Interpretation-based Code Summarization

Mingyang Geng*, Shangwen Wang*, Dezun Dong*, Haotian Wang*, Shaomeng Cao†, Kechi Zhang‡, Zhi Jin‡
* College of Computer Science, National University of Defense Technology, China,
{gengmingyang13, wangshangwen13, dong, wanghaotian13}@nudt.edu.cn
† Peng Cheng Laboratory, Shenzhen, China, caoshm@pcl.ac.cn
‡ Key Lab of High Confidence Software Technology, Peking University, Beijing, China, {zhangkechi, zhijin}@pku.edu.cn

***Abstract*—Code comment, i.e., the natural language text to describe the semantic of a code snippet, is an important way for developers to comprehend the code. Recently, a number of approaches have been proposed to automatically generate the comment given a code snippet, aiming at facilitating the comprehension activities of developers. Despite that state-of-the-art approaches have already utilized advanced machine learning techniques such as the Transformer model, they often ignore critical information of the source code, leading to the inaccuracy of the generated summarization. In this paper, to boost the effectiveness of code summarization, we propose a two-stage paradigm, where in the first stage, we train an off-the-shelf model and then identify its focuses when generating the initial summarization, through a model interpretation approach, and in the second stage, we reinforce the model to generate more qualified summarization based on the source code and its focuses. Our intuition is that in such a manner the model could learn to identify what critical information in the code has been captured and what has been missed in its initial summarization, and thus revise its initial summarization accordingly, just like how a human student learns to write high-quality summarization for a natural language text. Extensive experiments on two large-scale datasets show that our approach can boost the effectiveness of five state-of-the-art code summarization approaches significantly. Specifically, for the well-known code summarizer, DeepCom, utilizing our two-stage paradigm can increase its BLEU-4 values by around 30% and 25% on the two datasets, respectively.**

***Index Terms*—Automatic Comment Generation, Code Summarization, Model Interpretation**

I. Introduction

Program comprehension, where developers try to understand the semantic meaning of a code snippet, is critical to modern software development. However, it is a quite labor-intensive task (e.g., according to the investigation of Hallam *et al.*, 80% of the development time is spent on understanding the code [1]). It has been widely recognized that code comments, which are usually the brief natural language descriptions of the major functionalities of code snippets, can greatly facilitate program comprehension [2], [3]. In practice, however, developers may not write high-quality comments due to their negligence or time constraints [4], [5]. To release the burden of manually writing comments for developers, various of techniques have been proposed to automatically generate comments for code snippets during the years [6]–[9], making this task (a.k.a. code summarization) an active research domain in the software engineering community [10]–[12].

Traditionally, researchers aim to generate comments with a template by constructing a set of complex rules, which cannot generalize well considering the complexity of the real-world code [13], [14]. Recently, with the advanced deep learning techniques, state-of-the-art approaches mainly utilize Neural Machine Translation (NMT) to perform the code summarization task [6]–[9]. Specifically, such approaches adopt sequence-to-sequence neural networks to convert code snippets into comments. For instance, CodeNN [6] exploits a classical encoder-decoder framework in NMT that encodes code to context vectors and then generates summaries in the decoder with the attention mechanism. By leveraging the large-amount high-quality manually-written code comments available in the wild (*e.g.*, those within the popular projects hosted on GitHub), these neural comment generation (NCG) approaches can learn from human experiences, and is thus considered as a promising research direction. Unfortunately, state-of-the-art NCG approaches are far from providing significant usefulness to developers in practice, since they may still generate comments that are different from the human-written ones to a large extent, as evaluated by Shi *et al.* [15]. Moreover, through a user study, Mcburney *et al.* [16] found that usually, the critical information of the target code is missed in the generated summarizations.

The basic question targeted by our study is thus how to further improve the effectiveness of existing NCG approaches, given that they have already utilized state-of-the-art deep learning techniques. To address this challenge, we are not going to adjust the model architecture of the current approaches or apply more advanced learning techniques. Instead, we explore a new direction that tries to utilize the generated summarization to enhance code summarization inversely. We were inspired by noting the process of teaching students to perform the text summarization task, a common exercise in reading courses [17]. Specifically, in such an activity, students firstly give their answers that may not be completely right, after which the teacher will perhaps let them explain what information in the original text they focus on during the summarization. Then, the teacher will demonstrate the actual critical information in the text (i.e., the information that should be focused on). Finally, by referring to the differences between their focuses and the actual critical information, students learn to revise their summarizations and generate more accurate

*Shangwen Wang is the corresponding author.

This work was supported in part by Excellent Youth Foundation of Hunan Province under Grant No. 2021JJ10050.

results. We note from this case that students learn to write high-quality summarizations in a two-stage manner in general: proposing their answers and learning to revise them. The key reason for the high-quality summarization from students is not that they can summarize the text well at the beginning, on the contrary, it is that by comparing their initial focuses with the actual critical information, students understand what they capture and what they miss in their initial results. By learning to revise their answers based on such information, they further obtain the knowledge about how to perform better in the future. Motivated by the above observation, we propose that the effectiveness of code summarization could be improved if a model can be further trained based on its initial results. By doing so, the model may also learn how to refine its (probably inaccurate) summarization based on what critical information it has captured and what critical information it has missed.

To explore such a direction, in this paper, we propose a new two-stage paradigm for code summarization. In the first stage, we train an off-the-shelf NCG model and produce its initial summarization, and then extract the code tokens which are relied on by the model to generate the initial summarization (i.e., which is referred to as the *summarization interpretation* in this study), utilizing interpretation approaches. Interpretation approaches are widely studied in the machine learning domain [18]–[21], whose target is to explain why an output can be obtained given a specific input to the modal. In our approach, we utilize them to help identify which part of the code is focused on by the model. In the second stage, we reinforce the model by training it with the inputs being the original code snippets and the information it relies on to produce its initial summarization. We postulate that in this reinforcement phase, the model could implicitly learn to understand which parts of its focuses are indeed critical and further which parts of the original code that contain critical information are missed. The reinforced model is therefore expected to generate more accurate summarizations on top of its initial results. In this paradigm, the NCG model is the "student" while the loss during training plays the role of "teacher" and directs the student to the correct answer. We refer to this new paradigm as ICSER, which represents for Interpretation based Code SummarizER.

We performed extensive experiments to investigate the effectiveness of ICSER. Specifically, we selected five baseline NCG models and evaluated the effectiveness of the vanilla model and the corresponding two-stage paradigm on two large-scale datasets from the CodeSearchNet benchmark [22]. Results show that ICSER can significantly outperform the corresponding vanilla NCG models with respect to three widely-used metrics for assessing the quality of code summarizations, i.e., BLEU-4, METEOR, and ROUGE-L. For instance, after adopting our two-stage paradigm, the BLEU-4 values of the well-known code summarizer, DeepCom [7], are increased by around 30% and 25% on the two evaluation datasets. We also perform a human evaluation to assess the generated comments and results show that ICSER can generate more useful comments compared with the vanilla NCG model.

The main contributions of this paper are summarized as follows:

- We propose a new two-stage paradigm for code summarization which mimics the practice of human students to learn how to accurately summarize natural language texts.
- We design a black-box and a white-box interpretation approaches to identify the focuses of NCG models. With such information, the NCG model can be further trained to refine its initial summarizations based on the critical code parts that are captured and missed by it.
- Experiments on top of five state-of-the-art NCG models have shown that our approach can boost the effectiveness of code summarization significantly. We open source our replication package at **https://github.com/gmy2013/Icser** for follow-up studies.

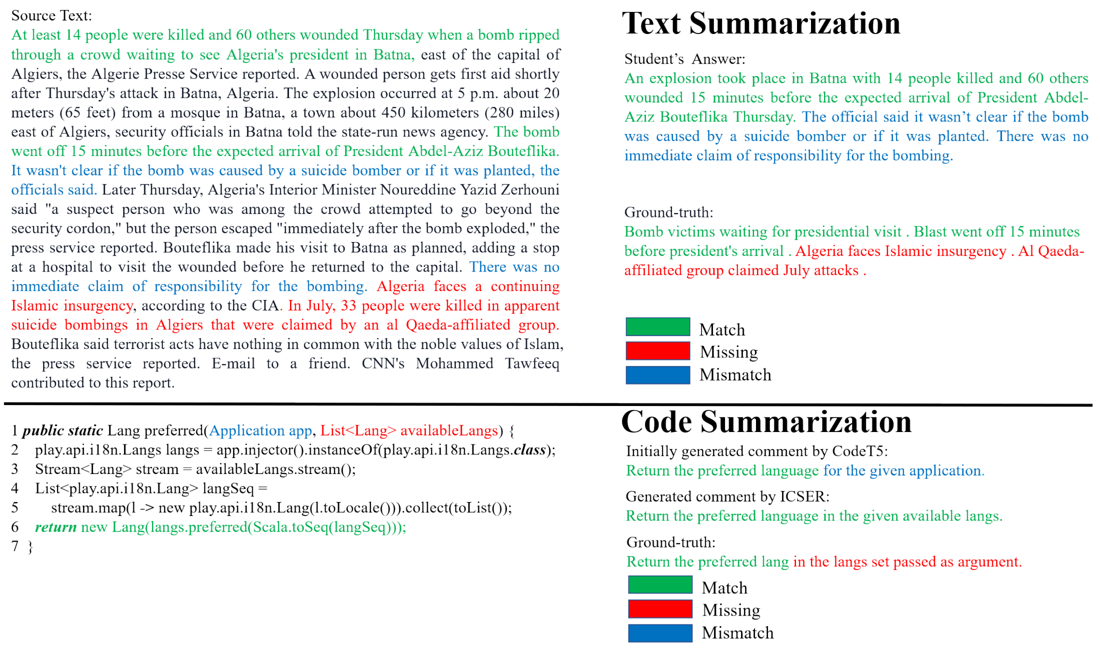
II. Background and Motivation

A. Background

The background of our work includes automatic comment generation and the interpretation of deep learning models.

1) Automatic Comment Generation: In the early stage of automatic source code summarization, template-based approaches [13], [14] are widely exploited. However, a well-designed template requires huge expert domain knowledge, which is time-consuming. Therefore, information retrieval (IR) based approaches [13], [23], [24] are proposed, with the basic idea to retrieve terms from source code to generate term-based summarization or to retrieve similar source code and reuse its summarization. However, the retrieved summarizations may not correctly describe the semantics and behavior of the target code snippets, leading to the mismatches between code and summarizations.

Recently, Neural Machine Translation (NMT) based models are exploited to generate summarizations for code snippets [6], [7], [25]. In concrete, Iyer *et al.* [6] proposed CodeNN, which adopts LSTM networks with attention module to generate natural language describing C# code snippets and SQL queries. Later, hybrid approaches [26], [27] that combine the NMT-based and IR-based methods are also proposed and have been shown to be promising. Allamanis *et al.* [28] proposed an attentional neural network to perform the code summarization task by viewing source code as plain text and generating comments using semantic features. Code structures are also incorporated for generating summaries. Hu *et al.* [7] proposed DeepCom, which exploits abstract syntax tree (AST) to annotate the methods of Java and take the flattened sequence as input to the NMT models. Code2seq [29] also leverages syntactic structure of programming languages to better encode source code. LeClair *et al.* [8] exploited a novel model *ast-attendgru* to perform the NCG task by combining tokens with the AST. Similarly, Hu *et al.*proposed Hybrid-DeepCom [25], improving DeepCom with hybrid lexical and syntactical information. A branch of studies focus on updating outdated comments according to code changes, which can be considered as another form of comment generation problem [30], [31].



Source Text:
At least 14 people were killed and 60 others wounded Thursday when a bomb ripped through a crowd waiting to see Algeria's president in Batna, east of the capital of Algiers, the Algerie Presse Service reported. A wounded person gets first aid shortly after Thursday's attack in Batna, Algeria. The explosion occurred at 5 p.m. about 20 meters (65 feet) from a mosque in Batna, a town about 450 kilometers (280 miles) east of Algiers, security officials in Batna told the state-run news agency. The bomb went off 15 minutes before the expected arrival of President Abdel-Aziz Bouteflika. It wasn't clear if the bomb was caused by a suicide bomber or if it was planted, the officials said. Later Thursday, Algeria's Interior Minister Noureddine Yazid Zerhouni said "a suspect person who was among the crowd attempted to go beyond the security cordon," but the person escaped "immediately after the bomb exploded," the press service reported. Bouteflika made his visit to Batna as planned, adding a stop at a hospital to visit the wounded before he returned to the capital. There was no immediate claim of responsibility for the bombing. Algeria faces a continuing Islamic insurgency, according to the CIA. In July, 33 people were killed in apparent suicide bombings in Algiers that were claimed by an al Qaeda-affiliated group. Bouteflika said terrorist acts have nothing in common with the noble values of Islam, the press service reported. E-mail to a friend. CNN's Mohammed Tawfeeq contributed to this report.

Text Summarization

Student's Answer:
An explosion took place in Batna with 14 people killed and 60 others wounded 15 minutes before the expected arrival of President Abdel-Aziz Bouteflika Thursday. The official said it wasn't clear if the bomb was caused by a suicide bomber or if it was planted. There was no immediate claim of responsibility for the bombing.

Ground-truth:
Bomb victims waiting for presidential visit . Blast went off 15 minutes before president's arrival . Algeria faces Islamic insurgency . Al Qaeda-affiliated group claimed July attacks .

🟩 Match
🟥 Missing
🟦 Mismatch

```
public static Lang preferred(Application app, List<Lang> availableLangs) {
  play.api.i18n.Langs langs = app.injector().instanceOf(play.api.i18n.Langs.class);
  Stream<Lang> stream = availableLangs.stream();
  List<play.api.i18n.Lang> langSeq =
     stream.map(l -> new play.api.i18n.Lang(l.toLocale())).collect(toList());
  return new Lang(langs.preferred(Scala.toSeq(langSeq)));
}
```

Code Summarization

Initially generated comment by CodeT5:
Return the preferred language for the given application.

Generated comment by ICSER:
Return the preferred language in the given available langs.

Ground-truth:
Return the preferred lang in the langs set passed as argument.

🟩 Match
🟥 Missing
🟦 Mismatch

Fig. 1: A text summarization example from the CNN/Daily dataset and a code summarization example from our evaluation dataset.

2) Interpretation of Deep Learning Models: Although sophisticated deep learning models have achieved great success in various applications, it is still hard to interpret how and why a deep learning model produces a specific result [32]. Despite that extensive efforts have been made towards explaining the results of learning-based models, it still remains an open problem [33]–[36]. In the natural language processing domain, existing works generally focus on classification models. Poulin et al. [37] proposed ExplainD to provide a graphical interpretation of the classification result. Erik et al. [38] interpreted the classification by using the game theory. LIME provides local model-agnostic interpretations of any classifier by constructing a linear model to locally fit a complex DNN model [39]. Riberio et al [40] further proposed Anchor, an extension of LIME based on the decision rules, where an anchor is a decision rule that leads to the result. LORE [41] is another local rule-based interpretation approach similar to Anchor.

Regarding to the intelligent source code domain, there has been plenty of interests in improving interpretability of models used in software engineering tasks [18]–[21]. Two representative works [20], [21] proposed to simplify the code while retaining the model prediction. Another approach called AutoFocus [18] aims to rate and visualize the relative importance of different code elements by using a combination of attention layers in the neural network and deleting statements in the program. Recently, Cito *et al.* [19] targeted global interpretability and helps model developers identify which types of inputs result in the poor performances of the model. They further exploited counterfactual explanation generation technique to generate the explanations by constituting minimal changes to the source code under which the model "changes its mind" [42].

B. Motivation

In this section, we illustrate the motivation of this work through a real-world text summarization exercise example for students.

The first part of Fig. 1 gives a text summarization example from the CNN/Daily dataset,[1] which is widely used as exercise examples for students on this task. From the source text, we can see that it introduces some information about a bombing case in Algeria such as the casualties and the time of the bombing. Beyond the basic information of the current bombing, the text also introduces the recent trend of bombings in Algeria, recalling another suicide bombing in the month of July. Consequently, in the ground-truth summarization of this piece of news (which is manually-annotated by researchers [43]), information for both the just-happened and previous bombings is mentioned together. As we can see, a student's summarization (1) captures some critical information like the casualties of the bombing (the contents in the color of green), (2) misses some critical information (in this case, the recent bombing trend and all relevant information of the previous bombing case are missed, cf. the contents in the color of red), and (3) mistakenly emphasizes some non-critical contents and includes them in the summarization (in this case, the student adds some information about the potential attacker but it is considered as non-critical probably because it is not clear enough now, cf. the contents in the color of blue). Correspondingly, consider the highlighted contents in the source text, those in green and blue are focused by the student, while those in green and red are actually needed to obtain the ground-truth summarization. Given that, if a teacher could tell the student that some of his/her focuses are right

[1]https://huggingface.co/datasets/cnn_dailymail

(those in green) while others are unnecessary (those in blue), and some critical information in the source text is lost (those in red), the student may produce more accurate answers.

This case is very similar to how NCG approaches perform the code summarization task. Specifically, after the initial training, an NCG model could learn certain domain knowledge so that it is able to capture some critical functionalities of a given code snippet. However, just like the human beings, a machine learning model could also miss some critical information in the code or mistakenly focuses on certain non-critical information, both of which will lead to the inaccuracy of its summarization. A concrete example is shown in the second part of Fig. 1, where we illustrate a code snippet from the CodeSearchNet dataset [44], its corresponding ground-truth summarization written by developers, and the summarization generated by CodeT5 [45], a well-known pre-training based NCG approach. The main functionality of the given code snippet is to select the preferred language from a set of candidate languages. We note that the summarization of CodeT5 (1) correctly captures the main functionality of the code snippet, which is returning the preferred language (the contents in the color of green), (2) misses the information about the range constraint (i.e., the preferred language is selected from a set of candidate languages passed as an argument, cf. the contents in the color of red), and (3) mistakenly captures information from another argument which does not contain much semantic information (this is reflected by the fact that the majority of the code is related to the argument `availableLangs` instead of `app`, cf. the contents in the color of blue). Therefore, we postulate that the effectiveness of CodeT5 could be improved if it could be trained to learn that it has focused on something critical (those in green as highlighted in the code snippet), it misses something critical (those in red), and it mistakenly focuses on something non-critical (those in blue). Indeed, as also shown in the figure, after using our proposed approach, the model generates a summarization that is semantically-identical to the ground-truth (we suppose the words "the given" indicates that the variable is passes as an argument). In this case, the value of Rouge-L, a widely-used metric for evaluating the quality of generated texts [46], is increased from 23.5% to 44.4%, an increase of nearly 90%.

III. Methodology

A. Approach Overview

The architecture of ICSER, which works in a two-stage manner, is shown in Figure 2. In the first stage, we train an off-the-shelf neural comment generation (NCG) model. Then, given a piece of source code as the input, we use the trained model to generate its corresponding summarization. After that, we identify the meaningful words in the generated comments, based on which we can extract the information in the source code utilized by the model to generate the current summarization (which is referred as *summarization interpretation*), with the help of interpretation approaches. Such information is considered as the original focuses of the comment generation model. In the second phase, we concatenate the source code and the original focuses, take them together as the input to the model, and train the model to produce summarizations based on such inputs. The aim of this step is reinforcing the model to learn about how to produce more accurate comments by identifying the critical information in the source code that has been captured and missed by it. After the second round of training, the trained model is supposed to learn the knowledge of revising its initial summarization based on its initial focuses, and therefore should be able to produce more qualified summarizations when given the code snippet and its initial summarization. We next introduce the meaningful word dentification, the interpretation approach we utilize, and the reinforcement training in the second phase in detail.

B. Meaningful Word Identification

We recall that one of the important targets in the first stage of our approach is to identify the code tokens that are relied on by the model to generate the initial summarization (we refer to as *summarization interpretation*). To achieve so, we first need to filter non-meaningful words in the summarization. Our intuition is that not all the words in a natural language text possess rich semantic information, which is a common sense in the natural language processing domain [47]–[49]. Therefore, we can safely focus on the meaningful words (which are usually the verbs and nouns) in the summarization when performing the summarization interpretation to obtain more accurate results. For instance, in Fig. 1, the semantic meaning of the sentence initially generated by CodeT5, i.e., "return the preferred language for the given application", relates largely to the following words "return", "preferred language", and "given application", but relates little to the definite article "the" and the preposition "for".

We choose to rely on the concept **stop word** from the natural language processing to fulfill our target. Specifically, stop words refer to the words that are filtered out before the processing of natural language text since they are insignificant to the ongoing tasks [48], [49]. Stop word identification is required to perform a number of natural language processing tasks such as in information retrieval systems (IR), stem weighting, stemming and spelling standardization [47]. In our approach, we use the open-source package *rake-nltk* [2] to help us identify the meaningful words in the summarization.

C. Summarization Interpretation

After the meaningful word identification, we use interpretation approaches to extract the code tokens that are focused on by the model to generate the initial summarization. In this study, we explore two interpretation approaches, a black-box one and a white-box one. We next introduce the two approaches in detail.

1) Black-Box Interpretation Approach: The ultimate goal of the summarization interpretation is to build a mapping relation between the code and the summarization, that is, to identify which part of the code contributes to the generation

[2]https://pypi.org/project/rake-nltk

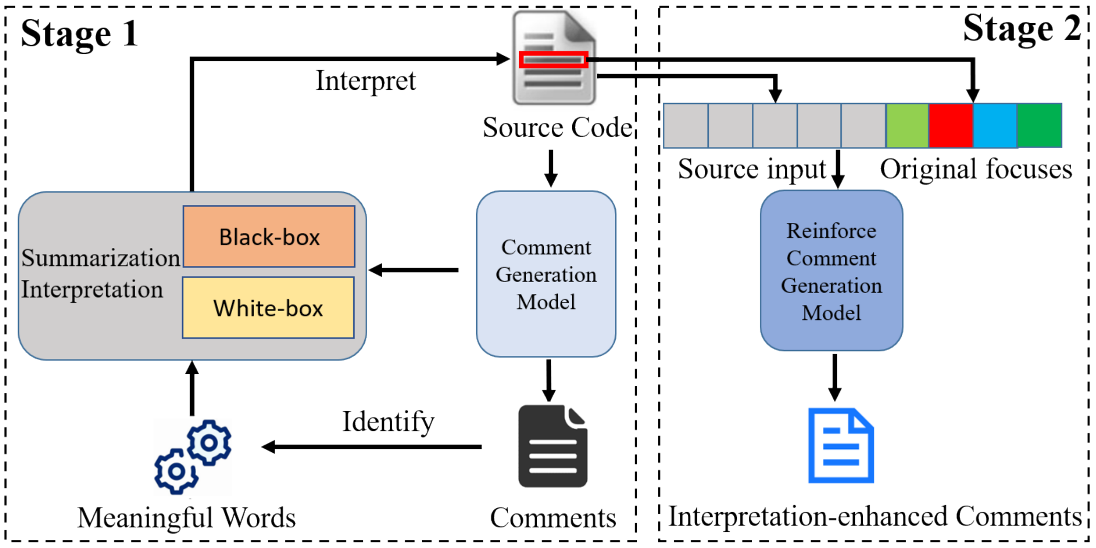


Fig. 2: Model architecture of our ICSER.

of a specific part of the summarization. In the black-box based approach, we have no direct access to the intrinsic information of the NCG model such as the values of its parameters. Therefore, we adopt a mutation-based strategy: we generate different mutants of the original source code by removing a small code segment each time, send the mutants to the NCG model and generate summarizations for such mutants, and then check if the meaningful words identified in the last step disappear in any of the newly generated summarization. If so, the removed code segment in the mutant is considered to have relation with the disappeared meaningful words, and a mapping relation is thus built. The behind intuition is straightforward: if a code segment can contribute to the generation of a specific part of the summarization, then such a part of summarization could not be generated if the source code does not contain this code segment. Note that such a strategy is model-agnostic and can be applied to any model that does not strictly require a syntactically-correct input (since removing a code segment may lead to the input code being syntactically-incorrect).

In our approach, the granularity of the code segment to be removed is in the expression level. The rationale is that when manually comprehending code, it could be difficult for developers to understand the semantic meaning of a standalone code token, on the contrary, they could easily understand the semantic meaning of a consecutive code token sequence. For example, for the code `lang.collect(toList())`, developers may easily understand its intention with the information of the APIs and the parameters. Based on that, we postulate that the trained NCG model could also generate summarizations by focusing on a consecutive sequence of code tokens. Also, we disregard statement level information since expression level information is more fine-grained and has achieved better effectiveness than statement level information in several software engineering tasks such as program repair [50], [51].

The whole process of our black-box interpretation approach is shown in Algorithm 1. Given a code snippet, we use a

Algorithm 1: Black-box Interpretation Approach.

```
Input: Neural comment generation model $NCG$, code snippet $c$.
Output: original focus set $list\_foc$.
1  Function Interpret($NCG$, $C$):
2      $list\_foc \to \phi$
3      $Initial\_comment \to NCG(c)$
4      $Meaningful\_words \to kw\_extract(Initial\_comment)$
5      $Expression\_list \to Exp\_extract(c)$
6      for $exp$ in $Expression\_list$ do
7          $mutant \to Remove(c,exp)$
8          $New\_comment \to NCG(mutant)$
9          /* Check if any meaningful word is lost
              in the newly generated summarization.
              */
10         if $\exists\ word \in Meaningful\_words,\ word \notin New\_comment$ then
11             $list\_foc$.append($exp$)
12     return $list\_foc$
```

language parser (*e.g.*, javalang [3] for the Java language and tree-sitter [4] for the Python language) to extract all the expressions in the code (line 5). For each extracted expression, we remove it from the code to generate a mutant of the original code (line 7). We then send the corresponding mutant to the NCG model to generate a new summarization (line 8). By comparing the newly-generated summarization and the original one, we check if any meaningful word in the summarization is lost. If so, the removed expression is recorded as part of the *original focuses* (lines 10-11). By iterating the process on all the expressions in the code, we finally extract all the original focuses.

2) White-Box Interpretation Approach: The white-box approaches allow us to inspect the intrinsic information of the NCG model. Inspired by existing studies [52], [53], we rely on the attention score of the input code token to identify the focuses. Specifically, for each meaningful word in the summa-

[3]https://github.com/c2nes/javalang
[4]https://tree-sitter.github.io/tree-sitter/

rization, we check the attention score of the model at that time stamp and consider the code token with the highest attention score as the focus (this is feasible for encoder-decoder based NCG approaches where the summarization is generated word by word). After that, all the identified code tokens are recorded as the original focuses of the NCG model. The intuition of such a strategy is simple and straightforward: since the attention mechanism usually represents the contribution of each input code token to the next word to be generated, we directly use it as an indicator of the model's focuses.

D. Reinforcement Training

In the second stage, we incorporate the original focuses identified through our summarization interpretation into the reinforcement training process of the model. Since such focuses could be extracted from different parts of the source code, they are thus not complete code snippets. Therefore, we decide to treat them as token sequences. Specifically, for each instance in the training set, we concatenate the original source code and the focuses to shape a new token sequence (as shown in Fig. 2 the focuses are appended to the end of the original source code). Then we exploit the newly-generated instances to reinforce our model to generate the oracle summarization.

After this reinforcement training in the second stage, the model is supposed to learn the ability to revise the summarization based on the original focuses. Taking the code snippet from the test set plus with the corresponding original focuses as inputs, the reinforced model can generate a summarization different from the original one, which is expected to be more qualified.

IV. Experiment Setting

A. Research Questions

In our evaluation, we aim to answer the following research questions:

- **RQ1: How effective is ICSER on generating high-quality summarizations based on the initial ones?** This RQ aims at comparing the summarization produced through our two-stage approach, ICSER, and the initial summarization produced by the original NCG model. By answering such a question, we can investigate how well the model learns to revise its initial summarization.
- **RQ2: To what extent do different summarization interpretation approaches affect the effectiveness of ICSER?** We design two interpretation approaches for ICSER, a black-box one and a white-box one. In this RQ, we aim to investigate how effective is ICSER with different interpretation approaches.

B. Dataset

We choose to use the CodeSearchNet (CSN) benchmark [22] as our evaluation dataset. CSN is a large-scale dataset mined from popular GitHub projects (indicated by the number of stars and forks), which contains code-comment pairwise data from six programming languages. It has been widely used in the code summarization domain upon released [15], [54]–[56]. Generally, the input to the NCG model is the function-level code snippet, and the corresponding comment written by the developers is used as the oracle summarization. In our study, to better evaluate the generalizability of our approach, we use the data from two popular programming languages, i.e., Java and Python. Following the existing study [15], the source code that cannot be parsed by our parser (i.e., javalang and tree-sitter) are filtered out and the detailed statistics of our evaluation dataset are listed in Table I.

TABLE I: The statistics of our evaluation datasets.

Dataset	**Training**	**Validation**	**Test**
CSN-Python	412,178	23,107	22,176
CSN-Java	446,607	19,855	29,778

C. Baseline NCG Approaches

As we have introduced in Section III, the application of ICSER requires that (1) the NCG model can take a piece of syntactically-incorrect code as input, and (2) the summarization is generated word by word. Given that, we choose the following five state-of-the-art approaches, which ensures the above prerequisites, as the baseline NCG approaches in our study.

- CodeNN [6] is the first neural approach that learns to generate summarizations of code snippets.
- DeepCom [7] is an SBT-based (Structure-based Traversal) model, which can capture the syntactic and structural information from AST. It is an attentional LSTM-based encoder-decoder neural network that encodes the SBT sequence and then generates summarizations.
- Astattggru [8] encodes both the code and AST to learn lexical and syntactic information. It exploits two GRUs to encode code and SBT sequences respectively.
- NCS [9] is the first attempt to replace the previous RNN units with Transformer model, incorporating the copy mechanism [57] to allow both generating words from vocabulary and copying words from the input source code.
- CodeT5 [45] is a unified pre-trained encoder-decoder Transformer model that is supposed to better leverage the code semantics conveyed from the developer-assigned identifiers.

We recall that all the selected baseline approaches take the sequential data as the inputs. Specifically, the input of CodeNN, NCS, and CodeT5 is the token sequence of the target code snippet. While DeepCom and Astattggru involve the information from ASTs, such information is encoded in the SBT sequence. Specifically, since our black-box approach targets expressions in the program, all the removed tokens are actually leaf nodes in the AST. We then remove the shortest AST paths that link each pair of the leaf nodes and finally obtain the SBT sequences of such post-processing ASTs. Therefore, all the selected baselines can accept the syntactically-incorrect code snippets as inputs. Also, they all output the word sequence of the summarization since they use the encoder-decoder architecture with the attention mechanism.

D. Metrics

To assess the quality of the generated summarization, we use three metrics in total:

- BLEU (Bilingual Evaluation Understudy) [58] is commonly used for evaluating the quality of the generated text [6], [7], [59]. In short, a BLEU score is a percentage number between 0 and 100 that measures the similarity between one sentence to a set of reference sentences using constituent n-grams precision scores. We exploit BLEU-4 (specifically, the BLEU-DC [15]) to measure the precision of the generated summaries. In concrete, a value of 0 indicates that the generated sentence has no overlap with the reference while a value of 100 means perfect overlap with the reference.
- METEOR [60], which denotes the Metric for Evaluation of Translation with Explicit ORdering, is another widely used metric to evaluate the quality of generated code summaries [25], [61], [62]. METEOR evaluates the generated summary g by aligning it to the reference summary r and calculating the similarity scores based on the unigram matching.
- ROUGE denotes the Recall-oriented Understudy for Gisting Evaluation [46]. It computes the count of several overlapping units such as n-grams, word pairs, and sequences. ROUGE has several different variants from which we consider the most popular one ROUGE-L [56], [63], [64], which is calculated based on the longest common subsequence (LCS).

E. Implementation Details

All the baseline approaches are open sourced by their original authors. Therefore, to avoid replication bias, we reused their official code. We exploited the default hyper-parameter settings provided by each approach. The maximum value of epoch is set to 40 and we adopted an early stopping mechanism to enable the convergence and generalization of the model, following the recent study [15]. In addition, to mitigate the randomness in the experiment, we repeated each experiment 3 times and display the mean and standard deviation in the form of **mean** $\pm$ **std**. All experiments were conducted on a DGX machine with 8 Tesla V100 32GB GPUs.

V. Experiment Results

A. RQ1-The effectiveness of ICSER

Table II and Table III demonstrate the effectiveness of baseline NCG models and ICSER on the two evaluation datasets. Note that we experimented with two different interpretation approaches (i.e., the white-box and black-box ones). In these two tables, we demonstrate the optimal effectiveness of ICSER and the differences between such interpretations will be discussed later.

From the results, we find that our two-stage code summarization approach, ICSER, can consistently outperform the corresponding baseline approach on both Java and Python datasets with respect to all the three metrics. For instance, for the pioneer NCG model in the literature, CodeNN, its values of BLEU-4, METEOR, and ROUGE-L are increased by 17.7%, 14.2%, and 9.7%, respectively, on the CSN-Java dataset after adopting the two-stage summarization paradigm. Similarly, on the CSN-Python dataset, the values of BLEU-4, METEOR, and ROUGE-L for CodeNN are increased by 12.6%, 25.2%, and 7.2%, respectively. The most significant effectiveness enhancement is from DeepCom. Specifically, comparing with the vanilla DeepCom, $Icser_{DeepCom}$ achieves increases of 28.2% (24.9%), 24.9% (24.2%), and 17.6% (14.2%) with respect to BLEU-4, METEOR, and ROUGE-L, on the CSN-Java (CSN-Python) dataset. We also note that NCG models with the Transformer architecture gain relatively low enhancement, probably because their vanilla models can already generate high-quality summarizations. For instance, on the CSN-Java dataset, NCS achieves the highest BLEU-4 score among all the baselines, i.e., 25.17%. After adopting the two-stage paradigm, $Icser_{NCS}$ achieves a BLEU-4 of 26.89%, experiencing a performance increase of 6.8%, which is lower than that of DeepCom (28.2%) to a large extent.

We also conducted Wilcoxon signed-rank tests [65] where for each vanilla NCG model and its two-stage paradigm, we compared their achieved values with respect to the three metrics on each individual code snippet from the test set. Results show that ICSER significantly outperforms its corresponding vanilla NCG model with respect to all the three metrics on both datasets. Specifically, all the p-values are less than 0.001 in the comparison results.

ICSER *can significantly improve the quality of the generated summarizations compared with those generated by the vanilla NCG models. Specifically, on the CSN-Java dataset, the BLEU-4 value of the summarizations generated by DeepCom is changed from 7.85% to 10.06%, an increase of 28.2%.*

B. RQ2-Comparson Between Black-Box and White-Box Interpretation Approaches

To investigate which interpretation approach can help ICSER achieve higher effectiveness (the black-box one or the white-box one), we compare the effectiveness of ICSER when integrated with different approaches and illustrate the results in Table IV.

From the results we note that for the majority of our selected NCG models, ICSER achieves higher effectiveness when using the black-box interpretation approach. For instance, when the vanilla model is CodeNN, the values achieved by ICSER with respect to the BLEU-4, METEOR, and ROUGE-L are 14.46% (14.96%), 12.66% (13.17%), and 31.97% (33.18%) when using the white-box (black-box) approach, on the CSN-Java dataset. We observe a similar trend on the CSN-Python dataset. The only exception is Astattgru, which gains higher effectiveness when using the white-box approach. Specifically, on the CSN-Java and CSN-Python datasets, the BLEU-4 values of the black-box approach are 16.74% and 16.83%,

TABLE II: Experiment results of different NCG models on the CSN-Java dataset (metric values in %).

Tool	BLEU-4	Improvement	METEOR	Improvement	ROUGE-L	Improvement
CodeNN	12.71±0.23	↑ 17.7%	11.53±0.12	↑ 14.2%	30.24±0.29	↑ 9.7%
$Icser_{CodeNN}$	**14.96±0.19**		**13.17±0.08**		**33.18±0.24**	
DeepCom	7.85±1.07	↑ 28.2%	7.47±0.25	↑ 24.9%	19.00±0.87	↑ 17.6%
$Icser_{DeepCom}$	**10.06±0.95**		**9.33±0.32**		**22.34±0.75**	
ASTattgru	15.83±0.17	↑ 8.1%	13.20±0.04	↑ 7.0%	34.21±0.64	↑ 5.5%
$Icser_{ASTattgru}$	**17.12±0.26**		**14.13±0.21**		**36.11±0.52**	
NCS	25.17±0.39	↑ 6.8%	15.62±0.24	↑ 10.7%	40.97±0.52	↑ 2.1%
$Icser_{NCS}$	**26.89±0.28**		**17.29±0.31**		**41.85±0.64**	
CodeT5	20.31±0.23	↑ 6.0%	15.32±0.26	↑ 10.1%	39.34±0.31	↑ 4.9%
$Icser_{CodeT5}$	**21.52±0.26**		**16.87±0.21**		**41.27±0.26**	

TABLE III: Experiment results of different NCG models on the CSN-Python dataset (metric values in %).

Tool	BLEU-4	Improvement	METEOR	Improvement	ROUGE-L	Improvement
CodeNN	13.92±0.14	↑ 12.6%	9.13±0.09	↑ 25.2%	30.89±0.12	↑ 7.2%
$Icser_{CodeNN}$	**15.68±0.25**		**11.43±0.12**		**33.12±0.18**	
DeepCom	8.09±1.13	↑ 24.9%	7.44±0.21	↑ 24.2%	20.02±0.79	↑ 14.2%
$Icser_{DeepCom}$	**10.11±0.95**		**9.24±0.30**		**22.87±0.71**	
ASTattgru	16.08±0.19	↑ 7.1%	12.94±0.08	↑ 13.3%	33.97±0.59	↑ 3.9%
$Icser_{ASTattgru}$	**17.22±0.33**		**14.66±0.18**		**35.29±0.47**	
NCS	25.24±0.42	↑ 6.7%	14.96±0.18	↑ 9.4%	38.47±0.29	↑ 2.2%
$Icser_{NCS}$	**26.93±0.37**		**16.37±0.26**		**39.31±0.31**	
CodeT5	20.01±0.19	↑ 5.8%	15.14±0.22	↑ 7.3%	37.89±0.38	↑ 3.2%
$Icser_{CodeT5}$	**21.18±0.19**		**16.24±0.19**		**39.11±0.33**	

TABLE IV: The effectiveness of the black-box and white-box interpretation approaches (in %).

Tool	CSN-Java			CSN-Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
$Icser_{CodeNN-white-box}$	14.46±0.21	12.66±0.11	31.97±0.28	14.71±0.19	10.39±0.11	32.64±0.16
$Icser_{CodeNN-black-box}$	**14.96±0.19**	**13.17±0.08**	**33.18±0.24**	**15.68±0.25**	**11.43±0.12**	**33.12±0.18**
$Icser_{DeepCom-white-box}$	9.24±1.02	8.38±0.19	21.12±0.79	9.14±0.88	8.13±0.26	21.45±0.83
$Icser_{DeepCom-black-box}$	**10.06±0.95**	**9.33±0.32**	**22.34±0.75**	**10.11±0.95**	**9.24±0.32**	**22.87±0.75**
$Icser_{Astattgru-white-box}$	**17.12±0.26**	**14.13±0.21**	**36.11±0.52**	**17.22±0.33**	**14.66±0.18**	**35.29±0.47**
$Icser_{Astattgru-black-box}$	16.74±0.20	13.89±0.11	35.79±0.61	16.83±0.29	14.15±0.16	37.92±0.39
$Icser_{NCS-white-box}$	26.43±0.33	16.78±0.29	41.69±0.68	26.67±0.44	16.01±0.22	39.11±0.22
$Icser_{NCS-black-box}$	**26.89±0.28**	**17.29±0.31**	**41.85±0.64**	**26.93±0.37**	**16.37±0.26**	**39.31±0.31**
$Icser_{CodeT5-white-box}$	21.38±0.28	16.24±0.23	40.88±0.24	20.79±0.17	15.99±0.21	38.79±0.35
$Icser_{CodeT5-black-box}$	**21.52±0.26**	**16.87±0.21**	**41.27±0.26**	**21.18±0.19**	**16.24±0.19**	**39.11±0.33**

respectively, slightly lower than that achieved when using the white-box approach, which are 17.12% and 17.22%, respectively. We also note that such effectiveness deviations caused by different interpretation approaches are not significant. For instance, regarding to the BLEU-4 metric of CodeT5 on the CSN-Java dataset, the black-box approach achieves a score of 21.52% while that of the white-box one is 21.38%, with only a difference of 0.14%. Our statistical test results also confirm this, with the p-values comparing the metric values achieved by the black-box and white-box approaches being larger than 0.05. Such results indicate that both approaches could be applied in practice.

Our results are also consistent with the previous studies [66], [67] which show that merely relying on the attention score may not obtain promising interpretation results in some cases. The behind reason could be explained as some inputs that do not possess the highest attention score may also contribute to certain outputs but such relations are ignored if only concentrating on the input with the highest attention score.

Generally, the black-box interpretation approach can achieve higher effectiveness than the white-box one when integrated with ICSER. *Nonetheless, such difference is not statistically significant.*

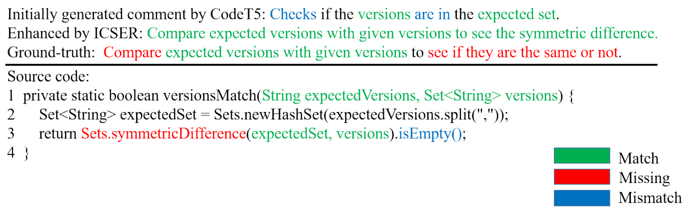
VI. DISCUSSION

A. Human Evaluation

All our metrics assess the lexical gap between the generated comments and the references. We also perform a human evaluation to assess the semantic quality of the generated comments. Specifically, we recruit eight participants, including four Ph.D students and four senior researchers, who are not co-authors of this paper. They all have at least three years of development experience for Java and Python. We randomly select 100 code snippets from the test sets (50 from CSN-Java and 50 from CSN-Python). We focus on comparing CodeT5 and $Icser_{CodeT5}$, and thus we obtain a total of 200

TABLE V: The statistic results of human evaluation.

	Approach	Avg.	Median	Std.
Naturalness	CodeT5	3.9	4.0	0.9
	$Icser_{CodeT5}$	**4.0**	4.0	0.8
Adequacy	CodeT5	3.1	3.0	1.2
	$Icser_{CodeT5}$	**3.6**	4.0	1.1
Usefulness	CodeT5	2.9	3.0	1.3
	$Icser_{CodeT5}$	**3.3**	3.5	1.1


Initially generated comment by CodeT5: Checks if the versions are in the expected set.
Enhanced by ICSER: Compare expected versions with given versions to see the symmetric difference.
Ground-truth: Compare expected versions with given versions to see if they are the same or not.

Source code:

```
private static boolean versionsMatch(String expectedVersions, Set<String> versions) {
    Set<String> expectedSet = Sets.newHashSet(expectedVersions.split(","));
    return Sets.symmetricDifference(expectedSet, versions).isEmpty();
}
```

🟩	Match
🟥	Missing
🟦	Mismatch

Fig. 3: An example from the CSN-Java test set.

generated comments. The 200 code-comment pairs are divided into four groups, and each group is checked by two participants independently. Following the existing studies [68], [69], each participant is asked to rate each generated comment from the three aspects: (1) **Naturalness** which reflects the fluency of generated comments from the perspective of grammar; (2) **Adequacy** which reflects the information richness of generated comments; and (3) **Usefulness** which reflects how can generated comments help developers, on a 5-point Likert scale (1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent).

Table V shows the statistic results of the human evaluation. We note that $Icser_{CodeT5}$ is better than CodeT5 in all the three aspects. The average scores of $Icser_{CodeT5}$ on naturalness, adequacy, and usefulness are 4.0, 3.6, 3.3 respectively, for the 100 selected code snippets. Specifically, in terms of naturalness, the average score of $Icser_{CodeT5}$ reaches 4.0, meaning that the generated comments are generally fluent and readable. In terms of adequacy, $Icser_{CodeT5}$ achieves 3.6 while CodeT5 achieves 3.1, which indicates that after adopting the two-stage paradigm, the generated comments contain more information. Such results confirm that the model does learn to capture some missed information in the initial summarization after the reinforcement training. Moreover, our user study shows that ICSER improves the usefulness perceived by the users from a point below 3 to a point over 3. This is a significant improvement since a point below 3 means that the users generally do not consider the summarization results as useful. Therefore, it indicates that ICSER effectively improves program comprehension from the developers' perspective.

Our human evaluation focuses on comparing CodeT5 and $Icser_{CodeT5}$. It should be noted that ICSER achieves the least improvement on CodeT5 with respect to the BLEU (cf. Table II and Table III). It is thus expected that compared with other vanilla models, ICSER can achieve at least similar improvements when judged by humans. Due to the scale restriction of human evaluation, we leave performing more manual analysis (*e.g.*, involving more code examples and participants from industry) as our future work.

B. Case Analysis

To further investigate the effectiveness achieved by ICSER, we analyzed a detailed case to compare the summarizations generated by the vanilla NCG model and ICSER. As shown in Fig. 3, the source code implements the functionality that compares two sets of version strings and judges if they are the same (returning `true` if so and `false` otherwise). Therefore, the corresponding comment written by the developers summarizes as "compare expected versions with given versions to see if they are the same or not". The initial summarization generated by CodeT5 successfully captures the two subjects (i.e., the argument `versions` and the expected set), but it inaccurately focuses on the method invocation `isEmpty()` and translates its meaning as *if something is in a specific range* ("if the versions are in the expected set"). With the second stage in ICSER (i.e., the reinforcement training process), the model learns that its aforementioned focus is inaccurate and turns to focus on another method invocation `symmetricDifference()`. This time, the model realises that the main functionality is checking if there is any difference between two objects, and thus generates a summarization that is semantically identical to the ground-truth, improving the BLEU-4 score from 0.07 to 0.49.

C. The Rationale of Using the Original Focuses

In our approach, we design a two-phase paradigm where in the second stage we use the initial focuses of the model for reinforcement learning. Our intuition is that in such a manner, the model could learn the critical information which has been captured and missed by it. To investigate the rationale of such a strategy, we performed another experiment where to reinforce the model, we randomly selected focuses from the source code whose numbers of tokens are identical to the identified original focuses.

From the results shown in Table VI, we note that after replacing the original focuses with randomly selected tokens, the effectiveness of ICSER decreases significantly. For instance, the BLEU-4, METEOR, and ROUGE-L of $Icser_{DeepCom}$ drop from 10.06% to 8.03%, from 9.33% to 7.73%, from 22.34% to 19.41% (from 10.11% to 8.16%, from 9.24% to 7.62%, from 22.87% to 20.21%) on the CSN-Java (CSN-Python) dataset. Similar phenomenons are observed for other baselines. Another interesting finding is that the effectiveness of ICSER with the random strategy is still better than that of the vanilla models. For instance, the BLEU-4 value of 8.03% on the CSN-Java dataset is higher than that of the vanilla DeepCom which is 7.85%. Such results indicate that the two-stage code summarization paradigm can generally improve the effectiveness of code summarizers, while the initial focuses of the model contribute significantly to such enhancements.

D. Threats to Validity

External threats. Generalizing to different programming languages is always a concern in the evaluation of NCG models [15]. To mitigate this threat, our study focuses on two widely-used languages (i.e., Python and Java) and results

TABLE VI: Experiment results after replacing the original focuses with randomly selected tokens (in %).

Tool	CSN-Java			CSN-Python		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CodeNN	12.71±0.23	11.53±0.12	30.24±0.29	13.92±0.14	9.13±0.09	30.89±0.12
$Icser_{CodeNN-Random}$	12.80±0.26	11.68±0.15	30.42±0.25	14.12±0.12	9.36±0.24	31.02±0.14
$Icser_{CodeNN}$	**14.96±0.19**	**13.17±0.08**	**33.18±0.24**	**15.68±0.25**	**11.43±0.12**	**33.12±0.18**
DeepCom	7.85±1.07	7.47±0.25	19.00±0.87	8.09±1.13	7.44±0.21	20.02±0.79
$Icser_{DeepCom-Random}$	8.03±1.13	7.73±0.24	19.41±0.82	8.16±1.05	7.62±0.24	20.21±0.76
$Icser_{DeepCom}$	**10.06±0.95**	**9.33±0.32**	**22.34±0.75**	**10.11±0.95**	**9.24±0.32**	**22.87±0.71**
Astattgru	15.83±0.17	13.20±0.04	34.21±0.64	16.08±0.19	12.94±0.08	33.97±0.59
$Icser_{Astattgru-Random}$	15.91±0.16	13.32±0.15	34.33±0.54	16.12±0.31	13.11±0.11	34.28±0.62
$Icser_{Astattgru}$	**17.12±0.26**	**14.13±0.21**	**36.11±0.52**	**17.22±0.33**	**14.66±0.18**	**35.29±0.47**
NCS	25.17±0.39	15.62±0.24	40.97±0.52	25.24±0.42	14.96±0.18	38.47±0.29
$Icser_{NCS-Random}$	25.39±0.27	15.81±0.28	41.16±0.61	25.41±0.39	15.32±0.17	38.72±0.33
$Icser_{NCS}$	**26.89±0.28**	**17.29±0.31**	**41.85±0.64**	**26.93±0.37**	**16.37±0.26**	**39.31±0.31**
CodeT5	20.31±0.23	15.32±0.26	39.34±0.31	20.01±0.19	15.14±0.22	37.89±0.38
$Icser_{CodeT5-Random}$	20.46±0.31	15.51±0.21	39.62±0.33	20.46±0.17	15.49±0.21	38.21±0.35
$Icser_{CodeT5}$	**21.52±0.26**	**16.87±0.21**	**41.27±0.26**	**21.18±0.19**	**16.24±0.19**	**39.11±0.33**

demonstrate similar trend. Also, due to the limitation of our black-box interpretation approach, we only evaluate our approach on code summarization approaches that are able to deal with syntactically-incorrect code snippets. Those that strictly require the input code being syntactically-correct are thus excluded from the evaluation [70]–[72]. This threat is mitigated considering that the five baselines in our evaluation represent the state of the art in the code summarization domain well. We leave refining our approach to generalize on more code summarization models as our future work.

Internal threats. The evaluation of ICSER requires a large-scale replication experiment on existing code summarization approaches. To ensure the reliability of our experiment, we directly reused the source code and hyper-parameter values from the existing studies. We further double checked our results and confirmed that they are consistent with the previously-reported ones.

VII. Related Work

In this section, we introduce existing studies in the literature that are related to this study from the following two domains: reinforcement learning and data augmentation technologies.

The aim of reinforcement learning technique is to find a strategy that makes the agent take certain actions to maximize the cumulative reward. Silver *et al.* [73] exploited deep reinforcement learning and Monte Carlo Tree Search to the computer Go game and reached a professional level. Reinforcement learning techniques can also be applied to the intelligent software engineering domain. Wan *et al.* [59] exploited an actor-critic reinforcement learning framework to alleviate the exposure bias issue of code summarization. Yao *et al.* [74] proposed an effective framework based on reinforcement learning by encouraging the code summarization model to generate summarizations that can be used for the retrieval task. Similarly, Le *et al.* [75] treated the code generation model as an actor and exploited a trained correctness-prediction critic network for better code generation. In our approach, we also reinforce the model to generate more qualified summarizations based on its initial summarization and focuses. The difference between ICSER and the traditional reinforcement learning techniques is that the actor space and the reward function are explicitly given in reinforcement learning but are implicitly learned by our approach. That is because manually labelling the ground-truth information (i.e., the critical part of each code snippet) is time-consuming and infeasible. We thus expect that the model could learn such information automatically.

Data augmentation aims to increase the data diversity and improve the generalizability by exploiting various transformation technology. In recent years, data augmentation techniques have also been exploited in the intelligent software engineering domain [76]–[78] to expose the vulnerability and improve the robustness of code models. Compared with data augmentation technology, ICSER adds auxiliary information as the inputs of the model without generating additional data, while data augmentation techniques usually add manual labeled or synthetic training samples for learning.

VIII. Conclusion

In this paper, we propose a new two-stage paradigm for code summarization. The basic idea of our new paradigm is to mimic the practice of human students to generate high-quality summarizations for natural language texts. Specifically, students usually write an initial answer first and then revise it by understanding what critical information they have captured and what they have missed. Therefore, in our paradigm, we first train an off-the-shelf code summarization model and identify its focuses with the help of *summarization interpretation*, then we reinforce the model using its focuses, with the hope that the model can learn how to revise its intial summarization automatically, just like a human student. Experiments on five state-of-the-art code summarization models show promising results: after adopting our two-stage paradigm, the effectiveness of these models increases significantly on two large-scale datasets for Java and Python languages.

References

[1] P. Hallam, "What do programmers really do anyway," *Microsoft Developer Network (MSDN)—C# Compiler*, 2006.

[2] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[3] M. Geng, S. Wang, D. Dong, S. Gu, F. Peng, W. Ruan, and X. Liao, "Fine-grained code-comment semantic interaction analysis," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 585–596.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 422–431.

[5] W. Maalej and H.-J. Happel, "Can development work describe itself?" in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 2010, pp. 191–200.

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

[7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[8] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[9] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[10] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[11] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[12] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1857–1871, 2019.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

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

[15] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "On the evaluation of neural code summarization," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1597–1608.

[16] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.

[17] A. Shelton, C. J. Lemons, and J. Wexler, "Supporting main idea identification and text summarization in middle school co-taught classes," *Intervention in School and Clinic*, vol. 56, no. 4, pp. 217–223, 2021.

[18] N. D. Bui, Y. Yu, and L. Jiang, "Autofocus: interpreting attention-based neural networks by code perturbation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 38–41.

[19] J. Cito, I. Dillig, S. Kim, V. Murali, and S. Chandra, "Explaining mispredictions of machine learning models using rule induction," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 716–727.

[20] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 441–452.

[21] S. Suneja, Y. Zheng, Y. Zhuang, J. A. Laredo, and A. Morari, "Probing model signal-awareness via prediction-preserving input minimization," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 945–955.

[22] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[23] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 13–22.

[24] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 acm/ieee 32nd international conference on software engineering*, vol. 2. IEEE, 2010, pp. 223–226.

[25] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[26] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.

[27] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[28] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.

[29] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[30] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "Automated comment update: How far are we?" in *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2021, pp. 36–46.

[31] B. Lin, S. Wang, Z. Liu, X. Xia, and X. Mao, "Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach," *IEEE Transactions on Software Engineering*, pp. 1–20, 2022.

[32] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[33] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.-R. Müller, "Explaining nonlinear classification decisions with deep taylor decomposition," *Pattern recognition*, vol. 65, pp. 211–222, 2017.

[34] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[35] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International conference on machine learning*. PMLR, 2017, pp. 3145–3153.

[36] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2921–2929.

[37] B. Poulin, R. Eisner, D. Szafron, P. Lu, R. Greiner, D. S. Wishart, A. Fyshe, B. Pearcy, C. MacDonell, and J. Anvik, "Visual explanation of evidence with additive classifiers," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 21, no. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, p. 1822.

[38] E. Strumbelj and I. Kononenko, "An efficient explanation of individual classifications using game theory," *The Journal of Machine Learning Research*, vol. 11, pp. 1–18, 2010.

[39] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[40] ——, "Anchors: High-precision model-agnostic explanations," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[41] R. Guidotti, A. Monreale, S. Ruggieri, D. Pedreschi, F. Turini, and F. Giannotti, "Local rule-based explanations of black box decision systems," *arXiv preprint arXiv:1805.10820*, 2018.

[42] J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 125–134.

[43] R. Nallapati, B. Zhou, C. Gulcehre, B. Xiang *et al.*, "Abstractive text summarization using sequence-to-sequence rnns and beyond," *arXiv preprint arXiv:1602.06023*, 2016.

[44] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[45] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[46] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[47] B. Alhadidi and M. Alwedyan, "Hybrid stop-word removal technique for arabic language." *Egypt. Comput. Sci. J.*, vol. 30, no. 1, pp. 35–38, 2008.

[48] H. Saif, M. Fernández, Y. He, and H. Alani, "On stopwords, filtering and data sparsity for sentiment analysis of twitter," 2014.

[49] J. Kaur and P. K. Buttar, "Stopwords removal and its algorithms based on different methods," *International Journal of Advanced Research in Computer Science*, vol. 9, no. 5, pp. 81–88, 2018.

[50] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[51] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 275–286.

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

[53] H. Ghader and C. Monz, "What does attention in neural machine translation pay attention to?" *arXiv preprint arXiv:1710.03348*, 2017.

[54] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[55] W. Sun, C. Fang, Y. Chen, Q. Zhang, G. Tao, T. Han, Y. Ge, Y. You, and B. Luo, "An extractive-and-abstractive framework for source code summarization," *arXiv preprint arXiv:2206.07245*, 2022.

[56] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 184–195.

[57] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *arXiv preprint arXiv:1704.04368*, 2017.

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

[59] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.

[60] S. Banerjee and A. Lavie, "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*, 2005, pp. 65–72.

[61] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Transactions on software Engineering*, 2020.

[62] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 1–12.

[63] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 253–264.

[64] R. Shahbazi, R. Sharma, and F. H. Fard, "Api2com: On the improvement of automatically generated code comments using api documentations," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 411–421.

[65] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[66] S. Wiegreffe and Y. Pinter, "Attention is not not explanation," *arXiv preprint arXiv:1908.04626*, 2019.

[67] X. Li, G. Li, L. Liu, M. Meng, and S. Shi, "On the word alignment from neural machine translation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 1293–1303.

[68] F. Mu, X. Chen, L. Shi, S. Wang, and Q. Wang, "Automatic comment generation via multi-pass deliberation," *arXiv preprint arXiv:2209.06634*, 2022.

[69] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1105–1116.

[70] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka, "Tree-to-sequence attentional neural machine translation," *arXiv preprint arXiv:1603.06075*, 2016.

[71] Y.-S. Wang, H.-Y. Lee, and Y.-N. Chen, "Tree transformer: Integrating tree structures into self-attention," *arXiv preprint arXiv:1909.06639*, 2019.

[72] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[73] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[74] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference*, 2019, pp. 2203–2214.

[75] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *arXiv preprint arXiv:2207.01780*, 2022.

[76] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 479–496.

[77] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[78] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.