signed-rank [39] test (a nonparametric uni-variate test) in terms of BLEU, METEOR and ROUGE-L metrics. The hypothesis are as follows, Null hypothesis (H_0): There this no significant difference between ClassSum and other baselines. Alternative hypothesis (H_A): The difference between ClassSum and other baselines is significant. If the calculated p-value is less than 0.05 (most commonly used), one can reject the null hypothesis and accept the alternative hypothesis. The p-values are shown in Table 6. The results show that all p-values are lower than 0.05. Then we can conclude that we should reject the null hypothesis, which means that there is a significant difference between our model and the baselines in terms of BLEU, METEOR and ROUGE-L metrics. Because Tables 4 and 5 have shown that our model performs better than the baselines, it's safe to say that ClassSum can achieve better performance.

Table 6 P-value of hypothesis H_0

Baselines	BLEU	BLEU(s)	METEOR	ROUGE-L
CodeNN	6.68e−10	1.86e−13	6.46e−23	8.17e−29
Transformer	1.91e−38	8.07e−25	1.28e−08	1.39e−15
SBT	8.02e−21	6.94e−32	1.63e−19	4.05e−24
PLBART	4.19e−16	1.25e−17	2.03e−09	3.23e−23
AST-attendgru	8.72e−34	1.16e−31	9.45e−08	8.64e−13

According to the listed results of the ClassSum against baselines, we can conclude that ClassSum can effectively capture the lexical and structure information from reduced code and CSBT, and generate higher quality summaries.

Answer for RQ1: The BLEU score of ClassSum can reach 15.27 (hybrid-projects test) and 5.83 (cross-projects test), the BLEU(s) score can reach 11.78 and 22.25, the METEOR score can reach 13.46 and 20.42, and the ROUGE-L score can reach 28.03 and 39.97. In class-level code summarization task, ClassSum outperforms the state-of-the-art method-level code summarization baselines by a significant improvement on all automatic evaluation metrics.

5.2 RQ2: How effective are the different encoders of ClassSum?

We perform ablation studies to examine the effectiveness of different components of ClassSum. As it's shown in Fig. 8 and lines 6–8 of Tables 4 and 5, if we only use the lexical encoder or syntax encoder, the performance of ClassSum will be significantly lower, either in cross-projects test or in hybrid-projects test. Moreover, we test the effectiveness of switch network. Experimental results show that, without the switch network, the effectiveness of ClassSum will be worse. The results indicate that the switch network is able to help ClassSum to better combine lexical and structural information, as switch network can be interpreted as a dynamic weights combination.

Answer for RQ2: The lexical encoder and CSBT encoder are both important to ClassSum. Without any of them, the performance of ClassSum score will become worse. Besides, with the switch network, ClassSum can better combine semantic and structural information.

5.3 RQ3: What is the difference between using reduced source code and using original source code?

The reduced code means that the bodies of all methods and nested classes within a class are removed. Compared to the original code with in class, the reduced code is shorter and easier for the model to extract useful features. We conducted comparative experiments using different NMT models Seq2seq, Transformer and ClassSum. The input to the models is either the original code or the reduced code. Tables 7 and 8 show the results on hybrid-projects test and cross-projects test.

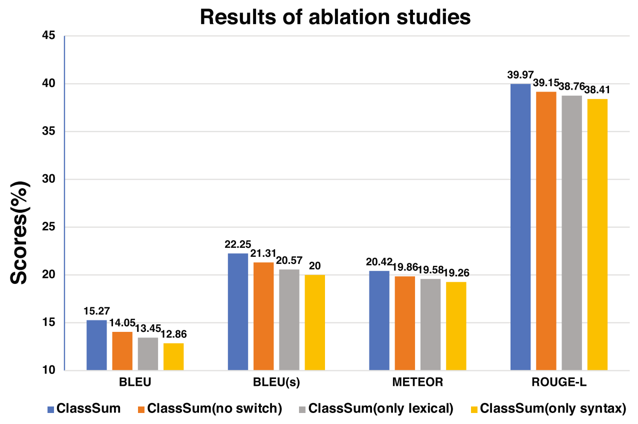


Fig. 8 The scores of metrics after removing different part of ClassSum. "No switch" means that we remove the switch network. "Only lexical(syntax)" means that we only use the lexical(syntax) encoder

According to the results, it's obvious that using reduced code performs better then original code, both in the cross-projects test and the hybrid-projects test. In the cross-projects test, with using reduced code, Seq2seq improves BLEU by 5%, Transformer improves BLEU by 7.9%, and ClassSum improves BLEU by 14.1% compared to using the original source code. In cross-projects test, Seq2seq model improves BLEU by 11.9%, Transformer improves BLEU by 12.7%, and ClassSum improves BLEU by 23%.

Answer for RQ3: In the class-level code summarization task, taking reduced code as input performs better than original code. Signatures and attributes contain more important information for the summary. This is similar to the human's habits. When a developer try to understand a class, he usually first reads and focuses on the class attributes and methods signatures. Thus, code summarization model can extract features from the reduced source code more easily. In addition, the training cost of using reduced code is lower.

5.4 RQ4: How does the dataset split strategy affect the performance of ClassSum?

Figure 9 shows the gap between cross-projects test and hybrid-projects test. In the cross-projects test, for ClassSum, the scores decrease 62% in terms of BLEU, 47% in terms of BLEU(s), 34% in terms of METEOR, and 30% in terms of ROUGE-L. For AST-attendgru, the scores decrease 71% in terms of BLEU, 49% in terms of BLEU(s), 33% in terms of METEOR, and 30% in terms of ROUGE-L. For SBT, the scores decrease 70% in terms of BLEU, 47% in terms of BLEU(s), 32% in terms of METEOR, and 30% in terms of ROUGE-L. This indicates that in the cross-projects test, the code summarization model will lead to worse performance. The reason is that there are a lot of similar code snippets and code reuse in same projects. The lack of similar code results in the gap between the scores of cross-projects test set and the training test set.

Answer for RQ4: In the cross-projects test, the BLEU score of ClassSum drops 62% than in the hybrid-projects test while other baselines drop more. This indicates that the split strategy of dataset can seriously affect the performance of the code summarization model. Because there is little similar code between cross-projects test set and training set.

5.5 RQ5: How do source code and summary length affect the performance of ClassSum?

We further analyze the effect of code length and summary length on the class-level code summarization model. Figures 10 and 11 show the experimental results.

Table 7 Comparison of performance between using reduced code and original code in hybrid-projects test

Model	Input	BLEU	BLEU(s)	METEOR	ROUGE-L
Seq2seq	code	9.99	16.59	16.40	33.19
Seq2seq	code(r)	10.63	17.13	16.67	33.77
Transformer	code	9.06	14.95	15.97	31.13
Transformer	code(r)	9.78	16.03	16.72	32.27
ClassSum	code + SBT	13.38	19.84	19.97	38.08
ClassSum	code(r) + CSBT	15.27	22.25	20.42	39.97

BLEU(s) denotes the smoothed BLEU-4, code(r) denotes the reduced code

Table 8 Comparison of performance between using reduced code and original code in cross-projects test

Model	Input	BLEU	BLEU(s)	METEOR	ROUGE-L
Seq2seq	code	3.35	8.68	10.82	22.81
Seq2seq	code(r)	3.75	9.33	11.39	23.53
Transformer	code	3.07	7.49	10.24	20.13
Transformer	code(r)	3.46	8.92	10.79	20.96
ClassSum	code + SBT	4.74	10.62	12.23	27.42
ClassSum	code(r) + CSBT	5.83	11.78	13.46	28.03

From Fig. 10, we can find a clear relationship between the summary length and the performance of the model. In the hybrid-projects test, ClassSum performs best when the summary length is 4, and 3 of cross-projects test. However, ClassSum performs worse when the summary length is too short or too long. For different summary lengths, ClassSum works better than the baselines.

However, either in both hybrid-projects or cross-projects test, Fig. 11 indicates that there is no significant relationship between the performance of ClassSum and the code length. But for different code lengths, ClassSum always works better than the baselines.

Answer for RQ5: The effect of code length is not significant, while the summary length has a large impact on the performance of class-level code summarization model. ClassSum works best when the summary length is 3 or 4, and worst when the summary length is too short or too long. However, in most cases, ClassSum has a better performance than the baselines.

5.6 Discussions

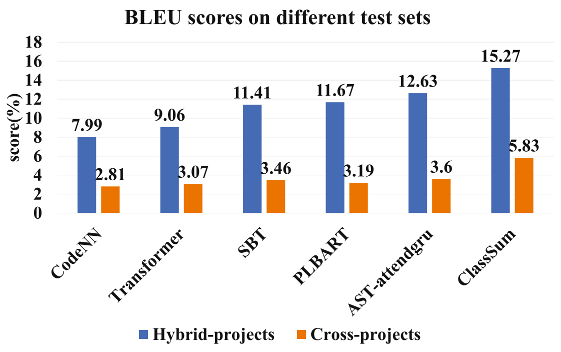
5.6.1 Output examples and error analysis

Table 9 shows some output examples generated by ClassSum and baselines (all derived from cross-project test). It's clear that compared with other baselines, our model generates the closest summaries to the gold truth.

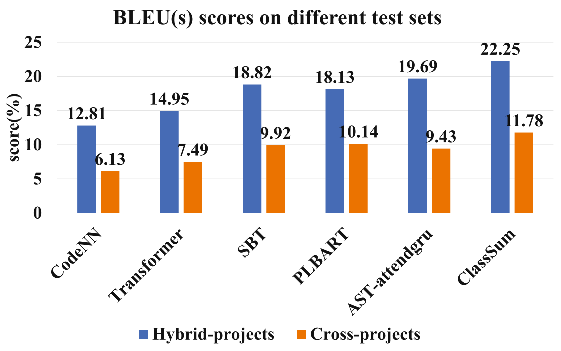
In the first example, the class signature is "discrete normal distribution". ClassSum is able to capture this information and add the quotation "on a finite set". While AST-attendgru fails in extracting "discrete", and SBT ignores the quotation "on a finite set". PLBART outputs an action-description, which is more likely an method summary. This is because both SBT and AST-attendgru use the original source code of the class as input, which makes it's difficult to extract useful features. ClassSum, on the other hand, uses a reduced source code, which can better assist the model to extract information features.

For the second example, ClassSum and the baselines are able to capture the keywords "bytes" and "iterator". However, the statement generated by SBT is simpler, and just a simple copy of the class signature. ClassSum is able to add "a sequence of" to bytes because it captures the "next" method signature of the iterator and determines that it is a sequence while AST-attendgru ignore this information. And PLBART outputs a method-like summary with the word "return".

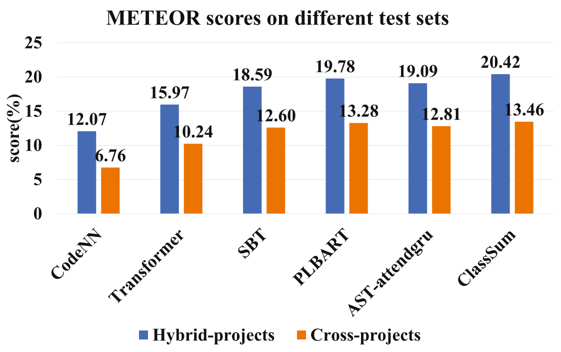
The third example is an error analysis where ClassSum fails to extract the keyword "sql server" and throws a <unk>. AST-attendgru captures this information and uses the synonym "ms sql", but it does not catch the keyword "update". SBT may realize the keywords "sql server", but it uses the word "database" instead, and the word "update" is not captured. And PLBART captures the keyword "HmilySQLServer", benefiting from its BPE-based tokenizer. The reason why ClassSum fail in this example is that in the third example, the original source code is similar to reduced source code. Thus, ClassSum is degraded and does not perform significantly better than the two baselines.



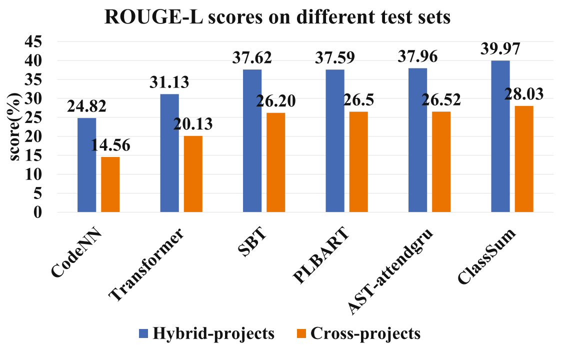
(a) BLEU



(b) BLEU(s)

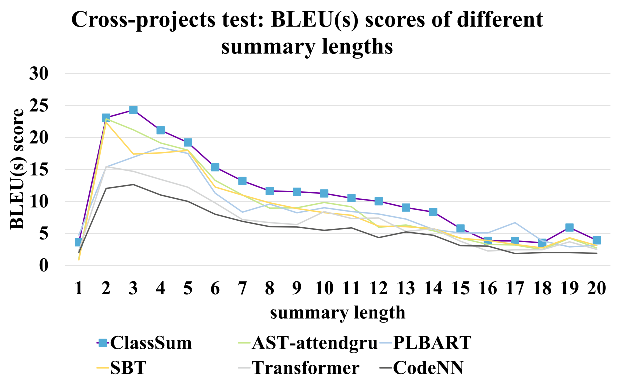


(c) METEOR

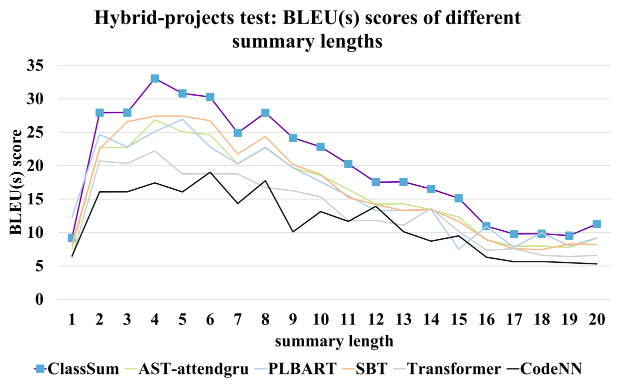


(d) ROUGE-L

Fig. 9 The scores of metrics of different test sets



(a) hybrid-projects test



(b) cross-projects test

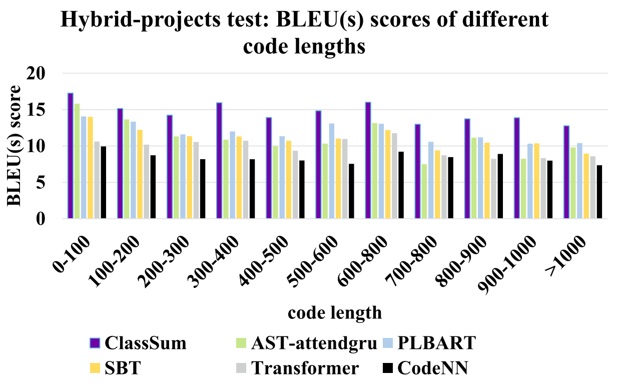
Fig. 10 The scores of BLEU(s) of different summary lengths

5.6.2 Time and memory cost

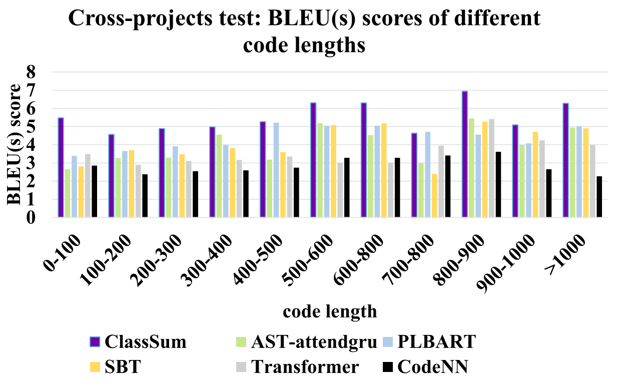
During the training process, we record the training time for per epoch, the memory usage of GPU, and the size of parameters of models. Table 10 shows the results.

In terms of training time, ClassSum takes more time to train one epoch than CodeNN. Because CodeNN have only an embedding-based encoder. However, we find that the time for training one epoch of ClassSum is less than PLBART, Transformer, SBT and AST-attendgru. The reason is that these models takes the long original source code as input. Besides, Transformer has too much parameters.

In terms of memory required for training, ClassSum occupies more GPU than CodeNN, but less than other baselines. The reason is same with above. An interesting finding is that: Though the Transformer model has a horrible 141 million parameters, occupying 18.9 GB GPU-RAM, but it does not outperform the LSTM model, which may be caused by the scale of the dataset. Transformer is more suitable for super-large scale dataset sets and pre-training tasks. This is one of our endeavors. We will



(a) hybrid-projects test



(b) cross-projects test

Fig. 11 The scores of BLEU(s) of different code lengths

expand our dataset and do super-large scale pre-training for class-level code summarization.

6 Threats to validity

There are following threats to validity in this paper.

6.1 Internal validity

To compare ClassSum with the baselines, we re-implement the baselines by ourselves. We have carefully checked our code and do many tests. However, the hyper-parameters settings of the method-level code summarization baselines may be mismatching with our dataset. To alleviate this threat, we have tried to adjust the parameters of these baselines as much as possible to make them perform better. For example, we use double layers for CodeNN instead of single layer used in the original paper and it boost BLEU from 4.60 to 7.99 on hybrid-projects test set and 1.60 to 2.81 on cross-projects test set. We adjust the hyper-parameters and settings for baselines and our model on the validation set.

6.2 External validity

The quality of our dataset relates to threats to external validity. Our corpus is downloaded from thousands of repositories from Github, but not all of these repositories are public software development project, such as blogs, notes, etc. The summaries in such repositories may not suitable for training code summarization model. We have tried our best to define heuristic rules to filter non-public software development repositories, but they may still exist in our dataset, which will pose threat to validity of our results. We regard the first sentence in the Javadoc extracted from the source code as gold truth. However, due to various reasons, this sentence may be meaningless or even wrong (e.g., “A Java class”). This is more obvious in the doc of class because some developers write their personal information in the first sentence. To address this issue, we have defined many rules to drop these dirty samples, but we may not be able to eliminate all the dirty samples. The potential low-quality summaries in our dataset is another external threat to this work. And some summaries are located on main function methods of classes, but we just ignore these samples. In the future work, we will use more effective methods for data extraction and filtering.

6.3 Construct validity

The construct validity is related to automatic metrics. We use automatic metrics, BLEU, ROUGE-L, METEOR to evaluate the generated summaries quickly, inexpensively and objectively. These metrics have been widely used in the machine translation tasks. However, informativeness and naturalness of summaries should also be considered besides the similarity to references. In our future work, we will take human evaluation into account.

7 Related works

Code summarization means automatically generating brief descriptions for source code. In recent years, this task has attracted a lot of attention and efforts in software engineering [4–6, 10, 15, 21, 40] and artificial intelligence [8, 9, 20, 25, 29, 41, 42].

In the past, the code summarization models are based on template and information retrieval techniques. For method or functions, Hill et al. [43, 44] use text retrieval (TR) methods, to capture source code semantics and generate summaries. Sridhara et al. [45–47] propose an approach to select import parts of Java methods called S_unit. Then they utilize Software Word Usage Model (SWUM) to

Table 9 Output examples of different models

ID	Source code	Generated summaries

1	`public class DiscreteUniformDistribution extends`
	`Distribution{`
	`  double values;`
	`  public DiscreteUniformDistribution(double a,`
	`  double b, double w){`
	`    setParameters(a, b, w);`
	`  }`
	`  public DiscreteUniformDistribution(){`
	`    this(1, 6, 1);`
	`  }`
	`  public void setParameters(double a,`
	`  double b, double w){`
	`     super.setParameters(a, b, w, DISCRETE);`
	`  }`
	`  public double getMaxDensity(){`
	`    return 1.0 / getDomain().getSize();`
	`  }`
	`  public double simulate(){`
	`    return getDomain().getLowerValue()`
	`    + Math.random() * getDomain().getSize()`
	`    * getDomain().getWidth();}`
	`}`
		Gold Truth: The discrete uniform distribution on a finite set.
		ClassSum: A discrete uniform distribution on a finite set.
		AST-attendgru: A uniform distribution on a finite set.
		PLBART: Specifies a uniform distribution with the specified values.
		SBT: A discrete distribution.
		Transformer: A distribution on a set.

2	`public final class IteratorOfBytes implements`
	`Iterator<Byte> {`
	`  private final byte[] items;`
	`  private final AtomicInteger position;`
	`  public IteratorOfBytes(final Text txt) {`
	`    this(new BytesOf(txt));}`
	`  public IteratorOfBytes(final String str) {`
	`    this(new BytesOf(str));}`
	`  @Override`
	`  public boolean hasNext() {`
	`    return this.position.intValue()`
	`    < this.items.length;}`
	`  @Override`
	`  public Byte next() {`
	`    if (!this.hasNext()) {`
	`      throw new NoSuchElementException(`
	`        "The iterator doesn't have any more items"`
	`      );}`
	`    return this.items[this.position.getAndIncrement()];}`
	`}`
		Gold Truth: Iterator that returns a set of bytes.
		ClassSum: An iterator over a sequence of bytes.
		AST-attendgru: Iterator of bytes.
		PLBART Returns an iterator of bytes for the given text.
		SBT: An iterator that wraps a byte array.
		Transformer: Bytes array iterator.

3	`public final class HmilySQLServerUpdateStatement`
	`  extends HmilyUpdateStatement implements`
	
	`  HmilySQLServerStatement {`
	`    private HmilyWithSegment withSegment;`
	`    public Optional<HmilyWithSegment>`
	`    getWithSegment() {`
	`      return Optional.ofNullable(withSegment);`
	`  }`
	`}`
		Gold Truth: sql server update statement
		ClassSum: represents <unk> update statement
		AST-attendgru: defines a database statement
		PLBART: The HmilySQL server instance.
		SBT: represents a ms sql statement
		Transformer: An update database statement

extract the them words and action words of S_unit. The words are used to fill the templates and generate summaries. Wong et al. [48, 49] use code clone detect method to search similar code segments from Github and Stackoverflow and generate summaries from the comments of these similar code. For classes, Moreno et al. [4] combines stereotype-based information with pre-defined heuristic templates based on method and data member access levels, and then generates summaries for the Java classes. Malhotra et al. [50] utilize micro-patterns (i.e., templates) to generate summaries for Java classes. The disadvantage of the template-based approaches is that the quality of the generated summaries depends on the properties of projects and naming conventions of developers.

Recently, many deep learning-based approaches have emerged. Iyer et al. [8] first applied deep learning in this task and propose the CodeNN method, which is based on attentional LSTM in decoding. Allamanis et al. [41] use

Table 10 Time and memory cost during the training process

Model	Parameters (M)	Time (s/epoch)
CodeNN	18.6	62
Transformer	141.0	734
SBT	21.1	429
AST-attendgru	28.6	623
ClassSum	34.4	489
PLBART	139.22	1203
Model	**GPU-usage (MB)**	**Disk-size (MB)**
CodeNN	2651	73
Transformer	19,378	566
SBT	7417	84
AST-attendgru	7189	136
ClassSum	5977	137
PLBART	18,532	532

CNN to encode the source code and use attentional GRU to generate summaries. Hu et al. [6] create a traversal method called Structure-based Traversal (SBT) to serialize ASTs of source code, then they use attentional Seq2Seq model to generate summaries. LeClair et al. [5] propose the AST-attendgru method, which takes two representations of source code: tokens-based representations and SBT-based representations as input. The model consists two unidirectional GRU encoders: one for encoding tokens another for encoding SBT. After that, the attentional GRU decoder generates summaries. Ahmad et al. [29] and Yang et al. [40] use Transformer model to generate summaries. Liu et al. [42] and Zhou et al. [51] apply graph neural network (GNN) to encode the source code and generate summaries. Chen et al. [52] study how to classify the method summaries. And for each class, they study which summarization model performs better. Zhang et al. [53] use retrieval and similar code to improve the performance of code summarization model. LeClair et al. [54] and Zhou et al. [20] study how to ensemble the method-level code summarization models.

However, most data-driven and deep learning-based approaches for code summarization focus on method-level source code. It is worth studying how to generate summaries for classes through deep learning methods. There has been fewer related studies. Liu et al. [10] use Knowledge Graph to select documents of other APIs (e.g., interfaces, methods, classes and packages) as the query-specific summaries of a Java classes. This method is mainly used for code and APIs retrieval. Liu et al. [55] also train a method-level code summarization seq2seq model and generate a summary of each method in a Java class and train a classifier to select one of method summaries as the final summary of the class. Thus, Liu's works are not directly taking the source code of Java classes as input. So, we won't take their methods as baselines. To the best of our knowledge, we are the first to directly ("directly" means that our model takes the source code of Java class as input and generate the class-level functionality summaries.) apply deep learning based techniques to generating summaries for classes and release a parallel corpus for class-level code summarization.

8 Conclusion and future work

In this work, we provide a large dataset and propose a deep neural network model for class-level code summarization. Previous models of code summarization mainly focus on method code, and there has been few researches about class code. Code of class is different with method. Class code is often longer and longer than method code, thus the long input challenge neural model to handle long-range dependencies and thus make it difficult to extract useful features in the source code. Besides, the source code of method bodies within a class is not always visible to developers. To handle with this problem, we propose a method to reduce the source code of class, and utilize a structured traversal method to serialize the AST of reduced source code. Experiment results show that taking reduced code as input performs better than using the original code. We also design an encoder–decoder framework with a switch network for class-level code summarization. During encoding, the encoders are used to encode the lexical information and structure information of classes. In the decoding process, it utilizes switch network to combine the context vectors of attention, and generate summaries. Experimental results show that ClassSum performs the best than directly applying method-level summarization models to the classes.

In the future, we will expand the dataset with additional methods for projects filtering and data extraction. We believe that the proposed summarization model should have the abilities for cross-language training and intend to apply code reducing method and ClassSum to other programming languages. Another potential future work is using reinforcement learning or adversarial training methods to enhance the performance of summarization models, as the method-level code summarization models do [20, 56, 57].

Funding This work was supported by the National Natural Science Foundation of China (No. 61772200), Shanghai Natural Science Foundation (No. 21ZR1416300), Capacity building project of local universities Science and Technology Commission of Shanghai Municipality No. 22010504100.

Data Availability The datasets generated during and/or analysed during the current study are available in the Github repository, https://github.com/classsum/ClassSum.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2018) Measuring program comprehension: a large-scale field study with professionals. IEEE Trans Softw Eng 44(10):951–976. https://doi.org/10.1109/TSE.2017.2734091
2. He H (2019) Understanding source code comments at large-scale. IN: ESEC/FSE 2019. Association for Computing Machinery, New York, NY, USA, pp 1217–1219. https://doi.org/10.1145/3338906.3342494
3. Song X, Sun H, Wang X, Yan J (2019) A survey of automatic generation of source code comments: algorithms and techniques. IEEE Access 7:111411–111428. https://doi.org/10.1109/ACCESS.2019.2931579
4. Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: 2013 21st International conference on program comprehension (ICPC), pp 23–32. https://doi.org/10.1109/ICPC.2013.6613830
5. LeClair A, Jiang S, McMillan C (2019) A neural model for generating natural language summaries of program subroutines. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 795–806. https://doi.org/10.1109/ICSE.2019.00087
6. Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Proceedings of the 26th conference on program comprehension. ICPC'18. Association for Computing Machinery, New York, NY, USA, pp 200–210. https://doi.org/10.1145/3196321.3196334
7. Github: GitHub (2022). https://github.com/. Accessed 3 Oct 2022
8. Iyer S, Konstas I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers). Association for Computational Linguistics, Berlin, Germany, pp 2073–2083. https://doi.org/10.18653/v1/P16-1195
9. Liang Y, Zhu K (2018) Automatic generation of text descriptive comments for code blocks. In: Proceedings of the AAAI conference on artificial intelligence, vol 32. https://doi.org/10.5555/3504035.3504676
10. Liu M, Peng X, Marcus A, Xing Z, Xie W, Xing S, Liu Y (2019) Generating query-specific class API summaries. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. ESEC/FSE 2019. Association for Computing Machinery, New York, NY, USA, pp 120–130. https://doi.org/10.1145/3338906.3338971
11. Hindle A, Barr ET, Gabel M, Su Z, Devanbu P (2016) On the naturalness of software. Commun ACM 59(5):122–131. https://doi.org/10.1145/2902362
12. Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: NIPS'14. MIT Press, Cambridge, pp 3104–3112. https://doi.org/10.5555/2969033.2969173
13. Luong T, Pham H, Manning CD (2015) Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 conference on empirical methods in natural language processing. Association for Computational Linguistics, Lisbon, Portugal, pp 1412–1421. https://doi.org/10.18653/v1/D15-1166
14. Elman JL (1990) Finding structure in time. Cogn Sci 14(2):179–211
15. Hu X, Li G, Xia X, Lo D, Jin Z (2020) Deep code comment generation with hybrid lexical and syntactical information. Empir Softw Eng 25(3):2179–2217. https://doi.org/10.1007/s10664-019-09730-9
16. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
17. Cho K, van Merrienboer B, Çaglar G, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: EMNLP, pp 1724–1734. http://aclweb.org/anthology/D/D14/D14-1179.pdf
18. SrcMM: SrcML (2022). https://www.srcml.org. Accessed 3 Oct 2022
19. See A, Liu PJ, Manning CD (2017) Get to the point: summarization with pointer-generator networks. In: Proceedings of the 55th Annual meeting of the association for computational linguistics (volume 1: long papers). Association for Computational Linguistics, Vancouver, Canada, pp 1073–1083. https://doi.org/10.18653/v1/P17-1099
20. Zhou Z, Yu H, Fan G (2021) Adversarial training and ensemble learning for automatic code summarization. Neural Comput Appl 33(19):12571–12589. https://doi.org/10.1007/s00521-021-05907-w
21. Zhou Z, Yu H, Fan G (2020) Effective approaches to combining lexical and syntactical information for code summarization. Softw Pract Exp 50(12):2313–2336. https://doi.org/10.1002/spe.2893
22. Cheng C, Li B, Li Z, Liang P (2018) Automatic detection of public development projects in large open source ecosystems: an exploratory study on Github. In: Proceedings of the 30th international conference on software engineering and knowledge engineering. https://doi.org/10.18293/seke2018-085
23. HellFirePvP: ModularMachinery (2022). https://github.com/HellFirePvP/ModularMachinery. Accessed 3 Oct 2022
24. javalang: javalang (2022). https://github.com/c2nes/javalang. Accessed 3 Oct 2022
25. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: a pre-trained model for programming and natural languages. In: Findings of the association for computational linguistics: EMNLP 2020. Association for Computational Linguistics, pp 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139
26. Maldonado ES, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 2015 IEEE 7th international workshop on managing technical debt (MTD), pp 9–15. https://doi.org/10.1109/MTD.2015.7332619
27. NLTK: GitHub (2022). https://www.nltk.org/. Accessed 3 Oct 2022
28. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: Proceedings of the 31st international conference on neural information processing systems. NIPS'17. Curran Associates Inc., Red Hook, NY, USA, pp. 6000–6010. https://doi.org/10.5555/3295222.3295349

29. Ahmad W, Chakraborty S, Ray B, Chang K-W (2020) A transformer-based approach for source code summarization. In: Proceedings of the 58th annual meeting of the association for computational linguistics. Association for Computational Linguistics, pp 4998–5007, Online. https://doi.org/10.18653/v1/2020.acl-main.449
30. Ahmad W, Chakraborty S, Ray B, Chang K-W (2021) Unified pre-training for program understanding and generation. In: Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies. Association for Computational Linguistics, pp 2655–2668, Online. https://doi.org/10.18653/v1/2021.naacl-main.211
31. Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L (2020) BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th annual meeting of the association for computational linguistics. Association for Computational Linguistics, pp 7871–7880, Online. https://doi.org/10.18653/v1/2020.acl-main.703
32. Papineni K, Roukos S, Ward T, Zhu W-J (2002) BLEU: a method for automatic evaluation of machine translation. In: ACL'02. Association for Computational Linguistics, USA, pp 311–318. https://doi.org/10.3115/1073083.1073135
33. Denkowski M, Lavie A (2014) Meteor universal: language specific translation evaluation for any target language. In: Proceedings of the ninth workshop on statistical machine translation. Association for Computational Linguistics, Baltimore, Maryland, USA, pp 376–380. https://doi.org/10.3115/v1/W14-3348
34. Lin C-Y (2004) Rouge: a package for automatic evaluation of summaries. In: Text summarization branches out. Association for Computational Linguistics, Barcelona, Spain, pp 74–81
35. Chen B, Cherry C (2014) A systematic comparison of smoothing techniques for sentence-level BLEU. In: Proceedings of the ninth workshop on statistical machine translation. Association for Computational Linguistics, Baltimore, Maryland, USA, pp 362–367. https://doi.org/10.3115/v1/W14-3346
36. py-rouge: py-rouge (2022). https://github.com/Diego999/py-rouge. Accessed 3 Oct 2022
37. Kingma DP, Ba J (2015) Adam: a method for stochastic optimization, pp 1412–6980. arXiv:1412.6980
38. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, Klingner J, Shah A, Johnson M, Liu X, Łukasz K, Gouws S, Kato Y, Kudo T, Kazawa H, Stevens K, Kurian G, Patil N, Wang W, Young C, Smith J, Riesa J, Rudnick A, Vinyals O, Corrado G, Hughes M, Dean J (2016) Google's neural machine translation system: bridging the gap between human and machine translation. arXiv:1609.08144
39. Conover WJ (1999) Practical nonparametric statistics
40. Yang Z, Keung J, Yu X, Gu X, Wei Z, Ma X, Zhang M (2021) A multi-modal transformer-based code summarization approach for smart contracts. In: 2021 IEEE/ACM 29th international conference on program comprehension (ICPC), pp 1–12. https://doi.org/10.1109/ICPC52881.2021.00010
41. Allamanis M, Peng H, Sutton C (2016) A convolutional attention network for extreme summarization of source code. In: International conference on machine learning (ICML)
42. Liu S, Chen Y, Xie X, Siow J, Liu Y (2020) Retrieval-augmented generation for code summarization via hybrid GNN. arXiv:2006.05405
43. Haiduc S, Aponte J, Moreno L, Marcus A (2010) On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working conference on reverse engineering, pp 35–44. https://doi.org/10.1109/WCRE.2010.13
44. Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: 2010 ACM/IEEE 32nd international conference on software engineering, vol 2, pp 223–226. https://doi.org/10.1145/1810295.1810335
45. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ASE'10. Association for Computing Machinery, New York, NY, USA, pp 43–52. https://doi.org/10.1145/1858996.1859006
46. Sridhara G, Pollock L, Vijay-Shanker K (2011) Generating parameter comments and integrating with method summaries. In: 2011 IEEE 19th international conference on program comprehension, pp 71–80. https://doi.org/10.1109/ICPC.2011.28
47. Sridhara G, Pollock L, Vijay-Shanker K (2011) Automatically detecting and describing high level actions within methods. In: 2011 33rd International conference on software engineering (ICSE), pp 101–110. https://doi.org/10.1145/1985793.1985808
48. Wong E, Yang J, Tan L (2013) Autocomment: mining question and answer sites for automatic comment generation. In: 2013 28th IEEE/ACM international conference on automated software engineering (ASE), pp 562–567. https://doi.org/10.1109/ASE.2013.6693113
49. Wong E, Liu T, Tan, L.: Clocom: Mining existing source code for automatic comment generation. In: 2015 IEEE 22nd International conference on software analysis, evolution, and reengineering (SANER), pp 380–389. https://doi.org/10.1109/SANER.2015.7081848
50. Malhotra M, Kumar Chhabra J (2018) Class level code summarization based on dependencies and micro patterns. In: 2018 Second international conference on inventive communication and computational technologies (ICICCT), pp 1011–1016. https://doi.org/10.1109/ICICCT.2018.8473199
51. Zhou Y, Shen J, Zhang X, Yang W, Han T, Chen T (2022) Automatic source code summarization with graph attention networks. J Syst Softw 188:111257. https://doi.org/10.1016/j.jss.2022.111257
52. Chen Q, Xia X, Hu H, Lo D, Li S (2021) Why my code summarization model does not work: code comment improvement with category prediction. ACM Trans Softw Eng Methodol 30(2). https://doi.org/10.1145/3434280
53. Zhang J, Wang X, Zhang H, Sun H, Liu X (2020) Retrieval-based neural source code summarization. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering. ICSE'20. Association for Computing Machinery, New York, NY, USA, pp 1385–1397. https://doi.org/10.1145/3377811.3380383
54. LeClair A, Bansal A, McMillan C (2021) Ensemble models for neural source code summarization of subroutines. In: IEEE International conference on software maintenance and evolution, ICSME 2021, Luxembourg, September 27–October 1, 2021. IEEE, pp 286–297. https://doi.org/10.1109/ICSME52107.2021.00032
55. Liu M, Peng X, Meng X, Xu H, Xing S, Wang X, Liu Y, Lv G (2020) Source code based on-demand class documentation generation. In: 2020 IEEE International conference on software maintenance and evolution (ICSME), pp 864–865. https://doi.org/10.1109/ICSME46990.2020.00114
56. Wan Y, Zhao Z, Yang M, Xu G, Ying H, Wu J, Yu PS (2018) Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM, pp 397–407. https://doi.org/10.1145/3238147.3238206. Accessed 25 Sept 2021

57. Wang W, Zhang Y, Sui Y, Wan Y, Zhao Z, Wu J, Yu PS, Xu G (2020) Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Trans Softw Eng 48(1):102–119. https://doi.org/10.1109/TSE.2020.2979701