An Extractive-and-Abstractive Framework for Source Code Summarization

WEISONG SUN, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHUNRONG FANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
YUCHEN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
QUANJUN ZHANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
GUANHONG TAO, Purdue University, USA
YUDU YOU, State Key Laboratory for Novel Software Technology, Nanjing University, China
TINGXU HAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
YIFEI GE, State Key Laboratory for Novel Software Technology, Nanjing University, China
YULING HU, State Key Laboratory for Novel Software Technology, Nanjing University, China
BIN LUO, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

(Source) Code summarization aims to automatically generate summaries/comments for given code snippets in the form of natural language. Such summaries play a key role in helping developers understand and maintain source code. Existing code summarization techniques can be categorized into *extractive methods* and *abstractive methods*. The *extractive methods* extract a subset of important statements and keywords from the code snippet using retrieval techniques and generate a summary that preserves factual details in important statements and keywords. However, such a subset may miss identifier or entity naming, and consequently, the naturalness of the generated summary is usually poor. The *abstractive methods* can generate human-written-like summaries leveraging encoder-decoder models. However, the generated summaries often miss important factual details.

To generate human-written-like summaries with preserved factual details, we propose a novel extractive-and-abstractive framework. The extractive module in the framework performs the task of extractive code summarization, which takes in the code snippet and predicts important statements containing key factual details. The abstractive module in the framework performs the task of abstractive code summarization, which

*Chunrong Fang is the corresponding author.

Authors' addresses: Weisong Sun, weisongsun@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Chunrong Fang, fangchunrong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Yuchen Chen, yuc.chen@outlook.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Quanjun Zhang, quanjun.zhang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Guanhong Tao, taog@purdue.edu, Purdue University, West Lafayette, Indiana, USA, 47907; Yudu You, nju_yyd@163.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Tingxu Han, txhan@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Yifei Ge, gyf991213@126.com, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Yuling Hu, yulinghu@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Bin Luo, luobin@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093; Zhenyu Chen, zychen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 210093.

1049-331X/2022/0-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

takes in the code snippet and important statements in parallel and generates a succinct and human-written-like natural language summary. We evaluate the effectiveness of our technique, called EACS, by conducting extensive experiments on three datasets involving six programming languages. Experimental results show that EACS significantly outperforms state-of-the-art techniques for all three widely used metrics, including BLEU, METEOR, and ROUGH-L. In addition, the human evaluation demonstrates that the summaries generated by EACS have higher naturalness and informativeness and are more relevant to given code snippets.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: Code Summarization, Extractive Code Summarization, Abstractive Code Summarization Program Comprehension

ACM Reference Format:
Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo, and Zhenyu Chen. 2022. An Extractive-and-Abstractive Framework for Source Code Summarization. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 1 (2022), 40 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code comments play a significant role in facilitating code comprehension [30, 101, 114, 117] and software maintenance [10, 15, 36, 120]. Writing high-quality code comments has been recognized as a good programming practice [15, 120]. The code comment is one of the most common summaries used during software development. However, writing code comments is a labor-intensive and time-consuming task [15, 45]. As a result, good comments are often absent, unmatched, and outdated during code evolution [41]. The prior work [41] shows that the lack of high-quality code comments is a common problem in the software industry. (Source) code summarization is a hot research topic [1, 10, 18, 30, 33, 34, 44, 53, 82, 87, 96, 125, 127], which aims to design advanced techniques to support automatic generation of code summaries (i.e., comments). For a code snippet (a method or function) given by the developer, code summarization techniques can automatically generate natural language summaries related to it. Figure 1 shows an example. The code snippet in Figure 1(a) is provided by the developer. The summary "removes the mapping for the specified key from this cache if present." in Figure 1(b) is a summary that satisfies the developer's requirement. The summary is usually a succinct description in natural language summarizing the intention/functionality of the desired code snippet [10].

```
public V remove(final K key) {
   V oldValue = cacheMap.remove(key);
   if (oldValue != null) {
     LOG.debug("Removed cache entry for '{}'", key);
   }
   return oldValue;
}
```

(a) A Code Snippet c_1

removes the mapping for the specified key from this cache if present.

(b) A Summary s_1

Fig. 1. Example of Code Snippet and Summary

Code summarization can be viewed as a special kind of text summarization task, where the text is not written by a traditional natural language but a programming language. Indeed, as

early as ten years ago, automatic text summarization techniques were introduced to automatic code summarization [33], detailed in Section 2.1. Therefore, similar to text summarization [19, 24], code summarization can be subdivided into extractive code summarization (*extractive methods*) and abstractive code summarization (*abstractive methods*). Most of the early code summarization techniques are *extractive methods*, which widely use an indexing-retrieval framework to generate summaries [18, 32, 33, 81]. They first index terms in code snippets and then retrieve top-*n* key terms as summaries. The terms in summaries are extracted from the current code snippet [32, 95], context code snippets [65, 69], or similar code snippets [18, 113]. Therefore, *extractive methods* can produce a summary that preserves the conceptual integrity and factual information of the input code snippet. In addition, *extractive methods* do not require ontology [79] or training data [33]. However, the terms extracted from code snippets may be meaningless words or abbreviations when the identifiers and methods are poorly named. The summaries consisting of these terms are not informative. Besides, the naturalness of the summaries generated by *extractive methods* is usually poor and not as human-written [43, 55], detailed in Section 3.

Recently, with the success of deep learning (DL) in abstractive text summarization, the DL-based code summarization techniques have been proposed one after another [2, 22, 40, 41, 43, 51, 52, 56, 68, 75, 86, 104, 106, 110, 122]. Compared with *extractive methods*, similar to abstractive text summarization [90], the DL-based code summarization techniques have stronger abstract expression capabilities and can generate human-written-like summaries. Therefore, we call the DL-based code summarization techniques as *abstractive methods*. *Abstractive methods* widely adopt the neural network model based on the encoder-decoder architecture and then train the model on a large code-comment corpus. The encoder first transforms code snippets into embedding representations (also called context vectors), and then the decoder decodes the context vectors into short natural language summaries. As in many research areas and as chronicled by Allamanis et al. [3], traditional methods have largely given way to deep learning methods based on big data input. Although the *abstractive methods* have the ability to generate novel words and phrases not featured in the code snippet – as a human-written abstract usually does, the generated summaries often miss important factual details in the code snippet, detailed in Section 3.

In this paper, we propose an extractive-and-abstractive framework for code summarization, which inherits the advantages of extractive and abstractive methods and shields their respective disadvantages. Specifically, we utilize pairs of code snippets and comments to train an extractor (an extractive method) and an abstracter (an abstractive method). The well-trained extractor can be used to predict important statements in code snippets. These important statements and the entire code snippet are input to the abstracter to generate a short natural language summary. The well-trained abstracter first utilizes two separate encoders to transform important statements and the entire code snippet into two context vectors. Then, the two context vectors are fused to produce a fusion vector, which will be passed to a decoder to generate a natural language summary. Compared with existing *abstractive methods*, our extractive-and-abstractive framework is equipped with an extractor, substantially balancing attention on important information and global contextual information, reducing the risk of missing important factual details and improving the overall performance.

In summary, we make the following contributions.

- To the best of our knowledge, we are the first to propose an extractive-and-abstractive framework for code summarization, which inherits the advantages of the extractive and abstractive methods and shields their respective disadvantages. It is a general framework and can be combined with multiple advanced models (see experimental results in Section 5.2.2).

- We implement a code summarization prototype called EACS based on the extractive-and-abstractive framework. EACS is able to generate a succinct natural language summary that is not only human-written-like, but also preserves important factual details.
- We conduct extensive quantitative experiments on three widely used datasets to evaluate EACS. Experimental results show that EACS significantly outperforms state-of-the-art baselines in terms of all three widely used automatic metrics, BLEU, METEOR, and ROUGE-L (detailed in Section 5.2.1). The source code of EACS and all the data used in this paper are released and can be downloaded from the website [99].
- We conduct a qualitative human evaluation to evaluate the summaries generated by EACS and baselines in terms of four aspects: similarity, naturalness, informativeness, and relevance. The statistical results of human scores show that the summaries generated by EACS are more informative and relevant to code snippets (detailed in Section 5.2.5).

The remainder of this paper is organized as follows. Section 2 provides the background of automatic text summarization and neural machine translation. Section 3 shows the motivating example. Section 4 introduces our methodology, i.e., the design of EACS. Section 5 presents automatic evaluation, human evaluation, and case studies in detail. Section 6 introduces some threats to validity. Section 7 presents the related work. We conclude the paper in Section 8.

2 BACKGROUND

2.1 Automatic Text Summarization

When designing EACS, we drew on the advanced ideas and techniques of automatic text summarization. Therefore, we first introduce the background of automatic text summarization.

Automatic text summarization is the task of automatically condensing a piece of text to a shorter version summary while maintaining the important points [19, 24]. According to technical characteristics, text summarization is subdivided into extractive text summarization (*extractive methods*) and abstractive text summarization (*abstractive methods*) [64, 90]. *Extractive methods* assemble summaries by directly selecting words, phrases, and sentences from the source text that capture its most salient content. The generated summaries usually persist salient information of source text [85, 116, 124]. In the early days, researchers adopted various similarity scores based on specific sentence features (keywords, position, length, frequency, linguistic) and metrics (structure-based, vector-based, and graph-based) to estimate salience between a sentence in a text and its reference summary [20, 98]. Recently, with advances in distributed representations of words, phrases, and sentences, researchers have proposed to use these distributed representations to compute similarity scores [76]. Such techniques are further refined by [11, 12, 71] where the representations learned by the encoder are used to choose the most salient sentences. In contrast, *abstractive methods* can generate novel words and phrases not featured in the source text – as a human-written abstract usually does [85]. The abstractive methods can be categorized into three categories [19, 31]: 1) structure-based: using pre-defined structures (e.g. graphs [25], trees [50], rules [27], and templates [73]), 2) semantic-based: using the text semantic representation and the natural language generation systems (e.g. based on information items, predicate arguments, and semantic graphs) [46], and 3) deep-learning-based methods [14, 119]. Recently, research in abstractive text summarization has made significant progress with the help of large pre-trained models [54, 123]. These works show that abstractive methods have a strong potential to produce high-quality summaries that are verbally innovative [90].

Automatic text summarization techniques were introduced to automatic code summarization as early as ten years ago [18, 33]. For example, in 2010, Sonia Haiduc et al. [32] proposed an extractive code summarization technique for the automatic generation of extractive summaries for

source code entities. Extractive summaries are generated by selecting the most important terms in code snippets. In addition, they present a study in [33] to investigate the suitability of various text summarization techniques for generating code summaries. The study results indicate that a combination of text summarization techniques is most appropriate for code summarization and that developers generally agree with the produced summaries. Existing extractive methods usually use text retrieval (TR) techniques to determine the most important n terms for each code snippet. The common TR techniques include Vector Space Model [84], Latent Semantic Indexing [16], and Hierarchical PAM [67]. Considering that the quality of the summaries generated by extractive methods depends heavily on the process of extracting the subset, Paige Rodeghero et al. [81] present an eye-tracking study of programmers and propose a tool for selecting keywords based on the findings of the eye-tracking study. Recently, DL-based code summarization techniques have been proposed one after another [30, 39, 109, 115]. For example, Srinivasan Iyer et al. [43] present the first DL-based method for generating code comments. To produce semantic-preserving code embedding representations, multiple aspects of the code snippet have been explored, including tokens [2, 40, 41, 68, 75], abstract syntactic trees (ASTs) [4, 39, 40, 52, 56, 91, 122], control flows [106] and code property graphs [58]. In addition, these DL-based methods have tried various neural network architectures, such as LSTM [39, 43, 106], Bidirectional-LSTM [55, 110, 122], GRU [40, 52], Transformer [2, 56] and GNN [51, 58]. Similar to abstractive text summarization, these techniques also widely adopt the encoder-decoder models borrowed from neural machine translation (detailed in Section 2.2) to generate natural language summaries. Therefore, we can consider DL-based code summarization techniques as abstractive methods. More extractive and abstractive code summarization techniques are introduced in Section 7.

In this paper, we utilize extractive and abstractive methods together. The former is responsible for extracting important factual details while the latter is responsible for generating human-written-like natural language summary, detailed in Section 4.

Existing works combining extractive and abstractive summarization methods are mainly proposed in NLP, such as [38, 76, 105]. Different from these works whose extractors select important sentences from the source text, our extractor extracts important statements from the source code. Additionally, our abstracter takes in important statements and the source code in parallel to generate summaries instead of considering only important sentences as in [76, 105] or only source text with sentence-level attention as in [38].

2.2 Neural Machine Translation

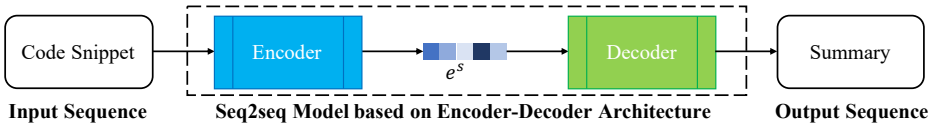


Fig. 2. Framework of DL-based Code Summarization

Neural machine translation (NMT) aims to automatically translate one language (e.g., French) into another language (e.g., English) while preserving semantics [6, 13, 39]. NMT has been shown to achieve great success for natural language corpora [6, 39]. It is typically thought of in terms of sequence to sequence (seq2seq) learning, in which a natural language sentence (e.g., French sentence) is one sequence and is converted into a semantically equivalent target sequence (e.g., English sentence) [52]. Code summarization can also be regarded as a kind of translation task, which translates programming language into natural language [30]. Several recent papers [4, 39, 43, 52]

have explored the idea of applying the seq2seq model to translate code snippets into natural language comments. Similar to NMT, code summarization methods based on seq2seq models also widely adopt the DL-based encoder-decoder architectures. Figure 2 shows the general framework of the code summarization technique based on the encoder-decoder architecture. From the figure, we can observe that the DL-based code summarization technique usually consists of two key components, an encoder and a decoder. Both encoder and decoder are neural networks, so the DL-based code summarization is also known as neural code summarization [87, 122]. The encoder is an embedding network that can encode the code snippet c given by the developer into a d-dimensional embedding representation $\boldsymbol{e}^c \in \mathbb{R}^d$. To train such an encoder, existing DL-based code summarization techniques have tried various neural network architectures, such as LSTM [39, 43, 106], Bidirectional-LSTM [55, 110, 122], GRU [40, 52, 107], Transformer [2, 56, 86] and GNN [51, 58]. The decoder is also a neural network that can decode the embedding representation $\boldsymbol{e}^c$ into a natural language summary. To train such a decoder, existing DL-based code summarization techniques usually adopt the same neural network architecture as the encoder. In DL-based code summarization studies, it is a common practice to use code comments as summaries during the training process [39, 55]. Code comments are natural language descriptions used to explain what the code snippets want to do [39]. For example, Figure 1(b) is a comment for the code snippet c_1. Therefore, we do not strictly distinguish the meaning of the two terms *comment* and *summary*, and use the term *comment* during the training process, and *summary* at other time.

3 MOTIVATING EXAMPLE

```
1. V oldValue = cacheMap.remove(key);
2. if (oldValue != null) {
3. LOG.debug("Removed cache entry for '{}'", key);
```

(a) Important Statements Selected by Our Extractor

1. **Reference Summary:** removes the mapping for the specified key from this cache if present.
2. **Extractive Summary:** removed key cache old value.
3. **Abstractive Summary:** removes the entry from the cache.
4. **ExAbstractive Summary:** removes the value for the given key from the cache if it exists.

(b) Summaries Generated by Different Techniques

Fig. 3. Motivating Example

In this section, we take the code snippet c_1 in Figure 1(a) as an example and apply different tools to generate summaries for comparison. It is a real-world example from the CodeXGLUE dataset [62] (see details in Section 5.1.1). Figure 1(b) shows the comment (s_1) written by the developer for c_1. We consider this as a reference summary (the ground truth), as shown in the first line of Figure 3(b). According to the grammar rules in natural language, we can simply divide the reference summary into four parts: “removes the mapping” (Blue font), “for the specified key” (Green font), “from this cache” (Red font), and “if present”(Orange font).

We study two existing techniques, an extractor method [32] and an abstracter method [21], in generating summaries for the given example. The extractor method [32] adopts the Latent Semantic Analysis (LSA) techniques [97] to determine the informativity of every term in the code snippet and then select the top k important terms to compose the summary. The second summary in Figure 3(b) (Extractive Summary) is generated by [32]. Observe that although the extractive summary has a poor naturalness and is far from the reference summary, it contains important factual details that

should be included in the summary, e.g., the important terms "key" and "cache". The abstractive method [21] first trains a model called CodeBERT for obtaining code representations, and then fine-tunes it on the code summarization task. The third summary in Figure 3(b) (Abstractive Summary) shows the result by [21]. Observe that 1) intuitively, the abstractive summary has a good naturalness and is like written by a human; 2) the abstractive summary can cover the first and the third parts (Blue and Red fonts) of the reference summary; 3) the abstractive summary can not cover the second and the fourth parts (Green and Orange fonts), i.e., missing some factual details. In summary, the main drawbacks of current extractive and abstractive methods are the poor naturalness of the generated code summaries and the loss of important factual details, respectively.

To overcome the drawbacks of existing extractive and abstractive methods, we propose an extractive-and-abstractive framework, i.e., EACS. EACS simultaneously inherits the advantages of both extractive and abstractive methods. For example, the last summary (i.e., ExAbstractive Summary) in Figure 3(b) is generated by EACS for c_1. We can observe that 1) compared with the extractive summary generated by [32], the exabstractive summary has a good naturalness and is like written by a human; 2) compared with the abstractive summary generated by [21], the exabstractive summary can cover all of the four parts and is closer to the reference summary. In the next section, we will introduce the design details of EACS.

4 DESIGN

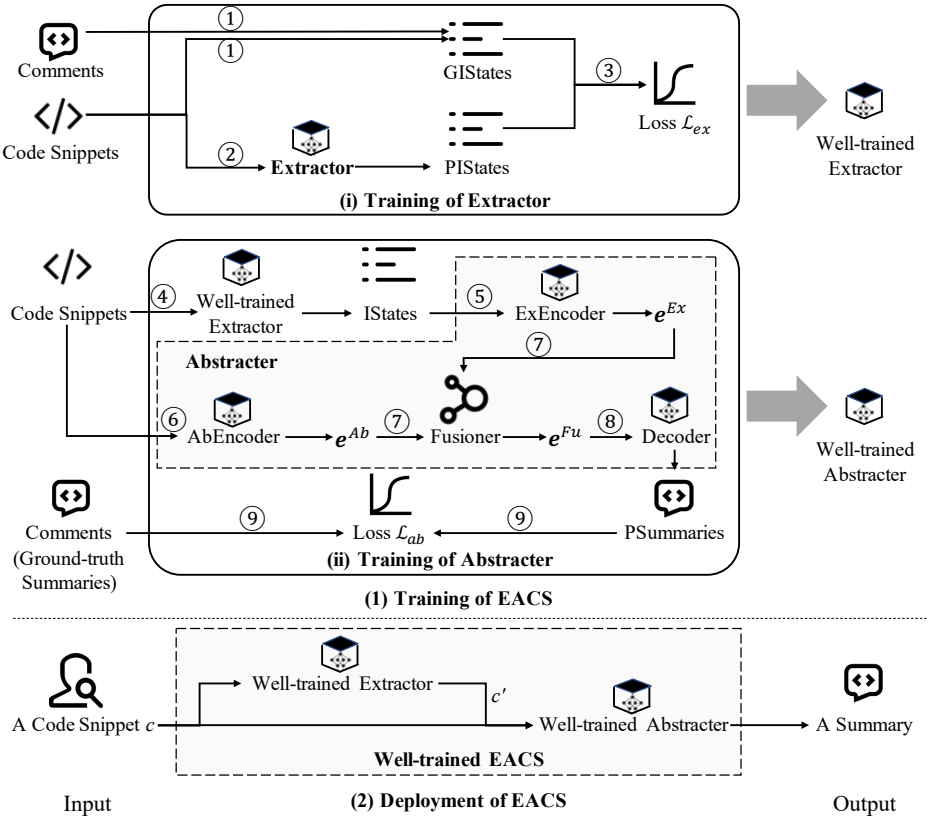


Fig. 4. The Overview of EACS

4.1 Overview

Figure 4 illustrates the overview of our approach EACS. The top and bottom parts show the training and deployment of EACS, respectively. EACS decomposes the training process into two phases: (i) training of extractor and (ii) training of abstracter. Both phases leverage the same two types of input data: comments and code snippets.

The goal of phase (i) is to train an extractor capable of extracting important statements from a given code snippet. To train such an extractor, EACS first produces ground-truth important statements (GIStates) based on the informativity of each statement in the code snippet. Then, EACS uses the extractor to extract predicted important statements (PIStates). During this phase, the model parameters of the extractor are initialized randomly and updated iteratively according to the loss $\mathcal{L}_{Ex}$. $\mathcal{L}_{Ex}$ is computed based on PIStates and GIStates. Details of training of the extractor are described in Section 4.2.1.

The goal of phase (ii) is to train an abstracter capable of generating a succinct natural language summary for a given code snippet. To train such an abstracter, given a code snippet, EACS first uses the well-trained extractor to extract the important statements (IStates). IStates will be further transformed into the embedding representation $\boldsymbol{e}^{Ex}$ through an encoder named ExEncoder. Meanwhile, EACS uses another encoder named AbEncoder to transform the entire code snippet into the embedding representation $\boldsymbol{e}^{Ab}$. Then, EACS produces the fused embedding representation $\boldsymbol{e}^{Fu}$ by fusing $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$. Further, $\boldsymbol{e}^{Fu}$ will be passed to a decoder (Decoder) to generate predicted summaries (PSummaries). During this phase, the model parameters of the abstracter (including ExEncoder, AbEncoder, and Decoder) are also randomly initialized. EACS iteratively updates the abstracter's parameters according to the loss $\mathcal{L}_{Ab}$. $\mathcal{L}_{Ab}$ is computed based on PSummaries and ground-truth summaries (i.e., comments). Details of training of the abstracter are described in Section 4.2.2.

The well-trained extractor and abstracter are the two core components of EACS to support the code summarization service. When EACS is deployed for usage, it takes in a code snippet from the developer and produces a succinct natural language summary for the code snippet, detailed in Section 4.3.

4.2 Training of EACS

4.2.1 *Part (i): Training of Extractor.*

Let c denote a code snippet containing a set of statements $stat = [stat_1, stat_2, \cdots, stat_n]$ where $stat_i$ is the i-th statement in c. Similar to extractive text summarization [59], extractive code summarization can be defined as the task of assigning a label $l_i \in \{0, 1\}$ to each $stat_i$. The label indicates whether the factual details contained in the statement $stat_i$ should be included in the summary. It is assumed that the summary represents the important content of the code snippet. Therefore, we can consider the extractor as a classifier that takes in all statements of a code snippet and predicts the important ones.

As shown in part (i) of Figure 4, the training of the extractor is completed through three steps: ① producing ground-truth important Statements (GIStates), ② producing predicted important statements (PIStates), and ③ calculating the loss ($\mathcal{L}_{Ex}$) based on GIStates and PIStates and updating the model parameters. We discuss the three steps in detail in the following sections.

Step ①: Producing Ground-truth Important Statements. As mentioned earlier, we aim to train an extractor that can predict the important statements in the code snippet c. We denote the labels of the statements in c as $l = [l_1, l_2, \cdots, l_n]$. $l_i \in \{0, 1\}$ is the label of the statement $stat_i$. $l_i = 1$ means the $stat_i$ is an informative (important) statement; otherwise, it is not. To train such an

extractor, we build a training dataset where each sample is a pair of $stat$ and the corresponding labels $\hat{l}$. We consider $\hat{l}$ as the ground-truth label of $stat$.

To obtain the ground-truth labels, like [38], we first measure the informativity of each statement $stat_i \in stat$. Specifically, the informativity is measured by the ROUGE-L score [57] between the statement $stat_i$ and the reference statements. We consider comments of code snippets as reference statements. Then, we sort the statements by their informativity and select them in order of high to low informativity. We select one statement each time. If a newly selected statement can increase the informativity of all the selected statements, it will be regarded as a ground-truth important statement. Finally, we obtain the ground-truth labels, i.e., all ground-truth important statements (GIStates).

In the above process, the ROUGE-L score is computed based on the longest common subsequence. The previous work [126] shows that there are often overlapped strings between identifiers (e.g., "QuickSort") in code snippets and words (e.g., "sort") in summaries. After code preprocessing, all identifiers will be split into multiple words (also called tokens). For example, "QuickSort" would be split into "quick" and "sort". Hence, both token sequences of code snippets and word sequences of summaries can be treated as string sequences (i.e., the same modality) and used to compute the ROUGE-L score.

Existing extractors [11, 38, 60, 70] in NLP widely adopt the ROUGE-L score to select important sentences from the source text. The work [11] gives a reasonable explanation of why the ROUGE score is used, that is, "If the extractor chooses a good sentence, after the abstractor rewrites it the ROUGE match would be high, and thus the action is encouraged. If a bad sentence is chosen, though the abstractor still produces a compressed version of it, the summary would not match the ground truth, and the low ROUGE score discourages this action". In other words, if the ROUGE score is used to guide the selection of important sentences, the summaries generated based on such important sentences can get higher ROUGE scores. We follow the production process of ground-truth labels widely adopted in existing extractors [11, 38, 60, 70]. This process utilizes a greedy strategy to select important sentences [60]. In this way, the ROUGE score between the selected important statements and the reference summary is the highest. And it also indicates that the selected important statements contain the most factual detail.

Step ②: Producing Predicted Important Statements. As shown in part (i) Figure 4, we use an extractor (Extractor) to predict important statements in the code snippet. The extractor is a neural network model consisting of an encoder and a classification layer. The encoder transforms statements into embedding representations. The classification layer predicts labels of statements based on their embedding representations.

The encoder essentially performs the code representation task of transforming complex source code into a numerical (embedding) representation while preserving semantics. Such embedding representations are convenient for program computation. Therefore, a large number of existing deep learning-based code representation techniques can be adopted by our EACS to design the encoder. Specifically, the embedding representations of the statements $\boldsymbol{e}^{stat}$ can be formalized as $\boldsymbol{e}^{stat} = encoder(stat)$. $encoder(\cdot)$ is a neural network architecture (e.g., LSTM [37]) or a pre-trained model (e.g., CodeBERT [21]) that can numericalize sequences of code statements with preserving semantics (i.e., embedding representations). In practice, we tried multiple neural network architectures (e.g., LSTM [37] and Transformer [103]) and pre-trained models (e.g., CodeBERT [21] and CodeT5 [108]). We experimentally found that, in the code summarization task, the encoder obtained by fine-tuning a pre-trained model performed better than that trained from scratch based on the neural network architecture, detailed in Section 5.2.2. We do not repeatedly present the design of the neural network architectures or pre-training models involved in the paper.

Please read the corresponding paper for more details. The classification layer is connected to the embedding representation $\boldsymbol{e}^{stat}$. In practice, we adopt $softmax$ as the classification layer, i.e., $l = softmax(\boldsymbol{e}^{stat})$.

Step ③: Model Training. During training, we update the model parameters Θ of the extractor regarding the sigmoid cross-entropy loss, i.e., $\mathcal{L}_{Ex}(\Theta)$ computed as:

$$\mathcal{L}_{Ex}(\Theta) = -\frac{1}{N}\sum_{n=1}^{N} \hat{l}_n log l_n + (1-\hat{l}_n) log(1-l_n) \tag{1}$$

where $\hat{l}_n \in 0, 1$ is the ground-truth label for the nth statement, and N is the number of statements. $\hat{l}_n = 1$ indicates that the model should pay attention to the factual details contained in the nth statement to facilitate final summary generation.

Once the well-trained extractor is produced, we can use it to extract important statements from the given code snippet. Taking the code snippet c_1 shown in Figure 1(a) as an example, our well-trained extractor extracts three important statements for it, as shown in Figure 3(a). These important statements play a key role in generating the final summary. For example, the ExAbstractive summary shown in the last line of Figure 3(b) is generated based on these important statements. Intuitively, the second part (Green font) is a translation or summary of the factual details (key terms "key") contained in the important statements *V oldValue = cacheMap.remove(key);* and *Log.debug("Removed cache entry for'{}' ", key);*. The fourth part (Orange font) is a translation or summary of the factual details contained in the important conditional statement *if(oldValue) != null)*. We also find that the last two important statements may be related to the logging behavior for debugging purposes and may not be part of the essential code of the function. Considering that the human developers would expect exactly the same summary even without those lines related to logging, we further study the summaries generated by EACS by deleting each important statement in the motivating example and obtain the following results:

(1) deleting the first line yields a summary "removes a value from the cache if it exists";

(2) deleting the second line yields "removes an entry from the cache";

(3) deleting the last line yields "removes the value associated with the key from the cache if present";

Observe that deleting either the first or the second line causes the missing factual detail "for the specified key", while without the last line, the summary is the same as the ground truth. It indicates that the first two lines are critical for having a completely accurate summary. Additionally, the factual detail "if it exists" can only be obtained when the second line presents. This is because the second line checks the return value of the remove function, which validates the existence of the key. Our well-trained extractor precisely captures these details, and EACS can produce the same summary as the reference without the logging line. This example demonstrates the effectiveness of our well-trained extractor.

4.2.2 *Part(ii): Training of Abstracter.*

As shown in part (ii) of Figure 4, the training of the abstracter is completed through six steps: ④ extracting important statements (IStates), ⑤ and ⑥ producing embedding representations ($\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$) of the important statements and the entire code snippet, ⑦ producing the fused representation $\boldsymbol{e}^{Fu}$ based on $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$, ⑧ generating predicted summary, and ⑨ computing the loss $\mathcal{L}_{Ab}$ based on the predicted summaries (PSummaries) and the ground-truth summaries (comments) to update the model parameters. We discuss the six steps in detail as follows.

Step ④: Extracting Important Statements. To generate summaries without missing factual details, our EACS pays more attention to the important statements of code snippets. These important statements contain factual details that should be included in final generated summaries. Therefore,

different from abstracters in existing abstract code summarization techniques, the abstracter of EACS treats important statements as part of the input. In this step, we first use the well-trained extractor to predict the labels of statements of the given code snippet. Then, the statements with the label 1 will be selected as important statements (IStates).

Step ⑤ and Step ⑥: Producing Embedding Representations. Step ⑤ and Step ⑥ do a similar thing, i.e. leveraging an encoder to transform the source code into an embedding representation. The difference is that Step ⑤ deals with important statements selected by the extractor, while Step ⑥ deals with the entire code snippet. Therefore, in EACS, we can use the same neural network architecture or pre-trained model to design ExEncoder and AbEncoder. Given a code snippet $c = [stat_1, stat_2, \cdots, stat_n]$, let $c' \subseteq c$ denote a set of the important statements selected by the extractor from c, the tasks performed by ExEncoder and AbEncoder can be formalized as follows:

$$\boldsymbol{e}^{Ex} = encoder(c'), \quad \boldsymbol{e}^{Ab} = encoder(c) \tag{2}$$

where $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ represent the embedding representations of c' and c; $encoder(\cdot)$ is a neural network architecture (e.g., LSTM [37], Transformer [103]) or pre-trained model (e.g., CodeBERT [21]) that can process sequential input. ExEncoder and AbEncoder perform the similar task — producing embedding representations of the code, so we design them with the same code representation techniques as the extractor's encoder.

Step ⑦: Producing Fused Representation. In this step, EACS fuses $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ to produce a fused embedding representation e^{Fu} through the Fusioner component. Considering that $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ are not aligned, we fuse them in a concatenated fashion. We try two concatenated ways as follows:

$$\boldsymbol{e}^{Fu} = [\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}] \text{ or } [\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}] \tag{3}$$

where $[\cdot;\cdot]$ denotes the concatenation of two vectors. The effects of both ways on the performance of EACS are discussed in Section 5.2.3.

Step ⑧: Generating Predicted Summaries. In this section, we utilize the decoder to generate natural language summaries. The decoder takes in the fused embedding representation $\boldsymbol{e}^{Fu}$ and predicts words one by one. Specifically, the decoder based on a neural network (e.g., LSTM) is to unfold the context vector $\boldsymbol{e}^{Fu}$ into the target sequence (i.e., the word sequence of the summary), through the following dynamic model,

$$\begin{aligned} \boldsymbol{h}_t &= f(y_{t-1}, \boldsymbol{h}_{t-1}, \boldsymbol{e}^{Fu}) \\ p(y_t|Y_{<t}, X) &= g(y_{t-1}, \boldsymbol{h}_t, \boldsymbol{e}^{Fu}) \end{aligned} \tag{4}$$

where $f(\cdot)$ and $g(\cdot)$ are activation functions; $\boldsymbol{h}_t$ is the hidden state of the neural network at time t; y_t is the predicted target word at t through $g(\cdot)$ with $Y_{<t}$ denoting the history $\{y_1, y_2, \cdots, y_{t-1}\}$. The prediction process is typically a classifier over the vocabulary. It can be seen from Equation (4) where the probability of generating a target word is related to the current hidden state, the history of the target sequence and the context $\boldsymbol{e}^{Fu}$. The essence of the decoder is to classify the vocabularies by optimizing the loss function in order to generate the vector representing the feature of the target word y_t. After the vector passes through a *softmax* function, the word corresponding to the highest probability is the result to be output.

Step ⑨: Model Training. During the training of the abstracter, the three components (ExEncoder, AbEncoder, and Decoder) are jointly trained to minimize the negative conditional log-likelihood, i.e, $\mathcal{L}_{Ab}(\Theta)$ computed as:

$$\mathcal{L}_{Ab}(\Theta) = -\frac{1}{N}\sum_{n=1}^{N} logp(\boldsymbol{y}_n|\boldsymbol{x}_n;\Theta) \tag{5}$$

where Θ is the model parameters of the abstracter and each $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ is a (code snippet, comment) pair from the training set.

4.3 Deployment of EACS

After EACS is trained, we can deploy it online for code summarization service. Part (2) of Figure 4 shows the deployment of EACS. For a code snippet c given by the developer, EACS first uses the well-trained extractor to extract important statements from c, represented c'. Then, EACS uses the well-trained abstracter to generate the summary. In practice, we can consider the well-trained EACS as a black-box tool that takes in a code snippet given by the developer and generates a succinct natural language summary.

5 EVALUATION

To evaluate our approach, in this section, we aim to answer the following six research questions:

RQ1: How does EACS perform compared to the state-of-the-art baselines?
RQ2: How does EACS perform in terms of generality?
RQ3: How does the fusion way of the extractor and abstracter affect the performance of EACS?
RQ4: How does the robustness of EACS perform when varying the code length and comment length?
RQ5: How does EACS perform in human evaluation?
RQ6: How does the similarity metric (e.g., BLEU, METEOR, and ROUGE-L) used in the extractor affect the performance of EACS?

5.1 Experimental Setup

5.1.1 Dataset. We conduct experiments on three datasets, including a Java dataset (JCSD) [41], a Python dataset (PCSD) [9], and a CodeSearchNet corpus [42], which have been widely used by existing code summarization studies [2, 21, 29, 109, 115, 122]. JCSD is provided by Hu et al. [41], which contains 69,708 pairs of Java methods and their comments collected from GitHub [28]. They have split training/validation/test sets with 69,708/8,714/8,714. PCSD is provided by Barone et al. [9], which contains 108,726 pairs of Python functions and their comments collected from GitHub. It uses docstrings (document strings) as comments. We follow SiT [115] and evaluate our EACS on the PCSD dataset provided by Wan et al. [104], which has split training/validation/test sets with 65,236/21,745/21,745. The CodeSearchNet corpus provided by Husain et al. [42] contains a large number of pairs of code snippets and comments across six programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby. Lu et al. [62] showed that some comments contain content unrelated to the code snippets and performed data cleaning on the CodeSearchNet corpus. Therefore, in this paper, we follow [21, 108] and use the clean version of the CodeSearchNet corpus called the CodeXGLUE dataset provided by Lu et al. [62]. The statistics of the CodeXGLUE dataset are listed in Table 1. Existing works [87, 88] show that code pre-processing choices can have a large impact on the summarization performance and should not be neglected. For code pre-processing, we follow the work [88] and use the same pre-processing result in EACS and baselines.

5.1.2 Evaluation Metrics. We use three metrics BLEU [74], METEOR [7], and ROUGE [57], to evaluate the model, which are widely used in code summarization [29, 41, 43, 103, 104, 115].

BLEU, the abbreviation for BiLingual Evaluation Understudy [74], is widely used for evaluating the quality of generated code summaries [41, 43, 104]. It is a variant of precision metric, which calculates the similarity by computing the n-gram precision of a generated summary to the reference

Table 1. Statistics of three datasets

Dataset		Training Set Size	Validation Set Size	Test Set Size
JCSD		69,708	8,714	8,714
PCSD		65,236	21,745	21,745
CodeXGLUE	Go	167,288	7,325	8,122
	Java	164,923	5,183	10,955
	JavaScript	58,025	3,885	3,291
	PHP	241,241	12,982	14,014
	Python	251,820	13,914	14,918
	Ruby	24,927	1,400	1,262

summary, with a penalty for the overly short length [74]. It is computed as:

$$BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n) \tag{6}$$

$$BP = \begin{cases} 1, & \text{if } |g| > |r| \\ e^{(1-\frac{|r|}{|g|})}, & \text{if } |g| \leq |r| \end{cases} \tag{7}$$

where $N = 1, 2, 3, 4$ and $w_n = \frac{1}{N}$. p_n is the n-gram precision [88]. BP represents the brevity penalty. g and r denote a generated (predicted) summary and a reference summary, respectively. $|g|$ and $|r|$ denote the lengths of g and r, respectively. In this paper, we follow [29, 115] and show the standard BLEU score which provides a cumulative score of 1-, 2-, 3-, and 4-grams [10].

METEOR, the abbreviation for Metric for Evaluation of Translation with Explicit ORdering [7], is also widely used to evaluate the quality of generated code summaries [106, 118, 122]. For a pair of summaries, METEOR creates a word alignment between them and calculates the similarity scores. Suppose m is the number of mapped unigrams between the reference summary r and the generated summary g, respectively. Then, precision (P_{unig}), recall (P_{unig}), and METEOR are computed as:

$$P_{unig} = \frac{m}{|g|}, \quad R_{unig} = \frac{m}{|r|} \tag{8}$$

$$METEOR = (1 - \gamma * frag^{\beta}) * \frac{P_{unig} * R_{unig}}{\alpha * P_{unig} + (1-\alpha) * R_{unig}} \tag{9}$$

where $frag$ is the fragmentation fraction. As in [122], α, β, and γ are three penalty parameters whose default values are 0.9, 3.0 and 0.5, respectively.

ROUGE-L. ROUGE is the abbreviation for Recall-oriented Understudy for Gisting Evaluation [57]. ROUGE-L, a variant of ROUGE, is computed based on the longest common subsequence (LCS). ROUGE-L is also widely used to evaluate the quality of generated code summaries [8, 56, 86]. Specifically, the LCS-based F-measure (F_{lcs}) is called ROUGE-L [57], and F_{lcs} is computed as:

$$R_{lcs} = \frac{LCS(r,g)}{|r|}, \quad P_{lcs} = \frac{LCS(r,g)}{|g|} \tag{10}$$

$$F_{lcs} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{11}$$

where r and g also denote the reference summary and the generated summary, respectively. Notice that ROUGE-L is 1 when $g = r$; while ROUGE-L is 0 when $LCS(r, g) = 0$, i.e., which means r and g are completely different. β is set to 1.2 as in [104, 107, 122].

The scores of BLEU, ROUGE-L and METEOR are in the range of [0,1] and usually reported in percentages. The higher the scores, the closer the generated summary is to the reference summary, and the better the code summarization performance. All scores are computed by the same implementation provided by [122].

Table 2. Parameter settings

Model	mini-batch size	word embedding size	learning rate	dropout
LSTM	256	256	0.2	0.2
Transformer	32	512	5e-5	0.1
CodeBERT	32	512	5e-5	0.1
CodeT5	32	512	5e-5	0.1

5.1.3 Experimental Settings. Our EACS is a general framework. In this paper, we try to combine EACS with different neural networks (e.g., LSTM and Transformer) and pre-trained models of code (CodeBERT and CodeT5). Therefore, during the experiments, when EACS is combined with different network architectures/pre-trained models, our experimental settings are different. As we all know, neural network hyperparameter setting is a complex task, not only to consider the model size and data size, but also very dependent on the developer's experience. Therefore, the common practice of hyperparameter setting is to follow the successful experience of existing work. Of course, this paper is no exception. For example, when combining EACS with CodeT5, we follow CodeT5 [108] and set the mini-batch size to 32, the word embedding size to 512, the learning rate to 5e-5, and the dropout to 0.1, and update the parameters via AdamW optimizer [47]. The code snippets are padded with a special token $\langle PAD \rangle$ to the maximum length. Table 2 presents the settings of several crucial parameters when EACS is combined with LSTM, Transformer, CodeBERT, and CodeT5. We refer to [110] to set LSTM, and refer to [2] to set Transformer. All models are implemented using the PyTorch 1.7.1 framework with Python 3.8. All experiments are conducted on a server equipped with one Nvidia Tesla V100 GPU with 31 GB memory, running on Centos 7.7. All the models in this paper are trained for the same epochs as their original paper, and we select the best model based on the lowest validation loss.

5.2 Experimental Results

5.2.1 ***RQ1:*** *EACS vs. Baselines.*

1) *Baselines:* To answer this research question, we compare our approach EACS to the following DL-based code summarization techniques.

CODE-NN [43] adopts a LSTM-based encoder-decoder architecture with attention mechanism. It is a classical encoder-decoder framework in NMT that encodes tokens of code snippets into embedding representations and then generates summaries in the decoder with the attention mechanism.

DeepCom [39] also adopts a LSTM-based encoder-decoder architecture with attention mechanism. In addition, to capture the structural information, DeepCom proposes a structure-based traversal method to traverse AST sequences of the code snippet. The AST sequences are further passed to the encoder and decoder to generate summaries.

Hybrid-DRL [104] (also shortened to RL+Hybrid2Seq in [2], Hybrid2Seq in [115]) also adopts a LSTM-based encoder-decoder architecture and is trained with reinforcement learning. It also designs additional encoder based on an AST-based LSTM to capture the structural information

of the code snippet. It uses reinforcement learning to solve the exposure bias problem during decoding, which obtains better performance.

TL-CodeSum [41] (also shortened to API+Code in [115]) adopts a GRU-based encoder-decoder architecture with attention mechanism. It encodes the Application Programming Interface (API) sequence along with the code token sequence, and then generates a summary from the source code with transferred API knowledge. It introduces an API sequence summarization task, aiming to train an API sequence encoder by using an external dataset so that it can learn more abundant representations of the code snippet.

Dual Model [109] also adopts a LSTM-based encoder-decoder architecture with an attention mechanism. It treats code summarization and code generation as a dual task. It trains the two tasks jointly by a dual training framework to simultaneously improve the performance of code summarization and code generation tasks.

Transformer-based [2] (also shortened to Transformer in [115], NCS in [87]) adopts a Transformer-based encoder-decoder architecture. It incorporates the copying mechanism [85] in the Transformer to allow both generating words from vocabulary and copying from the source code.

SiT [115] adopts a Transformer-based encoder-decoder architecture. It proposes structure-induced transformer to capture long-range dependencies and more global information in AST sequences of code snippets.

Re2Com [110] adopts an LSTM-based encoder-decoder architecture with an attention mechanism. It first uses an information retrieval technique to retrieve a similar code snippet and treat its comment as an exemplar. Then, it uses an LSTM-based seq2seq neural network that takes the given code, its AST, its similar code, and its exemplar as input, and leverages the information from the exemplar to generate summaries.

SCRIPT [29] adopts a Transformer-based encoder-decoder architecture. It proposes two types of Transformer encoders to capture the structural relative positions between tokens for better learning code semantics.

CAST [89] hierarchically splits an AST into a set of subtrees and devises a recursive neural network to encode the subtrees. The embeddings are then aggregated for generating the summary. They adopt Transformer as the backbone of the decoder.

CodeBERT [21] is a representative pre-trained model for source code. CodeBERT uses the same model architecture as RoBERTa-base [61]. It is trained with the Masked Language Modeling (MLM) task and the Replaced Token Detection (RTD) task. The authors of CodeBERT fine-tune and test it on the code summarization task (also called the code documentation generation task in their paper).

CodeT5 [108] is the state-of-the-art pre-trained model for source code. CodeT5 builds on an encoder-decoder framework with the same architecture as T5 [78]. It is trained with four pre-training tasks, including Masked Span Prediction (MSP) task, Identifier Tagging (IT), Masked Identifier Prediction (MIP), and Bimodal Dual Generation (BDG). Different from CodeBERT, CodeT5 has a pre-trained decoder. The authors of CodeT5 also conduct experiments on the code summarization task.

It should be noted that, strictly speaking, CodeBERT and CodeT5 are two pre-trained models for source code, not code summarization techniques. Both of them can be used for multiple downstream software engineering tasks (such as code search, code clone detection, and code summarization) by fine-tuning on the corresponding downstream task datasets. CodeBERT is essentially a pre-trained encoder, which can be used to transform input code snippets into numerical representations (i.e., embeddings). CodeT5 consists of a pre-trained encoder and a pre-trained decoder. CodeT5's pre-trained encoder does the same thing as that of CodeBERT. CodeT5's pre-trained decoder can decode

the embeddings produced by the pre-trained encoder into the expected output, such as natural language summaries in the code summarization task. In this paper, we fine-tune the pre-trained CodeBERT and CodeT5 on the code summarization task in the three datasets, i.e., JCSD, PCSD, and CodeXGLUE.

In addition to the abstractive methods introduced above, we implement the following two extractive methods as baselines.

TR-based [32] is an extractive method based on text retrieval (TR). As described in [32, 33], TR-based methods implement the generation of summaries through the following two processes: (1) Extract the text from the code snippet and convert it into a corpus. (2) Determine the most relevant terms for the given code snippet in the corpus and include them in the summary. In process (2), various TR techniques can be integrated, such as Vector Space Model (VSM) [84], Latent Semantic Analysis (LSA) [16, 48], Hierarchical Pachinko Allocation Model (hPAM) [67], to generate code summaries [18]. Since the implementation code for early TR-based methods is no longer available, in this paper, we follow [32, 33] and reproduce a LSA-based method (LSA-based, for short). The LSA-generated summary can contain terms that do not appear in the summarized code snippet but that appear somewhere else in the corpus [18].

Extractor-based (Ex-based, for short) is an extractive method based on our extractor. The design of the extractor is borrowed from extractive text summarization [38]. It should be noted that existing extractors cannot be directly used to summarize code as the extracted important texts essentially are still code. We hence try to generate summaries by directly feeding the important texts extracted by the extractor to CodeT5. In other words, the extractor-based method fine-tunes the pre-trained CodeT5 with only the extracted important statements.

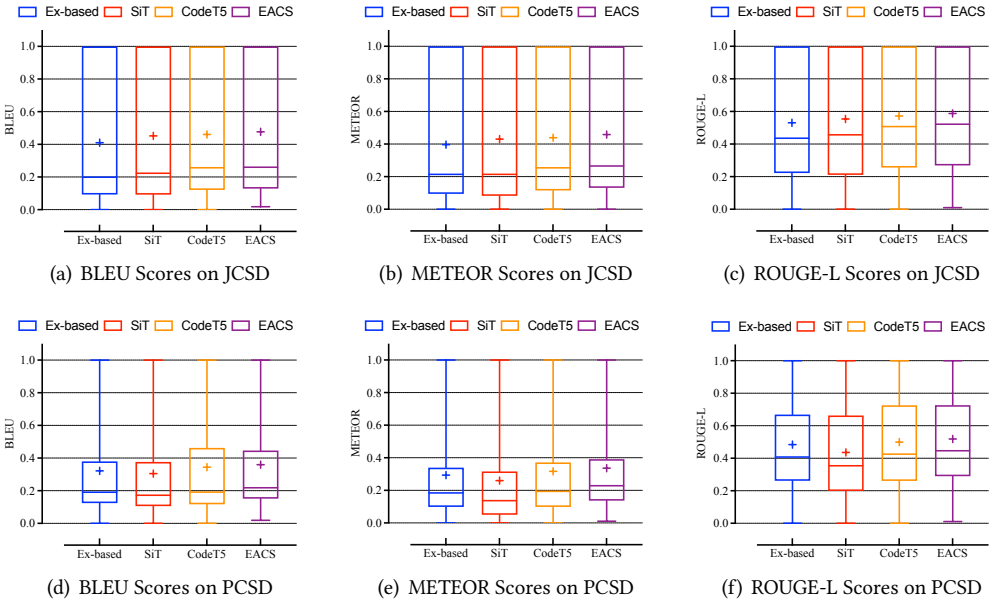
EACS. As described in Section 4, our EACS has two core components, i.e., an extractor and an abstracter. The extractor consists of an encoder and a classification layer. The abstracter consists of two encoders (i.e., ExEncoder and AbEncoder) and a decoder. In total, EACS has three encoders and one decoder. When combining EACS with CodeBERT, we build the three encoders on the pre-trained encoder provided by CodeBERT, and build the decoder on the Transformer architecture. When combining EACS with CodeT5, we build the three encoders and the decoder on the pre-trained encoder and decoder provided by CodeT5, respectively. Compared to fine-tuning CodeBERT and CodeT5 directly, although EACS needs to train an additional extractor, the training of the extractor is a one-time offline task. In addition, the abstracter of EACS has two encoders that are used to transform the important statements and the entire code snippet into embeddings in parallel.

2) *Results:* According to the description in baselines' papers, except for CodeBERT and CodeT5 evaluated on the CodeXGLUE dataset, the rest of the baselines (e.g., Transformer-based, SiT, and SCRIPT) are mainly evaluated on the JCSD and PCSD datasets. Therefore, for a fair and convenient comparison, we are consistent with most baselines and compare our EACS with them on the JCSD and PCSD datasets. In addition, several important baselines (including SiT and SCRIPT) require special data preprocessing, which can not be applied to some programming languages in the CodeXGLUE dataset (e.g., Go, PHP, and Ruby). Therefore, in this section, we only present experimental results on the JCSD and PCSD datasets. We compare Transformer-based, CodeBERT, and CodeT5 on the CodeXGLUE dataset and the results are discussed in Section 5.2.2.

Table 3 shows the performance of our EACS and baselines in terms of the three evaluation metrics, i.e., BLEU, METEOR, and ROUGE-L. From rows 9–11 of Table 3, we can observe that SiT outperforms Transformer-based and earlier baselines on both datasets in terms of all three metrics. Therefore, we rerun the best baseline SiT. Rows 12–16 show the results of the five baselines we rerun. We can observe that SCRIPT slightly outperforms SiT and CodeT5 on the JCSD dataset in terms of BLEU and METEOR, while CodeT5 performs best on the PCSD dataset, followed by SiT and SCRIPT. Overall, SiT is comparable to SCRIPT, and CodeT5 is better than SiT and SCRIPT. It

Table 3. Performance of Our EACS and Baselines. The results in rows 3–8 are reported from Transformer-based [2]. The results in rows 9–10 are reported from SiT [115]. † refers to the method that SiT rerun. The results in row 11 is reported from CAST [89], which currently only supports the Java programming language. ‡ refers to methods we rerun. $\mathcal{A}$: average value ↑; $\mathcal{M}$: median ↑; $\mathcal{S}$: standard deviation ↓. ↑ denotes bigger is better, and ↓ denotes smaller is better.

Methods (Year)	JCSD									PCSD								
	BLEU			METEOR			ROUGE-L			BLEU			METEOR			ROUGE-L		
	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$
CODE-NN (2016)	27.60	–	–	12.61	–	–	41.10	–	–	17.36	–	–	9.29	–	–	37.81	–	–
DeepCom (2018)	39.75	–	–	23.06	–	–	52.67	–	–	20.78	–	–	9.98	–	–	37.35	–	–
Hybrid-DRL (2018)	38.22	–	–	22.75	–	–	51.91	–	–	19.28	–	–	9.75	–	–	39.34	–	–
TL-CodeSum (2018)	41.31	–	–	23.73	–	–	52.25	–	–	15.36	–	–	8.57	–	–	33.65	–	–
Dual Model (2019)	42.39	–	–	25.77	–	–	53.61	–	–	21.80	–	–	11.14	–	–	39.45	–	–
Transformer-based (2020)	44.58	–	–	26.43	–	–	54.76	–	–	32.52	–	–	19.77	–	–	46.73	–	–
Transformer-based†(2020)	44.87	–	–	26.58	–	–	54.95	–	–	32.85	–	–	19.86	–	–	46.93	–	–
SiT (2021)	**45.76**	–	–	**27.58**	–	–	**55.58**	–	–	**34.11**	–	–	**21.11**	–	–	**48.35**	–	–
CAST (2021)	45.19	–	–	27.88	–	–	55.08	–	–	–	–	–	–	–	–	–	–	–
Re2Com‡ (2020)	35.65	15.23	0.36	16.26	12.54	0.35	44.95	32.05	0.35	14.68	13.89	0.09	6.43	4.13	0.05	25.16	20.96	0.12
SiT‡ (2021)	45.22	22.28	0.41	27.10	21.46	0.46	55.38	45.66	0.37	33.75	17.25	0.32	21.02	18.42	0.41	48.33	35.36	0.31
SCRIPT‡ (2022)	**46.41**	23.67	0.42	**28.47**	23.51	0.41	56.57	48.16	0.37	33.52	19.07	0.32	20.80	15.86	0.32	48.09	38.61	0.30
CodeBERT‡ (2020)	43.23	21.96	0.40	26.13	21.09	0.31	54.74	45.57	0.33	32.65	18.30	0.32	20.55	17.50	0.36	48.45	39.87	0.32
CodeT5‡ (2021)	46.08	**25.69**	0.40	27.93	**25.40**	0.51	**57.28**	**50.73**	0.35	**34.39**	**19.15**	0.33	**22.66**	**19.41**	0.43	**49.90**	**42.55**	0.29
TR-based‡	5.59	4.37	0.04	5.24	5.61	0.06	7.58	7.38	0.07	6.95	8.97	0.06	7.12	6.84	0.06	8.04	8.94	0.08
Ex-based‡	**41.00**	19.90	0.39	**25.41**	21.44	0.44	**53.10**	22.24	0.35	**32.15**	19.07	0.30	**21.30**	18.42	0.41	**48.45**	40.76	0.28
EACS	**47.66**	**25.99**	0.40	**30.39**	**26.54**	**0.52**	**58.77**	**52.16**	0.35	**35.96**	**21.79**	0.30	**23.70**	**22.86**	**0.45**	**51.83**	**44.66**	0.28



(a) BLEU Scores on JCSD (b) METEOR Scores on JCSD (c) ROUGE-L Scores on JCSD

(d) BLEU Scores on PCSD (e) METEOR Scores on PCSD (f) ROUGE-L Scores on PCSD

Fig. 5. Distributions of Metrics' Scores on the JCSD and PCSD datasets

should be noted that we rerun SCRIPT on its preprocessed JCSD and PCSD datasets, which differ from the data splits provided by [41] and [104]. Therefore, in the rest of this paper, we mainly compare EACS with SiT and CodeT5. Rows 17–18 show the results of the two extractive methods, i.e., TR-based and Ex-based. We can observe that the TR-based extractive method is significantly worse than the extractor-based extractive method. The last row of Table 3 shows the results of our EACS. On the JCSD dataset, compared with the the-state-of-the-art (CodeT5), EACS improves by 3.56% in BLEU, 8.81% in METEOR, and 2.60% in ROUGE-L. In addition, EACS has a higher median score than CodeT5 on each metric. For example, in ROUGE-L, EACS gets a median score of 52.16, while CodeT5 gets 50.73. In terms of standard deviation, EACS is comparable to CodeT5 with a mean of 0.42. On the PCSD dataset, EACS also outperforms CodeT5, improving by 4.57% in BLEU, 4.59% in METEOR, and 3.87% in ROUGE-L. EACS also has a higher median score than CodeT5 on each metric. For example, in METEOR, EACS gets a median score of 22.86 (19.41 for CodeT5). In terms of standard deviation, EACS obtains a mean of 0.34, better than CodeT5 (0.35). Based on the above observations, we can conclude that our EACS consistently outperforms CodeT5 in terms of three metrics across both datasets.

It should be noted that the values in Table 3 are the average scores of all test samples. For a more comprehensive comparison, we further analyze the distribution of the scores of Ex-based, SiT, CodeT5, and EACS on all test samples, and the statistical results are shown in Figure 5. In Figure 5, '+' denotes the mean, which is the value filled in Table 3. Figure 5(a)-(c) show the box-and-whiskers plots of the BLEU, METEOR, and ROUGE-L scores obtained by the three baselines and EACS on the JCSD dataset. It can be observed that on all three metrics, the median and first quartile associated with EACS are better than those associated with all three baselines. Figure 5(d)-(f) show the box-and-whiskers plots of the scores by the three baselines and EACS on the PCSD dataset. From left to right, the figure shows: (1) in BLEU, the median and first quartile associated with EACS are better than those associated with all three baselines; (2) in METEOR, the median, first quartile, and third quartile associated with EACS are better than those associated with all three baselines; and (3) in ROUGE-L, the median and first quartile associated with EACS are better than those associated with all three baselines. Overall, the metric score distribution of EACS is better than that of Ex-based, SiT, and CodeT5.

We further run a statistical test to assess whether there is enough empirical evidence to claim that there is a difference among the four techniques regarding the three metrics. The statistical test that must be followed depends on the properties of the data [112]. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric test methods. In addition, to provide an answer to the question under study ("Which of the techniques gives the best performance?"), each technique should be individually compared against all other alternatives [5]. We need to perform a pair-wise comparison among the results of each technique, providing a p-value that determines whether statistically significant differences exist.

Specifically, we use the statistical tool GraphPad Prism [93] for non-parametric tests. With an experimental design where each row represents matched data and the data do not follow a Gaussian distribution, Prism performs the Friedman test [92] and Dunn's multiple comparisons test [94] at a significance level of 5% by default. The outcome of the Friedman test tells if there are differences among the groups, but does not tell which groups are different from other groups. In order to determine which groups are different from others, the post-hoc test can be conducted. Probably the most common post-hoc test for the Friedman test is the Dunn test [17]. Therefore, we look at post-test results (reported by Dunn's multiple comparisons test) to see which techniques differ from which other techniques.

Table 4 shows the p-values reported by Prism after performing the Friedman test and Dunn's multiple comparisons test among Ex-based, SiT, CodeT5, and our EACS. From rows 9, 11, and 13 of

Table 4. p-values and effect sizes for metrics used in the automatic evaluation. Effect size: < 0.1: small effect; $0.1- < 0.3$: medium effect; and ≥ 0.3: large effect.

Comparison Group	Result	JCSD			PCSD		
		BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
SiT vs. Ex-based	p-value	< 0.0001	< 0.0001	> 0.9999	< 0.0001	< 0.0001	< 0.0001
	effect size	0.17	0.05	0.00	0.06	0.18	0.14
CodeT5 vs. Ex-based	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	0.27	0.16	0.15	0.06	0.06	0.05
CodeT5 vs. SiT	p-value	< 0.0001	< 0.0001	< 0.0001	> 0.9999	< 0.0001	< 0.0001
	effect size	0.10	0.20	0.15	0.00	0.12	0.08
EACS vs. Ex-based	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	0.38	0.21	0.26	0.29	0.21	0.13
EACS vs. SiT	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	0.21	0.26	0.26	0.35	0.38	0.26
EACS vs. CodeT5	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	0.11	0.06	0.11	0.35	0.27	0.18

Table 4, it is observed that, in all three metrics (i.e., BLEU, METEOR, and ROUGE-L), all p-values between EACS and baselines are less than 0.0001 and smaller than the significant threshold value of 0.05. Based on this, combining the results in Table 3 and Figure 5, we can conclude that the performance of EACS is significantly better than the three baselines.

Further, to assess the magnitude of the improvement of one technique (e.g., EACS) relative to another (e.g., Ex-based), we calculate the effect size for each comparison group (e.g., EACS vs. Ex-based). Dunn's test is a normal approximation to the exact rank sum test statistics. Therefore, according to [23, 102], we use correlation coefficients to estimate the effect size of each comparison group, which is computed as $r = z/\sqrt{n}$. z is Z-score and n is the number of observations in all groups [23]. Rows 4, 6, 8, 10, 12, and 14 of TABLE 4 show the effect size of each comparison group. From rows 10, 12, and 14, in terms of BLEU, the effect size values of the comparison groups EACS vs. Ex-based, EACS vs. SiT, and EACS vs. CodeT5 are 0.38, 0.21, and 0.11, respectively. It means that EACS has a relatively large improvement over Ex-based on BLEU, and a relatively medium improvement over SiT and CodeT5. In terms of METEOR, EACS has a relatively medium improvement over Ex-based and SiT, and a relatively small improvement over CodeT5. In terms of ROUGE-L, EACS has a relatively medium improvement over all three baselines. Similarly, the effect size values of the comparison groups EACS vs. Ex-based, EACS vs. SiT, and EACS vs. CodeT5 on the PCSD dataset also show a superiority of EACS.

In summary, the results and observations above demonstrate that under all experimental settings, our EACS consistently achieves higher performance in all three metrics, which indicates better code summarization performance.

5.2.2 ***RQ2:*** *Generality of EACS.*

The extractive-and-abstractive framework we proposed is a general code summarization framework. We mainly verify the generality of EACS from two aspects: scalability (model-agnostic) and generalizability (language-agnostic). To demonstrate that EACS is model-agnostic, we verify the feasibility of combining EACS with many advanced neural network architectures or code pre-trained models. To demonstrate that EACS is language-agnostic, we evaluate the effectiveness of EACS on a multiple-language dataset.

In this section, we combine EACS with several popular and advanced neural network architectures (including LSTM [37] and Transformer [103]) and pre-trained models (including CodeBERT [21] and

CodeT5 [108]) and explore the potential of EACS in multilingual code summarization tasks. These neural network architectures and pre-trained models are widely used in code summarization [2, 21, 39, 91, 108, 115]. Although intuitively "EACS+architecture/pre-trained model" seems always better than architecture/pre-trained model, further experimental verification is still necessary. Only when the experimental results show that "EACS+architecture/pre-trained model" is better than architecture/pre-trained model can we conclude that EACS does make an additional contribution to improving code summarization. The experimental results are shown in Table 5 and Table 6. In the two tables, rows 3 ("LSTM") and 5 ("Transformer") represent that we implement purely LSTM-based and Transformer-based encoder-decoder frameworks, respectively, to perform the code summarization task. Rows 4 ("EACS + LSTM") and 6 ("EACS + Transformer") represent that on the basis of the above (Rows 3 and 5), the well-trained extractor module is added. Rows 7 and 9 represent that we fine-tune the pre-trained models released on GitHub (i.e., CodeBERT [66] and CodeT5 [83]) on the code summarization task. Analogously, rows 8 ("EACS + CodeBERT") and 10 ("EACS + CodeT5") represent that on the basis of the above (Rows 7 and 9), the well-trained extractor module is added. As mentioned in Section 5.2.1, CodeBERT and CodeT5 can be used for many downstream software engineering tasks. For the code summarization task, CodeBERT and CodeT5 use the CodeXGLUE dataset for evaluation, so we also use this dataset and keep in line with them. In addition, the CodeXGLUE dataset contains a large number of code-comment pairs across six programming languages (including Go, Java, JavaScript, PHP, Python, and Ruby), which can also be used to evaluate the language-agnostic of EACS.

Table 5. Effectiveness of EACS when Combined with Different Neural Network Architectures (LSTM and Transformer) and Pre-trained Models (CodeBERT and CodeT5) on the CodeXGLUE dataset, including Go, Java, and JavaScript.

Methods	Go			Java			JavaScript		
	BLEU ($\mathcal{B}$)	METEOR ($\mathcal{M}$)	ROUGE-L ($\mathcal{R}$)	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$
LSTM	17.8	15.1	35.6	12.2	10.1	24.6	10.4	6.2	17.2
EACS + LSTM	**17.9**	**15.2**	**35.8**	**13.4**	**10.2**	**24.7**	**10.5**	**6.4**	**17.4**
Transformer	19.8	16.2	38.4	15.3	11.8	30.6	11.2	7.4	20.5
EACS + Transformer	**20.1**	**16.8**	**39.2**	**15.8**	**12.3**	**31.2**	**11.5**	**7.6**	**21.3**
CodeBERT	21.1	17.5	43.6	18.0	12.4	35.5	13.3	8.7	24.3
EACS + CodeBERT	**22.2**	**18.7**	**44.8**	**19.3**	**13.8**	**36.8**	**14.3**	**9.6**	**25.1**
CodeT5	21.5	18.9	45.9	20.2	15.3	39.3	15.8	11.2	28.9
EACS + CodeT5	**23.9**	**20.0**	**47.3**	**21.3**	**16.8**	**41.1**	**16.7**	**12.1**	**30.3**

From rows 3–6 of both tables, we can observe that the performance of the combination of EACS and architecture is significantly better than that of only architecture-based methods in terms of all three metrics. Among them, the combination of EACS and Transformer (i.e., "EACS + Transformer") performs the best. In addition, from rows 7–10, we can observe that the performance of the combination of EACS and pre-trained models is significantly better than that of purely pre-trained model-based methods in terms of all three metrics. Among them, the "EACS + CodeT5" performs the best. We can also observe that the combinations of EACS with pre-trained models significantly outperform the combinations with neural network architectures. This is mainly because pre-trained models for code representation are usually trained on larger-scale datasets, so they have stronger code representation capabilities and can represent code semantics more accurately. Based on the above results and observations, we can make a conclusion that our EACS is a general code summarization framework and can be easily combined with advanced neural network architectures

Table 6. Effectiveness of EACS when Combined with Different Neural Network Architectures (LSTM and Transformer) and Pre-trained Models (CodeBERT and CodeT5) on the CodeXGLUE dataset, including PHP, Python, and Ruby.

Methods	PHP			Python			Ruby		
	BLEU ($\mathcal{B}$)	METEOR ($\mathcal{M}$)	ROUGE-L ($\mathcal{R}$)	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$
LSTM	19.5	12.2	29.8	13.9	9.1	23.3	9.4	5.3	16.3
EACS + LSTM	**19.6**	**12.4**	**30.0**	**14.1**	**9.2**	**23.4**	**9.9**	**5.3**	**16.5**
Transformer	21.5	13.9	34.2	15.8	10.6	31.3	10.3	6.4	18.3
EACS + Transformer	**22.3**	**14.1**	**34.8**	**16.3**	**10.9**	**32.1**	**10.7**	**6.7**	**18.8**
CodeBERT	24.6	15.3	39.4	18.7	12.4	34.8	11.2	7.1	20.6
EACS + CodeBERT	**25.4**	**16.1**	**40.2**	**19.4**	**13.2**	**35.8**	**12.4**	**8.2**	**22.6**
CodeT5	25.9	18.2	43.6	20.0	15.1	37.8	14.9	10.8	27.9
EACS + CodeT5	**26.9**	**18.8**	**44.2**	**20.5**	**16.0**	**38.9**	**15.2**	**11.8**	**29.2**

or pre-trained models. We have reasons to believe that EACS can be combined with more advanced neural network architectures or models to exert more powerful performance in the future.

5.2.3 ***RQ3:*** *Influence of Fusion Ways of Extractor and Abstracter on EACS.*

In this section, we further explore the influence of fusion ways between the extractor and abstracter on the performance of EACS. In practice, as mentioned earlier, we try to fuse the outputs of the ExEncoder and AbEncoder by the following two concatenated ways: $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$ and $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$. The experimental results are shown in Table 7. In the table, rows 3 and 4 ($[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$ and $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$) mean that we use the way of concatenation to fuse the embeddings generated by ExEncoder/AbEncoder and AbEncoder/ExEncoder.

Table 7. Influence of Fusion Ways on EACS

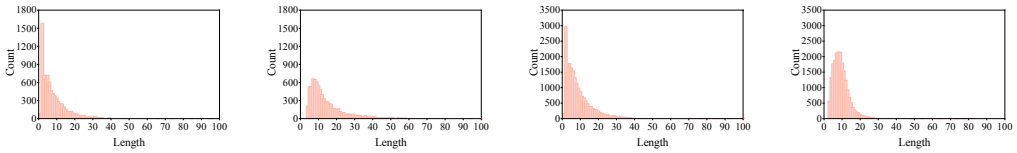
Fusion Ways	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
$[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$	47.27	30.08	58.14	35.56	23.53	51.69
$[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$	**47.66**	**30.39**	**58.77**	**35.96**	**23.70**	**51.83**
p-value	< 0.0001	0.0052	0.0001	< 0.0001	< 0.0001	< 0.0001
effect size	0.21	0.11	0.19	0.11	0.06	0.06

From Table 7, we can observe that, in general, the fusion way $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ is better than $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$. For each metric, we perform the paired Wilcoxon-Mann-Whitney signed-rank test on all scores of both fusion ways at a significance level of 5%. The test results are presented in the penultimate row of Table 7. It is observed that all p-values are smaller than the significant threshold value of 0.05. The last row shows the effect size values of the comparison group $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ vs. $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$. It is observed that on the JCSD dataset, $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ has a relatively medium improvement over $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$ in terms of all three metrics. On the PCSD dataset, $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ has a relatively medium improvement over $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$ in terms of BLEU, while the improvement in METEOR and ROUGE-L is relatively small. It also indicates that $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ has a significantly larger effect on the performance of EACS than $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$.

During model training, the word sequences w^{AbEx} and w^{ExAb} predicted based on $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ and $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$ would be compared with the same reference summary to calculate the loss and update the model parameters. We find that, under the same training epoch, the loss value computed based on w^{AbEx} is less than that based on w^{ExAb}. We conjecture that, compared with w^{ExAb}, the

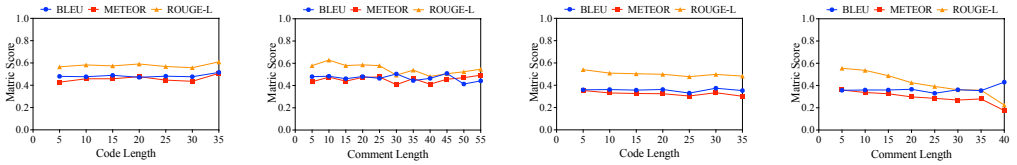
order of words in w^{AbEx} is closer to that in the reference summary. In other words, compared with $[\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}]$, the fusion way of $[\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}]$ makes the model easily learn features of code snippets.

5.2.4 ***RQ4:*** *Robustness of EACS.*



(a) Code Snippets in JCSD (b) Comments in JCSD (c) Code Snippets in PCSD (d) Comments in PCSD

Fig. 6. Length distribution of code snippets and comments in test sets

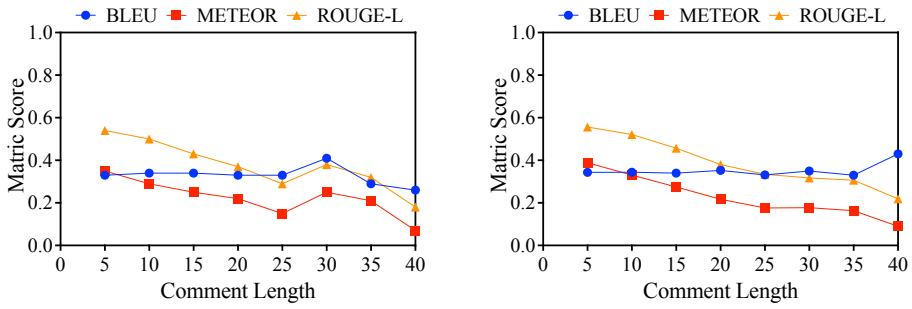


(a) Code Snippets in JCSD (b) Comments in JCSD (c) Code Snippets in PCSD (d) Comments in PCSD

Fig. 7. Effect of code snippet and comment length on the robustness of EACS

To study the robustness of EACS, we analyze two parameters (i.e., code length and comment length) that may have an effect on the embedding representations of code snippets and comments. Figure 6 shows the length distributions of code snippets and comments on the test sets of the JCSD and PCSD datasets. For a code snippet, its length refers to the number of lines of the code snippet. For a comment, its length refers to the number of words in the comment. From Figure 6(a) and (c), it can be observed that the lengths of most code snippets are less than 20. This was also observed in the quote in [63, 100] "Functions should hardly ever be 20 lines long". From Figure 6(b) and (d), it is noticed that almost all comments are less than 20 in length. This also confirms the challenge of capturing the correlation between the long code snippet with its corresponding short comment (summary).

Figure 7 shows the performance of EACS based on different evaluation metrics with varying parameters. From Figure 7(a) and (b), it can be observed that on the JCSD test set, EACS maintains stable performance even though the code snippet length or comment length increases, which can be attributed to the extractive-and-abstractive framework we proposed. On the PCSD test set, from Figure 7(c) and (d), it can be observed that EACS also maintains stable performance when the code snippet length increases, however, the performance of EACS degrades significantly in terms of METEOR and ROUGE-L as the comment length increases. We further analyze the performance of SiT and CodeT5 on varying the PCSD comments, and the results are shown in Figure 8. From this figure, we can observe the same phenomenon as EACS. It means that as the expected length of the generated summary continues to increase, it will be more challenging to generate. Overall, the results verify the robustness of our EACS.



(a) Robustness of SiT (b) Robustness of CodeT5

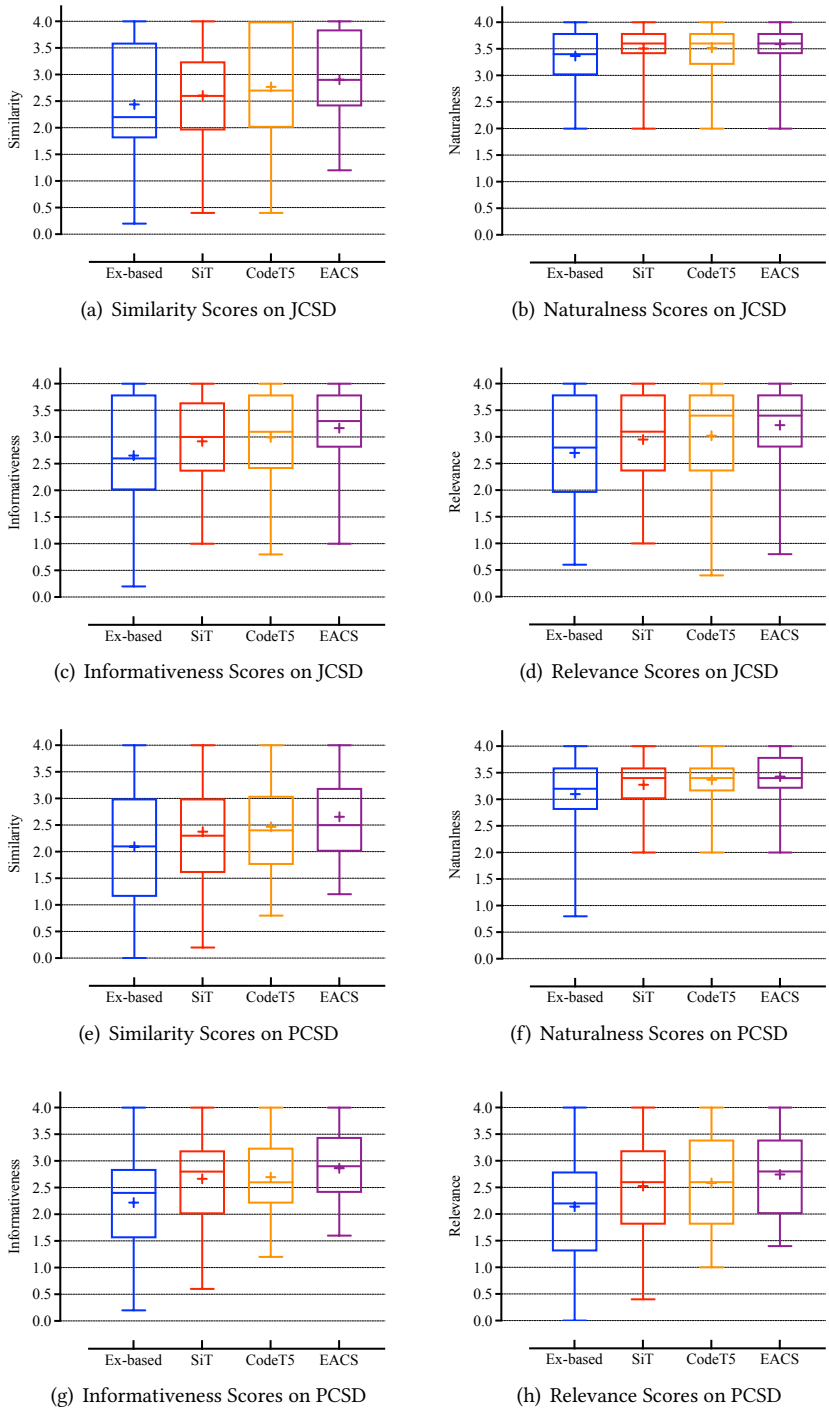
Fig. 8. Effect of comment length on the robustness of SiT and CodeT5

5.2.5 ***RQ5:*** *Human Evaluation.*

Many works [40, 43, 55, 82, 88, 89, 110, 122] show that the automatic evaluation metrics (BLEU, METEOR, and ROUGE-L) mainly calculate the textual similarity rather than the semantic similarity between the reference summaries and generated summaries. Hence, we conduct a human evaluation by following the previous works [40, 89, 110, 115, 122] to evaluate the summaries generated by the baselines Ex-based, SiT, CodeT5, and our EACS. Specially, we invite 10 volunteers with more than 3 years of software development experience and excellent English ability to carry out the evaluation. Each volunteer is asked to assign scores from 0 to 4 (the higher, the better) to generated summaries from four aspects: *similarity* (similarity of the generated summaries and the reference summaries), *naturalness* (grammaticality and fluency), *informativeness* (the amount of content carried over from the input code snippets to the generated summaries, ignoring fluency) and *relevance* (the degree to which the generated summaries are relevant with the input code snippets). We randomly select 100 code snippets, including 50 from the JCSD dataset and 50 from the PCSD dataset, the corresponding summaries generated by Ex-based, SiT, CodeT5, and our EACS, and the reference summaries (i.e., ground-truth), respectively. We divide the 100 samples into two groups, and each of them includes 50 samples, of which 25 belong to the JCSD dataset and 25 belong to the PCSD dataset. To reduce the workload of volunteers and ensure the fairness of experimental results, each volunteer randomly evaluates only one group of samples. Each summary is evaluated by five volunteers, and the final score is the average of them.

The results of the human evaluation are shown in Table 8. The standard deviations of all methods (the values in all parentheses) are small, which indicates that their scores by humans are about the same degree of concentration [55]. From Table 8, it can be observed that overall our EACS consistently outperforms Ex-based, SiT, and CodeT5 in all four aspects. On the JCSD dataset, compared with Ex-based, SiT, and CodeT5, EACS improves on average by 15.41%, 7.33%, and 4.55% in four aspects, respectively, while on the PCSD dataset, EACS improves on average by 22.18%, 7.75%, and 5.04%, respectively. Although from the average values in rows 6 and 11 of Table 8, EACS is slightly better than the best baseline CodeT5, the improvement of EACS to CodeT5 is greater than that of CodeT5 to SiT. Specifically, EACS improves CodeT5 by 4.55% and 5.04% on the JCSD and PCSD datasets, while CodeT5 improves SiT by 2.67% and 2.58%, respectively. In other words, EACS improves CodeT5 nearly twice as much as CodeT5 improves SiT.

For a more comprehensive comparison, we further analyze the distribution of the scores of Ex-based, SiT, CodeT5, and EACS on all samples, and the statistical results are shown in Figure 9.



(a) Similarity Scores on JCSD

(b) Naturalness Scores on JCSD

(c) Informativeness Scores on JCSD

(d) Relevance Scores on JCSD

(e) Similarity Scores on PCSD

(f) Naturalness Scores on PCSD

(g) Informativeness Scores on PCSD

(h) Relevance Scores on PCSD

Fig. 9. Distributions of scores of human evaluation on the JCSD and PCSD datasets

Table 8. Results of human evaluation. The values in parentheses represent standard deviations.

Dataset	Metrics	Ex-based	SiT	CodeT5	EACS
JCSD	Similarity	2.44 (0.48)	2.60 (0.48)	2.77 (0.40)	**2.90 (0.37)**
	Naturalness	3.36 (0.56)	3.51 (0.47)	3.52 (0.44)	**3.57 (0.43)**
	Informativeness	2.65 (0.52)	2.92 (0.56)	3.00 (0.51)	**3.17 (0.52)**
	Relevance	2.70 (0.59)	2.95 (0.53)	3.02 (0.51)	**3.23 (0.50)**
Average		2.79	3.00	3.08	**3.22**
PCSD	Similarity	2.09 (0.47)	2.38 (0.49)	2.47 (0.56)	**2.66 (0.41)**
	Naturalness	3.10 (0.65)	3.28 (0.60)	3.37 (0.54)	**3.43 (0.54)**
	Informativeness	2.22 (0.59)	2.66 (0.61)	2.70 (0.65)	**2.86 (0.59)**
	Relevance	2.14 (0.55)	2.53 (0.63)	2.58 (0.66)	**2.74 (0.52)**
Average		2.39	2.71	2.78	**2.92**

Table 9. p-values and effect sizes for metrics used in the human evaluation. Effect size: < 0.1: small effect; $0.1- < 0.3$: medium effect; and ≥ 0.3: large effect.

Comparison Group	Result	JCSD				PCSD			
		Similarity	Naturalness	Informativeness	Relevance	Similarity	Naturalness	Informativeness	Relevance
SiT vs. Ex-based	p-value	> 0.9999	0.5302	0.9794	0.728	> 0.9999	> 0.9999	0.1208	0.6229
	effect size	0.14	0.24	0.20	0.22	0.15	0.09	0.33	0.23
CodeT5 vs. Ex-based	p-value	0.0791	0.1089	0.1339	0.2641	> 0.9999	0.1339	**0.0358**	0.2406
	effect size	0.35	0.33	0.32	0.28	0.18	0.32	0.39	0.29
CodeT5 vs. SiT	p-value	0.7856	> 0.9999	> 0.9999	> 0.9999	> 0.9999	0.575	> 0.9999	> 0.9999
	effect size	0.21	0.09	0.13	0.07	0.03	0.24	0.06	0.06
EACS vs. Ex-based	p-value	**0.0002**	**0.0078**	**0.0005**	**0.0003**	**0.0014**	**0.0172**	**0.0003**	**0.0019**
	effect size	0.59	0.45	0.55	0.57	0.52	0.42	0.58	0.51
EACS vs. SiT	p-value	**0.009**	0.7856	0.0709	0.0791	0.0568	0.1089	0.4882	0.2895
	effect size	0.45	0.21	0.36	0.35	0.37	0.33	0.25	0.28
EACS vs. CodeT5	p-value	0.575	> 0.9999	0.6229	0.2641	0.098	> 0.9999	> 0.9999	0.728
	effect size	0.24	0.21	0.23	0.28	0.34	0.10	0.19	0.22

In Figure 9, '+' denotes the mean, which is the value filled in Table 8. Figure 9(a)-(d) show the box-and-whiskers plots of the similarity, naturalness, informativeness, and relevance scores obtained by the three baselines and EACS on the JCSD dataset. It can be observed that on all four metrics, the first quartile associated with EACS is better than those associated with all three baselines. Figure 5(e)-(h) show the box-and-whiskers plots of the scores by the three baselines and EACS on the PCSD dataset. In similarity, the median, first quartile, and third quartile associated with EACS are better than those associated with all three baselines. In naturalness, the third quartile associated with EACS is better than those associated with all three baselines. In informativeness, the median, first quartile, and third quartile associated with EACS are better than those associated with all three baselines. In relevance, the median and first quartile associated with EACS are better than those associated with all three baselines. Overall, the metric score distribution of EACS is better than that of Ex-based, SiT, and CodeT5. In particular, the first quartile associated with EACS is better than those associated with the best baseline CodeT5 on all four metrics. This means that EACS generates fewer low-scoring summaries compared to CodeT5.

We further run a statistical test to assess whether there is enough empirical evidence to claim that there is a difference among the four techniques regarding the four metrics. Table 9 shows the p-values reported by Prism after performing the Friedman test and Dunn's multiple comparisons test among Ex-based, SiT, CodeT5, and our EACS. From this table, we can observe that the p-values

Table 10. p-values and effect sizes for metrics used in the human evaluation. In this table, p-values are reported by the paired Wilcoxon-Mann-Whitney signed-rank test [111] at a significance level of 5%. Effect size: < 0.1: small effect; $0.1- < 0.3$: medium effect; and ≥ 0.3: large effect.

Result	JCSD				PCSD			
	Similarity	Naturalness	Informativeness	Relevance	Similarity	Naturalness	Informativeness	Relevance
p-value	0.031	0.070	0.032	0.018	0.002	0.105	0.024	0.012
effect size	0.24	0.21	0.23	0.28	0.34	0.10	0.19	0.22

for most comparison groups are larger than the significance threshold of 0.05. Only the scores of EACS and Ex-based on all four indicators are significantly different, as shown in row 9 of Table 9. The main reason is that the experimental samples in human evaluation are relatively small. In other words, more samples are needed to detect statistically significant differences. This is corroborated by the automated evaluation (as the p-values in Table 4).

To assess the magnitude of the improvement of one technique relative to another, we calculate the effect size for each comparison group (e.g., EACS vs. Ex-based). Rows 4, 6, 8, 10, 12, and 14 of TABLE 9 show the effect size of each comparison group. For example, from row 10, we can observe that all effect size values of the comparison group EACS vs. Ex-based are larger than 0.3, which means that EACS has a relatively large improvement over Ex-based on all four metrics.

The p-values shown in Table 9 are reported by Dunn's multiple comparisons test with correction. If we compare EACS and CodeT5 alone, we can get the statistical test results as shown in Table 10. From row 2 of Table 10, it is observed that except for naturalness, the p-values are less than the significance threshold of 0.05. Combining the effect sizes shown in row 3 of Table 10, we can conclude that overall EACS is superior to CodeT5, especially with significantly medium improvements in similarity, informativeness, and relevance of generated summaries.

Table 11. Number statistic. Values in bold in each column are the maximum values. Columns ≥ 1, ≥ 2, and ≥ 3 show the number of samples that get a metric score larger than 1, 2, and 3, respectively.

Dataset	Methods	Similarity			Naturalness			Informativeness			Relevance		
		≥ 1	≥ 2	≥ 3	≥ 1	≥ 2	≥ 3	≥ 1	≥ 2	≥ 3	≥ 1	≥ 2	≥ 3
JCSD	Ex-based	45	36	17	**50**	**50**	41	47	39	19	48	39	24
	SiT	48	38	20	**50**	**50**	46	**50**	44	26	**50**	43	27
	CodeT5	48	40	23	**50**	**50**	44	49	43	28	48	44	31
	EACS	**50**	**42**	**25**	**50**	**50**	**47**	**50**	**47**	**34**	**50**	**47**	**35**
PCSD	Ex-based	41	28	14	**50**	48	37	45	33	13	42	32	11
	SiT	48	30	17	**50**	**50**	40	48	42	19	48	35	15
	CodeT5	48	32	15	**50**	**50**	44	**50**	41	19	**50**	37	14
	EACS	**50**	**42**	**20**	**50**	**50**	**47**	**50**	**47**	**25**	**50**	**40**	**18**

We count the number of samples for different score bands. Table 11 shows the statistical results. From the table, it is observed that, EACS overall outperforms all three baselines on all four metrics, especially for scores ≥ 2 and ≥ 3. For example, among the randomly selected 100 code snippets, in similarity, our EACS can generate comments with scores ≥ 3 for 45 code snippets, while the best baseline CodeT5 can only achieve 38. In naturalness, EACS can generate comments with scores ≥ 3 for 94 code snippets, outperforming CodeT5 (88). In informativeness, EACS can generate comments with scores ≥ 3 for 59 code snippets and better than CodeT5 (47). In relevance, our EACS can generate comments with scores ≥ 3 for 53 code snippets and outperforms CodeT5 (45). Referring

to [122], a summary with a score ≥ 3 can be regarded as a good summary. Based on this, we can conclude that EACS can generate good summaries for more code snippets than CodeT5.

5.2.6 ***RQ6:*** *Effect of Similarity Metrics.*
In ground-truth important statement generation, we follow existing extractors [11, 38, 60, 70] in NLP and use the ROUGE-L score to measure the informativity of each statement. In existing research (including code summarization and text summarization), ROUGE-L is only one of many commonly used similarity metrics. Other similar metrics, such as BLEU and METEOR, can also be used as a replacement for ROUGE-L in the extractor. In addition, some recent studies [35, 82] have also found that some metrics (e.g., chrF [77] and Sentence-BERT [80]) are more suitable for code summarization tasks than the above three metrics. chrFz works solely on character n-grams rather than word n-grams [77]. It can be seen as a character n-gram F-score. Haque et. al [35] experimentally found that the Sentence-BERT [80] produced vectorized representations of the summaries that have the highest correlation to perceived similarity by the human experts. These metrics may also be good choices. Therefore, we conduct an experiment to explore the effect of different similarity metrics on the performance of EACS. And experimental results are shown in Table 12.

Table 12. Effect of similarity metrics used in the extractor on EACS. SBERT: Sentence-BERT. $\mathcal{A}$: average value ↑; $\mathcal{M}$: median ↑; $\mathcal{S}$: standard deviation ↓.

Metric	JCSD									PCSD								
	BLEU			METEOR			ROUGE-L			BLEU			METEOR			ROUGE-L		
	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$	$\mathcal{A}$	$\mathcal{M}$	$\mathcal{S}$
BLEU	46.86	25.60	0.41	29.32	**26.80**	0.40	58.60	**52.70**	**0.35**	35.33	20.30	0.32	23.49	20.59	0.32	51.67	44.44	0.29
METEOR	46.74	25.27	0.41	29.21	26.67	0.40	58.34	52.14	**0.35**	35.31	20.16	0.32	23.52	20.50	0.32	51.68	44.31	0.29
ROUGE-L	**47.66**	**25.99**	**0.40**	**30.39**	26.54	**0.39**	**58.77**	52.16	**0.35**	**35.96**	**21.79**	**0.30**	**23.70**	**22.86**	**0.30**	**51.83**	**44.66**	**0.28**
chrF	46.73	25.41	0.41	29.23	26.64	**0.39**	58.45	52.36	**0.35**	35.30	20.30	0.32	23.53	20.63	0.32	51.72	44.44	0.29
SBERT	46.80	25.56	**0.40**	29.30	26.74	**0.39**	58.53	52.66	**0.35**	35.31	20.31	0.31	23.50	20.61	0.31	51.70	44.32	**0.28**

In Table 12, the first column shows the similarity metrics used in the extractor. From this table, it is observed that compared with BLEU, METEOR, chrF, and Sentence-BERT (SBERT), the similarity metric ROUGE-L overall contributes more to the score of the final generated summaries (e.g., with the highest average values and the lowest standard deviations). In other words, if we use the similarity metric ROUGE-L to guide the selection of ground-truth important statements in the extractor, the final summaries generated by the abstracter will achieve higher BLEU, METEOR, and ROUGE-L scores. Although some differences between programming languages and natural language, we get the same observation as researchers in NLP, that ROUGE-L is suitable for generating ground-truth important statements/sentences and thereby helps to improve code/text summarization.

5.3 Case Study

In this section, we provide case studies to understand the generated summaries of EACS compared with Ex-based, SiT, and CodeT5 to demonstrate the usefulness of our EACS. Specifically, we will present several cases, including two successful cases, two moderate cases, and two weak cases. We apply Ex-based, SiT, CodeT5, and our EACS to generate summaries for comparison. In all examples, as in Section 3, we treat comments of code snippets as reference summaries.

5.3.1 Case Study on Java Code Summarization.
In this section, we present three Java cases, including a successful case, a moderate case, and a weak case. They are real-world examples from the JCSD test set.

```
public synchronized void listen(final Set<NotificationChannel> channelNames){
  m_channels.addAll(channelNames);
}
```

(a) A code snippet c_2 from the test set of the JCSD database

```
m_channels.addAll(channelNames);
```

(b) Important statements selected by our extractor

Reference Summary: adds the set of channel names to the set of listened channels.
Ex-based: adds the given channel names to the filter
SiT: remove the channel names from the channel names.
CodeT5: add a set of channel names to the list of those currently registered.
EACS: add a set of channel names to the set of listened channels.

(c) Summaries generated by different techniques

Fig. 10. Successful Java Case

Successful Case. Figure 10 shows a successful Java case. The first line of Figure 10(c) shows the reference summary. The summaries generated by Ex-based, SiT, CodeT5, and EACS for c_2 are shown in lines 2–5. From the figure, we can observe that compared with the reference summary, the summary generated by SiT only covers the main semantics of the second part, i.e., "the channel names" (Red font). What is even worse is that the first core word "remove" generated by SiT and that of "adds" in the reference summary have completely opposite semantics, which makes the semantics of the generated summary and the reference summary completely opposite. Compared with SiT, the summaries generated by Ex-based and CodeT5 cover the core word "add" and the main semantics of the second part ("channel names"). In summary, in all three baselines, CodeT5 performs the best, followed by Ex-based and SiT. Compared to CodeT5, the summary generated by EACS covers all three parts of the reference summary. Technically, CodeT5 can be considered as an abstracter. The abstracter of EACS has one additional component, i.e., ExEncoder, which is responsible for embedding important statements extracted by the extractor. Therefore, we can attribute the successful generation of the third part of the reference summary (i.e., "to the set of listened channels") to the extractive-and-abstractive framework we proposed. Intuitively, the text "the set of listened channels" in the third part is the summary of the code identifier "m_channels" contained in the important statement in Figure 10(b). Based on the above, we can conclude that our EACS is also a very competitive technique for the Java code summarization task.

Moderate Case. Figure 11 shows a moderate Java case. In this example, according to the specific semantics of the code snippet c_3, the reference summary in the first line of Figure 11(c) can be split into two clauses: "creates a new item" and "binds the parameters to the new item". From lines 2-5 of Figure 11(c), it is observed that 1) all four techniques cover the core semantic of the first clause, i.e., "creates a new item" (highlighted in Blue and Orange font); 2) SiT and EACS cover the most factual details (e.g., "bind" and "parameters") contained in the reference summary and outperform Ex-based and CodeT5. However, for the second clause "binds the parameters to the new item", the summary generated by SiT is incorrect. Similarly, EACS erroneously summarizes "bind it (i.e., the new item) to the (given) parameters". Of course, it is undeniable that our extractor still performs well in extracting important statements. From this case, we can find that it is still

```
private static Object createAndBindItem(ParameterAccess parameters, Class < ? > itemType) {
    try {
        Object value = itemType.newInstance();
        bind(value, parameters);
        return value;
    } catch(InstantiationException | IllegalAccessException e) {
        throw new BindingException(_STR + itemType, e);
    }
}
```

(a) A code snippet c_3 from the test set of the JCSD database

```
private static Object createAndBindItem(ParameterAccess parameters, Class < ? > itemType) {
Object value = itemType.newInstance();
throw new BindingException(_STR + itemType, e);
```

(b) Important statements selected by our extractor

Reference Summary: creates and binds the parameters to a new item.
Ex-based: creates and returns an instance of the specified type.
SiT: create an instance of a given item type on the given parameters and bind it.
CodeT5: creates a single item of given items.
EACS: create a new instance of the given class and bind it to the given parameters.

(c) Summaries generated by different techniques

Fig. 11. Moderate Java Case

challenging for the current code summarization models (including our EACS) to understand the fine-grained positional relationships among code elements, especially when such relationships are not explicitly presented in the code (e.g., "bind(value, parameters)") or in a form that is prone to misleading the model (e.g., "createAndBindItem").

Weak Case. Figure 12 shows a weak Java case. In this example, the reference summary in the first line of Figure 12(c) contains two sentences. The first sentence introduces the functionality of the code snippet c_4. The second sentence describes the condition that triggers such a functionality (highlighted in Orange font). From lines 2-5 of Figure 12(c), it is observed that 1) Ex-based, CodeT5, and our EACS only cover the first half of the first sentence, while SiT only covers the second half; 2) all four techniques fail to cover the second sentence. As shown in Figure 12(b), our extractor successfully extracts the important statement, which provides key information for the abstracter to summarize "drops it from the connection pool". Unfortunately, the abstracter still fails to summarize the second half of the first sentence. This means that our abstracter needs to be further optimized and improved. In addition, although the second sentence does not describe the core functionality of c_4, it provides very valuable information for developers to understand and use c_4. From this case, we can find that the summaries generated by current code summarization models (including our EACS) are still not as good as professional human developers, and there is still much room for improvement.

5.3.2 Case Study on Python Code Summarization.

In this section, we present three Python cases, including a successful case, a moderate case, and a weak case. They are real-world examples from the PCSD test set.

Successful Case. Figure 13 shows a successful Java case. c_5 is a long piece that consists of 51 lines of code. Figure 13(b) presents the important statements selected by our extractor from c_3. Figure 13(c) contains five summaries, the first from the comment of c_5, the second to five summaries are generated by Ex-based, SiT, CodeT5, and our EACS, respectively. From the figure, we can observe that compared with the reference summary, 1) the summary generated by SiT misses

```
private void destroy() {
   if(log.isLoggable(Level.FINEST)) {
      log.finest(this + _STR);
   }
   try {
      _socket.forceShutdown();
   } catch(Throwable e) {}
   try {
      closeConnection();
   } catch(Throwable e) {
      log.log(Level.FINER, e.toString(), e);
   }
   _port.removeConnection(this);
}
```

(a) A code snippet c_4 from the test set of the JCSD database

```
_port.removeConnection(this);
```

(b) Important statements selected by our extractor

Reference Summary: destroy kills the connection and drops it from the connection pool. destroy should only occur if the connection state machine has failed.
Ex-based: destroys the connection. this implementation closes the connection.
SiT: removes this connection from the port.
CodeT5: close the connection without removing all resources.
EACS: destroy the connection (force shutdown).

(c) Summaries generated by different techniques

Fig. 12. Weak Java Case

```
def findRefPath(startObj, endObj, maxLen=8, restart=True, seen={}, path=None, ignore = None):
   refs = []
   if (path is None):
      path = [endObj]
   …
   newRefs = [r for r in gc.get_referrers(endObj) if (id(r) not in ignore)]
   …
   for r in newRefs:
      …
      refs.append((p + [r]))
   return refs
```

(a) A code snippet c_5 from the test set of the PCSD database

```
def findRefPath(startObj, endObj, maxLen=8, restart=True, seen={}, path=None, ignore = None):
```

(b) Important statements selected by our extractor

Reference Summary: determine all paths of object references from startobj to endobj.
Ex-based: find a path between start obj and end obj.
SiT: search for a path.
CodeT5: find a path of reference objects to the start and end object.
EACS: find reference paths between start and end objects.

(c) Summaries generated by different techniques

Fig. 13. Successful Python Case

many factual details, such as “from startobj to endobj”; 2) although the summary generated by CodeT5 is literally similar to the reference summary, it has completely different semantics; 3) the summaries generated by Ex-based and EACS cover the main semantics of the reference summary

although they look different literally. In addition, compared with Ex-based, EACS can correctly cover more complete semantics. For example, the summary text "reference paths" generated by EACS is semantically equivalent to the text "paths of object references" in the reference summary, although the expressions are different. In addition, the expression "find reference paths" is aligned with the function name "findRefPath". It should be noticed that, as shown in Figure 13(b), our extractor successfully extracts the important statement in which the factual detail "reference" appears. Therefore, we can attribute the successful generation of the text "reference paths" to the extractive-and-abstractive framework we proposed. In summary, our EACS significantly outperforms the other three baselines in terms of the semantic completeness of the generated summaries. Based on the above, we can conclude that our EACS is also a very competitive technique for the Python code summarization task.

```
def addAttributeWord(evaluatorWords, word):
    if (len(word) < 2):
        evaluatorWords.append(word)
        return
    if (word[0] != '.'):
        evaluatorWords.append(word)
        return
    dotIndex = word.find('.', 1)
    if (dotIndex < 0):
        evaluatorWords.append(word)
        return
    evaluatorWords.append(word[:dotIndex])
    addAttributeWord(evaluatorWords, word[dotIndex:])
```

(a) A code snippet c_6 from the test set of the PCSD database

```
def addAttributeWord(evaluatorWords, word):
```

(b) Important statements selected by our extractor

Reference Summary: add attribute word and remainder if the word starts with a dot.
Ex-based: adds a word to the given word.
SiT: add a single word to the evaluator words.
CodeT5: add the word and return a list of word.
EACS: add the word to the list of attribute words.

(c) Summaries generated by different techniques

Fig. 14. Moderate Python Case

Moderate Case. Figure 14 shows a moderate Python case. Similarly, we can divide the reference summary of c_6 shown in the first line of Figure 14(c) into four parts: "add", "attribute word", "and remainder", and "if the word starts with a dot". Lines 2-5 show the summaries generated by four techniques. From Figure 14(a), we can know that "evaluatorWords" is a list used to store the attribute words. From a semantic point of view, only SiT and EACS correctly cover the first and second parts of the reference summary. The third part of the reference summary (i.e., "and remainder") is a summary of the "word[:dotIndex]" shown in the penultimate line of Figure 14(a). This summary is highly abstract, reflecting the summarization ability and word habits of professional human developers. Obviously, current code summarization models (including our EACS) do not yet have such capabilities. The fourth part of the reference summary, i.e., "if the word starts with a dot", is a summary of lines 5-7 of c_6 (highlighted in Orange font). This summary is also very abstract. "the word starts with a dot" and "word[0]!='.'" are opposite conditions. The developer uses "if (false) $\cdots$ return $\cdots$" to achieve the same functionality as "if (true) $\cdots$ else $\cdots$", thereby achieving the goal of simplifying code. All four techniques fail to generate such abstract summaries. This case shows

that current code summarization models (including our EACS) are not yet able to make a high-level abstract summarization of the code, and there is still much room for improvement. In addition, as shown in Figure 14(b), our extractor also fails to extract those important statements related to highly abstract summaries. We will further explore how to capture the alignment between highly abstract summaries and code statements in future work.

```
def django_boolean_icon(field_val, alt_text=None, title=None):
    BOOLEAN_MAPPING = {True: u'yes', False: u'no', None: u'unknown'}
    alt_text = (alt_text or BOOLEAN_MAPPING[field_val])
    if (title is not None):
        title = (u'title="%s" ' %title)
    else:
        title = u''
    icon_url = static((u'feincms/img/icon-%s.gif' % BOOLEAN_MAPPING[field_val]))
    return mark_safe((u'<img src="%s"alt="%s" %s/>' % (icon_url, alt_text, title)))
```

(a) A code snippet c_7 from the test set of the PCSD database

```
def django_boolean_icon(field_val, alt_text=None, title=None):
```

(b) Important statements selected by our extractor

Reference Summary: return html code for a nice representation of true/false.
Ex-based: wrapper for a boolean.
SiT: generates an icon for a boolean field.
CodeT5: returns the boolean icon for a form field.
EACS: generates an icon for a boolean field.

(c) Summaries generated by different techniques

Fig. 15. Weak Python Case

Weak Case. Figure 15 shows a weak Python case. The reference summary of c_7 shown in the first line of Figure 15(c) can be divided into three parts: "return", "html code", and "for a nice representation of true/false". Combining the code snippet of c_7 shown in Figure 15(a), we can know that the "a nice representation of true/false" indeed refers to a boolean icon. Therefore, from a semantic perspective, the summaries generated by SiT, CodeT5, and EACS correctly cover the third part of the reference summary. Of course, literally, the four generated summaries are far from the reference summary. Most of the words used in the generated summaries come from the code itself. Similar to what is observed in the previous case, these models are not yet able to make a high-level abstract summarization of the code like professional human developers. It is worth noting that all four techniques failed to cover the second part. Figure 15(a), we can observe that the last statement is closely related to the keyword "html". Although "<img />" is one of the commonly used tags in HTML, the four code summarization models do not seem to learn this association. Our extractor also fails to extract this important statement. We check the vocabulary and find that the tokenizer used in data preprocessing would break HTML tags. For example, the "<img src" in the last statement of Figure 15(a) is split into "< ", "imgs", and "rc" by the CodeT5 tokenizer. Obviously, this not only breaks the tag mode, but also wrongly splits the "img src" into "imgs" and "rc". This may be the main reason why the four code summarization models and our extractor do not work well on this example. How to strengthen the model to learn special domain knowledge (e.g., HTML tags) may be a future research point, which is worthy of further exploration.

6 THREATS TO VALIDITY

There are three main threats to the validity of our approach as follows:

Dataset. The first threat to validity is the evaluation dataset. The JCSD and PCSD datasets may contain duplications (as mentioned in [122]) in the training and test sets to make the model overfit easily. To compare fairly with previous baselines, we follow [41, 115] to directly use the same data as them and do not perform any additional processing on the two datasets. To alleviate this threat, we also conduct experiments on a large dataset CodeXGLUE [62], which is very carefully de-duplicated and includes six programming languages, and thus the performance numbers we report here can be expected to be fairly good. We will use the preprocessing script provided by [122] to deduplicate the dataset and conduct experiments in future work.

Evaluation Method. The second threat to validity is the evaluation method. In this paper, we employ three automatic evaluation metrics (i.e., BLEU, METEOR, and ROUGE-L) to evaluate the quality of the generated summaries. Although these metrics are widely used in existing code summarization works, they still have limitations. The three metrics can calculate the textual difference between generated summaries and references. However, in some cases, the model may produce valid summaries that do not align with the ground truth, i.e., the metrics cannot truly reflect the semantic similarity [26, 29]. To alleviate this threat, we also perform a human evaluation from four aspects (i.e., similarity, naturalness, informativeness, and relevance). Furthermore, we use the average results of multiple evaluators to alleviate the subjectivity of human evaluation.

Baselines. The third threat to validity is about baselines. Since the implementation code for early TR-based methods is no longer available, in this paper, to alleviate this threat, we try our best to re-implement a TR-based method with the guide of the early works [32, 33]. Since this paper adopts the very recent datasets (i.e., JCSD and PCSD), the early TR-based methods did not conduct experimental evaluations on these datasets before. To proceed with our study, we perform LSA on our evaluation dataset. We process these data through our understanding of papers [32, 33] and choose recommended settings to minimize experimental bias. We make the implementation code public for subsequent researchers to check and use.

7 RELATED WORK

Code summarization has always been one of the hottest research topics in software engineering. As mentioned earlier, we can categorize existing work related to code summarization into extractive methods, abstractive methods, and others.

Extractive methods. Most early (prior to 2016) code summarization techniques [32, 33, 69, 95] are extractive methods. Such methods work by extracting a subset of the statements and keywords from the code, and then including information from those statements and keywords in summary. For example, Sonia Haiduc et al. [32] first propose an extractive method to generate extractive summaries for code snippets automatically. Extractive summaries are obtained from the contents of a document by selecting the most important information in that document. *Extractive methods* use text retrieval (TR) techniques (e.g., Vector Space Model [84], Latent Semantic Indexing [16], and Hierarchical PAM [67]) to determine the most important n terms for each code snippet. Considering that the quality of the summaries generated by extractive methods depends heavily on the process of extracting the subset, Paige Rodeghero et al. [81] present an eye-tracking study of programmers and propose a tool for selecting keywords based on the findings of the eye-tracking study. The extractive methods rely on high-quality identifier names and method signatures from the source code. These techniques may fail to generate accurate comments if the source code contains poorly named identifiers or method names [113].

Abstractive methods. Nowadays, DL-based (abstractive) code summarization techniques have been proposed one after another. By combining Seq2Seq models trained on large-scale code-comment datasets, abstractive methods can generate words that do not appear in the given code snippet to overcome the limitations of extractive methods. For example, Srinivasan Iyer et al. [43]

present the first completely abstractive method for generating short, high-level summaries of code snippets. Their experimental results demonstrate that abstractive methods significantly outperform extractive methods in the naturalness of generated summaries. In other words, the summaries generated by abstractive methods are human-written-like. Code representation plays a key role in abstractive methods. To produce semantic-preserving code embedding representations, multiple aspects of the code snippet have been explored, including tokens [2, 22, 40, 41, 43, 51, 52, 56, 68, 75, 104, 106, 110, 122], abstract syntactic trees (ASTs) [4, 39, 40, 51, 52, 56, 91, 104, 106, 110, 118, 122], control flows [106], code property graph [58]. In addition, existing abstractive methods have tried various neural network architectures, such as LSTM [39, 43, 106], Bidirectional-LSTM [55, 110, 122], GRU [40, 52], Transformer [2, 56] and GNN [51, 58]. Although deep learning-based abstractive methods show great potential for generating human-written-like summaries, we find that the generated summaries often miss important factual details.

Others. Paul W. McBurney [65] takes the context of code snippets into account when generating summaries. The context includes the dependencies of the method and any other methods which rely on the output of the method [49]. The works [8, 86, 107, 121] also use the information beyond the code snippet itself, e.g., project or class context [8, 107, 121], application programming interface documentations/knowledge (API Docs) [86]. These techniques cannot be tested on existing commonly used datasets (e.g., JCSD, PCSD, and CodeSearchNet Corpus) because methods (code snippets) in these datasets are context-missing. There are techniques using summaries of similar code snippets as the summary of the current snippet or as a starting point to generate a new summary. For example, Edmund Wong et al. [113] apply code clone detection techniques to discover similar code snippets and use the comments from some code snippets to describe the other similar code snippets. The works [55, 110, 122] retrieve similar code snippets and then use the information contained in the similar code snippets or their summaries to generate a summary of the current code snippet. They rely on whether similar code snippets can be retrieved and how similar they are. All the works mentioned above emphasize using external knowledge sources (e.g., contexts, API Docs, and similar code snippets) to improve the quality of the generated comments. We will explore the effect of combining our EACS with the above external knowledge in future work.

Our method. EACS is a general code summarization framework that adopts extractive and abstract schemes at the same time. Different from existing extractive methods that use TR techniques to extract important statements/words, the extractive module of EACS applies a deep learning-based classifier to predict the importance of each statement. Unlike existing abstractive methods, our abstracter receives and processes the entire code snippet and important statements as input in parallel.

8 CONCLUSION

In this paper, we propose an extractive-and-abstractive framework named EACS for source code summarization. EACS consists of two modules, extractor and abstracter. The extractor has the ability to extract important statements from the code snippet. The important statements contain important factual details that should be included in the final generated summary. The abstracter takes in the important statements extracted by the extractor and the entire code snippet and is able to generate human-written-like summary in natural language. We conduct comprehensive experiments on three databases to evaluate the performance of EACS. And the experimental results demonstrate that our EACS is an effective code summarization technique and significantly outperforms the state-of-the-art. Extensive human evaluations demonstrate that the summaries generated by EACS have higher naturalness and informativeness and are more relevant to given code snippets.

As mentioned earlier, the extractive-and-abstractive framework we proposed is highly scalable that does not depend on a specific deep learning network/model. Based on the experimental results

of **RQ2** shown in Section 5.2.2, we can speculate that replacing the pre-trained model in the framework with more advanced large language models (LLMs), such as ChatGPT [72] (if available), has the opportunity to further improve code summarization performance. We plan to combine EACS with more advanced LLMs, to exert more powerful performance in the future.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by National Natural Science Foundation of China (61932012, 62141215, 62372228) and the Program B for Outstanding PhD Candidate of Nanjing University (202201B054).

REFERENCES

[1] Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. 2019. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the 11th Symposium on Eye Tracking Research & Applications*. ACM, Denver , CO, USA, 13:1–13:9.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 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, Online, 4998–5007.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (2018), 81:1–81:37.

[4] Uri Alon, Omer Levy, and Eran Yahav. 2018. Code2seq: Generating Sequences from Structured Representations of Code. *CoRR* abs/1808.01400 (2018).

[5] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. 2019. An approach for bug localization in models using two levels: model and metamodel. *Software and Systems Modeling* 18, 6 (2019), 3551–3576.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the 3rd International Conference on Learning Representations*. OpenReview.net, San Diego, CA, USA, 1–15.

[7] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Association for Computational Linguistics, Ann Arbor, Michigan, USA, 65–72.

[8] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 253–264.

[9] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *Proceedings of the 8th International Joint Conference on Natural Language Processing*. Asian Federation of Natural Language Processing, Taipei, Taiwan, 314–319.

[10] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021), 25:1–25:29.

[11] Yen-Chun Chen and Mohit Bansal. 2018. Fast Abstractive Summarization with Reinforce-Selected Sentence Rewriting. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Melbourne, Australia, 675–686.

[12] Jianpeng Cheng and Mirella Lapata. 2016. Neural Summarization by Extracting Sentences and Words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Berlin, Germany, 1–11.

[13] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. In *Proceedings of the 8th Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, Doha, Qatar, 103–111.

[14] Sumit Chopra, Michael Auli, and Alexander M. Rush. 2016. Abstractive Sentence Summarization with Attentive Recurrent Neural Networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. The Association for Computational Linguistics, San Diego California, USA, 93–98.

[15] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of*

Communication: documenting & Designing for Pervasive Information. ACM, Coventry, UK, 68–75.
[16] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. 1990. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.
[17] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.
[18] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. 2013. Evaluating Source Code Summarization Techniques: Replication and Expansion. In *Proceedings of the 21st International Conference on Program Comprehension*. IEEE Computer Society, San Francisco, CA, USA, 13–22.
[19] Wafaa S. El-Kassas, Cherif R. Salama, Ahmed A. Rafea, and Hoda K. Mohamed. 2021. Automatic text summarization: A comprehensive survey. *Expert Systems with Applications* 165, 1 (2021), 113679–113704.
[20] Günes Erkan and Dragomir R. Radev. 2004. LexRank: Graph-based Lexical Centrality as Salience in Text Summarization. *Journal of Artificial Intelligence Research* 22, 1 (2004), 457–479.
[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*. Association for Computational Linguistics, Online Event, 1536–1547.
[22] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured Neural Summarization. *CoRR* abs/1811.01824 (2018).
[23] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. 2012. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology: General* 141, 1 (2012), 2–18.
[24] Mahak Gambhir and Vishal Gupta. 2017. Recent Automatic Text Summarization Techniques: A Survey. *Artificial Intelligence Review* 47, 1 (2017), 1–66.
[25] Kavita Ganesan, ChengXiang Zhai, and Jiawei Han. 2010. Opinosis: A Graph Based Approach to Abstractive Summarization of Highly Redundant Opinions. In *Proceedings of the 23rd International Conference on Computational Linguistics*. Tsinghua University Press, Beijing, China, 340–348.
[26] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach based on Transformer for Source Code Summarization. *CoRR* abs/2203.09707 (2022).
[27] Pierre-Etienne Genest and Guy Lapalme. 2012. Fully Abstractive Approach to Guided Summarization. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Jeju Island, Korea, 354–358.
[28] Inc. GitHub. 2008. GitHub. site: https://github.com. Accessed: 2022.
[29] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. *CoRR* abs/2202.06521 (2022).
[30] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 746–757.
[31] Som Gupta and Sanjai Kumar Gupta. 2019. Abstractive summarization: An overview of the state of the art. *Expert Systems with Applications* 121, 1 (2019), 49–65.
[32] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting Program Comprehension with Source Code Summarization. In *Proceedings of the 32nd International Conference on Software Engineering*. ACM, Cape Town, South Africa, 223–226.
[33] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE Computer Society, Beverly, MA, USA, 35–44.
[34] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. 2021. Action Word Prediction for Neural Source Code Summarization. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 330–341.
[35] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th International Conference on Program Comprehension*. ACM, Virtual Event, 36–47.
[36] Carl S. Hartzman and Charles F. Austin. 1993. Maintenance Productivity: Observations based on An Experience in A Large System Environment. In *Proceedings of the 3rd Conference of the Centre for Advanced Studies on Collaborative Research*. IBM, Toronto, Ontario, Canada, 138–170.
[37] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[38] Wan Ting Hsu, Chieh-Kai Lin, Ming-Ying Lee, Kerui Min, Jing Tang, and Min Sun. 2018. A Unified Model for Extractive and Abstractive Summarization using Inconsistency Loss. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Melbourne, Australia, 132–141.

[39] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, Gothenburg, Sweden, 200–210.
[40] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.
[41] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. ijcai.org, Stockholm, Sweden, 2269–2275.
[42] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019).
[43] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Berlin, Germany, 2073–2083.
[44] Peddamail Jayavardhan Reddy, Ziyu Yao, Zhen Wang, and Huan Sun. 2018. A Comprehensive Study of StaQC for Deep Code Summarization. In *Proceedings of the 24th International Conference on Knowledge Discovery and Data Mining*. ACM, London, UK, 1–8.
[45] Mira Kajko-Mattsson. 2005. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55.
[46] Atif Khan, Naomie Salim, and Yogan Jaya Kumar. 2015. A framework for multi-document abstractive summarization based on semantic role labelling. *Applied Soft Computing* 30, 1 (2015), 737–747.
[47] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3th International Conference on Learning Representations – Poster*. OpenReview.net, San Diego, CA, USA, 1–15.
[48] Kirill Kireyev. 2008. Using Latent Semantic Analysis for Extractive Summarization. In *Proceedings of the 1st Text Analysis Conference*. NIST, Gaithersburg, Maryland, USA, 1–4.
[49] Jens Krinke. 2006. Effects of Context on Program Slicing. *Journal of Systems and Software* 79, 9 (2006), 1249–1260.
[50] Litton J. Kurisinkel, Yue Zhang, and Vasudeva Varma. 2017. Abstractive Multi-document Summarization by Partial Tree Extraction, Recombination and Linearization. In *Proceedings of the 8th International Joint Conference on Natural Language Processing*. Asian Federation of Natural Language Processing, Taipei, Taiwan.
[51] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, Seoul, Republic of Korea, 184–195.
[52] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE / ACM, Montreal, QC, Canada, 795–806.
[53] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 23th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Minneapolis, MN, USA, 3931–3937.
[54] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 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, Online, 7871–7880.
[55] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *Proceedings of the 36th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 155–166.
[56] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 184–195.
[57] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics – workshop on Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81.
[58] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Automatic Code Summarization via Multi-dimensional Semantic Fusing in GNN. *CoRR* abs/2006.05405 (2020).
[59] Yang Liu. 2019. Fine-tune BERT for Extractive Summarization. *CoRR* abs/1903.10318 (2019).
[60] Yang Liu and Mirella Lapata. 2019. Text Summarization with Pretrained Encoders. In *Proceedings of the 23rd Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Hong Kong, China, 3728–3738.
[61] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692

(2019), 1–13.

[62] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. virtual, 1–14.

[63] Robert C Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

[64] Mani Maybury. 1999. *Advances in Automatic Text Summarization*. MIT press.

[65] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[66] Microsoft. 2021. CodeBert. site: https://github.com/microsoft/CodeBERT. Accessed August, 2022.

[67] David M. Mimno, Wei Li, and Andrew McCallum. 2007. Mixtures of Hierarchical Topics with Pachinko Allocation. In *Proceedings of the 24th International Conference Machine Learning*. ACM, Corvallis, Oregon, USA, 633–640.

[68] Jessica Moore, Ben Gelman, and David Slater. 2019. A Convolutional Neural Network for Language-Agnostic Source Code Summarization. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, Heraklion, Crete, Greece, 15–26.

[69] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the 21st International Conference on Program Comprehension*. IEEE Computer Society, San Francisco, CA, USA, 23–32.

[70] Ramesh Nallapati, Feifei Zhai, and Bowen Zhou. 2017. SummaRuNNer: A Recurrent Neural Network based Sequence Model for Extractive Summarization of Documents. In *Proceedings of the 31st Conference on Artificial Intelligence*. AAAI Press, San Francisco, California, USA, 3075–3081.

[71] Ramesh Nallapati, Bowen Zhou, and Mingbo Ma. 2016. Classify or Select: Neural Architectures for Extractive Document Summarization. *CoRR* abs/1611.04244 (2016).

[72] OpenAI. 2022. ChatGPT. site: https://chat-gpt.org/. Accessed March, 2023.

[73] Tatsuro Oya, Yashar Mehdad, Giuseppe Carenini, and Raymond T. Ng. 2014. A Template-based Abstractive Meeting Summarization: Leveraging Summary and Source Text Relationships. In *Proceedings of the 8th International Natural Language Generation Conference*. The Association for Computer Linguistics, Philadelphia, PA, USA, 45–53.

[74] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL, Philadelphia, PA, USA, 311–318.

[75] Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. *CoRR* abs/2105.08645 (2021).

[76] Jonathan Pilault, Raymond Li, Sandeep Subramanian, and Chris Pal. 2020. On Extractive and Abstractive Neural Document Summarization with Transformer Language Models. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online, 9308–9319.

[77] Maja Popovic. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the 10th Workshop on Statistical Machine Translation*. The Association for Computer Linguistics, Lisbon, Portugal, 392–395.

[78] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with A Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[79] Sarah Rastkar. 2010. Summarizing Software Concerns. In *Proceedings of the 32nd International Conference on Software Engineering*. ACM, Cape Town, South Africa, 527–528.

[80] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, Hong Kong, China, 3980–3990.

[81] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney K. D'Mello. 2014. Improving Automated Source Code Summarization via An Eye-tracking Study of Programmers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, Hyderabad, India, 390–401.

[82] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks. In *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 1105–1116.

[83] Salesforce. 2021. CodeT5. site: https://github.com/salesforce/CodeT5. Accessed August, 2022.

[84] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[85] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Vancouver, Canada, 1073–1083.
[86] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 411–421.
[87] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2021. Neural Code Summarization: How Far Are We?. In *CoRR*, Vol. abs/2107.07112.
[88] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the Evaluation of Neural Code Summarization. In *Proceedings of the 44th International Conference on Software Engineering*. IEEE, Pittsburgh, USA, 1597––1608.
[89] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 4053–4062.
[90] Tian Shi, Yaser Keneshloo, Naren Ramakrishnan, and Chandan K. Reddy. 2021. Neural Abstractive Text Summarization with Sequence-to-Sequence Models. *Transactions on Data Science* 2, 1 (2021), 1:1–1:37.
[91] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. In *Proceedings of the 18th International Joint Conference on Neural Networks*. IEEE, Hungary, 1–8.
[92] GraphPad Software. 1995. Friedman's test. site: https://www.graphpad.com/guides/prism/latest/statistics/how_the_friedman_test_works.htm. Accessed March, 2023.
[93] GraphPad Software. 1995. GraphPad Prism. site: https://www.graphpad.com. Accessed March, 2023.
[94] GraphPad Software. 1995. GraphPad Prism. site: https://www.graphpad.com. Accessed March, 2023.
[95] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the 25th International Conference on Automated Software Engineering*. ACM, Antwerp, Belgium, 43–52.
[96] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, Seoul, Republic of Korea, 2–13.
[97] Josef Steinberger and Karel Jezek. 2009. Update Summarization based on Latent Semantic Analysis. In *Proceedings of the 12th International Conference on Text, Speech and Dialogue*. Springer, Pilsen, Czech Republic, 77–84.
[98] Josef Steinberger, Karel Jezek, et al. 2004. Using latent semantic analysis in text summarization and summary evaluation. In *Proceedings of the 7th International Conference ISIM*. 1–8.
[99] Weisong Sun and Yuchen Chen. 2022. EACS. site: https://github.com/wssun/EACS. Accessed August, 2022.
[100] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 388–400.
[101] Ted Tenny. 1988. Program Readability: Procedures Versus Comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.
[102] Maciej Tomczak and Ewa Tomczak. 2014. The need to report effect size estimates revisited. An overview of some recommended measures of effect size. *Trends in sport sciences* 21, 1 (2014), 19–25.
[103] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., Long Beach, CA, USA, 5998–6008.
[104] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM/IEEE, Montpellier, France, 397–407.
[105] Shuai Wang, Xiang Zhao, Bo Li, Bin Ge, and Daquan Tang. 2017. Integrating Extractive and Abstractive Models for Long Text Summarization. In *Proceedings of the 6th International Congress on Big Data, BigData Congress*. IEEE Computer Society, Honolulu, HI, USA, 305–312.
[106] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Transactions on Software Engineering (Early Access)* (2020), 1–19.
[107] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021. CoCoSum: Contextual Code Summarization with Multi-Relational Graph Neural Network. *CoRR* abs/2107.01933 (2021), 1–24.

[108] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 8696–8708.

[109] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Proceedings of the 33rd Annual Conference on Neural Informatiom Processing Systems (NeurIPS 2019)*. Vancouver, BC, Canada, 6559–6569.

[110] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 349–360.

[111] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1963. *Critical Values and Probability Levels for the Wilcoxon Rank Sum Test and the Wilcoxon Signed Rank Test.* American Cyanamid Company.

[112] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer.

[113] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining Existing Source Code for Automatic Comment Generation. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Computer Society, Montreal, QC, Canada, 380–389.

[114] Scott N. Woodfield, Hubert E. Dunsmore, and Vincent Yun Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE Computer Society, San Diego, California, USA, 215–223.

[115] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Proceedings of the Findings of the 59th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online Event, 1078–1090.

[116] Yuxiang Wu and Baotian Hu. 2018. Learning to Extract Coherent Summary via Deep Reinforcement Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. AAAI Press, New Orleans, Louisiana, USA, 5602–5609.

[117] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976.

[118] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 1–12.

[119] Wenyuan Zeng, Wenjie Luo, Sanja Fidler, and Raquel Urtasun. 2016. Efficient Summarization with Read-Again and Copy Mechanism. *CoRR* abs/1611.03382 (2016), 1–11.

[120] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, Seoul, South Korea, 1359–1371.

[121] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Raymond J. Mooney, Junyi Jessy Li, and Milos Gligoric. 2021. Learning to Generate Code Comments from Class Hierarchies. *CoRR* abs/2103.13426 (2021).

[122] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, Seoul, South Korea, 1385–1397.

[123] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2020. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, Virtual Event, 11328–11339.

[124] Qingyu Zhou, Nan Yang, Furu Wei, Shaohan Huang, Ming Zhou, and Tiejun Zhao. 2018. Neural Document Summarization by Jointly Learning to Score and Select Sentences. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Melbourne, Australia, 654–663.

[125] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2021. Adversarial Robustness of Deep Code Comment Generation. *ACM Transactions on Software Engineering and Methodology* 1, 1 (2021), 1–30.

[126] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An Overlapping-Aware Code Retriever. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 883–894.

[127] Yuxiang Zhu and Minxue Pan. 2019. Automatic Code Summarization: A Systematic Literature Review. *CoRR* abs/1909.04352 (2019).