Integrating Extractive and Abstractive Models for Code Comment Generation

Weisong Sun, Yuling Hu, Yingfei Xu, Yuchen Chen, and Chunrong Fang*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai, China
{weisongsun, yulinghu, yingfeixu}@smail.nju.edu.cn, yuc.chen@outlook.com, fangchunrong@nju.edu.cn
*corresponding author

Abstract—Code comments play an essential role in aiding developers understand and maintain source code. Current code comment generation techniques can be classified into categories: extractive methods and abstractive methods. Extractive methods use text retrieval techniques to extract important code tokens to constitute comments. Such comments contain important factual details articulated explicitly in code tokens, but are poor in naturalness. Abstractive methods usually regard code comment generation as a neural machine translation task. By leveraging powerful deep learning-based language models, abstractive methods can generate comments that resemble human writing. However, compared with natural language, programming language code is more complex. Comments generated by abstractive methods often leave out important factual details. In this paper, we propose a novel method for code comment generation by integrating extractive and abstractive models. Our extractive model is built on the Latent Semantic Analysis (LSA) model, effectively extracting important factual details in code snippets. Meanwhile, our abstractive model is built on a deep learning-based encoder-decoder model, enabling it to generate concise and human-written-like comments. We evaluate the effectiveness of our method, called ICS, by conducting extensive experiments on the CodeSearchNet dataset involving six programming languages. The results demonstrate that ICS outperforms state-of-the-art techniques in three widely used metrics: BLEU, METEOR, and ROUGE-L. Moreover, the outcomes of the human evaluation indicate that the comments generated by ICS exhibit superior naturalness and informativeness, and closely align with the provided code snippets.

Keywords–Code Comment Generation; Code Summarization; Latent Semantic Analysis; Text Summarization; Deep Learning

1. Introduction

Code comments play a crucial role in enhancing code understanding and facilitating software maintenance [1]. Writing high-quality code comments has been regarded as a fundamental programming practice, but it demands significant time and effort [2]. Consequently, valuable comments often become absent, unmatched, and outdated as the code evolves [3]. Code comment generation, also known as code summarization, represents a prominent research area [4], [5]. It focuses on developing advanced techniques for automatically generating comments for code snippets. Given a code snippet (at the method/function level), code comment generation techniques can automatically produce a concise natural language comment for it. Figure 1 illustrates an example. The code snippet presented in Figure 1(a) is provided by the developer. The comment "get a compatible constructor for the given value type." in Figure 1(b) fulfills the developer's requirement.

```
public static Constructor getCompatibleConstructor(final Class type, final Class valueType){
    try{
        return type.getConstructor(new Class[]{valueType});
    } catch (Exception ignore) {
        Class[] types = type.getClasses();
        for (int i = 0; i < types.length; i++) {
            try{
                return type.getConstructor(new Class[]{types[i]});
            } catch (Exception ignore2) {}
        }
    }
    return null;
}
```

(a) A Code Snippet s_1

get a compatible constructor for the given value type.

(b) A Comment c_1

Figure 1: Example of Code Snippet and Comment

Code comment generation represents a unique form of text summarization task, where the text is expressed in a programming language, diverging from conventional natural languages. Automatic text summarization was introduced to code comment generation over a decade ago [4]. Hence, similar to text summarization, current code comment generation techniques can be classified into *extractive methods* and *abstractive methods*. Initially, code comment generation techniques primarily relied on *extractive methods*, utilizing an index-retrieval framework to produce comments [4], [6]. They first index terms/tokens in code snippets and then retrieve top-n key terms as comments. The terms in comments can be extracted from the present code snippets [7], context code snippets [8] or similar code snippets [6]. Consequently, *extractive methods* can produce comments that uphold the conceptual integrity and factual information of the input code snippet. However, the key terms extracted from code snippets may comprise frivolous words or abbreviations, especially when the identifiers are poorly named. Furthermore, the comments generated by *extractive methods* are less natural and lack the appearance of being written by humans [9], detailed in Section 3.

In recent years, with the boom of deep learning (DL) in abstractive text summarization, DL-based code comment generation techniques have been proposed successively [9], [10], [11], [12]. Compared with *extractive methods*, DL-based code comment generation techniques possess superior abilities in

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00027

abstract expression and can produce comments that resemble human writing, akin to abstractive text summarization [13]. Therefore, we classify the DL-based code comment generation techniques as *abstractive methods*. *Abstractive methods* typically employ the neural network model built on the encoder-decoder architecture and train the model on a large code-comment corpus. The encoder first converts code snippets into numerical context vectors (also known as embeddings), which are then decoded by the decoder to produce concise natural language text. As documented by Allamanis et al. [14], traditional methods have largely been replaced by DL methods that rely on substantial input from big data. However, although *abstractive methods* can generate novel words and phrases that are not present in the code snippet, they may omit crucial factual details in the code snippet, detailed in Section 3.

In this paper, we propose a novel method for code comment generation by integrating extractive and abstractive models. Our method retains the advantages of extractive and abstractive methods while mitigating their respective shortcomings. Specifically, large code corpora are used to train an extractive model (like an extractive method) through the Latent Semantic Analysis (LSA) and an abstractive model (like an abstractive method). The extractive model serves as an extractor, responsible for extracting essential statements from the code snippets. The abstractive model functions as an abstracter, taking both the entire code snippet and important statements extracted by the extractor as input and generating a concise natural language comment. The abstracter first utilizes two distinct encoders to transform the entire code snippet and important statements into two embeddings, and then integrates them to create an integration embedding. The integration embedding will subsequently be fed to a decoder to generate comments. In comparison to existing abstractive methods, our method integrates an extractive method, effectively balancing attention between global contextual information and crucial contextual information, thus minimizing the risk of omitting essential factual details and enhancing the overall performance.

In summary, we make the following contributions.

- We propose to integrate extractive and abstractive models to enhance code comment generation. Our method preserves the advantages of both extractive and abstractive methods while mitigating their respective shortcomings.
- We conduct extensive quantitative experiments on a widely used dataset called CodeSearchNet (CSN), and the results demonstrate that the proposed method named ICS significantly outperforms state-of-the-art baselines in three widely used automatic metrics.
- We conduct a qualitative human evaluation to assess the comments generated by ICS and baselines based on four aspects: similarity, naturalness, informativeness, and relevance. The statistical results of human scores indicate that the comments generated by ICS are more informative and relevant to code snippets.
- We release the source code of ICS at the project homepage [15] for experiment replication, future research, and practical use.

2. Background

2.1 Automatic Text Summarization

We draw inspiration from the advanced techniques of automatic text summarization. Therefore, we begin by introducing the background of automatic text summarization.

Automatic text summarization refers to the process of automatically condensing a piece of text into a concise summary while preserving its key points [16]. Based on their technical characteristics, text summarization can be categorized into extractive text summarization (*extractive methods*) and abstractive text summarization (*abstractive methods*) [13]. *Extractive methods* directly extract words, phrases, and sentences from the source text to assemble summaries. The generated summaries typically contain salient information from the source text [17]. In contrast, *abstractive methods* can generate summaries that include novel words and phrases that are not present in the source text – similar to the abstracts written by humans [17]. As a result, they are more likely to generate summaries that are fluid, easy to understand, and of high quality [13].

As early as ten years ago, automatic text summarization techniques were introduced to automatic code comment generation [4], [6]. For instance, in 2010, Sonia Haiduc et al. [7] proposed an extractive code comment generation technique that automatically generates extractive comments for source code entities. Extractive comments are generated by selecting the most important terms in code snippets. Recently, DL-based code comment generation techniques have been successively proposed [18], [19], [20], [21]. Similar to abstractive text summarization, these techniques also extensively adopt the encoder-decoder models borrowed from neural machine translation [18] to generate natural language comments. Therefore, DL-based code comment generation techniques can be viewed as abstractive methods.

In this paper, we integrate extractive and abstractive methods together. The extractor is responsible for extracting important factual details while the abstracter is responsible for generating human-written-like natural language comments, detailed in Section 4. Existing works that combine extractive and abstractive summarization methods are primarily proposed in NLP [22], [23]. Unlike these works, where extractors select important sentences from the source text, our extractor extracts important statements from the source code. Additionally, our abstracter takes both the source code and important statements as input to generate comments, rather than considering only important sentences as in [23] or focusing solely on the source text with sentence-level attention as in [22].

2.2 Latent Semantic Analysis

The extractor of ICS is based on Latent Semantic Analysis (LSA). Thus, in this subsection, we will provide an introduction to LSA.

LSA is an unsupervised method that discovers topic-based semantic relationships between documents and words through matrix decomposition based on a large text corpus [24]. LSA represents documents and words as vectors in a high-dimensional space, where each dimension corresponds to a

word. The technique then employs singular value decomposition (SVD) to reduce the dimensionality of the space, representing the semantic content of documents and words through topic vectors. This ability allows it to measure similarities more accurately in the topic vector space and handle synonyms effectively [25].

LSA was originally developed for information retrieval tasks, such as indexing and retrieval of large document collections [26]. Over two decades ago, LSA was first introduced to the task of text summarization [27], [28], [29]. This method can summarize the event/topic of a set of documents related to a specific event/topic.

In this paper, we apply LSA in our extractor since it can accurately capture the semantic similarity between texts. Furthermore, compared with DL-based methods, LSA has a significantly lower training cost in terms of both time and computational resources. Specifically, we treat each code snippet as a document, each token in the code snippet as a word, and pick the top-n relevant tokens in each code snippet as key tokens. Next, we identify the statements that contain these key tokens as important statements. The details will be discussed in Section 4-B1.

3. Motivating Example

```
Public static Constructor getCompatibleConstructor(final Class type, final Class valueType){}
return type.getConstructor(new Class[]{valueType});
Class[] types = type.getClasses();
for (int i = 0; i < types.length; i++) {
return type.getConstructor(new Class[]{types[i]});
```

(a) Important Statements Selected by Extractor

Reference Comment: get a compatible constructor for the given value type.
Extractive Comment: type compatible constructor classes value
Abstractive Comment: get compatible constructor.
Integrated Comment: returns the compatible constructor for the given type.

(b) Comments Generated by Different Techniques

Figure 2: Motivating Example

In this section, we use the code snippet s_1 in Figure 1(a) as an example and apply different techniques to generate comments for comparison. It is a real-world example from the CodeSearchNet dataset (detailed in Section 5-A1). Figure 1(b) displays the comment written by the developer for s_1, which we consider as the reference comment (the ground truth) indicated in the first line of Figure 2(b). According to the grammar rules in natural language, we can divide the reference comment into two parts: "get a compatible constructor" (Blue font), and "for the given value type" (Green font).

We use two techniques, an extractive method [29] and an abstractive method [30] to generate comments for s_1. The extractive method [29] adopts the Latent Semantic Analysis (LSA) techniques [31] to assess the informativity of each token in the code snippet and then selects the top 5 key tokens. The second line of Figure 2(b) shows an extractive comment generated by the extractive method. It contains significant factual details that should be included in the comment, e.g., the important words "type", "compatible", and "constructor". The abstractive method [30] first trains a model called CodeT5 to obtain code representations, and then fine-tunes it on the code comment generation task. The third line of Figure 2(b) shows the comment generated by the abstractive method. Observing the generated abstractive comment, we notice the following: 1) intuitively, it possesses good naturalness, resembling human-written text; 2) the abstractive comment covers the first part (Blue font) of the reference comment but fails to cover the second part (Green font), i.e., it misses some factual details.

Our solution. The last comment (i.e., Integrated Comment) in Figure 2(b) is generated by our ICS. It is observed that 1) the integrated comment contains the key tokens selected by the LSA-based extractive method; 2) compared with the abstractive comment generated by [30], the integrated comment can cover both parts of the reference. Based on the above observations, it is evident that our method can generate human-written-like comments while preserving important factual details. The good performance of our method is attributed to its ability to give more attention to essential statements. Intuitively, the first part (Green font) serves as a translation or comment of the factual details (keywords "compatible" and "constructor") contained in the important method declaration statement *public static Constructor getCompatibleConstructor(final Class type, final Class valueType)*{}. The second part (Green font) represents a translation or summary of the factual details (e.g., the key token "type") contained in the remaining important statements shown in Figure 2(a).

4. Methodology

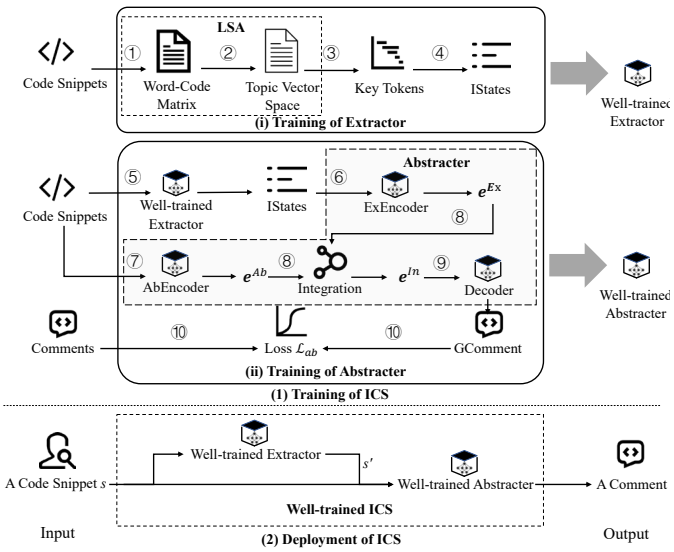


Figure 3: The overview of ICS

4.1 Overview

Figure 3 provides the overview of our approach ICS. The top part illustrates the training process of ICS, while the bottom part shows the deployment (usage) of ICS. ICS divides the training process into two phases: (i) training of extractor and (ii) training of abstracter. The goal of the extractor's training is to produce a well-trained model capable of extracting

important statements from a given code snippet. The training of the abstracter aims to produce a well-trained abstracter capable of generating a succinct natural language comment for a given code snippet. The training of the extractor leverages one type of input data: code snippets, and the training of the abstracter also leverages their corresponding comments. To construct the extractor, ICS first treats code snippets as texts and builds up an LSA model, detailed in Section 4-B1-**Step** ① and **Step** ②. Then, ICS determines the most relevant top-5 key tokens for each code snippet, detailed in Section 4-B1-**Step** ③ and extracts important statements based on the key tokens, as explained in Section 4-B1-**Step** ④. To train the abstracter, ICS uses its well-trained extractor to extract the important statements from a given code snippet, detailed in Section 4-B2-**Step** ⑤. The extracted important statements will be further transformed into the embedding representation $\boldsymbol{e}^{Ex}$ by an encoder named ExEncoder, detailed in Section 4-B2-**Step** ⑥. Then, ICS uses another encoder named AbEncoder to transform the entire code snippet into the embedding representation $\boldsymbol{e}^{Ab}$, detailed in Section 4-B2-**Step** ⑦. Furthermore, ICS produces the integrated embedding representation $\boldsymbol{e}^{In}$ by integrating $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$, detailed in Section 4-B2-**Step** ⑧. The embedding $\boldsymbol{e}^{In}$ will be passed to a decoder to generate predicted comments (PComments), detailed in Section 4-B2-**Step** ⑨. During this procedure, the model parameters of the abstracter (including ExEncoder, AbEncoder, and Decoder) are randomly initialized. Finally, based on the loss ($\mathcal{L}_{Ab}$) between the predicted comments (PComments) and ground-truth comments, ICS can iteratively update the model parameters of the abstracter, detailed in Section 4-B2-**Step** ⑩. The well-trained extractor and abstracter are the two core components of ICS to support the code comment generation service. When ICS is deployed for usage, it receives a code snippet from the developer and generates a concise natural language comment for the code snippet, detailed in Section 4-C.

4.2 Training of ICS

4.2.1 Part (i): Training of Extractor

The primary goal of the extractor is to identify highly informative statements, ensuring that the extracted statements contain essential information needed to generate an abstractive summary. As shown in part (i) of Figure 3, the training of the extractor is divided into four steps: ① and ② involve building up the LSA model, with ① constructing a token-code matrix and ② performing a truncated singular value decomposition on it. Subsequently, ③ calculates top-5 most relevant key tokens for a given code snippet, and ④ selects the statements containing the top-5 key tokens as important statements. Next, we will discuss these four steps in detail.

Step ① and Step ②: Building up the LSA model. LSA starts by obtaining a word-document matrix [29]. We treat each code snippet as a document and build a token-code matrix. Specifically, during the code preprocessing phase, we begin by considering a code snippet as an entire string and then tokenize it into a list of tokens. It's worth noting that all identifiers will be split into multiple token (also called words). For instance, "MergeSort" would be split into "merge" and "sort". Second, we remove stop-words from the token list, since most stop-words are uninformative pronouns, prepositions, conjunctions, and so on, such as "it", "in" and "or". In addition, we eliminate the keywords (also known as reserved words) specific to programming languages in use. Finally, we represent each code snippet by a set of meaningful tokens $s = [t_1, t_2, \cdots, t_a]$, and construct the token-code matrix X:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \tag{1}$$

where m is the number of unique tokens in the entire training set, and n is the size of the training set. This matrix essentially represents each token by a n-dimensional vector and each code snippet by a m-dimensional vector. The element x_{ij} represents the frequency or weight of the token t_i in the code snippet s_j. It is usually the number of occurrences of the word (token) in the document (code snippet), or the word's (token's) $tf\text{-}idf$ (term frequency-inverse document frequency) [32], and we apply the latter one. Given a set of code snippet S, a token t_i, and an individual code snippet $s_j \in S$, x_{ij} is computed as:

$$x_{ij} = f_{t_i,s_j} * log(n/f_{t_i,S}) \tag{2}$$

where f_{t_i,s_j} is the number of times t appears in s_j; $f_{t_i,S}$ is the number of code snippets in which t_i appears in S.

After the token-code matrix is constructed, a truncated singular value decomposition is performed on it:

$$X \approx U_k \Sigma_k V_k = [u_1, u_2, \cdots, u_k] \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \begin{bmatrix} {v_1}^T \\ {v_2}^T \\ \vdots \\ {v_k}^T \end{bmatrix} \tag{3}$$

where $k \leq n \leq m$. U_k is the topic vector space. Each column of it represents a topic, and each row of it represents a token. Each column of the matrix $\Sigma_k V_k$ represents a code snippet. After this, all but the first (largest) k values in the diagonal of the singular matrix Σ are set to zero, leading to a kind of principal component analysis. This also reduces the dimension of each code snippet and token to k.

Step ③: Calculating top-5 most relevant key tokens for a code snippet. After Step ① and Step ②, we get the k-dimensional vector representation of each token and code snippet in the matrix X. Note that the vector representation of a new code snippet that is not contained in the training set can be computed as:

$$\boldsymbol{v}_s = q^T U_k \Sigma_k^{-1} \tag{4}$$

where q represents the array containing $tf\text{-}idf$ value for tokens in the code snippet, and q^T is the transpose of it. This is equivalent to the geometric addition of constituent token vectors corresponding to tokens in a code snippet. Further, the similarity between $t_i \in s$ and s, denoted $sim(t_i, s)$:

$$sim(t_i, s) = cosine(\boldsymbol{v}_{t_i} \Sigma_k^{1/2}, \boldsymbol{v}_s \Sigma_k^{1/2}) \tag{5}$$

Finally, we follow [7] and select top-5 tokens with the highest similarity as key tokens. We also try to vary top-n to explore its impact on ICS's performance, detailed in Section 5-B4.

Step ④: Selecting important statements containing the key tokens. For the i-th statement in s, it can be represented by a set of tokens $stat_i = [t_1, t_2, \cdots, t_a]$ after code preprocessing. We can also get its top-5 key tokens $k = [w_1, w_2, \cdots, w_5]$ after Step ③. If

$$\exists w \in stat_i, w \in k \tag{6}$$

then we consider $stat_i$ an important statement. After traversing all the statements of a code snippet, we can get a set of important statements for it. It means that the model should pay special attention to the factual details contained in these statements to facilitate comment generation.

4.2.2 Part (ii): Training of Abstracter

As shown in part (ii) of Figure 3, the training of the abstracter is completed through six steps, i.e., ⑤–⑩. Next, we will discuss these six steps in detail.

Step ⑤: Extracting Important Statements. To generate comments without missing factual details, our ICS pays more attention to the important statements of code snippets where the factual details are contained in. Therefore, different from abstracters in existing abstractive code summarization techniques, the abstracter of ICS treats important statements as part of the input. In this step, we first use the well-trained extractor to calculate top-5 key tokens. Then, the statements containing these key tokens will be selected as important statements (IStates).

Step ⑥ and Step ⑦: Producing Embedding Representations. These two steps do a similar thing, i.e. leveraging an encoder to transform the source code into an embedding representation. The difference is that Step ⑥ deals with important statements selected by the extractor and Step ⑦ deals with the entire code snippet. Therefore, ICS can use the same neural network architecture or pre-trained model to design ExEncoder and AbEncoder. Given a code snippet $s = [stat_1, stat_2, \cdots, stat_r]$, let $s' \subseteq s$ denote a set of the important statements selected by the extractor from s, the tasks performed by ExEncoder and AbEncoder can be formalized:

$$\boldsymbol{e}^{Ex} = encoder(s'), \quad \boldsymbol{e}^{Ab} = encoder(s) \tag{7}$$

where $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ represent the embedding representations of s' and s, respectively; the encoder is a neural network architecture (e.g., LSTM [33] and Transformer [34]) or pre-trained model (e.g., CodeBERT [35] and CodeT5 [30]) that can process sequential input. We build our ExEncoder and AbEncoder over the pre-trained encoder provided by CodeT5 because CodeT5 performs better than other models in the task of code comment generation. The performances of different models are shown in Table II.

Step ⑧: Producing Integrated Representation. In this step, ICS integrates $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ to produce a integrated embedding representation e^{In} through the component Integrater. Since $\boldsymbol{e}^{Ex}$ and $\boldsymbol{e}^{Ab}$ are not aligned, we integrate them in a concatenated fashion. We try two concatenated ways as follows:

$$\boldsymbol{e}^{In} = [\boldsymbol{e}^{Ex}; \boldsymbol{e}^{Ab}] \text{ or } [\boldsymbol{e}^{Ab}; \boldsymbol{e}^{Ex}] \tag{8}$$

where $[\cdot;\cdot]$ denotes the concatenation of two vectors. The effects of both ways on the performance of ICS are discussed in Section 5-B3.

Step ⑨: Generating Predicted Comments. This step makes use of the decoder to generate natural language comment, which takes in the embedding representation $\boldsymbol{e}^{In}$ and predicts words one by one. Specifically, we build our decoder on the pre-trained encoder provided by CodeT5 [30] to unfold the context vector $\boldsymbol{e}^{In}$ into the target sequence (i.e., comment) through the following dynamic model,

$$\begin{aligned} \boldsymbol{h}_t &= f(y_{t-1}, \boldsymbol{h}_{t-1}, \boldsymbol{e}^{In}) \\ p(y_t|Y_{<t}, X) &= g(y_{t-1}, \boldsymbol{h}_t, \boldsymbol{e}^{In}) \end{aligned} \tag{9}$$

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 9 that the probability of generating a target word is related to the current hidden state, the history of the target sequence, and the context $\boldsymbol{e}^{In}$. The essence of the decoder is to classify the vocabularies by optimizing the loss function to generate the vector representing 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 ⑩: 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:

$$\mathcal{L}_{Ab}(\Theta) = -\frac{1}{N}\sum_{n=1}^{N} logp(\boldsymbol{y}_n|\boldsymbol{x}_n; \Theta) \tag{10}$$

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

4.3 Deployment of ICS

After ICS is trained, we can deploy it online for code comment generation service. Part (2) of Figure 3 shows the deployment of ICS. For a given code snippet s, ICS first uses the well-trained extractor to extract important statements from s, represented s'. Next, ICS uses the well-trained abstracter to generate the comment. In practice, we can treat the well-trained ICS as a black-box tool that takes in a code snippet given by the developer and generates a concise natural language comment.

5. Evaluation

To evaluate the proposed method, we aim to answer the following five research questions:

RQ1: How does ICS perform compared to baselines?
RQ2: How does ICS perform in human evaluation?
RQ3: How does the integration way of the extractor and abstracter affect ICS?
RQ4: How does the number of key tokens affect ICS?

RQ5: How does the robustness of ICS perform when varying the code length and comment length?

5.1 Experimental Setup

5.1.1 Dataset

We conduct experiments on the clean version of the CodeSearchNet (CSN) dataset [36] provided by [37]. This dataset has a large number of pairs of code snippets and comments across six programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby. It has been widely used by existing code comment generation studies [19], [35], [11], [21]. The statistics of it are listed in Table I.

TABLE I: Statistics of the CSN dataset

Language	Training Set Size	Validation Set Size	Test Set Size
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

5.1.2 Evaluation Metrics

We use three metrics BLEU [38], METEOR [39], and ROUGH-L [40] to evaluate our model ICS, which are widely used in code comment generation [9], [34], [41], [21]

BLEU, the abbreviation for BiLingual Evaluation Understudy [38], is commonly used to assess the quality of generated code summaries [9], [41]. It is a variant of precision metric, which measures the similarity of a generated summary to the reference summary by computing the n-gram precision, with a penalty for overly short lengths [38]. In this paper, we follow [21], [42] and report the standard BLEU score which provides a cumulative score of 1-, 2-, 3-, and 4-grams [43].

METEOR, the abbreviation for Metric for Evaluation of Translation with Explicit ORdering [39], is another widely used metric for evaluating the quality of generated code summaries [44], [45]. For a pair of summaries, METEOR establishes a word alignment between them and calculates the similarity scores.

ROUGE-L. ROUGE is the abbreviation for Recall-oriented Understudy for Gisting Evaluation [40]. 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 [41], [44], [46], [47].

5.1.3 Experimental Settings

In this paper, we build our encoder and decoder on the pre-trained encoder and decoder provided by CodeT5. Following CodeT5 [30] 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 [48]. The code snippets are padded with a special token $\langle PAD \rangle$ to the maximum length. 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 models are trained for the same epochs as their original paper, and we select the best model based on the lowest validation loss.

5.1.4 Baselines

LSTM [33] (Long Short-Term Memory) is a type of recurrent neural network architecture with multiple memory cells, each controlled by a set of gates to retain or discard information based on input data and the network's previous state.

Transformer-based [11] (also shortened to **Transformer** in [21], **NCS** in [49]) adopts a Transformer-based encoder-decoder architecture. It incorporates the copying mechanism [17] in the Transformer, enabling it to both generate words from vocabulary and copy from the source code.

CodeBERT [35] is a representative pre-trained model for source code. It uses the same model architecture as RoBERTa-base [50]. It is trained with the Masked Language Modeling (MLM) task and the Replaced Token Detection (RTD) task.

CodeT5 [30] is one of the state-of-the-art pre-trained models for source code. CodeT5 builds on an encoder-decoder framework with the same architecture as T5 [51]. CodeT5 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.

UniXcoder [52] is a unified cross-modal pre-trained model for programming language. It is based on a multi-layer Transformer and follows Dong et al. [53] to use mask attention matrices with prefix adapters to control the access to context for each token. It is trained with four tasks, including MLM (Masked Language Modeling), ULM (Unidirectional Language Modeling), DNS (Denoising Objective DeNoiSing), and CFRL (Code Fragment Representation Learning).

It should be noted that, strictly speaking, CodeBERT, CodeT5, and UniXcoder are three pre-trained models for source code, not code comment generation techniques. All of them can be used for multiple downstream software engineering tasks (such as code search, code clone detection, and code comment generation) by fine-tuning them on the corresponding downstream task datasets. In this paper, we fine-tune the pre-trained CodeBERT, CodeT5, and UniXcoder on the code comment generation task on the CSN dataset.

In addition to the DL-based abstractive methods introduced above, we also conduct experiments on the following two extractive methods as baselines.

TR-based [7] is an extractive method based on text retrieval (TR). As described in [4], TR-based methods generate comments 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 incorporate them in the comment. In process (2), various TR techniques can be integrated, such as Vector Space Model (VSM) [54], Latent Semantic Analysis (LSA) [26], Hierarchical Pachinko Allocation Model (hPAM) [55], to generate code comments. In this paper, we directly use the top-5 key tokens extracted through LSA introduced in Section 4-B1 as the TR-based summary.

TABLE II: Overall performance of ICS and baselines.

	Technique	Go			Java			JavaScript			PHP			Python			Ruby		
		$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$
3	LSTM	17.8	15.1	35.6	12.2	10.1	24.6	10.4	6.2	17.2	19.5	12.2	29.8	13.9	9.1	23.3	9.4	5.3	16.3
4	Transformer	19.8	16.2	38.4	15.3	11.8	30.6	11.2	7.4	20.5	21.5	13.9	34.2	15.8	10.6	31.3	10.3	6.4	18.3
5	CodeBERT	21.1	17.5	43.6	18.0	12.4	35.5	13.3	8.7	24.3	24.6	15.3	39.4	18.7	12.4	34.8	11.2	7.1	20.6
6	CodeT5	22.1	18.5	44.8	20.4	14.5	38.1	15.8	11.2	28.9	25.9	18.0	43.0	20.0	15.1	37.8	14.9	10.8	27.9
7	UniXcoder	19.2	14.8	38.9	20.1	13.4	36.7	15.6	10.0	26.9	26.2	16.5	41.2	19.9	13.5	36.8	15.0	9.7	26.8
8	TR-based	5.1	4.1	9.3	7.4	5.1	10.2	6.1	4.2	8.4	11.0	6.8	12.7	5.8	3.9	7.6	6.3	4.1	8.5
9	EX-based	21.6	18.9	43.6	19.8	14.6	37.5	14.9	10.5	27.7	24.7	17.2	41.1	18.6	13.6	35.5	14.9	10.4	27.9
10	ICS	**23.4**	**20.4**	**46.9**	**20.9**	**15.2**	**39.1**	**15.9**	**11.4**	**29.1**	**26.2**	**18.1**	**43.7**	**20.4**	**15.1**	**38.6**	**15.2**	**11.1**	**28.7**

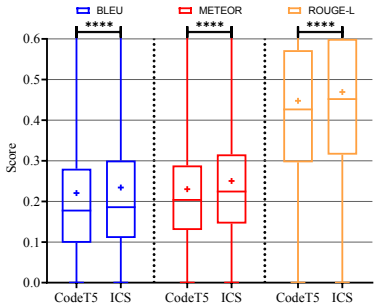


Figure 4: Distributions of Metrics' Scores of in language Go

Extractor-based (Ex-based, for short) is an extractive method implemented through our extractor. In this approach, we generate summaries by directly feeding the important statements extracted by the extractor to CodeT5.

5.2 Experimental Results

5.2.1 RQ1: ICS vs. Baselines

Table II shows the performance of our ICS and baselines in BLEU ($\mathcal{B}$), METEOR ($\mathcal{M}$), and ROUGE-L ($\mathcal{R}$). Rows 3–7 of Table II indicate that CodeT5 overall outperforms UniXcoder and previous baselines across all six languages of the CSN dataset, based on all three metrics. Consequently, in the rest of our paper, we mainly focus on comparing ICS with CodeT5. Rows 8–9 show the results of the two extractive methods, i.e., TR-based and Ex-based. It is observed that TR-based performs significantly worse than Ex-based. In general, the best abstractive method CodeT5 outperforms the best extractive method Ex-based overall. The last row of Table II displays the results of our ICS. We observed that ICS is better than the best baseline CodeT5 overall, with particular strength on the Go dataset.

We further analyze the distribution of the metric scores of CodeT5 and ICS on the test samples of the Go dataset, and the statistical results are shown in Figure 4. In Figure 4, '+' denotes the mean, which is the value filled in Table II. It can be seen that the first, median, and third quartiles associated with ICS are better than those associated with CodeT5. To test whether there is a statistically significant difference between the two techniques, we perform the paired Wilcoxon-Mann-Whitney signed-rank test at a significance level of 5%, following previously reported guidelines for inferential statistical analysis involving randomized algorithms [56]. In Figure 4, '*' ($0.01 < p < 0.05$), '**' ($0.001 < p < 0.01$), '***' ($0.0001 < p < 0.001$), and '****' ($p < 0,0001$) represent the differences between the two groups are Significant, Very significant, Extremely significant, and Extremely significant, respectively. Plus, 'ns' ($p \geq 0.05$) means Not significant. From the figure, it is observed that all p-values between ICS and CodeT5 are smaller than the significant threshold value of 0.05. Based on this, combining the results in Table II and Figure 4, it can be concluded that the performance of ICS is significantly better than CodeT5 on the Go dataset.

TABLE III: Results of human evaluation. The values in parentheses represent standard deviations.

Dataset	Metrics	UniXcoder	CodeT5	ICS
Java	Similarity	2.63 (0.96)	2.79 (0.94)	**2.85** (0.90)
	Naturalness	3.36 (0.78)	3.36 (0.79)	**3.40** (0.73)
	Informativeness	2.61 (1.02)	2.79 (1.07)	**2.87** (1.09)
	Relevance	2.89 (0.99)	3.04 (0.97)	**3.12** (0.97)
Average		2.87	2.99	**3.07**
Python	Similarity	2.48 (0.98)	2.64 (0.89)	**2.84** (0.71)
	Naturalness	3.21 (0.82)	3.32 (0.72)	**3.49** (0.77)
	Informativeness	2.35 (1.00)	2.72 (0.95)	**2.88** (0.75)
	Relevance	2.55 (1.00)	2.87 (0.88)	**3.0** (0.80)
Average		2.65	2.89	**3.05**

5.2.2 RQ2: Human Evaluation

Many works [9], [57], [58], [44], [59], [60] have demonstrated that automatic evaluation metrics (BLEU, METEOR, and ROUGE-L) primarily calculate the textual similarity rather than the semantic similarity between reference comments and generated comments. Hence, following the previous works [21], [57], [58], [44], we perform a human evaluation to assess the comments generated by the baselines UniXcoder, CodeT5, and our ICS. Specially, we invite 6 volunteers with over 3 years of experience in software development and strong English proficiency to carry out the evaluation. Each volunteer is requested to rate the generated comments on four aspects using a scale from 0 to 4 (higher score indicating better performance): *similarity* (similarity between generated and reference comments), *naturalness* (grammatical correctness and fluency), *informativeness* (amount of content preserved from input code snippets in the generated comments, disregarding fluency), and *relevance* (how well the generated comments match the input code snippets). We randomly select 50 code snippets, 25 each from the Java dataset and the Python dataset sourced from the CSN dataset, along with their corresponding comments generated by UniXcoder, CodeT5, and our ICS, as well as the reference comments (i.e., ground-truth). We divide the 50 samples into two groups, each containing 25 samples. To ensure fairness and reduce volunteer workload, each volunteer evaluates only one group of samples randomly. Each comment is evaluated by 3 volunteers, and the final score is calculated as the average of their individual assessments.

The results of the human evaluation are shown in Table III. From Table III, it reveals that overall our ICS consistently outperforms UniXcoder and CodeT5 in all four aspects. On the Java dataset, compared with UniXcoder and CodeT5,

TABLE IV: Influence of Integration Ways on ICS

Integration Way	Java			Python		
	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$
$[e^{Ab}; e^{Ex}]$	**20.86**	15.22	39.10	20.4067	15.13	38.56
$[e^{Ex}; e^{Ab}]$	20.51	**15.46**	**39.15**	**20.4071**	**15.15**	**38.57**
p-value	**ns**	**	**ns**	**ns**	**ns**	**ns**

TABLE V: Influence of the Number of Key Tokens on ICS

Number of Key Tokens	Java			Python		
	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$
3	20.56	**15.72**	39.25	20.23	14.98	38.03
5	**20.86**	15.22	39.10	**20.41**	**15.13**	**38.56**
7	20.69	15.09	38.88	20.11	14.92	38.10
9	20.70	15.58	**39.38**	20.17	14.9	38.05
p-value	****	****	**ns**	****	****	****

ICS shows an average improvement of 6.97% and 2.68% in four aspects, respectively. While on the Python dataset, ICS improves on average by 15.09% and 5.54%, respectively.

5.2.3 RQ3: Influence of Integration Ways on ICS

We also conduct an experiment to investigate the influence of the integration ways (i.e., $[e^{Ab}; e^{Ex}]$ and $[e^{Ex}; e^{Ab}]$.) on the performance of ICS. The experimental results are shown in Table IV. From rows 3-4 of Table IV, it is observed that except for the BLEU metric on the Java dataset, $[e^{Ex}; e^{Ab}]$ is slightly better than $[e^{Ab}; e^{Ex}]$. For each metric, we further perform the paired Wilcoxon-Mann-Whitney signed-rank test on all scores of both integration ways at a significance level of 5%. The results are listed in the last row of Table IV, where the symbol "*" has the same meaning as that in Figure 4. For example, in the $\mathcal{B}$ column indicates that there is no significant difference between $[e^{Ab}; e^{Ex}]$ and $[e^{Ex}; e^{Ab}]$ in terms of the BLEU score (i.e., p-value > 0.05). From the last row of Table IV, it is observed that except for the METEOR metric on the Java dataset, there is no significant difference between $[e^{Ex}; e^{Ab}]$ and $[e^{Ab}; e^{Ex}]$. Based on the above observations, it can be concluded that the integration way has less influence on the performance of ICS.

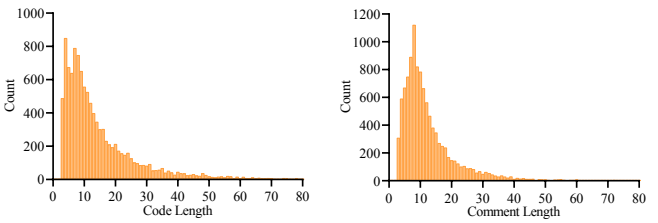
5.2.4 RQ4: Influence of the Number of Key Tokens on ICS

As described in Section 4-B1, key tokens decide the selection of important statements, directly impacting the extractor's performance. Therefore, the quantity of key tokens also affects the quality of the selected important statements. Specifically, a smaller number of key tokens will result in fewer important statements being identified, possibly omitting certain factual details. Conversely, more key tokens will result in a larger number of important statements, where there may exist redundant information. Accordingly, we conduct an experiment on the Java and Python datasets to investigate the influence of the number of key tokens on the performance of ICS, setting it to 3, 5, 7, and 9. The results are shown in Table V.

From rows 3–6 of Table V, we note that 5 key tokens yield the best scores, with the exception of the METEOR and ROUGE-L metrics on the Java dataset. Furthermore, we conduct the Friedman test at a significance level of 5% to examine the statistical significance of the results. The results are presented in the final row of Table V, and the symbol '*' has the same meaning as that in Figure 4. The analysis reveals that all scores, other than the ROUGE-L metric on the Java dataset, exhibit statistically significant differences. We further perform Dunn's multiple comparisons test [61] on these scores to identify the groups with significant differences. It turns out that on the BLEU metric of the Java dataset and all the metrics of the Python dataset, the results of 5 key tokens significantly differ from other key token numbers. Based on the experimental results, setting the number of key tokens to 5 is deemed reasonable as it performs better overall.

5.2.5 RQ5: Robustness of ICS

To study the robustness of ICS, we analyze two parameters (i.e., code length and comment length) that may affect the embedding representations of code snippets and comments. Figure 5 shows the length distributions of code snippets and comments on the test sets of the Java and Python datasets. Code length is measured by the number of lines in a code snippet. Comment length is determined by the number of words in a comment. From Figures 5(a) and 5(c), it is observed that most code snippets consist of less than 50 lines. Similarly, Figures 5(b) and 5(d) show that the majority of comments are less than 30 words.

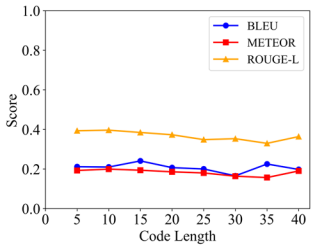


(a) Java code length distribution (b) Java comment length distribution

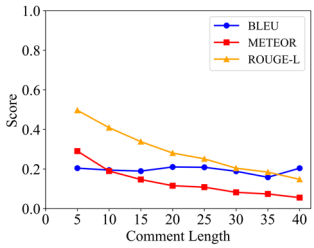
(c) Python code length distribution (d) Python comment length distribution

Figure 5: Length distribution of samples in test sets

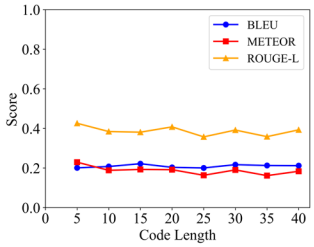
Figure 6 presents the performance of ICS on different evaluation metrics with varying parameters. As shown in Figures 6(a) and 6(c), ICS can maintain stable performance even with the increase of the code snippet length. Figures 6(b) and 6(d) reveal that, with the increase of comment length, ICS also maintains stable performance in terms of BLUE, but the scores of METEOR and ROUGE-L decrease steadily. We further assess the performance of CodeT5 on varying the comment length of the Java and Python datasets, and the results, shown in Figure 7, demonstrate similar behavior to ICS. It indicates that generating high-quality comments becomes



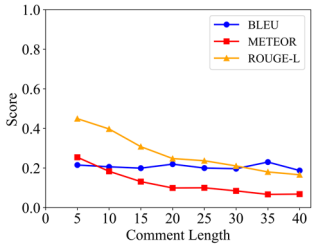
(a) Robustness on varying Ja codes



(b) Robustness on varying Ja comments

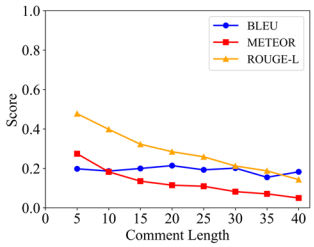


(c) Robustness on varying Py codes

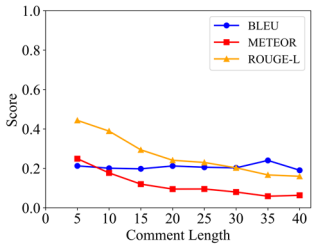


(d) Robustness on varying Py comments

Figure 6: Effect of code snippet and comment length on the robustness of ICS. Ja: Java; Py: Python.



(a) Robustness on varying Ja comments



(b) Robustness on varying Py comments

Figure 7: Robustness of CodeT5 on varying comments

more challenging as the expected length of the generated comment increases. Overall, the results provide evidence for the robustness of our ICS approach.

5.3 Case Study on Go Code Comment Generation

This section provides a case study aiming to compare the generated comments of ICS with the state-of-the-art CodeT5 and to demonstrate the effectiveness of our ICS approach. We take the code snippet s_2 in Figure 8 as an example and apply both CodeT5 and our ICS to generate comments for comparison. The reference comment is displayed in the first line of Figure 8(c). The comments generated by CodeT5 and ICS for s_2 are shown in the second and third lines, respectively. From the figure, we observe that, compared to the reference comment, 1) the comment generated by CodeT5 only covers the red and blue part but omits a crucial factual detail, namely "to the watcher"; 2) although different tenses are used, the comment generated by our ICS successfully covers all three parts and remains semantically equivalent. It is worth noting that, as depicted in Figure 8(b), our extractor successfully extracts the important statement containing the factual detail "watcher" appears. In summary, our ICS outperforms CodeT5 in terms of the semantic completeness of the generated comments. This success can be attributed to the integration of the extractor and abstracter, enabling the successful generation of the text "to the watcher." Based on the above, we can conclude that our ICS is a competitive technique for the Go code comment generation task. Additional cases can be viewed in the publicly reproducible artifact [15].

```
func (w *Watcher) Add(stream *Stream) {
    coal.Init(stream.Model)
    if w.streams[stream.Name()] != nil {
        ...
    }
    w.streams[stream.Name()] = stream
    coal.OpenStream(stream.Store, steam.Model, nil,
        func(e coal.Event, id bson.ObjectId, m coal.Model, token []byte) {
            if stream.SoftDelete && e == coal.Deleted {
                return
            }
            if stream.SoftDelete && e == coal.Updated {
                ...
            }
            evt := &Event{Type: e, ID: id, Model: m, Stream: stream,}
            w.manager.broadcast(evt)
        },
    ...
}
```

(a) A Code Snippet s_2 from the Test Set of the CSN Dataset

```
func (w *Watcher) Add(stream *Stream) {
```

(b) Important Statements Selected by Our Extractor

Reference Comment: Add will add a stream to the watcher.
CodeT5: Add adds a new stream.
ICS: Add adds a stream to the watcher.

(c) Comments Generated by Different Techniques

Figure 8: Case study on Go code comment generation

6. Related Work

Code comment generation has consistently been one of the hottest research topics in software engineering. The majority of early code comment generation techniques [7], [4], [62] are extractive methods. These methods involve extracting a subset of the statements and keywords from the code, and then incorporating information into the generated comment. Typically, text retrieval (TR) techniques (e.g., Vector Space Model [54], Latent Semantic Indexing [26], and Hierarchical PAM [55]) are utilized to determine the most important n terms. Acknowledging that the quality of the comments generated by extractive methods relies heavily on the extracting process, Paige Rodeghero et al. [63] conducted an eye-tracking study of programmers and proposed a tool for selecting keywords based on the study's findings. The extractive methods depend on high-quality identifier names and method signatures from the source code. However, these techniques may fail to generate accurate comments if the source code contains poorly named identifiers or method namess [64].

Currently, DL-based (abstractive) code comment generation techniques are being continuously proposed. Abstractive methods, which combine Seq2Seq models trained on large-scale

code-comment datasets, can generate words that do not explicitly appear in the given code snippet, thereby overcoming the limitations of extractive methods. Srinivasan Iyer et al. [9] introduce the first fully abstractive method for generating high-level summaries of code snippets. Their results demonstrate that abstractive methods surpass extractive methods significantly in terms of the naturalness of generated summaries. Code representation plays a pivotal role in abstractive methods. To produce semantic-preserving code representations, researchers have explored various aspects of the code snippet, including tokens [9], abstract syntactic trees (ASTs) [18], control flows [65], code property graph [66]. In addition, existing abstractive methods have experimented various networks, e.g., LSTM [9], Bi-LSTM [44], GRU [57], Transformer [11] and GNN [67]. Despite the great potential of DL-based abstractive methods in generating human-like summaries, we observed that the generated summaries often lack important factual details. Our ICS combines both extractive and abstractive techniques. The extractive module of ICS applies LSA to calculate the top-5 keywords for each code snippet and extract important statements. Unlike existing abstractive methods, our abstracter receives both of the entire code snippet and important statements as input and processes them.

7. Conclusion

This paper provides an extractive-and-abstractive method to enhance code comment generation, namely ICS. The extractor utilizes LSA to identify crucial statements from the code snippet. The important statements comprise essential factual details that should be incorporated into the final generated comment. The abstracter of ICS takes in both the important statements extracted by the extractor and the entire code snippet as input, enabling it to generate human-written-like comments. The experimental results on the CSN dataset demonstrate that ICS is effective in code comment generation and outperforms the state-of-the-art. Furthermore, extensive human evaluations indicate that the comments generated by ICS exhibit superior informativeness and relevance to given code snippets.

Acknowledgment

This work is supported partially by the National Natural Science Foundation of China (62141215, 62272220, 62372228), the Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

References

[1] C. S. Hartzman and C. F. Austin, "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*, 1993, pp. 138–170.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "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*, 2005, pp. 68–75.

[3] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 2269–2275.

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

[5] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.

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

[7] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 223–226.

[8] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for Java methods," *Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

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

[10] J. Moore, B. Gelman, and D. Slater, "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*, 2019, pp. 15–26.

[11] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[12] L. N. Phan, H. Tran, D. Le, H. Nguyen, J. T. Anibal, A. Peltekian, and Y. Ye, "CoTexT: Multi-task learning with code-text transformer," *CoRR*, vol. abs/2105.08645, 2021.

[13] T. Shi, Y. Keneshloo, N. Ramakrishnan, and C. K. Reddy, "Neural abstractive text summarization with sequence-to-sequence models," *Transactions on Data Science*, vol. 2, no. 1, pp. 1:1–1:37, 2021.

[14] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 81:1–81:37, 2018.

[15] W. Sun and Y. Hu, "ICS." site: https://github.com/wssun/ICS, 2023, accessed July 2023.

[16] M. Gambhir and V. Gupta, "Recent automatic text summarization techniques: A survey," *Artificial Intelligence Review*, vol. 47, no. 1, pp. 1–66, 2017.

[17] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 1073–1083.

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

[19] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proceedings of the 33rd Annual Conference on Neural Informatiom Processing Systems*, 2019, pp. 6559–6569.

[20] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "translation": Data, metrics, baselining & evaluation," in *Proceedings of the 35th International Conference on Automated Software Engineering*, 2020, pp. 746–757.

[21] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," in *Proceedings of the Findings of the 59th Annual Meeting of the Association for Computational Linguistics*, 2021, pp. 1078–1090.

[22] W. T. Hsu, C. Lin, M. Lee, K. Min, J. Tang, and M. Sun, "A unified model for extractive and abstractive summarization using inconsistency loss," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018, pp. 132–141.

[23] J. Pilault, R. Li, S. Subramanian, and C. Pal, "On extractive and abstractive neural document summarization with transformer language models," in *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing*, 2020, pp. 9308–9319.

[24] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[25] S. T. Dumais *et al.*, "Latent semantic analysis," *Annu. Rev. Inf. Sci. Technol.*, vol. 38, no. 1, pp. 188–230, 2004.

[26] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[27] J. Steinberger, K. Jezek *et al.*, "Using latent semantic analysis in text summarization and summary evaluation," *Proc. ISIM*, vol. 4, no. 93-100, p. 8, 2004.

[28] J. Steinberger and M. Křišťan, "Lsa-based multi-document summarization," in *Proceedings of 8th International PhD Workshop on Systems and Control*, vol. 7. Citeseer, 2007, pp. 1–5.

[29] K. Kireyev, "Using latent semantic analysis for extractive summarization," in *Proceedings of the First Text Analysis Conference*, 2008, pp. 1–4.

[30] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "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*, 2021, pp. 8696–8708.

[31] J. Steinberger and K. Jezek, "Update summarization based on latent semantic analysis," in *Proceedings of the 12th International Conference on Text, Speech and Dialogue*, 2009, pp. 77–84.

[32] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1, 2003, pp. 29–48.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 5998–6008.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.

[36] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[37] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021, pp. 1–14.

[38] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

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

[40] C.-Y. Lin, "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*, 2004, pp. 74–81.

[41] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[42] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source code summarization with structural relative position guided transformer," *CoRR*, 2022.

[43] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *Transactions on Software Engineering and Methodology*, vol. 30, no. 2, pp. 25:1–25:29, 2021.

[44] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[45] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *Proceedings of the 29th International Conference on Program Comprehension*, 2021, pp. 1–12.

[46] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *Proceedings of the 29th International Conference on Program Comprehension*, 2021, pp. 253–264.

[47] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *Proceedings of the 29th International Conference on Program Comprehension*, 2021, pp. 184–195.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3th International Conference on Learning Representations – Poster*, 2015, pp. 1–15.

[49] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "Neural code summarization: How far are we?" in *CoRR*, vol. abs/2107.07112, 2021.

[50] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, pp. 1–13, 2019.

[51] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[52] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022, pp. 7212–7225.

[53] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H. Hon, "Unified language model pre-training for natural language understanding and generation," in *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems*, 2019, pp. 13 042–13 054.

[54] G. Salton, A. Wong, and C. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[55] D. M. Mimno, W. Li, and A. McCallum, "Mixtures of hierarchical topics with pachinko allocation," in *Proceedings of the 24th International Conference Machine Learning*, 2007, pp. 633–640.

[56] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

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

[58] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proceedings of the 35th International Conference on Automated Software Engineering*, 2020, pp. 349–360.

[59] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *Proceedings of the 36th International Conference on Automated Software Engineering*, 2021, pp. 155–166.

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

[61] O. J. Dunn, "Multiple comparisons using rank sums," *Technometrics*, vol. 6, no. 3, pp. 241–252, 1964.

[62] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 23–32.

[63] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. K. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 390–401.

[64] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 380–389.

[65] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization using hierarchical attention," *Transactions on Software Engineering*, pp. 1–19, 2020.

[66] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Automatic code summarization via multi-dimensional semantic fusing in GNN," *CoRR*, vol. abs/2006.05405, 2020.

[67] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.