Expert Systems

WILEY

Expert Systems

ORIGINAL ARTICLE

Automatic Code Summarization Using Abbreviation Expansion and Subword Segmentation

Yu-Guo Liang[1] | Gui-Sheng Fan[1] | Hui-Qun Yu[1] | Ming-Chen Li[1] | Zi-Jie Huang[1,2]

[1]School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China | [2]Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, China

Correspondence: Gui-Sheng Fan (gsfan@ecust.edu.cn) | Hui-Qun Yu (yhq@ecust.edu.cn)

Received: 24 June 2024 | **Revised:** 22 November 2024 | **Accepted:** 26 December 2024

Funding: This work was supported by National Natural Science Foundation of China (No. 62372174), the Computational Biology Program of Shanghai Science and Technology Commission (No. 23JS1400600), the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality (No. 22010504100), the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies (No. 2021-GYHLW-01007), and the Shanghai 2024 Science and Technology Innovation Action Plan Star Cultivation (Sailing Program, No. 24YF2719900 and 24YF2720000).

Keywords: automatic code summarization | code abbreviation expansion | deep learning | program understanding | subword segmentation

ABSTRACT

Automatic code summarization refers to generating concise natural language descriptions for code snippets. It is vital for improving the efficiency of program understanding among software developers and maintainers. Despite the impressive strides made by deep learning-based methods, limitations still exist in their ability to understand and model semantic information due to the unique nature of programming languages. We propose two methods to boost code summarization models: context-based abbreviation expansion and unigram language model-based subword segmentation. We use heuristics to expand abbreviations within identifiers, reducing semantic ambiguity and improving the language alignment of code summarization models. Furthermore, we leverage subword segmentation to tokenize code into finer subword sequences, providing more semantic information during training and inference, thereby enhancing program understanding. These methods are model-agnostic and can be readily integrated into existing automatic code summarization approaches. Experiments conducted on two widely used Java code summarization datasets demonstrated the effectiveness of our approach. Specifically, by fusing original and modified code representations into the Transformer model, our Semantic Enhanced Transformer for Code Summarizsation (SETCS) serves as a robust semantic-level baseline. By simply modifying the datasets, our methods achieved performance improvements of up to 7.3%, 10.0%, 6.7%, and 3.2% for representative code summarization models in terms of *BLEU-4*, *METEOR*, *ROUGE-L* and *SIDE*, respectively.

1 | Introduction

Program understanding is essential to software development and maintenance (Storey 2005). The presence of high-quality natural language descriptions for code can significantly enhance the readability and understandability of the program, thereby boosting the work efficiency of software development and maintenance personnel (He 2019). Automatic code summarization, as a task of automatically generating corresponding functional descriptions for code, is currently a hot research topic in the field of program understanding (Moreno and Marcus 2018; Rai, Belwal, and Gupta 2022).

As advances in deep learning techniques and the enrichment of open-sourced code summarization corpora, data-driven deep learning methods have significantly improved the efficiency

and quality of auto-generated summaries. Iyer et al. (2016) pioneered the integration of deep neural networks in automatic code summarization, employing the sequence-to-sequence (Seq2Seq) model within the end-to-end NMT framework to generate code summaries. Since the Transformer (Vaswani et al. 2017) emerged in recent years has advantages in representing long sequences, researchers have continuously proposed advanced code summarization frameworks based on this prevailing model. Most deep learning based automatic code summarization approaches draw inspiration from NMT solutions in NLP, and concentrate on exploring the relationship between code-related semantic as well as structural information and natural language descriptions (Rai, Belwal, and Gupta 2022).

Pre-trained code models, which build upon the architectures of existing deep learning models, are initially trained on extensive multi-language datasets and subsequently fine-tuned on smaller, task-specific datasets. These models leverage elaborated pre-training tasks to obtain universal code representation suitable for multiple programming languages. This makes them versatile for various downstream software engineering tasks, including automatic code summarization. Similarly, these models borrow key concepts from pre-trained language models in the NLP field, with a primary focus on designing innovative pre-training tasks that accommodate the unique characteristics of code (Niu, Li, Luo, et al. 2022).

Although deep learning based automatic code summarization approaches have achieved impressive results, we discover that existing code summarization models are still facing difficulties in understanding and modelling complex information contained in code. For instance, Table 1 presents a Java code snippet (part of the code is truncated for the sake of brevity) and the corresponding summary description in the Funcom dataset (LeClair, Jiang, and McMillan 2019), where information of the abbreviated formal parameter 'u' is reflected in the summary. Since Java is a strongly typed language, the type 'URL' of the formal parameter in this example may aid models in generating an accurate summary to some extent. However, basic data types like 'int' and 'char' in other code snippets can offer limited information, making it challenging for these models. This necessitates the conversion of abbreviations nested in source code, particularly in identifiers, into corresponding full terms, which is the goal of the code abbreviation expansion task. Code abbreviation expansion is able to enhance both the understandability of source code and the accuracy of natural language analysis techniques (Newman et al. 2019). Ideally, the uncertainty of abbreviations' semantic information can be eliminated by means of code abbreviation expansion, which not only helps code summarization models better understand codes but enables them to focus on critical identifiers themselves rather than their types, fostering better text alignment between programming and natural language. Exploratory experiments suggest that an increase in code abbreviations deteriorates the performance of a code summarization model. Therefore, this article's primary objective is to investigate whether code abbreviation expansion is capable of improving the performance of code summarization models.

Moreover, the out-of-vocabulary (OOV) issue is another challenge in automatic code summarization (Sharma, Chen, and Fard 2022; Cheng et al. 2022). This problem usually arises when the model encounters identifiers that it has not seen during training, therefore, they are not included in its vocabulary. To mitigate this issue, current code summarization approaches split code and summary sequences into individual words using predefined split functions based on the CamelCase and snake_case naming conventions (LeClair, Jiang, and McMillan 2019; Hu et al. 2020; Ahmad et al. 2020). For example, if the 'imgname' identifier included in the code snippet appears infrequently across the dataset, it may not be included in the model's vocabulary. In such cases, during both model training and inference stages, the identifier would be replaced by a special symbol (usually denoted as <unk>), representing an unknown word. This replacement leads to the loss of critical information because the model cannot learn the semantic meaning of it. However, even if the identifier is frequent enough to be included in the vocabulary, it can still be challenging for code summarization models to understand its actual meaning and generate an accurate summary. This is because the traditional naming convention-based split functions cannot split 'imgname' into the more meaningful tokens 'img' and 'name'. As a result, the model might struggle to generate the corresponding summary 'image'.

Although subword segmentation methods, initially developed for NMT, have effectively addressed the OOV problem and have

TABLE 1 | A code snippet containing abbreviations and identifiers that does not comply with naming conventions.

Function ID	36110318

Code	`public void load(URL u){`
	`  FileCacheSeekableStream s = new FileCacheSeekableStream(`
	`  u.openStream());`
	`  load(s);`
	`  imgname = u.toString();`
	`  ......`
	`}`

Summary	Loads the image from a URL.

been widely adopted in pre-trained language models, these methods have yet to be considered in automatic code summarization approaches. Existing pre-trained code models have directly utilised subword algorithms from referenced pre-trained language models, without making necessary adjustments to accommodate the unique characteristics of code (Niu, Li, Luo, et al. 2022). As a result, their usefulness in addressing the aforementioned challenges is limited. Consequently, the second aim of this article is to explore how to effectively employ subword segmentation algorithms to tokenize words that traditional functions fail to split, and to validate their effectiveness in code summarization models. The main contributions of our work include:

- We propose the use of code abbreviation expansion to weaken the negative impact of abbreviations on program understanding and strengthen the language alignment ability of code summarization models. A series of context-based heuristic algorithms are adopted to expand abbreviations nested in code snippets of Java code summarization datasets.
- We introduce the unigram subword segmentation algorithm to expose more semantic information and further enhance the program understanding performance of code summarization models. Code-specific tokenizers are developed to tokenize code-summary pairs into more granular and semantically preserved subword sequences.
- We present a framework Semantic Enhanced Transformer for Code Summarization (SETCS) to better leverage the semantic information introduced by above methods. A robust baseline is designed by fusing embeddings of original and newly generated subtoken sequences, allowing for effective capture of critical information.
- To the best of our knowledge, this is the first work that incorporates code abbreviation expansion and subword segmentation into the automatic code summarization task. These methods are model-agnostic and can be easily integrated into existing automatic code summarization approaches. Experiments conducted on two widely evaluated datasets demonstrate the effectiveness of our proposed methods.

The remainder of this article is structured as follows. Section 2 summarises related work. Section 3 details our proposed methods. The experimental setup and results are explained and analysed in Sections 4 and 5, respectively. Finally, we conclude the article and discuss potential avenues for future research in Section 6.

2 | Related Work

2.1 | Automatic Code Summarization

Automatic code summarization approaches focus on leveraging code-related information to generate high-quality summary descriptions. Based on the type of information leveraged, existing research can be divided into two categories.

2.1.1 | Structure-Driven Code Summarization Models

Hu, Li, Xia, Lo, and Jin (2018) first proposed a method of using the abstract syntax tree (AST) representation of source code to improve the performance of the code summarization model. Subsequent works tried to adopt, optimise AST, or introduce more advanced structural information, such as combined usage of AST and serialised code (LeClair, Jiang, and McMillan 2019; Hu et al. 2020; Zhou et al. 2022; Tang et al. 2022), fine-grained split ASTs (Zhang et al. 2019; Lin et al. 2021), and utilisation of code property graph (Liu et al. 2021), multi-view graph (Wu, Zhao, and Zhang 2021), dataflow graph (Gao et al. 2023), as well as heterogeneous code graph (Guo et al. 2024).

2.1.2 | Semantic-Driven Code Summarization Models

TL-CodeSum (Hu, Li, Xia, Lo, Lu, et al. 2018) and API2Com (Shahbazi, Sharma, and Fard 2021) demonstrated the effectiveness of application programming interface (API) information for code summarization. DMACOS (Xie et al. 2021) exploited the deliberation network and adopted method name prediction as an auxiliary training task to improve the quality of generated summaries. Li et al. (2024) utilised multi-task joint learning to incorporate action word prediction into code summarization models. Both Rencos (Zhang et al. 2020) and Re2Com (Wei et al. 2021) combined traditional information retrieval techniques with deep neural networks to exploit the information contained in retrieved similar code snippets or corresponding summaries. MLCS (Zhou et al. 2023), a code summarization method based on meta-learning and code retrieval, and MPCos (Xie et al. 2023) designed meta-learning frameworks for the automatic code summarization task in different scenarios, among which the key idea is to use similar code samples to obtain specific summary generators optimised for each target code snippet.

Existing pre-trained code models can also be classified into the above two categories according to different types of model input in the pre-training stage. For example, in addition to source code, GraphCodeBERT (Guo et al. 2021) and SPT-Code (Niu, Li, Ng, et al. 2022) took control flow graph and AST as additional code-related structural input respectively, while CodeBERT (Feng et al. 2020), CodeT5 (Wang et al. 2021) and PLBART (Ahmad et al. 2021) took code-related semantic information such as summaries and posts as additional model inputs.

Both code abbreviation expansion and subword segmentation methods introduced in this article fall into the second category, as the former method utilises related identifiers to expand abbreviations nested in the source code and the latter method assists in code summarization models by exposing more semantic information included in the code snippet.

2.2 | Code Abbreviation Expansion

Due to the limitations of abbreviation dictionaries and general English dictionaries, more advanced approaches for code abbreviation expansion focus on contextual information of

abbreviations, including comments, methods, classes, and projects. In addition, most researchers generally adopt certain predefined matching rules to find potential expansions by identifying different types of abbreviations. According to our survey, a series of works made by Jiang et al., (Jiang, Liu, and Zhang 2019; Jiang et al. 2020; Jiang et al. 2021; Jiang et al. 2022) in recent years have significantly improved the recall and precision scores of the code abbreviation expansion task in multiple open-source applications.

Literature Jiang, Liu, and Zhang (2019) used the semantic relationships between software entities and construct knowledge graphs for entities, semantically related entities, and their relationships to obtain full terms of abbreviations in software entities. Literature Jiang et al. (2020) designed a series of heuristic methods utilising specific fine-grained context to expand the abbreviations in both formal and actual parameters. In response to the question of whether target abbreviations should be replaced with the corresponding full names, literature Jiang et al. (2021) proposed an automatic decision-making tool for abbreviation expansion. On the basis of Jiang, Liu, and Zhang (2019), literature Jiang et al. (2022) further proposed an automatic identifier abbreviation expansion method that leverages the semantic relationship between software entities and migration expansion within the same application.

To expand abbreviations nested source codes of code summarization datasets, we re-implement and refine three heuristic algorithms so that abbreviations in identifiers such as parameters and variable names can be expanded as much as possible. These algorithms have been proved to be highly precise when tested across a range of well-known open-source projects (Jiang et al. 2020).

2.3 | Subword Segmentation

Byte pair encoding (BPE) (Gage 1994) is a data compression technology and the original idea is to iteratively replace the most frequently occurred byte pairs in a sequence with a single, unused byte. It was later adopted by Sennrich, Haddow, and Birch (2016) to solve the OOV problem in the NMT task and became the dominant method for subword segmentation. By continuously merging frequently occurred character pairs or sequences, BPE can retain the most frequently occurred subwords in the process of segmenting rare words. It is worth to note that both CodeBERT and CodeT5 adopt the tokenizer of Roberta (Liu et al. 2019), which is a pretrained language model that utilises this algorithm.

Similarly, the WordPiece algorithm (Wu et al. 2016) also starts from a small vocabulary and continuously learns the merging rules during the training of a tokenizer. The difference is that WordPiece prioritises character pairs with lower frequencies in each part of the vocabulary, and it does not use merging rules learned in the training stage but looks for the longest subword from the vocabulary for segmentation in the tokenisation stage.

Contrary to the above two methods, the Unigram algorithm (Kudo 2018) continuously removes unnecessary words from a large vocabulary until the desired vocabulary size is reached. In addition, both BPE and WordPiece segment sentences or words into unique subword sequences, while Unigram is capable to produce multiple subword segmentation results based on probability.

To ensure the selection of the most suitable result from tokenized subword candidates, we employ the Unigram algorithm to train code-specific tokenizers for each code summarization dataset, aiming to preserve the original semantic information of the data samples to the greatest extent possible.

3 | Methods

Figure 1 shows the flowchart of our approach. Initially, we extract code snippets and corresponding summaries from source code files. Subsequently, these codes are parsed into Abstract Syntax Trees (ASTs), enabling the extraction of key information

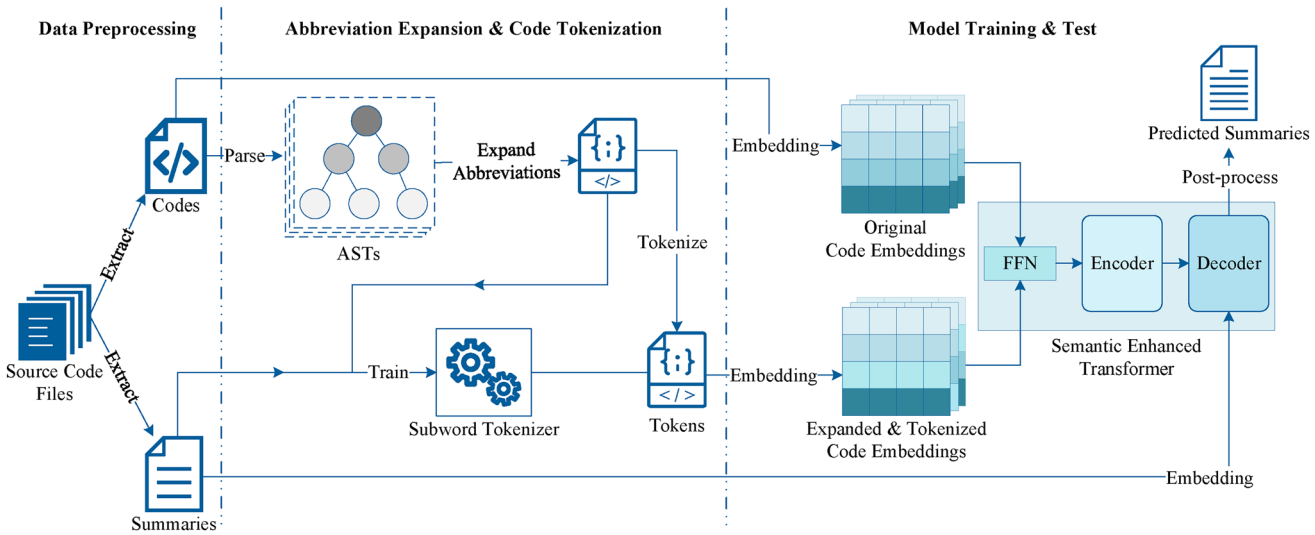


FIGURE 1 | Flowchart of our approach.

to assist in expanding nested abbreviations within the code. Following this, we utilise the subword segmentation algorithm to train a tokenizer based on words in the new corpus, which comprises sequences of expanded codes and original summaries. Ultimately, tokenized codes and corresponding summaries are used to train a code summarization model.

The method of fusing the embeddings of both source and modified code using a Feature Fusion Network (FFN) is not strictly necessary, as the expanded and tokenized code can be directly used to train a new code summarization model. However, the technique of feature fusion is significant and has been employed in many automatic code summarization approaches. To better leverage the critical semantic information introduced by methods proposed in this article, we further present a new encoder-decoder-based model, namely the SETCS.

3.1 | Context-Based Code Abbreviation Expansion

Figure 2 illustrates the AST corresponding to the code snippet shown in Table 1, while only part of the key attributes and values are displayed for brevity. Non-terminal nodes in the AST represent various attributes, such as parameters, name, and body of the method declaration. Terminal nodes represent values of related attributes, such as identifiers and keywords contained in the code snippet. In the process of parsing source codes into ASTs, four sets of auxiliary information for each code snippet are extracted and stored:

1. Method ID, project ID, method name, called methods, and passed actual parameters.
2. Formal parameters as well as their types, split parameters, and involved abbreviations.
3. Parameters and their types within the method, split parameters, and involved abbreviations.
4. Variables and their types within the method, split variables, and involved abbreviation.

The method name, actual parameters passed in the called methods, and types of formal parameters are used as reference words for expanding abbreviations involved in split formal parameters. Types of parameters and variables are used as the reference words to expand abbreviations involved in split parameters and variables, respectively. The method ID and project ID are used to locate specific methods in the project when expanding abbreviations. For example, in the illustrated AST, the method name of 'load' (extracted name of the method declaration) and formal parameter's type of 'URL' (extracted reference type of the formal parameter) will be used to expand the abbreviation 'u' in the formal parameter; the variable's type of 'FileCacheSeekableStream' (extracted reference type of the local variable declaration) will be utilised to expand abbreviation of 's' in the variable name.

Note that before identifying abbreviations, corresponding identifiers are split using a traditional predefined split function, which splits identifiers based on naming conventions and converts all split words to lowercase. For example, either 'fileName' or 'file_name' would be split into 'file' and 'name'. In addition, all abbreviation expansion algorithms utilise the function to split reference words. Code abbreviation expansion algorithms are shown as follows:

For longer identifiers that are composed of multiple words, developers often select the initial characters of each word as an abbreviation during programming, and such an abbreviation form is termed as acronym. For example, the identifier of 'timePerFrame' may be abbreviated as 'tpf'. When expanding such kind

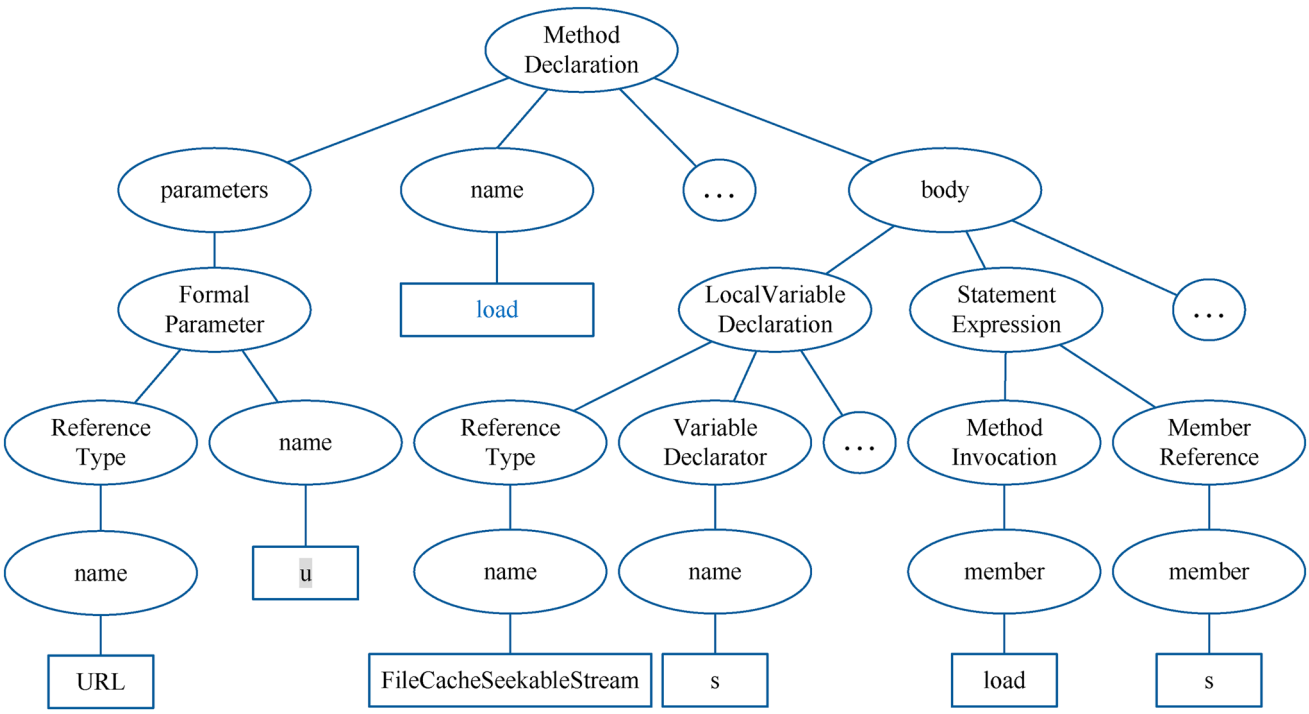


FIGURE 2 | Illustration of AST.

of abbreviations, the initial characters of each split word are extracted (lines 1–4) and used to compare with the abbreviation, if the abbreviation and combinations of initial characters are the same, the split word will be considered as the expansion candidate of the abbreviation (lines 5–6). It should be noted that the abbreviation may be equivalent to the initial characters of part split words, such as the situation of abbreviating 'setKeystoreFilename' as 'kf', instead of 'skf'. To leverage method names to expand abbreviations present in formal parameters, this case is also considered during the implementation of the algorithm (Algorithm 1).

Prefix abbreviations are commonly found in identifier definition statements, among which 'String str' is the most typical example. The idea of expanding these abbreviations is to find split words that begin with the abbreviation but are not exactly equivalent to it in the process of splitting the reference word and add them to the set of expansion candidates (lines 1–5). Since basic forms of words are usually short, the shortest one is selected as the final expansion candidate of the abbreviation if multiple candidate expansions are obtained (lines 6–10) (Algorithm 2).

The term of 'idx' is a common dropped letters abbreviation, and 'index' is usually its full name. In the process of splitting the reference word, every split word and each character in the abbreviation are compared sequentially, and if a split word (lines 8–9) contains all the characters of the abbreviation, it is appended to the list of expansion candidates. Then the next split word and each character in the abbreviation are compared again until all split words are traversed. The code logic of lines 11–15 is the same as lines 6–10 in Algorithm 2, where the shortest word in the list of expansion candidates is finally selected, while the purpose of which is to avoid introducing extraneous long words that contain abbreviations. Considering that this algorithm is prone to generate erroneous expansion results for single-letter abbreviations, in practice, the length of input abbreviations is limited to more than 1 (Algorithm 3).

ALGORITHM 1 | Acronym expansion.

```
   Input: abbreviation, reference word
   Output: expansion candidate
1  words ← split(reference)
2  initialCharacters = ''
3  for each word in words do
4  |  initialCharacters = initialCharacters + word[0]
5  if abbreviation equals initialCharacters then
6  |  expansion = initialCharacters
7  return expansion
```

ALGORITHM 2 | Prefix abbreviation expansion.

```
   Input: abbreviation, reference word
   Output: expansion candidates
1  words ← split(reference)
2  candidates = []
3  for each word in words do
4  |  if word starts with and not equals abbreviation then
5  |  |  candidates ← candidates ∪ word
6  if len(candidates) > 0 then
7  |  expansion ← candidates[0]
8  |  for each candidate in candidates do
9  |  |  if len(candidate) < len(expansion) then
10 |  |  |  expansion = candidate
11 return expansion
```

ALGORITHM 3 | Dropped-letters abbreviation expansion.

```
Input: abbreviation, reference word
Output: expansion candidates
words ← split(reference)
candidates = []
for each word in words do
    i ← 0
    for j in range(len(word)) do
        if abbreviation[i] equals word[j] then
            i ← i + 1
            if i equals len(abbreviation) then
                candidates ← candidates ∪ word
                break
if len(candidates) > 0 then
    expansion ← candidates[0]
    for each candidate in candidates do
        if len(candidate) < len(expansion) then
            expansion = candidate
return expansion
```

In summary, context information such as parameter types, method names, and actual parameters passed into called methods are utilised as reference words for formal parameter abbreviations in a specific method. Subsequently, the most frequent expansion candidate obtained by the three expansion algorithms is selected as the final choice. For abbreviations contained in parameters and variables within the method body, expansion candidates obtained by acronym and prefix abbreviation expansion algorithms are favoured based on their types. While the overall approaches of the three abbreviation expansion algorithms described above are generally consistent with that of Jiang et al. (2020), the main distinction arises from the original study's focus on expanding abbreviations in parameters and evaluation on 9 open-source projects, compared to our need to expand abbreviations nested in both parameters and variables within datasets containing approximately 4.7 k and 0.5 M projects, respectively. Consequently, in our implementation, we encounter more specific scenarios, such as the discovery of 'setKeystoreFilename' during expanding acronyms, and address these issues to balance precision and recall as effectively as possible. More detailed information can be found in our open-source code.

3.2 | Unigram-Based Subword Segmentation

As shown in Figure 1, after obtaining the expanded code snippets, the new corpus's word collection obtained by the traditional split method is deemed as the initial vocabulary; then a code-specific tokenizer is trained by leveraging the unigram subword segmentation algorithm, which based on the unigram language model; finally, the tokenizer is utilised to tokenize all code-summary pairs into more fine-grained subword sequences before they are fed into the code summarization model.

In the context of automatic code summarization, the unigram subword segmentation algorithm aims to segment code sequences and their corresponding summary sequences into subword units, considering subword-level probabilities. The algorithm follows the steps outlined below:

For a pair of code sequence C and summary sequence S in the new corpus D, let $c = (c_1, \ldots, c_x)$ and $s = (s_1, \ldots, s_y)$ correspond to subword sequences for C and S, respectively. The unigram language model assumes that each subword appears independently, so the occurrence probability of a subwords sequence $c = (c_1, \ldots, c_x)$ can be formalised as product of each subword's occurrence probability:

$$P(c) = \prod_{i=1}^{x} p(c_i)$$
$$\forall_i c_i \in \mathcal{V}, \sum_{i=1}^{|\mathcal{V}|} p(c_i) = 1 \tag{1}$$

where V is the pre-determined initial vocabulary. Let $T(C)$ represent the set of segmentation candidates for C, then the most likely segmentation sequence can be formulated as:

$$c^* = \underset{c \in T(C)}{\operatorname{argmax}} P(c) \tag{2}$$

After that, the expectation maximisation (EM) algorithm is used to maximise the following marginal likelihood $\mathcal{L}$, and estimate

the occurrence probability of subwords in the form of hidden variables $P(c_i)$.

$$\mathscr{L} = \sum_{j=1}^{|D|} \log\left(P\left(C^{(j)}\right)\right) = \sum_{j=1}^{|D|} \log\left(\sum_{c \in \mathcal{T}(c^j)} P(c)\right) \tag{3}$$

where $\mathcal{D} = \left\{\left\langle C^{(j)}, S^{(j)}\right\rangle\right\}_{j=1}^{|D|} = \left\{\left\langle c^{(j)}, s^{(j)}\right\rangle\right\}_{j=1}^{|D|}$ represents the new code-summary corpus, and $|D|$ is the size of the corpus.

Finally, following steps are iterated over until the desired vocabulary size $|V|$ is reached:

1. Maintain a fixed vocabulary and use the EM algorithm to optimise $P(c)$.
2. Calculate the loss ℓ_i for each subword c_i, where ℓ_i represents the change in the loss value of $\mathscr{L}$ when ℓ_i is removed from the current vocabulary.
3. Sort all subwords according to ℓ_i and retain the top $n\%$ of subwords.

Note that high-frequency basic words, including single characters and keywords in the programming language, should always be kept in the vocabulary to prevent issues of OOV and over-fine-grained tokenisation, so that critical semantic information in initial sequences can be preserved as much as possible. Finally, a vocabulary that contains subword tokens and their corresponding occurrence probabilities is obtained, and the trained tokenizer utilises Equation (2) to generate the most likely subword sequences c^* and s^* for each pair of C and S based on the final vocabulary.

In practice, the tokenizer is used to tokenize each word in the target sequence sequentially. If a word can be represented by a combination of multiple tokens in the vocabulary, it will be tokenized based on (1) whether the tokenized subword is included in the pre-split word set of code and the sequence in the same method, which gives subword candidates occurring in somewhere of the same method higher priority; (2) the number of tokens after tokenisation, which means shorter subword candidates would be a priority. Eventually, the semantically preserved and/or shortest tokenisation result from the Top-k subword combination candidates will be selected.

By leveraging the vocabulary that includes characters, common subwords, and words, rare words in almost all codes and summaries can be properly tokenized. Most importantly, the fine-grained and semantically preserved subword representation exposes more meaningful information, which is expected to further improve the performance of the code summarization model.

3.3 | Semantic Enhanced Transformer for Code Summarization

Figure 3 shows the framework of SETCS. Similar to most code summarization models, SETCS utilises the encoder-decoder framework, and adopts the Transformer model as backbone. Both encoder and decoder of the model are stacked with N identical layers, and each layer contains several sublayers. Specially, SETCS takes both original and modified code sequences as input of the encoder, while only original summary sequences are fed into the decoder. Besides, the relative positional encoding (Shaw, Uszkoreit, and Vaswani 2018), instead of Transformer's default positional encoding mechanism, is used to leverage representations of relative positions between elements of input sequences effectively.

Given that the modified code sequence is typically longer than the original one, after obtaining the optimal subword sequence c^* of a source code sequence C, we insert special <pad> tokens into the original code sequence, making its length equal to the modified code sequence. This operation aims to align these two sequences precisely and avoid improper concatenation in the latter. Using a predefined embedder class, these sequences are converted into dense vector representations that capture the lexical information of both the original and modified code. Following that, embeddings of C and c^* are generated and concatenated together:

$$e_C = \text{concat}\left(e_{c'}, e_{c*}\right) \tag{4}$$

where $e_{c'}$ and e_{c*} represent embedding of original and tokenized code sequence separately. The word embeddings shown in three

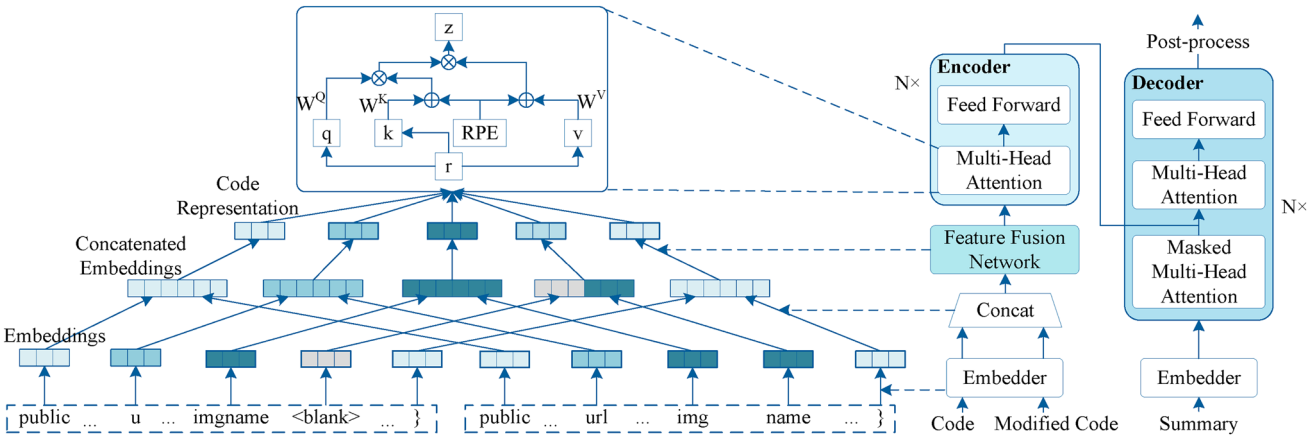


FIGURE 3 | Framework of SETCS.

are actually stacked together, similar to the representations shown in Figure 1. However, we have separated them for better understanding.

To obtain the code representation r_c that fuses features of both input sequences, the concatenated code embedding e_C is sent into a customised network consisting of a Linear layer and a ReLU activation followed:

$$r_c = \max\left(0, e_C W^e + b^e\right) \tag{5}$$

where W^e and b^e are learnable parameters in the form of matrix and vector, respectively. After that, r_c is fed into the encoder that is composed of a multi-head self-attention sublayer and a feedforward sublayer.

The multi-head self-attention sublayer consists of h heads to keep the model focused on information at different locations in the input representation. Each head performs the self-attention function in parallel and computes an output sequence $z = (z_1, \ldots, z_x)$ for the input representation of code, $r_c = (r_1, \ldots, r_x)$ of x elements:

$$z_i = \sum_{m=1}^{x} \alpha_{mn}\left(r_m W^V + p^V_{mn}\right) \tag{6}$$

where $r_m \in \mathbb{R}^{d_r}$, $z_i \in \mathbb{R}^{d_z}$. The involved weight coefficient α_{mn} can be formulated as:

$$\alpha_{mn} = \frac{\exp\left(e_{mn}\right)}{\sum_{o=1}^{x} \exp\left(e_{mo}\right)} \tag{7}$$

where e_{mn} is computed via a scaled dot-product attention:

$$e_{mn} = \frac{r_m W^Q \left(r_n W^K + p^K_{mn}\right)^T}{\sqrt{d_z}} \tag{8}$$

The parameter matrices W^Q, W^K, and $W^V \in \mathbb{R}^{d_r \times d_z}$ are unique per sublayer and head. The encoding vectors p^V_{mn} and $p^K_{mn} \in \mathbb{R}^{d_z}$ include the relative position information between the input elements r_m and r_n.

Similarly, the outputs of each head are then concatenated together and fed into the feedforward network sublayer. The only difference between our customised feature fusion network and the feedforward layer is that the latter consists of an additional linear transformation:

$$FeedForward(Z) = \max\left(0, ZW^1 + b^1\right)W^2 + b^2 \tag{9}$$

where W^1, W^2, b^1, and b^2 are trainable parameters, and Z represents output of the multi-head self-attention sublayer. Note that each sublayer in the model is followed by a residual connection and layer normalisation, which are omitted from Figure 3 for brevity.

Compared to the encoder, each layer of the decoder contains an additional masked multi-head self-attention sublayer. This sublayer is designed to prevent the model from seeing future information during the prediction of the next word. It achieves this by applying a mask to the part of the summary sequence that comes after the current word to be predicted. This ensures that the model's attention is focused only on the known part of the sequence during the training phase. After passing through the multi-head self-attention sublayer, the token representations are passed through a feedforward sublayer. Each token representation in the target summary sequence is generated sequentially, with each token's generation based on the current encoding state and the outputs generated for the previous tokens. This process allows the model to build up a context for the current prediction. Finally, the output of the decoder is passed through a softmax activation function. This function maps the raw model output to a probability distribution over the possible next tokens, making it possible to select the most likely next token for the summary.

4 | Experimental Setup

4.1 | Datasets

Given the indispensable role of project information in code abbreviation expansion, we exclude the dataset open-sourced by Hu, Li, Xia, Lo, and Jin (2018), even though it is relatively small in scale and has been more widely evaluated, due to its lack of project information. Instead, we conduct experiments using the Funcom dataset (LeClair, Jiang, and McMillan 2019) and the Java portion of the CodeSearchNet corpus (Husain et al. 2020), henceforth referred to as CSN-Java.

The CSN corpus, sourced from the GitHub open-source repository, comprises code snippets and corresponding summary descriptions across six programming languages. Among them, CSN-Java contains approximately 4.7 k samples from nearly 0.5 M projects. The Funcom dataset, originated from the Sourcerer repository open-sourced by Lopes et al. (2010), consists of 2.1 M Java samples from around 29 k projects, as preprocessed by LeClair, Jiang, and McMillan (2019).

Despite the preliminary filtering of these two code summarization datasets, we observed a significant number of low-quality samples. These could negatively impact or inflate the evaluation results of code summarization models (LeClair, Jiang, and McMillan 2019; Allamanis 2019). As a result, we remove samples that meet any of the following conditions during the extraction of code and summaries from source code files.

1. The code cannot be parsed, or it is not recognised as a method declaration. This step is necessary for the process of code abbreviation expansion.
2. The length of the split code or summary sequence is less than three. Most of these samples contain fragmented information with very limited meaning.
3. The summary is identified as Self-Admitted Technical Debt (SATD). These summaries are consisted of meaningless contents such as TODO/Fixme.
4. The summary includes auto-generated phrases such as ‘auto generated’ or ‘generated by’, which is usually

associated with auto-generated code that need to be removed according to previous studies (LeClair, Jiang, and McMillan 2019; Hu et al. 2020; Husain et al. 2020).

5. The contents of the summary are identical, occur more than 300 times, and do not relate to the actual functionality of the corresponding code.
6. The code is an exact or near duplicate, which may inflate model evaluation results (Allamanis 2019).

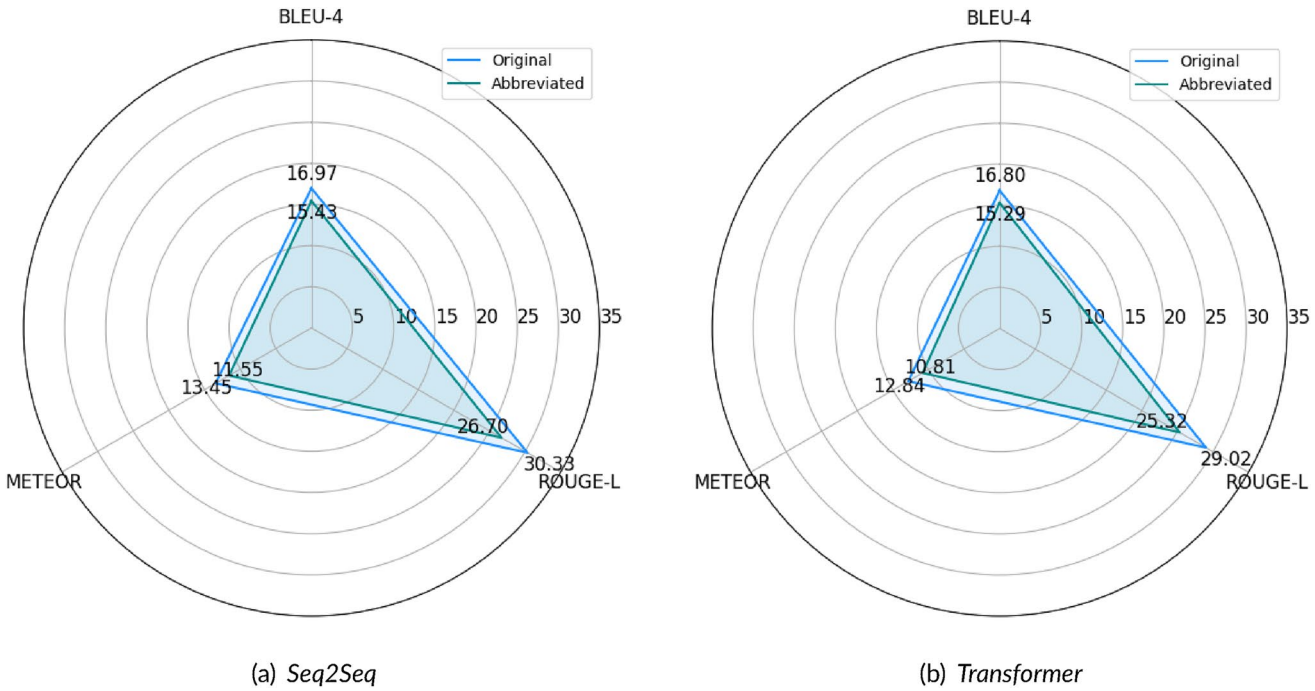
In the process of dataset filtering, we use the javalang[1] library to parse the code, the SATD detection tool[2] to identify SATDs, and the Near-Duplicate Code Detector[3] to detect cloned codes, respectively. Refer to LeClair, Jiang, and McMillan (2019), both filtered datasets are partitioned into training, validation, and test set by project, maintaining a ratio of 90:5:5. The third column in Table 2 shows the number of code-summary pairs in two filtered datasets. For clarify, these filtered dataset are referred to as the original dataset used in subsequent experiments.

4.2 | Exploratory Experiments

To investigate the potential adverse effects of abbreviations in code on code summarization models, we conduct exploratory experiments by actively augmented the prevalence of abbreviations in the code. We then observe the resultant changes in model performance on both the original and abbreviated datasets. This allowed us to assess the impact of abbreviation-rich code on the effectiveness of code summarization.

TABLE 2 | Statistics of code-summary pairs, parsed identifiers, split identifiers, identified abbreviations, and expanded abbreviations in two datasets.

Dataset	Partition	Code-summary pairs	Parsed identifiers	Split identifiers	Identified abbreviations	Expanded abbreviations
CSN-Java	Train	368,224	12,996,895	22,424,406	3,620,121	602,310
	Valid	16,846	602,239	1,028,123	187,108	30,129
	Test	16,746	595,283	994,543	137,407	26,048
	Total	401,816	14,194,417	24,447,072	3,944,636	658,487
Funcom	Train	1,371,687	16,896,844	28,956,036	4,368,006	931,854
	Valid	86,165	1,077,001	1,850,750	271,223	59,134
	Test	81,642	1,022,339	1,753,158	259,266	61,124
	Total	1,539,494	18,996,184	32,559,944	4,898,495	1,052,112



(a) *Seq2Seq* (b) *Transformer*

FIGURE 4 | Radar map showing performance degradation of models on original and abbreviated CSN-Java datasets.

Specifically, we crawl open-source Java projects with over 20 stars from GitHub and extract parameters, variables, and their corresponding types from the parsed code. If a specific parameter or variable was identified as an acronym, prefix, or dropped-letters abbreviation of its corresponding type, the parameter or variable and it's type will be added to the expansion-abbreviation library. Ultimately, we obtain a library containing 5956 pairs of expansion and abbreviation. Using this library, we replace identifiers in the CSN-Java dataset that match the expansions with corresponding abbreviations. To minimise the influence of manually introduced abbreviations on the original semantic meaning of code in the dataset, identical identifiers in a code snippet will be replaced with the same predetermined abbreviation. If an identifier can be replaced with multiple different abbreviations, it will be randomly replaced with an abbreviation that does not duplicate existing identifiers in the current code snippet. Subsequently, we train and test two representative code summarization models, Seq2Seq and Transformer, on both the original and abbreviated datasets, and evaluate the models' performance using common evaluation metrics, namely BLEU-4, METEOR, and ROUGE-L. Detailed information regarding the models and evaluation metrics used in the experiments will be provided in Sections 4.4 and 4.5, respectively.

The changes in evaluation metrics for Seq2Seq and Transformer models on the original and abbreviated datasets are depicted in Figure 4. It is evident from the results that increasing the proportion of abbreviations in the dataset negatively impacts the performance of code summarization models. Both models exhibit a decrease of approximately 1.5, 2, and 3.5 points in the BLEU-4, METEOR, and ROUGE-L metrics, respectively, when more abbreviations are introduced into the dataset. These findings suggest the potential for enhancing the performance of code summarization models by minimising the presence of abbreviations in the datasets.

4.3 | Preliminary Experiments

Studies in the code abbreviation expansion domain define a word as an abbreviation if it if not found in an English dictionary (Jiang et al. 2020; Di Martino, Maggio, and Corazza 2012). We employ the PyEnchant[4] library to identify abbreviations from split identifiers. Specifically, words not included in the 'en_US' dictionary of the enchant library are considered abbreviations. Additionally, single letters, with the exception of 'a', are also treated as abbreviations to complement the identification results. The last four columns in Table 2 show the number of parsed identifiers, split identifiers, identified abbreviations, and expanded abbreviations in two datasets respectively. It can be found that more than 25% of identifiers contain abbreviations. After leveraging abbreviation expansion algorithms, about 21% of the abbreviations in the Funcom data set are expanded, while this percentage in CSN-Java is approximately 17%. We attribute the difference to: (1) Compared with the Fucnom dataset, each code snippet in CSN-Java contains a larger number of abbreviations on average (about 3–10), indicating that there is substantial room for exploration in abbreviation expansion for this dataset. (2) The projects in CSN-Java contain partial methods, which means that only a fraction of the full method implementation is present in the dataset. Consequently, the amount of context information available for expanding abbreviations is inherently limited.

Given that the precision of abbreviation expansion directly or indirectly affects the performance of code summarization models in subsequent experiments, we randomly sampled 1000 expanded abbreviations from two datasets for manual evaluation. Specifically, we found two cases of expansion errors:

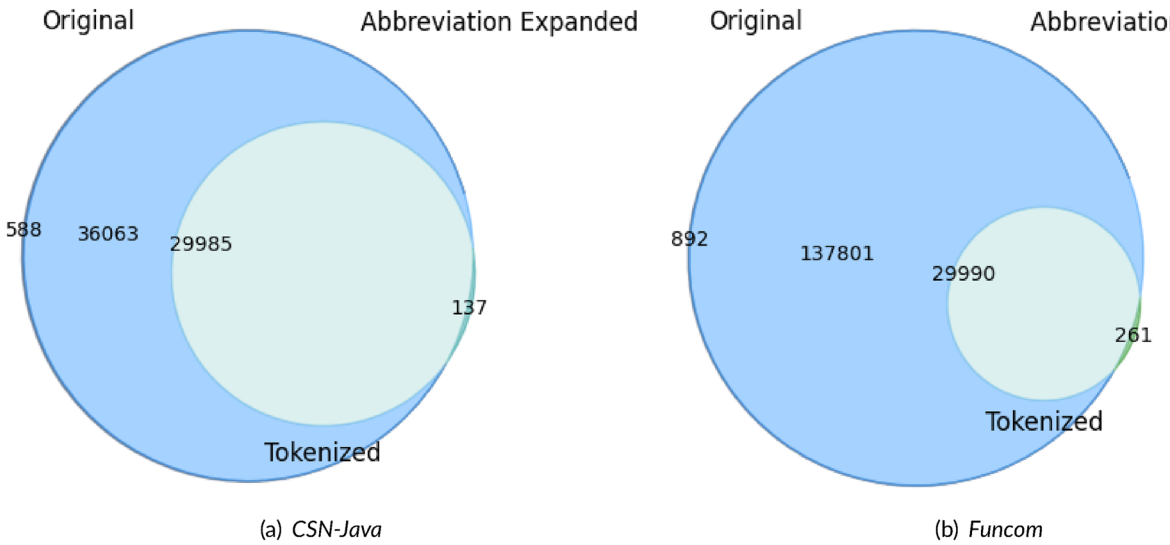
1. The term abbreviation is contained within the reference word. For example, 'uri' typically refers to the Uniform Resource Identifier. However, due to the presence of 'Security' in the method name 'getSecurityProtocol', the split 'security', as a reference word, was incorrectly interpreted by the Dropped Letters expansion algorithm as the full name of the abbreviated parameter 'uri'.
2. There are multiple expansion candidates in the reference words. For example, when expanding the abbreviated parameter 'p' using the Acronym expansion algorithm, the 'player' from the parameter type 'PlayerPreferences' was initially identified and determined as its expansion. However, based on the context of the function, the expansion corresponding to abbreviation 'p' should be 'preferences', or more precisely, 'player preferences'.

Overall, heuristic-based acronym expansion algorithms cannot achieve perfect precision and are susceptible to the influence of developer abbreviation habits. The two types of expansion errors mentioned above are unavoidable. Fortunately, both cases are rare (one case for each type found in 1000 manually evaluated samples), and in most times, developers use abbreviations that include the initials of all words in parameter or variable types, which are correctly expanded by the utilised algorithms.

During the training of the tokenizer, we set the expected vocabulary size to 30 k, and retain the top 90% subwords at the end of each iteration. In the process of dataset tokenisation, the final tokenisation result is selected from the Top-9 candidate subword combinations for both datasets. More detailed information about determining the 'k' value will be discussed in Section 5.3.

To prevent data leakage, we construct the initial vocabulary using only split code and summary words from the training and validation sets. When tokenizing words in the test set, we select the final tokenisation results by referring only to the split words from the code.

The distributions of shared and unique tokens for original, abbreviation expanded, and tokenized datasets are shown in Figure 5. The outermost navy blue, adjacent dodger blue, and innermost light cyan circles in the venn diagram represent the unique token distribution of the original, abbreviation expanded, and ULM tokenized datasets, respectively. Numbers in the middle represent the quantity of shared tokens of datasets in different status, where we can find that trained tokenizers



(a) *CSN-Java* (b) *Funcom*

FIGURE 5 | Venn diagram showing statistics of shared and unique tokens for original, abbreviation expanded, and tokenized results of two datasets.

effectively limit vocabulary size of tokenized datasets to less than 30 K; numbers on the leftmost part (coloured in navy blue) and rightmost part (coloured in light cyan) indicate the quantity of unique tokens in the original and ULM tokenized datasets respectively. The unique tokens in both ULM tokenized datasets are subsequences of longer numerical sequences. In addition, basic numeric tokens of 0–9 are also included in vocabularies to guarantee all fresh numbers appearing in the test set can be properly tokenized via existing numeric tokens. It is worth noting that code abbreviation expansion also reduces the number of unique tokens in original datasets to some extent. Even in small quantities, these eliminated tokens are usually relatively important abbreviated identifiers as mentioned earlier. If we don't expand these abbreviations, they will be generally identified as the <unk> symbols due to the low occurrence frequency. However, they will likely be tokenized into longer character sequences by trained tokenizers after introducing the subword segmentation algorithm. Both circumstances may result in the loss of critical information. Therefore, we believe that it is necessary to perform abbreviation expansion before training and adopting the tokenizer. Results of ablation experiments (Section 5.2) and example analysis (Section 5.4) on the Transformer baseline will demonstrate the effectiveness of abbreviation expansion as well as its usefulness in combining with the introduced Unigram-based subword segmentation method.

4.4 | Baseline Models

To verify the effectiveness of our proposed methods, we conduct experiments using four representative code summarization models:

4.4.1 | Seq2Seq

A classical open-sourced NMT framework (Klein et al. 2017), based on recurrent neural network (RNN) and equipped with an attention mechanism. Specifically, this baseline uses LSTM (Hochreiter and Schmidhuber 1997) to generate summaries for given code snippets and is adopted by Rencos (Zhang et al. 2020), Re2Com (Wei et al. 2021), MLCS (Zhou et al. 2023) as model backbone.

4.4.2 | Transformer

The vanilla Transformer (Vaswani et al. 2017) model incorporated with relative positional encoding mechanism. Specifically, it has been employed by the method of neural code summarization (NCS) (Ahmad et al. 2020), API2Com (Shahbazi, Sharma, and Fard 2021), SiT (Wu, Zhao, and Zhang 2021), AST-Trans (Tang et al. 2022) and the framework of SETCS presented in this article.

4.4.3 | Networked Control System

NCS (Ahmad et al. 2020): An enhanced Transformer designed for code summarization that utilises both relative positional encoding and copying mechanism (See, Liu, and Manning 2017) for the first time. The copying mechanism enables the Transformer to generate words from the vocabulary and copy from the input source code.

4.4.4 | MLCS

MLCS (Zhou et al. 2023): A state-of-the-art code summarization framework based on meta-learning and code retrieval. By optimising a unique code summarizer for each target code snippet knowledge learned from the retrieved similar examples, MLCS was able to outperform typical deep-learning models and retrieval-based neural models.

It is worth noting that since both code summarization datasets came from open-source communities, pre-trained code models typically utilise larger-scale open-source corpora for

TABLE 3 | Experimental results of SETCS and baselines on two datasets.

	CSN-Java				Funcom			
Model	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**
Seq2Seq	16.87	13.37	30.24	83.62	25.79	17.44	38.58	85.71
Transformer	16.65	12.76	28.92	83.55	25.11	17.31	37.60	84.06
MLCS	18.17	12.71	30.66	84.64	27.15	18.34	40.34	86.91
NCS	**18.22**	13.41	**31.72**	**85.86**	**27.81**	18.82	41.07	87.80
SETCS	18.14	**13.96**	31.66	85.78	**27.81**	**19.62**	**41.72**	**88.28**

Note: The best performances are highlighted in bold.

pre-training, these models should have encountered test samples from the datasets used in our study during the pre-training stage. Therefore, we excluded these models from the baselines to avoid threats of pre-training technique and data leakage to the internal validity of this study.

Referring to prior works (Ahmad et al. 2020; Lin et al. 2021; Zhou et al. 2023; Wei et al. 2019), we limit the maximum input and output lengths for all models to 150 and 30, correspondingly. Meanwhile, we set the batch size, vocabulary size, maximum training epochs, and beam size to 64, 30 K, 30, and 4, respectively. The best model for code summarization is determined based on the BLEU scores from the validation set, and the training process will be halted if there is no enhancement in the BLEU score over 10 successive epochs. All experiments are conducted on a Linux server, which is equipped with a NVIDIA Tesla P40 GPU. The duration of experiments executed on the CSN-Java dataset is less than a day, while those performed on the Funcom dataset typically require approximately 3 days.

4.5 | Evaluation Metrics

The commonly adopted evaluation metrics, BLEU (Papineni et al. 2002), METEOR (Banerjee and Lavie 2005), and ROUGE (Lin 2004), are predicated on the same underlying scenario. Specifically, for each candidate text, which is the prediction result generated by the trained model, there exists a corresponding reference text within the dataset, typically a reference summary authored by the developer. The computation of these evaluation metrics are fundamentally based on precision and recall scores:

$$P_n = \frac{\text{gram}_n(\text{pred}, \text{ref})}{\text{gram}_n(\text{pred})}, R_n = \frac{\text{gram}_n(\text{pred}, \text{ref})}{\text{gram}_n(\text{pred})} \tag{10}$$

where pred, ref, and $gram_n$ refers to the candidate text, reference text, and the overlapping n-grams, respectively.

The BLEU metric highlights precision, which calculates the geometric average of $gram_n$ matches between pred and ref.:

$$\text{BLEU} = \sigma \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log P_n\right) \tag{11}$$

The classical BLEU-4 is calculated by $gram_4$.

The METEOR metric further considers recall, word form, and synonym matching, which creates unigram alignment between pred and ref., while longer $gram_n$ alignment is prioritised in this stage.

$$\text{METEOR} = \sigma \cdot \frac{P_n R_n}{(1-\alpha)R_n + \alpha P_n} \tag{12}$$

where α is the default parameter used for evaluation.

Note that the penalty factor σ differs in different evaluation metrics. The ROUGE metric calculates $gram_n$ between pred and ref. The calculation formula can be expressed as:

$$\text{ROUGE} = \frac{2P_n R_n}{R_n + P_n} \tag{13}$$

The widely used ROUGE-L is calculated based on the longest common sequence.

However, the above-mentioned metrics primarily focus on evaluating textual similarity between candidate and reference texts, which may penalise semantically equivalent texts that differ in wording. To complement these metrics and capture the extent to which the candidate text aligns with the semantics of the corresponding code snippet, we also adopt the newly proposed SIDE metric (Mastropaolo et al. 2024), which has been shown to align well with human assessment. This metric measures the cosine similarity between embeddings of the candidate text and the corresponding code sequence:

$$\text{SIDE} = \cos\left(e_{pred}, e_C\right) \tag{14}$$

where e refers to embedding generated by a fine-tuned MPNet (Song et al. 2020) model via contrastive learning.

In all subsequent experiments, we employ the BLEU-4, METEOR, ROUGE-L and SIDE metrics to evaluate the quality of the summaries generated by the code summarization models, with higher metric scores representing better quality of generated summaries. For fair comparison, model predictions as well as ground-truth references before and after tokenisation are used for calculation, and the mean score is deemed as the final result for each evaluation metric.

5 | Analysis of Experimental Results

For simplicity, this section adopts CAE and ULM to represent Code Abbreviation Expansion and ULM-based subword segmentation, respectively. In addition, best results of each metric in tables are boldfaced.

5.1 | Method Validation

Experimental results of SETCS compared with baselines and improvements of the baselines after adopting CAE and ULM on two datasets are shown in Tables 3 and 4 respectively.

As shown in Table 3, compared to the Transformer baseline, the proposed SETCS, which further harnesses the critical semantic information provided by both CAE and ULM, yields an improvement of over 2 absolute points across almost all evaluation metrics on both the CSN-Java and Funcom datasets. As suggested by Roy, Fakhoury, and Arnaoudova (2021), this assures systematic enhancements in summarization quality, implying that our proposed methods, in conjunction with the feature fusion approach, could be effectively employed in other code summarization models that utilise a similar framework to SETCS. Notably, the NCS model, despite being proposed earlier, still outperforms the state-of-the-art MLCS and other baseline models that merely leverage code-related semantic information on both datasets. Besides, the improvement of SETCS over NCS is less significant, underscoring the potent potential of the copying mechanism. Nonetheless, the primary focus of this study is to validate the effectiveness and applicability of CAE and ULM on existing code summarization models, rather than proposing a new state-of-the-art model. More importantly, SETCS could serve as a robust baseline or backbone for future studies on two well-curated datasets.

Experimental results in Table 4 demonstrate that the performance of all code summarization models improves with the adoption of our proposed methods. Specifically, the following conclusions can be drawn:

1. Compared to the smaller CSN-Java dataset, the overall performance improvement of all baseline models on the Funcom dataset is more significant. Taking the prevailing Transformer model as an example, after adopting CAE and ULM, it can achieve score improvements of 7.3%, 6.9%, 6.7%, and 3.2% in terms of BLEU-4, METEOR, ROUGE-L, and SIDE, respectively. More significantly, collaboratively utilising both methods could yield 10.0% performance gain for Transformer regarding the METEOR metric on the CSN-Java dataset, which enables the baseline comparable to SETCS and the improved NCS.
2. In comparison to the other three metrics, the majority of models exhibit relatively larger absolute score gains with respect to the ROUGE-L metric on both datasets. We attribute this phenomenon to the extension of the reference summary by ULM, coupled with the more granular subword representation. This enables the model to capture more semantic information and contributes to the observed significant improvement.
3. Overall, the NCS model exhibits the least performance improvement following the adoption of the proposed methods. This outcome is reasonable given that the multiple identical expansion results introduced by CAE could potentially interfere with the copying mechanism employed by NCS. Furthermore, both methods, particularly ULM, might increase the code length. Any content that exceeds the maximum code length limitation is truncated during the stages of model training and inference, which could lead to the loss of crucial information.

TABLE 4 | Improvements of baselines after adopting both CAE and ULM on two datasets.

	CSN-Java				Funcom			
Model	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**
Seq2Seq	16.87	13.37	30.24	83.62	25.79	17.44	38.58	85.71
Seq2Seq w/Both	17.40	13.67	30.72	84.57	26.26	18.35	39.54	86.25
	(+3.1%)	(+2.2%)	(+1.6%)	(+1.1%)	(+1.8%)	(+5.2%)	(+2.5%)	(+0.6%)
Transformer	16.65	12.76	28.92	83.55	25.11	17.31	37.60	84.06
Transformer w/Both	17.60	**14.04**	30.82	84.83	26.95	18.51	40.12	86.72
	(+5.7%)	**(+10.0%)**	**(+6.6%)**	**(+1.5%)**	**(+7.3%)**	**(+6.9%)**	**(+6.7%)**	**(+3.2%)**
MLCS	18.17	12.71	30.66	84.64	27.15	18.34	40.34	86.91
MLCS w/Both	18.45	12.98	31.29	85.35	27.96	18.88	41.29	87.94
	(+1.5%)	(+2.1%)	(+2.1%)	(+0.8%)	(+3.0%)	(+2.9%)	(+2.4%)	(+1.2%)
NCS	18.22	13.41	31.72	85.86	27.81	18.82	41.07	87.80
NCS w/Both	**18.51**	13.99	**32.25**	**85.99**	**28.02**	**19.10**	**41.31**	**88.04**
	(+1.2%)	(+4.3%)	(+1.7%)	(+0.2%)	(+0.7%)	(+1.6%)	(+0.6%)	(+0.3%)

Note: The best performances are highlighted in bold.

TABLE 5 | Ablation experiment results of transformer and SETCS on two datasets.

	CSN-Java				Funcom			
Model	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**	**BLEU-4**	**METEOR**	**ROUGE-L**	**SIDE**
Transformer	16.65	12.76	28.92	83.55	25.11	17.31	37.60	84.06
Transformer w/CAE	17.13	13.38	30.20	84.98	25.47	17.50	38.14	84.31
Transformer w/ ULM	17.42	**14.13**	**30.87**	**85.36**	25.86	17.87	38.79	85.08
Transformer w/Both	**17.60**	14.04	30.82	84.83	**26.95**	**18.51**	**40.12**	**86.72**
SETCS w/o CAE	17.84	13.73	31.19	**85.89**	27.79	**19.66**	41.68	88.15
SETCS w/o ULM	17.96	**14.00**	31.61	85.73	27.71	19.64	41.65	88.14
SETCS	**18.14**	13.96	**31.66**	85.78	**27.81**	19.62	**41.72**	**88.28**

Note: The best performances are highlighted in bold.

5.2 | Ablation Experiments

Table 5 presents the experimental results of the Transformer and SETCS models after incorporating CAE, ULM, and both methods, separately, on two different datasets. The primary distinction between the two sets of ablation experiments lies in the fact that only the datasets are modified in the first set of experiments, whereas in the latter set, modifications are also made to the models. Besides, performance of the Transformer model can be seen as the ablation result of SETCS without the feature fusion network.

In ablation experiments results of the first group, it is clearly that both methods can improve the performance of Transformer to various degrees, among which ULM plays a more important role. The combination of two methods could bring further improvements in terms of almost all textual similarity-based evaluation metrics, where the minor degradation of the METEOR and ROUGE-L metrics on CSN-Java can be neglected as difference of the absolute value is less than 0.1. Notably, on the CSN-Java dataset, for both Transformer and SETCS, the proposed ULM could bring about the best improvement for the semantic similarity-based SIDE metric. In fact, compared with the traditional split method, code summarization models adopting ULM tokenizers have better evaluation results on reference summaries whatever before and after tokenisation. Moreover, the experimental results of 'Transformer w/Both' with 30 k vocabulary on the Funcom dataset are still better than that with a 50 k vocabulary. All these findings further prove that CAE and ULM can effectively introduce and expose more critical semantic information, which plays a key role in improving model's performance. In addition, when testing ULM tokenizers in preliminary experiments, we found that tokenizers trained with a smaller vocabulary will tokenize most nouns in plural forms, resulting in substantial score gains in terms of the ROUGE-L metric and decreased performance on other metrics, which indicates that the granularity of subword segmentation is not the finer the better. Therefore, when training a tokenizer for the code summarization model, factors such as size of the desired vocabulary and length limitations of model's input and output should be comprehensively considered.

For the second group of ablation experiments' results, it's interesting that CAE plays a more significant role in improving performance on the CSN-Java dataset, while ULM plays a more significant role in the Funcom dataset. Each of the two methods significantly boosts the performance of all metrics compared to the Transformer baseline, which indicate the effectiveness of the feature fusion network equipped by SETCS. However, the collaboration of the two methods yield relatively fewer improvements across most evaluation metrics, which contradicts the earlier findings. We speculate that the customised network operated in SETCS is capable of learning more specific transformations but struggles with learning complex patterns when both methods are combined. More specifically, the modification of code snippets introduced by CAE is fixed in most circumstances as its algorithms are predefined to expand abbreviations for parameters or variables in very specific places, while modifications brought by ULM are randomly distributed in different locations of the code. In short, this phenomenon can be attributed to the limitations of the feature fusion strategy employed by SETCS, and more effective approaches are yet to be discovered. Actually, we have explored many other feature fusion strategies but reaped relatively fewer improvements compared to method presented in this article. These tested strategies include concatenating embeddings of both original and modified code sequences from another dimension, concatenating embeddings of original and differences between both code sequences, concatenating both code representations directly, and utilising different customised networks when transforming concatenated embeddings to code representations. Therefore, we leave this challenge for future research. For the purpose of better illustration and broader applicability, experiments in the subsequent sections are conducted on the Transformer baseline.

5.3 | ULM Tuning and Comparison

In order to determine the appropriate 'k' value for the Top-k subword combination candidates, as discussed in Section 3.2, we carry out experiments using the 'Transformer w/ULM' model on CSN-Java, with 'k' values ranging from 1 to 13 and the span set to 2. Additionally, we conduct comparative experiments to further examine the effects of the introduced ULM algorithm against basic subword segmentation algorithms. The choice to perform these experiments on CSN-Java instead of Funcom is primarily driven by considerations of time efficiency.

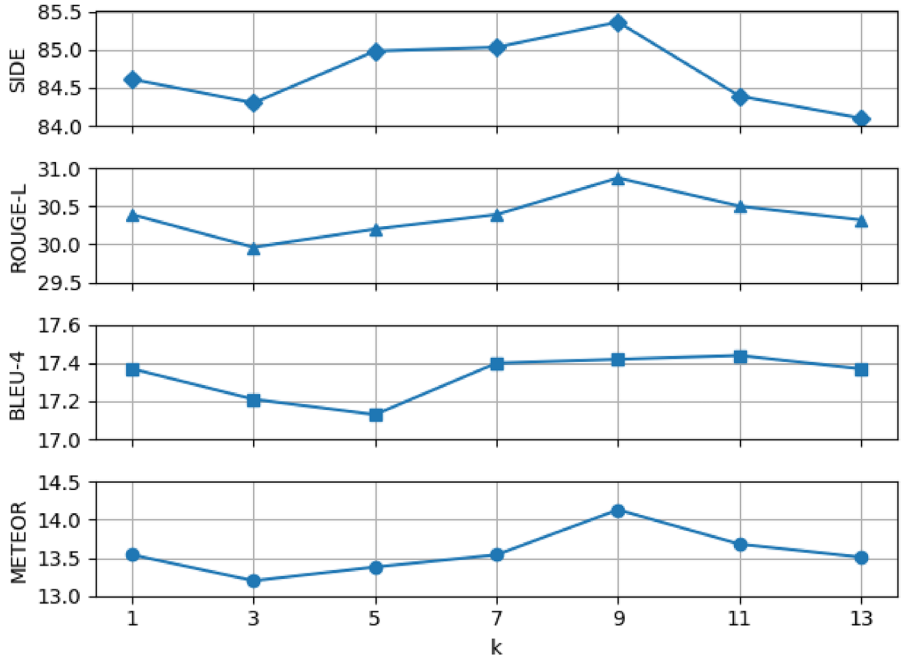


FIGURE 6 | Experimental results of 'Transformer w/ULM' with different *k* values on CSN-Java.

TABLE 6 | Experimental results of Transformer with different subword segmentation algorithms on CSN-Java.

Model	BLEU-4	METEOR	ROUGE-L	SIDE
Transformer	16.65	12.76	28.92	83.55
Transformer w/BPE_Basic	17.21	13.21	30.12	84.59
Transformer w/ULM_Basic	17.15	13.56	30.53	84.62
Transformer w/ULM_Top-9	**17.42**	**14.13**	**30.87**	**85.36**

Note: The best performances are highlighted in bold.

Figure 6 displays the changing curves of four evaluation metrics, where the trend of all curves goes down, up, and then down. Table 6 shows experimental results regarding different subword segmentation algorithms, where results of the Transformer baseline, Transformer with the basic BPE algorithm, Transformer with the basic ULM algorithm, and Transformer with the introduced ULM algorithm are listed from up to down. The essential algorithms operated by both basic and introduced ULM are the same, but the latter further optimised the training and tokenisation procedures of code-specific tokenizers to obtain semantic-preserved results. Besides, the basic WordPiece algorithm is not involved since it is not open-sourced. The 'k' value of the introduced ULM algorithm is set to 9 in all experiments on CSN-Java, as the overall performance of 'Transformer w/ULM' by selecting tokenisation results from Top-9 candidates is proved to be the best.

Overall, all subword segmentation algorithms could significantly improve the performance of Transformer in terms all metrics, which is expected. Specifically, the difference between each pair of subword segmentation algorithms is relatively small in terms of BLEU-4, but differences are obvious when it comes to other three metrics. The tactic of selecting most semantic-preserved tokenisation results from Top-k subword combination candidates introduced in this article is proved to be more effective compared with the direct adoption of basic ULM algorithm, which performance is sightly inferior to the introduced ULM with k set to 1. To sum up, the introduction of subword segmentation algorithms can bring about remarkable improvements for code summarizations models, and the performance could be further upgraded if more code-related semantic information can be preserved.

5.4 | Example Analysis

Table 7 illustrates two examples from Funcom. The last four rows of the table list generated summaries of the Transformer model before and after using the proposed method(s).

For the first code snippet, after using CAE to expand the abbreviation 'msg' nested in the formal parameter 'msgNumber' to 'message' Transformer accurately generates the corresponding summaries for the expanded formal parameter 'message number'. It is interesting that ULM also enables the model to generate the correct summary for the abbreviated formal parameter. We speculate the code summarization model has the potential to generate the corresponding full names for corresponding abbreviations, and semantic information exposed by trained tokenizers convinces the model that the full name of abbreviation 'msg' in the code should be 'message'. In other words, both methods effectively enhanced the ability of language alignment for code summarization models.

When it comes to the second code snippet, although the formal parameter 'feedbacktype' appears multiple times in the code, it

TABLE 7 | Two illustrative examples from the Funcom dataset.

Function ID	27906163	44895355
Code	`public SummaryItem getSummaryItemForMsg(int msgNumber){ return (SummaryItem) summaryItems.get(msgNumber-1); }`	`public void setFeedbacktype(String feedbacktype){ setPropertyString(QTI_RDFS+"feedbacktype)", feedbacktype)); }`
Summary	return the summary item info for a particular message number .	sets the feedbacktype to the given string.
Transformer	returns the summary item for the given msg number .	sets the name of the qti rdfs property.
Transformer w/ CAE	returns the summary item for a given message number .	sets the <unk>property.
Transformer w/ ULM	returns the summary item for the given message number .	sets the feedback type property.
Transformer w/ Both	returns the summary item for the given message number .	sets the feedback type .

is still being identified as <unk> due to its overall low frequency in the dataset, which is reflected in the summary generated by 'Transformer w/CAE'. Instead of generating <unk> with a relative small probability, the vanilla Transformer finally chose 'qti rdfs' as the summary, which appears in the code but has nothing to do with the actual functionality of the code. After tokenizing 'feedbacktype' into 'feedback type' using the Unigram subword algorithm, the model correctly understood its meaning and accurately generated a corresponding summary for it.

In summary, the methods proposed in this article improve the performance of the code summarization model at the semantic level, and the two methods complement each other. Code abbreviation expansion eliminates some rare words. It also avoids the unigram subword algorithm tokenizing them into overlong subwords. The subword algorithm can expose more abbreviation information. If the abbreviation 'img' nested in the identifier 'imgname' contained in the code snippet of Table 1 is accurately tokenized and expanded, code summarization models will be more likely to generate the correct summary 'image' for the code. Therefore, the subword segmentation algorithm also has practical implications for the study of abbreviation expansion, and proposing more advanced techniques to combine the copying mechanism with methods proposed in this article is worthy of further exploration as well.

6 | Conclusion and Future Work

In this article, we propose two methods to enhance the semantic performance of code summarization models. By expanding abbreviations within identifiers, we eliminate the uncertainty of the corresponding semantic information and allow the model to focus more on the identifiers themselves rather than their types. Moreover, by leveraging the Unigram subword segmentation algorithm, we train code-specific tokenizers to tokenize code into more granular subword sequences, which enables the code summarization model to capture more critical information during training and inference stages. Experimental results from three typical code summarization models and the presented SETCS on two datasets demonstrate the effectiveness of our proposed methods.

Future works include:

1. Incorporate advanced feature fusion techniques into SETCS to unlock the full potential of our proposed methods, or employ the framework to verify other automatic code summarization approaches at either the semantic or structural level.
2. Explore further how expanding code abbreviations in different proportions and types impacts the performance of code summarization models, and how the performance is influenced by different subword segmentation algorithms with varying vocabulary sizes.
3. Apply the methods proposed in this article to pre-trained code models and other program understanding or generation tasks, particularly in conjunction with prompt learning (Liu et al. 2023) or meta-learning techniques. This could potentially enhance the efficiency and performance of these models and tasks.

To facilitate future research, we have made datasets used in experiments, as well as the source code of SETCS, publicly available at https://github.com/Hugo-Liang/SETCS.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The authors declare that the data supporting the findings of this study are available within the paper.

Endnotes

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

[2] https://github.com/Tbabm/SATDDetector-Core.

[3] https://github.com/microsoft/near-duplicate-code-detector.

[4] https://pyenchant.github.io/pyenchant.

References

Ahmad, W., S. Chakraborty, B. Ray, and K. W. Chang. 2020. "A Transformer-Based Approach for Source Code Summarization." In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 4998–5007. Stroudsburg, PA: Association for Computational Linguistics.

Ahmad, W., S. Chakraborty, B. Ray, and K. W. Chang. 2021. "Unified Pre-training for Program Understanding and Generation." In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2655–2668. Stroudsburg, PA: Association for Computational Linguistics.

Allamanis, M. 2019. "The Adverse Effects of Code Duplication in Machine Learning Models of Code." In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 143–153. New York, NY: Association for Computing Machinery.

Banerjee, S., and A. Lavie. 2005. "METEOR: An Automatic Metric for MT Evaluation With Improved Correlation With Human Judgments." In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 65–72. Stroudsburg, PA: Association for Computational Linguistics.

Cheng, W., P. Hu, S. Wei, and R. Mo. 2022. "Keyword-Guided Abstractive Code Summarization via Incorporating Structural and Contextual Information." *Information and Software Technology* 150: 106987.

Di Martino, S., V. Maggio, and A. Corazza. 2012. "LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations." In *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, 233–242. Los Alamitos, CA: IEEE Computer Society.

Feng, Z., D. Guo, D. Tang, et al. 2020. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. Stroudsburg, PA: Association for Computational Linguistics.

Gage, P. 1994. "A New Algorithm for Data Compression." *C Users Journal* 12, no. 2: 23–38.

Gao, S., C. Gao, Y. He, et al. 2023. "Code Structure–Guided Transformer for Source Code Summarization." *ACM Transactions on Software Engineering and Methodology* 32, no. 1: 1–32.

Guo, D., S. Ren, S. Lu, et al. 2021. "GraphCodeBERT: Pre-Training Code Representations With Data Flow." In *International Conference on Learning Representations*. OpenReview.net.

Guo, J., J. Liu, X. Liu, and L. Li. 2024. "Summarizing Source Code Through Heterogeneous Feature Fusion and Extraction." *Information Fusion* 103: 102058.

He, H. 2019. "Understanding Source Code Comments at Large-Scale." In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering New York*, 1217–1219. NY, USA: Association for Computing Machinery.

Hochreiter, S., and J. Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9, no. 8: 1735–1780.

Hu, X., G. Li, X. Xia, D. Lo, and Z. Jin. 2018. "Deep Code Comment Generation." In *Proceedings of the 26th Conference on Program Comprehension*, 200–210. New Yorkm NY: Association for Computing Machinery.

Hu, X., G. Li, X. Xia, D. Lo, and Z. Jin. 2020. "Deep Code Comment Generation With Hybrid Lexical and Syntactical Information." *Empirical Software Engineering* 25, no. 3: 2179–2217.

Hu, X., G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin. 2018. "Summarizing Source Code with Transferred API Knowledge." In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2269–2275. Menlo Park, CA: AAAI Press.

Husain, H., H. H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. 2020. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search." arXiv:1909.09436.

Iyer, S., I. Konstas, A. Cheung, and L. Zettlemoyer. 2016. "Summarizing Source Code Using a Neural Attention Model." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2073–2083. Stroudsburg, PA: Association for Computational Linguistics.

Jiang, Y., H. Liu, J. Jin, and L. Zhang. 2022. "Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion." *IEEE Transactions on Software Engineering* 48, no. 2: 519–537.

Jiang, Y., H. Liu, and L. Zhang. 2019. "Semantic Relation Based Expansion of Abbreviations." In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 131–141. New York, NY: Association for Computing Machinery.

Jiang, Y., H. Liu, Y. Zhang, N. Niu, Y. Zhao, and L. Zhang. 2021. "Which Abbreviations Should be Expanded?" In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 578–589. New York, NY: Association for Computing Machinery.

Jiang, Y., H. Liu, J. Zhu, and L. Zhang. 2020. "Automatic and Accurate Expansion of Abbreviations in Parameters." *IEEE Transactions on Software Engineering* 46, no. 7: 732–747.

Klein, G., Y. Kim, Y. Deng, J. Senellart, and A. Rush. 2017. "OpenNMT: Open-Source Toolkit for Neural Machine Translation." In *Proceedings of ACL 2017, System Demonstrations*, 67–72. Stroudsburg, PA: Association for Computational Linguistics.

Kudo, T. 2018. "Subword Regularization: Improving Neural Network Translation Models With Multiple Subword Candidates." In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 66–75. Stroudsburg, PA: Association for Computational Linguistics.

LeClair, A., S. Jiang, and C. McMillan. 2019. "A Neural Model for Generating Natural Language Summaries of Program Subroutines." In *Proceedings of the 41st International Conference on Software Engineering*, 795–806. Piscataway, NJ: IEEE Press.

Li, M., H. Yu, G. Fan, Z. Zhou, and Z. Huang. 2024. "Enhancing Code Summarization With Action Word Prediction." *Neurocomputing* 563: 126777.

Lin, C., Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu. 2021. "Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting." In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 184–195. Los Alamitos, CA: IEEE Computer Society.

Lin, C. Y. 2004. "ROUGE: A Package for Automatic Evaluation of Summaries." In *Text Summarization Branches out*, 74–81. Stroudsburg, PA: Association for Computational Linguistics.

Liu, P., W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. 2023. "Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing." *ACM Computing Surveys* 55, no. 9: 1–35.

Liu, S., Y. Chen, X. Xie, J. K. Siow, and Y. Liu. 2021. "Retrieval-Augmented Generation for Code Summarization via Hybrid GNN." In *International Conference on Learning Representations*. OpenReview.net.

Liu, Y., M. Ott, N. Goyal, et al. 2019. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." arXiv:1907.11692.

Lopes, C., S. Bajracharya, J. Ossher, and P. Baldi. 2010. "UCI Source Code Data Sets."

Mastropaolo, A., M. Ciniselli, M. D. Penta, and G. Bavota. 2024. "Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization." In *In: 2024 IEEE/ACM 46th International Conference on Software Engineering*, 1002–1002. Los Alamitos, CA: IEEE Computer Society.

Moreno, L., and A. Marcus. 2018. "Automatic Software Summarization: The State of the Art." In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 530–531. New York, NY: Association for Computing Machinery.

Newman, C. D., M. J. Decker, R. S. Alsuhaibani, A. Peruma, D. Kaushik, and E. Hill. 2019. "An Empirical Study of Abbreviations and Expansions in Software Artifacts." In *2019 IEEE International Conference on Software Maintenance and Evolution*, 269–279. New York, NY: IEEE.

Niu, C., C. Li, B. Luo, and V. Ng. 2022. "Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code." In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence International Joint Conferences on Artificial Intelligence Organization*, 5546–5555. Vienna, Austria: IJCAI.

Niu, C., C. Li, V. Ng, J. Ge, L. Huang, and B. Luo. 2022. "SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations." In *2022 IEEE/ACM 44th International Conference on Software Engineering*, 1–13. Los Alamitos, CA: IEEE Computer Society.

Papineni, K., S. Roukos, T. Ward, and W. J. Zhu. 2002. "BLEU: A Method for Automatic Evaluation of Machine Translation." In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, 311–318. Stroudsburg, PA: Association for Computational Linguistics.

Rai, S., R. C. Belwal, and A. Gupta. 2022. "A Review on Source Code Documentation." *ACM Transactions on Intelligent Systems and Technology* 13, no. 5: 1–44.

Roy, D., S. Fakhoury, and V. Arnaoudova. 2021. "Reassessing Automatic Evaluation Metrics for Code Summarization Tasks." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1105–1116. New York, NY: Association for Computing Machinery.

See, A., P. J. Liu, and C. 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*, 1073–1083. Stroudsburg, PA: Association for Computational Linguistics.

Sennrich, R., B. Haddow, and A. Birch. 2016. "Neural Machine Translation of Rare Words With Subword Units." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1715–1725. Stroudsburg, PA: Association for Computational Linguistics.

Shahbazi, R., R. Sharma, and F. H. Fard. 2021. "API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations." In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 411–421. Los Alamitos, CA: IEEE Computer Society.

Sharma, R., F. Chen, and F. Fard. 2022. "LAMNER: Code Comment Generation Using Character Language Model and Named Entity Recognition." In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 48–59. New York, NY: Association for Computing Machinery.

Shaw, P., J. Uszkoreit, and A. Vaswani. 2018. "Self-Attention With Relative Position Representations." In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 464–468. Stroudsburg, PA: Association for Computational Linguistics.

Song, K., X. Tan, T. Qin, J. Lu, and T. Y. Liu. 2020. "MPNet: Masked and Permuted Pre-Training for Language Understanding." In *Proceedings of the 34th International Conference on Neural Information Processing Systems*. Red Hook, NY: Curran Associates Inc.

Storey, M. A. 2005. "Theories, Methods and Tools in Program Comprehension: Past, Present and Future." In *13th International Workshop on Program Comprehension*, 181–191. Los Alamitos, CA: IEEE Computer Society.

Tang, Z., X. Shen, C. Li, et al. 2022. "AST-Trans: Code Summarization With Efficient Tree-Structured Attention." In *Proceedings of the 44th International Conference on Software Engineering*, 150–162. New York, NY: Association for Computing Machinery.

Vaswani, A., N. Shazeer, N. Parmar, et al. 2017. "Attention Is all You Need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000–6010. Red Hook, NY: Curran Associates Inc.

Wang, Y., W. Wang, S. Joty, and S. C. H. Hoi. 2021. "CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation." In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 8696. Stroudsburg, PA: Association for Computational Linguistics.

Wei, B., G. Li, X. Xia, Z. Fu, and Z. Jin. 2019. *Code Generation as a Dual Task of Code Summarization*. Red Hook, NY: Curran Associates Inc.

Wei, B., Y. Li, G. Li, X. Xia, and Z. Jin. 2021. "Retrieve and Refine: Exemplar-Based Neural Comment Generation." In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 349–360. New York, NY: Association for Computing Machinery.

Wu, H., H. Zhao, and M. Zhang. 2021. "Code Summarization with Structure-Induced Transformer." In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 1078–1090. Stroudsburg, PA: Association for Computational Linguistics.

Wu, Y., M. Schuster, Z. Chen, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation." arXiv:1609.08144v2.

Xie, R., T. Hu, W. Ye, and S. Zhang. 2023. *Low-Resources Project-Specific Code Summarization. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY: Association for Computing Machinery.

Xie, R., W. Ye, J. Sun, and S. Zhang. 2021. "Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach." In *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 138–148. Los Alamitos, CA: IEEE Computer Society.

Zhang, J., X. Wang, H. Zhang, H. Sun, and X. Liu. 2020. *Retrieval-Based Neural Source Code Summarization*, 1385–1397. New York, NY: Association for Computing Machinery.

Zhang, J., X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. 2019. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree." In *Proceedings of the 41st International Conference on Software Engineering*, 783–794. Piscataway, NJ: IEEE Press.

Zhou, Z., H. Yu, G. Fan, Z. Huang, and K. Yang. 2023. "Towards Retrieval-Based Neural Code Summarization: A Meta-Learning Approach." *IEEE Transactions on Software Engineering* 49, no. 4: 3008–3031.

Zhou, Z., H. Yu, G. Fan, Z. Huang, and X. Yang. 2022. "Summarizing Source Code With Hierarchical Code Representation." *Information and Software Technology* 143: 106761.