ESALE: Enhancing Code-Summary Alignment Learning for Source Code Summarization

Chunrong Fang, Weisong Sun*, Yuchen Chen, Xiao Chen, Zhao Wei, Quanjun Zhang, Yudu You, Bin Luo, Yang Liu, Zhenyu Chen

Abstract—(Source) code summarization aims to automatically generate succinct natural language summaries for given code snippets. Such summaries play a significant role in promoting developers to understand and maintain code. Inspired by neural machine translation, deep learning-based code summarization techniques widely adopt an encoder-decoder framework, where the encoder transforms given code snippets into context vectors, and the decoder decodes context vectors into summaries. Recently, large-scale pre-trained models for source code (e.g., CodeBERT and UniXcoder) are equipped with encoders capable of producing general context vectors and have achieved substantial improvements on the code summarization task. However, although they are usually trained mainly on code-focused tasks and can capture general code features, they still fall short in capturing specific features that need to be summarized. In a nutshell, they fail to learn the alignment between code snippets and summaries (code-summary alignment for short).

In this paper, we propose a novel approach to improve code summarization based on summary-focused tasks. Specifically, we exploit a multi-task learning paradigm to train the encoder on three summary-focused tasks to enhance its ability to learn code-summary alignment, including unidirectional language modeling (ULM), masked language modeling (MLM), and action word prediction (AWP). Unlike pre-trained models that mainly predict masked tokens in code snippets, we design ULM and MLM to predict masked words in summaries. Intuitively, predicting words based on given code snippets would help learn the code-summary alignment. In addition, existing work shows that AWP affects the prediction of the entire summary. Therefore, we further introduce the domain-specific task AWP to enhance the ability of the encoder to learn the alignment between action words and code snippets. We evaluate the effectiveness of our approach, called ESALE, by conducting extensive experiments on four datasets, including two widely used datasets JCSD and PCSD, a cross-project Java dataset CPJD, and a multilingual language dataset CodeSearchNet. Experimental results show that ESALE significantly outperforms state-of-the-art baselines in all three widely used metrics, including BLEU, METEOR, and ROUGE-L. Moreover, the human evaluation proves that the summaries generated by ESALE are more informative and closer to the ground-truth summaries.

Index Terms—Source Code Summarization, Deep Learning, Multi-task Learning

1 Introduction

Code comments play a key role in facilitating code comprehension [1], [2], [3], [4] and software maintenance [5], [6], [7], [8], [9]. For example, prior works [1], [2], [10] show that code comments can help improve code readability. Commenting code has been recognized as a good programming practice [6], [8]. However, writing high-quality code comments is a labor-intensive and time-consuming task [6], [11]. As a result, good comments are often absent, unmatched, and outdated during the software evolution [12]. (Source) code summarization is an active research field [4], [9], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], which aims at designing advanced techniques to support automatic generation of code comments (also called summaries). Given a code snippet (a method or function) by the developer, code summarization can generate summaries describing the functionality of the code snippet.

Existing code summarization techniques can mainly be categorized into *keywords-based methods*, *retrieval-based methods*, and *deep learning-based methods*. *Keywords-based methods* extract critical terms from code snippets to constitute their summaries [13], [24]. Such methods may fail to generate accurate summaries if the source code contains poorly named identifiers or method names [25], [26]. *Retrieval-based methods* first leverage code clone detection techniques to retrieve similar code snippets and then use their corresponding comments to summarize other code snippets. Similar code snippets can be retrieved from existing open-source platforms (e.g., GitHub [25]) or software Q&A sites (e.g., Stack Overflow) [27]. Such methods rely on whether similar code snippets can be retrieved [28] and how similar the code snippets are [26]. In addition, code snippets may contain some information inconsistent with the content in comments of their similar code snippets [29], making retrieval-based methods ineffective in many cases. *Deep learning-based methods* leverage powerful generative models trained on a large-scale code-comment corpus to translate code snippets in programming languages into summaries in natural language [26], [30]. Such methods can model the semantic mapping relations between code snippets and summaries and can generate high-quality summaries [30].

Recently, with the success of the pre-training and fine-

- *Chunrong Fang, Yuchen Chen, Xiao Chen, Quanjun Zhang, Yudu You, Bin Luo and Zhenyu Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China and also with the Software Institute, Nanjing University, Nanjing, Jiangsu 210008, China. E-mail: fangchunrong@nju.edu.cn, yuc.chen@outlook.com, shawnchan@smail.nju.edu.cn, quanjun.zhang@smail.nju.edu.cn, fzuyyd@163.com, luobin@nju.edu.cn, zychen@nju.edu.cn.*
- *Weisong Sun and Yang Liu are with the College of Computing and Data Science, Nanyang Technological University, Singapore. E-mail: weisong.sun@ntu.edu.sg, yangliu@ntu.edu.sg.*
- *Zhao Wei is with Tencent Inc., China. Email: zachwei@tencent.com.*
- **Weisong Sun is corresponding author.*

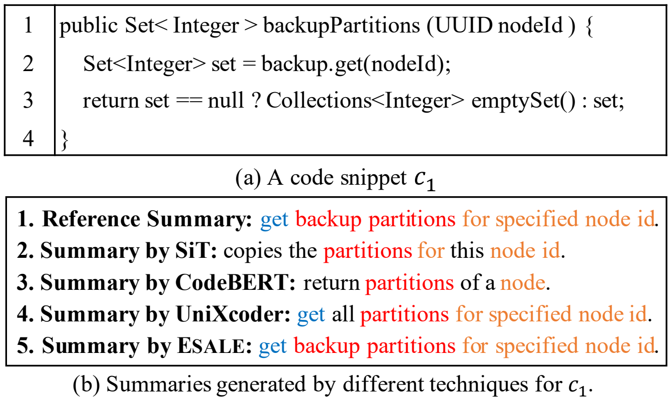
Manuscript received xxx xxx, 2023; revised xxx xxx, 2024.

tuning paradigm in the field of natural language processing (NLP) (e.g., BERT [31] and T5 [32]), many works in software engineering (SE) have introduced this paradigm to boost further code-related tasks, including code summarization (e.g., CodeBERT [33], CodeT5 [34], and UniXcoder [35]). In practice, these works typically pre-train a model with general language modeling tasks, such as masked language modeling (MLM) [31] and unidirectional language modeling (ULM) [36], followed by fine-tuning on code summarization tasks. In *deep learning-based methods*, code summarization is widely considered as the neural machine translation (NMT) task where the source text is the code snippet in programming language and the target text is the summary in natural language [4], [37], [38], [39]. Inspired by NMT, code summarization models widely adopt the encoder-decoder framework. The encoder is responsible for transforming the code snippet into a context vector. The decoder is responsible for decoding the context vector into a natural language summary. Intuitively, context vectors tell the decoder what content needs to be translated, which indicates that the encoder plays a significant role in a code summarization model. Therefore, to achieve high-quality code summarization, a good encoder should be able to produce context vectors that capture the code features that need to be translated by the decoder. However, although the advanced pre-trained encoders have achieved significant progress in producing general vector representations (i.e., context vectors) for given code snippets, they are still insufficient in capturing specific code features that need to be translated. These pre-trained encoders are primarily trained with code-focused tasks that teach them to learn the relationship among the tokens of code snippets rather than the relationship between code snippets and summaries. In other words, these pre-trained encoders are still insufficient in capturing the alignment between code snippets and summaries (i.e., code-summary alignment), detailed in Section 2.

In this paper, we propose a novel approach to improve code summarization. Our approach is built upon large-scale pre-trained code encoders that have been shown to be superior in capturing and representing general code features. To improve the ability of our encoder to learn the code-summary alignment, we exploit the multi-task learning (MTL) paradigm to train it. In the MTL paradigm, multiple tasks are simultaneously learned by a shared model [40], [41], [42]. Such a paradigm can improve data efficiency, reduce overfitting through shared representations, and accelerate learning speed by leveraging auxiliary information. SE researchers have also introduced MTL to address SE tasks. For example, Aung et al. [43] find that both two important tasks, i.e., developer recommendation and issue type classifying involved in the bug triage process, rely on historical bug descriptions and code snippet information. Therefore, they train a multi-triage model to resolve both tasks simultaneously via MTL, and demonstrate this model can reduce task repetition and leverage the correlating information between tasks. MTL has also demonstrated promise in training a pre-trained code representation model (e.g., MulCode [44] and UniXcoder) and achieved promising efficacy on various downstream SE tasks. In our setting, we perform MTL on three summary-focused pre-training tasks, including ULM, MLM, and action word prediction (AWP) [20]. The three tasks are simultaneously learned by a shared encoder. ULM and MLM are two general tasks borrowed from the field of NLP. ULM and MLM are important because they can facilitate the language model to capture the relationships of words in the text [31], [36]. Unlike pre-trained models (e.g., CodeBERT and UniXcoder) that predict masked code tokens based on unmasked parts of code snippets, we design ULM and MLM to predict masked words in summaries based on code snippets. Intuitively, predicting masked words in summaries based on code snippets would help the encoder learn the alignment between masked words and code snippets. In addition, existing work [20] shows that AWP affects the prediction of subsequent words and thus the prediction of the entire summary. Therefore, we further introduce the domain-specific task AWP to enhance the ability of the encoder to learn the alignment between action words and code snippets. In summary, all three summary-focused tasks can enhance the encoder to learn the code-summary alignment such that it can capture the specific code features that need to be summarized. In practice, to reduce the training cost, instead of training the shared encoder from scratch, we initialize the shared encoder with an existing pre-trained encoder, e.g., UniXcoder's encoder. After obtaining the pre-trained shared encoder, we further train a code summarization model capable of generating a succinct natural language summary for a given code snippet. Specifically, we fine-tune the pre-trained shared encoder on the code summarization task and simultaneously train a decoder.

In summary, we make the following contributions.

- We propose a novel approach named ESALE to improve code summarization. ESALE devises three summary-focused pre-training tasks (two general tasks ULM and MLM, and one domain-specific task AWP) to enhance the encoder to learn the code-summary alignment.
- We introduce a domain-specific task (i.e., AWP) as one of the important pre-training tasks. We perform an in-depth analysis of the effect of the AWP task, and statistical results show that this task can significantly improve code summarization performance (detailed in Section 4.2.2).
- We conduct extensive quantitative experiments on four datasets, including two widely used Java and Python datasets (called JCSD and PCSD), a cross-projected Java dataset (called CPJD), and a multilingual dataset (called CSN), to evaluate ESALE. Experimental results show that ESALE significantly outperforms baselines in terms of all three widely used automatic metrics BLEU, METEOR, and ROUGE-L (detailed in Section 4.2.1). The source code of ESALE and all the data used in this paper are released and can be downloaded from the website [45].
- We conduct a qualitative human evaluation to evaluate the summaries generated by ESALE and baselines in terms of four aspects: similarity, naturalness, informativeness, and relevance. And statistical results of human scores show that the summaries generated by ESALE are more informative and closer to the ground-truth summaries (detailed in Section 4.2.4).



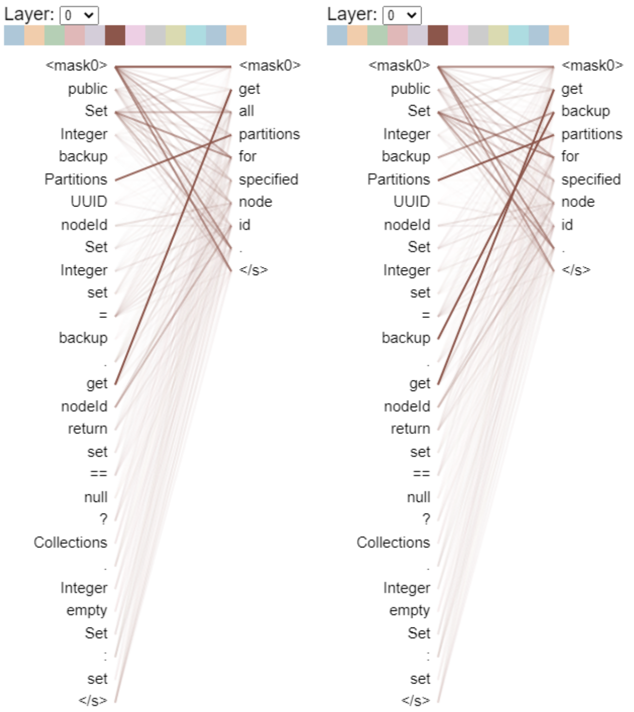
```
public Set< Integer > backupPartitions (UUID nodeId ) {
    Set<Integer> set = backup.get(nodeId);
    return set == null ? Collections<Integer> emptySet() : set;
}
```

(a) A code snippet c_1

1. Reference Summary: get backup partitions for specified node id.
2. Summary by SiT: copies the partitions for this node id.
3. Summary by CodeBERT: return partitions of a node.
4. Summary by UniXcoder: get all partitions for specified node id.
5. Summary by ESALE: get backup partitions for specified node id.

(b) Summaries generated by different techniques for c_1.

Fig. 1. Code snippet c_1 and summaries generated by different techniques for c_1.This example is from the test set of the JCSD dataset, numbered 8,335.



(a) UniXcoder (b) ESALE

Fig. 2. Visualization of cross attention

The rest of this paper is organized as follows. Section 2 illustrates the motivation of this paper. Section 3 introduces the design of ESALE. Section 4 presents the design of the experiments in detail and gives the details of experiment results and analysis. Section 5 presents a case study. Section 6 introduces threats to validity. Section 7 discusses the related work. We conclude the paper in Section 8.

2 MOTIVATING EXAMPLE

This section takes a real-world code snippet and corresponding summaries generated by different techniques as examples to illustrate our motivation. The code snippet c_1 in Fig. 1(a) is from the test set of the JCSD dataset (numbered 8,335) [12] (see details in Section 4.1.1). The first line of Fig. 1(b) shows the comment written by the developer for c_1. We consider the comment as a reference summary. According to the grammar rules in natural language, we can simply divide the reference summary into three parts: “get” (Blue font), “backup partitions” (Red font), and “for specified node id” (Orange font). Lines 2-5 show the summaries generated by baselines SiT [46], CodeBERT, UniXcoder, and our ESALE, respectively. SiT is one of the advanced code summarization techniques. CodeBERT is a representative pre-trained model for source code. UniXcoder is the state-of-the-art pre-trained model for source code. Both CodeBERT and UniXcoder can also be used for code summarization tasks by fine-tuning. More details on the three baselines are introduced in Section 4.2.1. From Fig. 1, it can be observed that compared with the reference summary, 1) SiT and CodeBERT can cover some words in the second and third parts (i.e., “partitions” and “node id”); 2) UniXcoder is better than SiT and CodeBERT, and can cover the word “partitions” in the second part and all words in the last part (i.e., “for specified node id”); 3) ESALE performs the best, successfully covering all three parts. Compared to UniXcoder, our ESALE can correctly generate the word “backup”.

To intuitively understand how ESALE can perform better in code summarization, we visualize the cross attention between encoder and decoder (also called encoder-decoder attention [47]) using the attention visualization tool BertViz[1] [48]. Attention can help interpret the model by showing how the model attends to different parts of the input [48], [49], [50]. Fig. 2(a) and (b) show the visualizations of the cross attentions of UniXcoder and our ESALE. In Fig. 2(a) and (b), the cross attention is depicted as lines connecting the attending tokens (right) with the tokens being attended to (left). In our setting, the left and right show code snippets and summaries, respectively. Colors identify the head(s), and the thickness of the line reflects the attention weights. It can be observed that each word of the generated summary is basically mapped from a certain code pattern (a set of tokens with different weights). In addition, compared to UniXcoder, ESALE can correctly predict the word “backup”, which can be attributed to the higher attention ESALE paid to the two “backup” tokens in the code snippet. This example demonstrates that the context vector produced by ESALE’s encoder captures the code pattern that needs to be translated into “backup”. It should be noted that the main difference between our ESALE and UniXcoder is the encoder. We initialize ESALE’s encoder with the parameters of the pre-trained encoder of UniXcoder, and then fine-tune it with three summary-focused tasks (details are described in Section 3.2.2). Hence, we can attribute the better performance of ESALE to its encoder. As well, we can boldly speculate that the three summary-focused tasks we designed could teach the encoder to learn the code-comment alignment and capture important code features that need to be translated.

To confirm our speculation, we further study the summaries generated by ESALE by deleting related code patterns in the motivation case. When deleting two related code

[1] https://github.com/jessevig/bertviz

TABLE 1
Summaries generated by different models when the token “backup” is deleted from c_1. ESALE w/o AWP, MLM, and ULM denote that we pre-train the shared encoder of ESALE without AWP, MLM, and ULM, respectively.

Model	Summary
UniXcoder	get partitions for a given node id.
ESALE	get partitions for specified node id.
ESALE w/o AWP	# of partitions for given node id.
ESALE w/o MLM	provide this method to get the partitions for the given node id.
ESALE w/o ULM	# of partitions for given node id.

tokens “backup” from c_1 (Lines 1 and 2 of Fig. 1(a)), the summaries generated by UniXcoder and ESALE are shown in the first two rows of TABLE 1. It can be observed that compared with the respective original summaries shown in lines 4 and 5 of Fig. 1(b), the new summaries generated by UniXcoder and ESALE have different changes. Specifically, the new summary generated by UniXcoder misses the word “all”, and “a given node id” is different from “specified node id” in the reference summary. For our ESALE, the word “backup” is not included in the newly generated summary because it is not present in the code any more. Even human developers cannot tell. It also proves in reverse that ESALE’s encoder can capture the code pattern related to the summary word “backup” when it appears in the input code snippet while UniXcoder’s encoder cannot. All these can be attributed to the three summary-focused tasks we designed, on which the encoder trained is able to learn the code-comment alignment and capture important code features that need to be summarized. In addition, when the word “backup” is omitted, the summaries generated by ESALE trained without the AWP, MLM, or ULM task are shown in the last three rows of TABLE 1. It is observed that all three versions of ESALE do not generate the word “backup”. It means that the three summary-focused tasks indeed help train ESALE’s encoder to capture the code pattern related to the summary word “backup” when it appears in the input code snippet while UniXcoder’s encoder cannot. Certainly, it is undeniable that each task may influence the generation of other parts of the summary content, and the three tasks may also affect each other. It is important to note that this still serves to demonstrate that the superior performance of ESALE originates from the three summary-focused tasks.

3 METHODOLOGY

3.1 Overview

Our approach produces a code summarization model via two sequential training phases: (a) training a shared encoder, followed by (b) training a code summarization model based on the encoder generated in phase (a).

In phase (a), ESALE decomposes the training procedure of the shared encoder into two steps: preprocessing and shared encoder training. In the first step, ESALE takes in pairs of code snippets and comments in the training data and produces two sequences, i.e., input sequences and masked input sequences, detailed in Section 3.2.1. In the second step, ESALE utilizes the two sequences to train a shared encoder. In this paper, we aim to enhance the encoder to learn the code-summary alignment, thereby improving code summarization performance. Therefore, we exploit the MTL paradigm to train a shared encoder on three summary-focused tasks. Specifically, for input sequences, ESALE utilizes a shared encoder to transform them into embedding representations, which will be used in the AWP and ULM tasks. For masked input sequences, ESALE utilizes the shared encoder to transform them into embedding representations, which will be used in the MLM task. The models for the three tasks are jointly trained and determine the parameters of the shared encoder, detailed in Section 3.2.2.

In phase (b), after getting the pre-trained shared encoder, ESALE fine-tunes it and trains a decoder simultaneously on the downstream code summarization task. The fine-tuned shared encoder and fine-tuned decoder compose a well-trained code summarization model, detailed in Section 3.3. When the well-trained code summarization model is deployed online, it can take in a code snippet given by the developer and generate a natural language summary, detailed in Section 3.4.

3.2 Training of Shared Encoder

3.2.1 Preprocessing

ESALE takes in raw training data where each sample consists of a code snippet and its corresponding comment. ESALE follows common practices [33], [35], [51] and uses the tokenizer provided by Roberta [52] to tokenize code snippets and comments and produce token sequences and word sequences, respectively. We also use Byte Pair Encoding (BPE) within Roberta to split tokens into subtokens. As [9], we call the basic unit of preprocessed source code a token and the basic unit of summary a word. ESALE further masks parts of words in word sequences to produce masked word sequences. Specifically, we follow existing works [33], [35] and randomly choose 15% of the words in a word sequence, and change them into a special token <MASK>. Next, two special tokens <SOS> and <EOS> are added at the beginning and the end of the token sequences, respectively. The special token <EOS> is appended as a suffix for word sequences and masked word sequences. Then we concatenate pairs of token sequences and word sequences to produce input sequences. We concatenate pairs of token sequences and masked word sequences to produce masked input sequences. Unlike pre-trained models (e.g., CodeBERT and UniXcoder) that first concatenate pairs of token sequences and word sequences and then mask parts of the input sequences, we only mask parts of word sequences. Both input sequences and masked input sequences will be used to train the shared encoder in the second step.

3.2.2 Shared Encoder Training

Shared Encoder. The shared encoder is a deep neural network or pre-trained model responsible for transforming the input sequences into vector representations (i.e., embeddings). In practice, we build our shared encoder upon the existing pre-trained encoder. There are two benefits to doing this: 1) compared to training the encoder from scratch, the scheme based on the pre-trained encoders can significantly reduce the training cost; 2) existing pre-trained encoders have achieved almost optimal performance on the code summarization task, providing a high starting point.

Specifically, we tried to build our shared encoder upon two pre-trained encoders provided by CodeBERT and UniXcoder. We first initialize the shared encoder with the parameters of their pre-trained encoders. Then, we fine-tune the shared encoder with three summary-focused tasks, i.e., AWP, ULM, and MLM. The experimental results show that the shared encoder built upon UniXcoder's encoder is better than that built on CodeBERT's encoder on the downstream code summarization task, detailed in Section 4.2.1.

Next, we introduce the design of the three summary-focused tasks.

(i) AWP. An "action word" in a summary is typically a verb that broadly classifies what the code does [20], such as "get", "add", and "remove". Programmers tend to write summaries containing only one action word, typically positioned at the beginning of the summary sentence (i.e., the first word).

AWP is a classification task, where the input of the model is a code snippet, and the output is the predicted label with respect to the action word [20]. In this paper, we use this task to train a model capable of predicting the action words of summaries based on given code snippets. Formally, let $c = \{t_1, t_2, \ldots, t_m\}$ denote the token sequence of the code snippet, where m is the length of the token sequence, and $y = \{y_1, y_2, \ldots, t_C\}$ denote the set of possible classes, where C is the number of classes of action words. The summary-focused AWP can be defined as follows:

Definition 1 (Summary-focused AWP). A summary-focused AWP is a multi-classification task denoted as $\hat{y} = \arg\max_{y \in Y} P(y|c)$, where:

- $P(y|c)$ represents the probability of the class y given the code snippet c.
- $\arg\max$ denotes the operation that selects the class label with the highest probability.

The model we train is composed of the shared encoder and a classification layer. The classification layer is a fully connected network of size $N * C$, where N is the output size of the shared encoder. Given an input sequence x, we first utilize the shared encoder to transform x into the embedding $\boldsymbol{e}^x$. Then, a classification layer is used to classify $\boldsymbol{e}^x$ into predicted action word classes. Given the embedding $\boldsymbol{e}^x$, we obtain the logits by $\hat{y}_i = \boldsymbol{W}\boldsymbol{e}^x + \boldsymbol{b}$, where $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{b}$ is the bias term. We optimize the model by minimizing the categorical cross-entropy loss:

$$\mathcal{L}_{AWP}(\Theta) = -\sum_{i=1}^{C} y_i \log \frac{\exp(\hat{y}_i)}{\sum_{j=1}^{C} \exp(\hat{y}_j)} \qquad (1)$$

where Θ denotes trainable parameters of the model (i.e., $\boldsymbol{W}$ and $\boldsymbol{b}$); $\hat{y}_i$ and y_i are the predicted score and target score for each class $i \in C$. In practice, we follow [20] and use the top-40 setting; that is, the model attempts to predict the forty most-common action words, or "other" if predicted to be a less-common action word. The top 40 action words are selected based on their frequency in the comments of all samples in each dataset.

TABLE 2
The performance of encoders of different seq2seq models on the AWP task. ESALE w/o AWP denotes that we pre-train the shared encoder of ESALE without AWP.

Model (Year)	Precision	Recall	F-measure
AttendGRU (2016)	63.49	62.43	62.55
CodeBERT (2020)	63.17	66.01	62.65
UniXcoder (2022)	63.19	66.28	62.87
ESALE w/o AWP	63.17	66.21	63.05
ESALE	63.54	66.26	63.27

Here, we give a brief explanation of why we consider AWP as one of the pre-training tasks. First, the production of good summaries relies on the production of the action words in those summaries. Code summarization models are widely built on the encoder-decoder framework, where the decoder predicts words one by one according to previous words and the context produced by the encoder. So, if the first word is wrong, it is difficult for the decoder to predict the entire summary correctly. This situation can be exaggerated by the aggressive use of attention mechanisms, which can attend previous words in the predicted summary to parts of the code snippet [20]. Therefore, it is crucial for code summarization models to predict accurate action words. Second, our experiments found that ESALE equipped with AWP performs better than without. In practice, before deciding to add AWP as one of the pre-training tasks, we also followed [20] and conducted experiments on the performance of encoders of different seq2seq models on the AWP task with 41 classes. We compared five techniques, including AttendGRU, CodeBERT, UniXcoder, ESALE w/o AWP, and ESALE. AttendGRU is representative of seq2seq-like approaches as proposed by Iyer et al. [30]. In the paper [20], AttendGRU performs the best, so we also consider it as a baseline. For AttendGRU, we build a classification model by appending a fully connected network to its encoder, and train the model from scratch. For CodeBERT, UniXcoder, ESALE w/o AWP, and ESALE, we build classification models by appending a fully connected network to their encoders as classification layers and train the models by fine-tuning.

TABLE 2 shows the experimental results where columns 2–4 report the weighted average precision, recall, and f-measure computed by the classification_report function provided by scikit-learn [2]. The experimental dataset consists of pairs of code snippets and action words extracted from the JCSD dataset. From the table, it is observed that, overall, 1) compared with the encoder from AttendGRU and trained from scratch, the pre-trained encoders from CodeBERT, UniXcoder, ESALE w/o AWP perform better in terms of f-measure; 2) compared with ESALE w/o AWP, ESALE's encoder treating AWP as one of the pre-training tasks further improves the AWP performance. More details of comparing ESALE w/o AWP and ESALE are described in Section 4.2.2.

In summary, equipping ESALE with AWP aims to enhance the shared encoder to learn the code pattern that is the key feature to predict the action word. In this way, the code summarization model based on the shared encoder can better generate the action word of the summary.

[2] https://scikit-learn.org/stable/modules/model_evaluation.html#classification-report

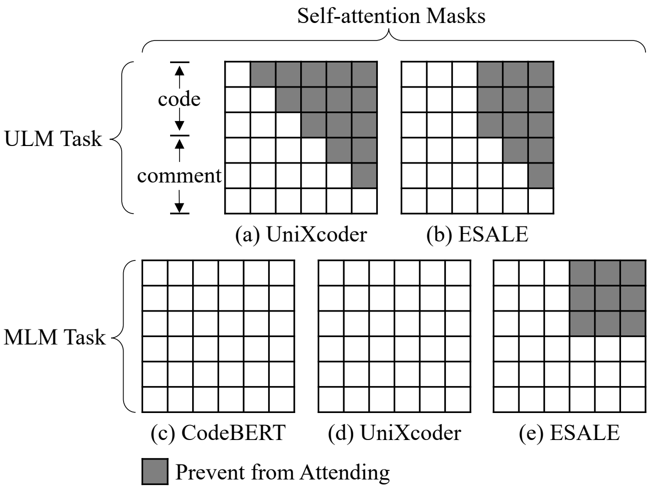


Fig. 3. Differences in preventing from attending

(ii) ULM. The ULM task is defined as the problem of left-to-right language modeling [36], which only allows words to attend the previous words and itself to predict the next word [35], [53]. Unlike existing works on predicting the next code token [35], [53], we use this task to train a model capable of predicting the next summary word one by one conditioned on the code token sequence and unmasked/preceding parts of the summary sequence. It can be done using a ULM mask matrix for the attention mask. We refer to such ULM as summary-focused ULM. Formally, $w = \{w_1, w_2, \ldots, w_n\}$ denote the word sequence of the summary, where n is the length of the word sequence. The summary-focused ULM can be defined as follows:

Definition 2 (Summary-focused ULM). A summary-focused ULM is a probabilistic model denoted as $P(w_1, w_2, \ldots, w_n|c) = \prod_{i=1}^{n} P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1, c)$, where:

- $P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1, c)$ represents the probability of the word w_i given the preceding summary words $w_{i-1}, w_{i-2}, \ldots, w_1$ and the code snippet c.
- $\prod$ denotes the product of probabilities over the entire word sequence of the summary.

The model we train also includes the shared encoder followed by a fully connected network. The size of the fully connected network is $N * |V|$, where N is the output size of the shared encoder and $|V|$ is the vocabulary size. Given an input sequence x, we first utilize the shared encoder to produce its embedding $\boldsymbol{e}^x$. Then, the fully connected network is used to predict the likelihood score of each word in the vocabulary being the next word. The model is optimized by minimizing the objective function:

$$\mathcal{L}_{ULM}(\Theta) = -\sum_{i=0}^{n-1} logp(w_i|\boldsymbol{e}^x_{t<i}). \tag{2}$$

where $\boldsymbol{e}^x_{t<i}$ represents the embedding of the word sequence appearing on the left of the word w_i.

Fig. 3(a) and (b) visually illustrate the self-attention masks used by UniXcoder and ESALE, respectively. The self-attention masks are used to control the behavior of the model, i.e., preventing from attending. UniXcoder directly exploits the general ULM in NLP [54], which uses a triangular matrix for attention mask, predicting the next token in the entire input sequence. Different UniXcoder, ESALE introduces a summary-focused ULM, which is used to train ESALE to predict the next summary word only in the summary word sequence by attending the entire code token sequence and the left summary words.

(iii) MLM. The MLM task is defined as the problem of predicting the original tokens of masked tokens based on their bidirectional contextual tokens [31]. Unlike ULM, which can only be trained unidirectionally, bidirectional conditioning in MLM allows each word to indirectly see itself, simplifying the prediction of the target word in a multi-layered context. Therefore, in this paper, this task is designed to train a model capable of predicting masked tokens based on all tokens in the code snippet and unmasked words in the summary. Similarly, we refer to such MLM as summary-focused MLM. Formally, the summary-focused MLM can be defined as follows:

Definition 3 (Summary-focused MLM). A summary-focused MLM is a probabilistic model denoted as $P(w_1, w_2, \ldots, w_n|c) = \prod_{i=1}^{n} P(w_i|w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n, c)$, where:

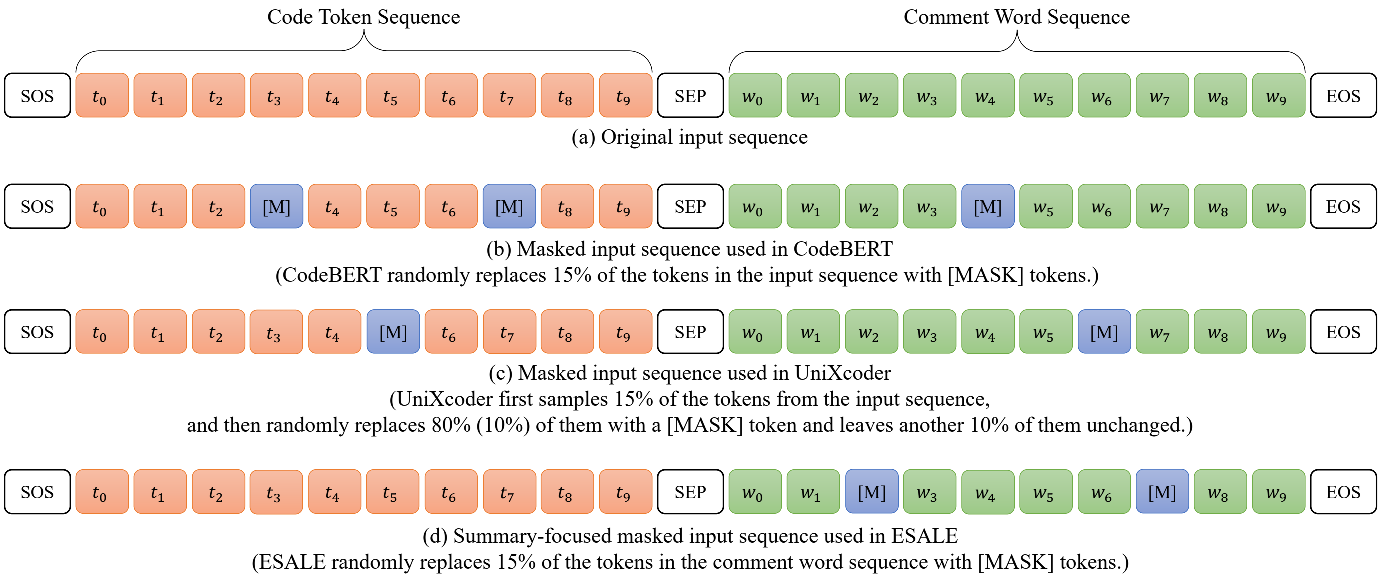
- $P(w_i|w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n, c)$ represents the probability of the masked word w_i given the unmasked summary words $w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n$ and the code snippet c.
- $\prod$ denotes the product of probabilities over the entire word sequence of the summary.

In this task, the model we train is composed of a shared encoder and a fully connected network. The design of the fully connected network is the same as in the summary-focused ULM task. Given a masked input sequence x, we first use the shared encoder to transform x into embedding $\boldsymbol{e}^x$. Then, the fully connected network is used to predict the likelihood score of each word in the vocabulary being the masked word. We optimize the model by minimizing the following objective function:

$$\mathcal{L}_{MLM}(\Theta) = -\sum_{w_i \in S_m} logp(w_i|\boldsymbol{e}^x_{mask}) \tag{3}$$

where $\boldsymbol{e}^x_{mask}$ is the embedding of x; S_m is the set of masked words that need to be predicted.

We use Fig. 4 to visually illustrate the differences in the masked proportion and position between the baselines (i.e., CodeBERT and UniXcoder) and our ESALE. Fig. 4(a) shows an example of an original input sequence consisting of a code token sequence and a comment word sequence. Fig. 4(b)–(d) show the masked input sequences used in CodeBERT, UniXcoder, and ESALE, respectively. In Fig. 4(b), we follow CodeBERT and randomly replace 15% of the tokens in the input sequence with [MASK] tokens (the blue blocks labeled [M] in the figure). In Fig. 4(c), we follow UniXcoder and first sample 15% of the tokens from the input sequence, and then randomly replace 80% (i.e., about 10% of the input sequence) of them with a [MASK] token and leave another 10% of them unchanged. In Fig. 4(d),


Code Token Sequence
Comment Word Sequence
SOS
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
SEP
w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9
EOS
[M]

(a) Original input sequence

(b) Masked input sequence used in CodeBERT
(CodeBERT randomly replaces 15% of the tokens in the input sequence with [MASK] tokens.)

(c) Masked input sequence used in UniXcoder
(UniXcoder first samples 15% of the tokens from the input sequence,
and then randomly replaces 80% (10%) of them with a [MASK] token and leaves another 10% of them unchanged.)

(d) Summary-focused masked input sequence used in ESALE
(ESALE randomly replaces 15% of the tokens in the comment word sequence with [MASK] tokens.)

Fig. 4. Differences in MLM between baselines (CodeBERT, UniXcoder) and our ESALE

our ESALE randomly replaces 15% of the tokens in the comment word sequence with [MASK] tokens. From the figure, it is observed that ESALE is summary-focused and significantly different from CodeBERT and UniXcoder in the masked proportion and position. Fig. 3(c), (d), and (e) also visually present the self-attention masks used by CodeBERT, UniXcoder, and ESALE, respectively.

Model Training. The training procedure of the above task models follows the existing MTL paradigm [53]. In MTL, models are trained with data from multiple related tasks simultaneously while using a shared representation to learn the common features between all these tasks, and what is learned for one task can help other tasks be learned better [55]. The shared representation increases data efficiency and can potentially yield a faster learning speed for related or downstream tasks, helping to alleviate the well-known weaknesses of deep learning: large-scale data requirements and computational demand [40].

In this paper, we exploit the MTL paradigm to train a shared encoder with three summary-focused tasks, i.e., AWP, ULM, and MLM. The weight parameters of the shared encoder are learned to minimize the sum of the cross-entropy losses of the three pre-training tasks, and are shared among all three tasks. The final loss function is computed as:

$$\min_{\Theta} \mathcal{L}_{AWP}(\Theta) + \mathcal{L}_{ULM}(\Theta) + \mathcal{L}_{MLM}(\Theta) \tag{4}$$

Intuitively, during pre-training, the AWP model predicts the label corresponding to action words based on the input sequence. Simultaneously, the ULM model predicts the next token based on the left tokens of the input sequence. Meanwhile, the MLM model predicts the original tokens of masked tokens of the input sequence. The AWP model, ULM model, and MLM model share an encoder (i.e., the encoder of ESALE). After obtaining a pre-trained encoder (referred to as the pre-trained shared encoder in this paper), we further use pairs of code snippets and summaries to fine-tune it along with the decoder. The fine-tuning process is elaborated upon in the subsequent section.

3.3 Training of Code Summarization Model

After obtaining the pre-trained shared encoder, we further train a code summarization model capable of generating a succinct natural language summary for a given code snippet. Specifically, we fine-tune the pre-trained shared encoder on the code summarization task and simultaneously train a decoder. Given training data consisting of pairs of code snippets and comments, ESALE first leverages the pre-trained shared encoder to transform code snippets into context vectors $\boldsymbol{e}^{Code}$. Then, ESALE leverages a decoder to generate predicted summaries. The decoder takes in $\boldsymbol{e}^{Code}$ and predicts words one by one, detailed in Section 3.3.1). Finally, ESALE computes the loss ($\mathcal{L}_{CS}(\Theta)$) based on predicted summaries and ground-truth summaries (i.e., comments) and iteratively updates the model parameters Θ, detailed in Section 3.3.2).

3.3.1 Decoder

In this step, we utilize the decoder to generate natural language summaries. The decoder takes in the context vectors $\boldsymbol{e}^{Code}$ and predicts words one by one. Specifically, the decoder based on a neural network (e.g., LSTM [56] and Transformer [47]) is to unfold the context vectors $\boldsymbol{e}^{Code}$ into the target sequence (i.e., the word sequence of the summary) through the following dynamic model,

$$\begin{aligned} \boldsymbol{h}_t &= f(y_{t-1}, \boldsymbol{h}_{t-1}, \boldsymbol{e}^{Code}) \\ p(y_t|Y_{<t}, X) &= g(y_{t-1}, \boldsymbol{h}_t, \boldsymbol{e}^{Code}) \end{aligned} \tag{5}$$

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 (5) that the probability of generating the target word is related to the current hidden state, the history of the target sequence, and the context vectors $\boldsymbol{e}^{Code}$. The essence of the decoder is to classify the vocabularies by optimizing the loss function in order to generate the vector representing the feature of the target word y_t. After the vector passes through a *softmax* function, the word corresponding to the highest probability is the result to be output.

In practice, we design our decoders with different schemes suggested by CodeBERT and UniXcoder, respectively. CodeBERT only provides a pre-trained encoder, while UniXcoder provides a pre-trained encoder and a pre-trained decoder. Therefore, when the shared encoder is built upon the pre-trained encoder provided by CodeBERT, we build our decoder upon Transformer [47]. When the shared encoder is built upon the pre-trained encoder provided by UniXcoder, we build our decoder upon the pre-trained decoder provided by UniXcoder.

3.3.2 *Model Training*

During the training of the code summarization model, the two components (pre-trained shared encoder and decoder) are jointly trained to minimize the following objective function:

$$\mathcal{L}_{CS}(\Theta) = -\frac{1}{N}\sum_{n=1}^{N} logp(\boldsymbol{y}_n|\boldsymbol{x}_n) \qquad (6)$$

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

3.4 Deployment of Code Summarization Model

After the model is trained, we can deploy it online for code summarization service. For a code snippet c given by the developer, ESALE first uses the fine-tuned encoder to transform c into a context vector, which will be fed to the fine-tuned decoder to generate a summary in natural language. In practice, we can consider the well-trained ESALE as a black-box tool that takes in a code snippet and generates a succinct natural language summary.

4 EVALUATION

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

RQ1: How does ESALE perform compared to the state-of-the-art baselines?

RQ2: How do the three pre-trained tasks (i.e., AWP, ULM, and MLM) affect the performance of ESALE (ablation study)?

RQ3: How does the robustness of ESALE perform when varying the code length and comment length?

RQ4: How does ESALE perform in human evaluation?

4.1 Experimental Setup

4.1.1 *Dataset*

We conduct experiments on four datasets. JCSD provided by Hu et al. [12] is a Java dataset. PCSD provided by Barone et al. [57] is a Python dataset. These two datasets are named JCSD and PCSD by Zhang et al. [58] and have been widely used by existing code summarization studies [33], [46], [58], [59], [60], [61]. Some studies point out that randomly splitting datasets can lead to leakage between the test set and training set, since code from the same projects tends to be very similar [17], [62], [63]. Therefore, we also conduct experiments on a recently released project-partitioned Java dataset (CPJD, for short) provided by Nie et al. [63]. To explore the performance of ESALE on multiple programming language datasets, we also conduct experiments on the CodeSearchNet corpus (CSN, for short). The CSN corpus provided by Husain et al. [64] contains a large number of pairs of code snippets and comments across six programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby. Lu et al. [65] reveal that some comments contain content unrelated to the code snippets and performed data cleaning on the CSN corpus. Therefore, in this paper, we follow [33], [34] and use the clean version of the CSN corpus provided by Lu et al. [65]. Since JCSD, PCSD, and CPJD already contain Java and Python datasets, to reduce the experimental workload and focus on evaluating ESALE on different languages, for CSN, we mainly conduct experiments on four languages: PHP, Go, JavaScript, and Ruby. The statistics of the four datasets are shown in TABLE 3, where the last column presents the data splitting methods used by the four datasets. It should be noted that we only use the training set in the pre-training phase of the shared encoder. Model training/pre-training on different datasets/programming languages is independent of each other.

TABLE 3
Dataset statistics. CPJD denotes the cross-project Java dataset. CSN denotes the CodeSearchNet corpus.

Dataset		Training Set	Validation Set	Test Set	Splitting Method
JCSD		69,708	8,714	8,714	Random
PCSD		57,203	19,067	19,066	Random
CPJD		51,408	7,180	7,409	Project-partitioned
CSN	PHP	241,241	12,982	14,014	Project-partitioned
	Go	167,288	7,325	8,122	
	JavaScript	58,025	3,885	3,291	
	Ruby	24,927	1,400	1,261	

4.1.2 *Evaluation Metrics*

We use three automatic metrics BLEU [66], METEOR [67], and ROUGE [68], to evaluate the model, which are widely used in code summarization [12], [30], [46], [47], [61], [69].

BLEU, the abbreviation for BiLingual Evaluation Understudy [66], is widely used for evaluating the quality of generated summaries [12], [30], [69]. It is a variant of precision metric, which calculates the similarity by computing the n-gram precision of a generated summary to the reference summary. It has a penalty for the overly short length [66]. In this paper, we follow [35], [46], [61] and show the standard BLEU score which provides a cumulative score of 1-, 2-, 3-, and 4-grams [9].

METEOR, the abbreviation for Metric for Evaluation of Translation with Explicit ORdering [67], is also widely used to evaluate the quality of generated summaries [58], [70], [71]. For a pair of summaries, METEOR creates a word alignment between them and calculates the similarity scores.

ROUGE-L. ROUGE is the abbreviation for Recall-oriented Understudy for Gisting Evaluation [68]. 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 [72], [73], [74].

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

4.1.3 Experimental Settings

To train models, we first shuffle the training data and set the mini-batch size to 32. For each batch, the code snippets are padded with a special token $\langle PAD \rangle$ to the maximum length. Following [33], [35], we set the maximum length of code snippets and comments to 256 and 128, respectively. We update the parameters via AdamW optimizer [75] for 100k steps, with a learning rate of 0.0005. To prevent overfitting, we use dropout with 0.1. For beam search, we set the beam size to 5. Finally, we select the best model based on the lowest validation loss. 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 32 GB memory, running on Ubuntu 18.04.

4.2 Experimental Results

4.2.1 *RQ1:* ESALE *vs. Baselines*

1) *Baselines:*

Re^{2}Com [29] adopts an LSTM-based encoder-decoder architecture with an attention mechanism. It first uses an information retrieval technique to retrieve a similar code snippet and treat its comment as an exemplar. Then, it uses an LSTM-based seq2seq neural network that takes the given code, its AST, its similar code, and its exemplar as input, and leverages the information from the exemplar to generate summaries.

SiT [46] adopts a Transformer-based encoder-decoder architecture. It proposes a structure-induced transformer to capture long-range dependencies and more global information in AST sequences of code snippets.

SCRIPT [61] adopts a Transformer-based encoder-decoder architecture. It proposes two types of Transformer encoders to capture the structural relative positions between tokens for better learning code semantics.

In addition to these non-pre-trained techniques above, since our method is based on the pre-training and fine-tuning paradigm, we also compare two techniques following such a paradigm.

CodeBERT [33] is a representative pre-trained model of code. It is trained with the MLM and Replaced Token Detection (RTD) tasks. The authors of CodeBERT fine-tune and test it on the code summarization task (also called the code documentation generation task in their paper).

UniXcoder [35] is the state-of-the-art pre-trained model of code. It is trained with four tasks: MLM, ULM, Denoising Objective DeNoiSing (DNS), and Code Fragment Representation Learning. Unlike CodeBERT which only pre-trains the encoder, UniXcoder pre-trains both the encoder and the decoder. The authors of UniXcoder also fine-tune and test it on the code summarization task.

TABLE 4
Overall performance of baselines and our ESALE

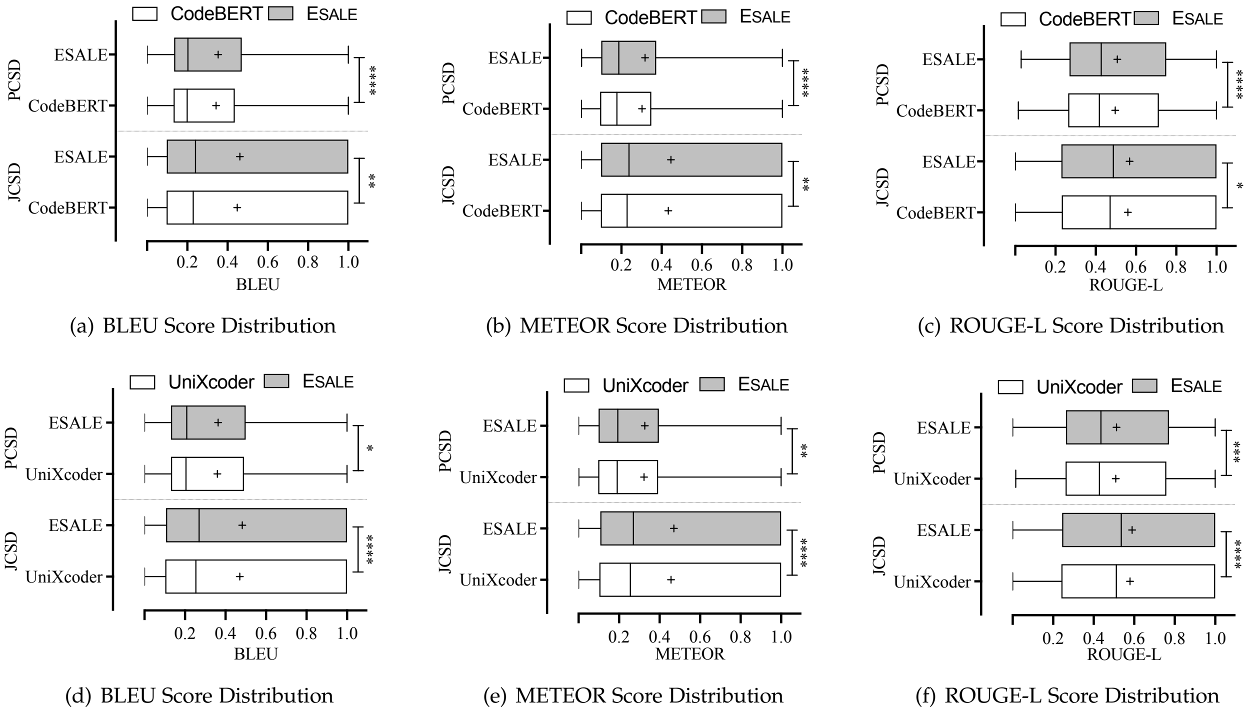
Techniques (Year)	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
Re^{2}Com (2020)	35.65	16.26	44.95	–	–	–
SiT (2021)	45.22	27.10	55.44	33.75	21.02	48.33
SCRIPT (2022)	46.41	28.47	56.57	33.52	20.80	48.09
CodeBERT (2020)	44.80	27.73	56.00	34.35	22.02	49.69
ESALE + CodeBERT	**46.21**	**29.37**	**56.82**	**35.34**	**23.00**	**50.76**
UniXcoder (2022)	47.15	30.02	57.98	36.03	23.37	50.93
ESALE + UniXcoder	**48.31**	**30.79**	**58.98**	**36.36**	**23.60**	**51.34**

For non-pre-training-based models (i.e., Re^{2}Com, SiT, and SCRIPT) and pre-training-based models (i.e., CodeBERT and UniXcoder), we train and fine-tune them separately on the training set of each code summarization dataset, and evaluate them on the corresponding test set.

2) *Results:* TABLE 4 shows the performances of our ESALE and baselines in terms of the three evaluation metrics, i.e., BLEU, METEOR, and ROUGE-L. In TABLE 4, "ESALE + CodeBERT" refers to that we build the shared encoder based on the pre-trained encoder provided by CodeBERT. Analogously, in "ESALE + UniXcoder", the shared encoder and decoder are built upon the pre-trained encoder and decoder provided by UniXcoder. In practice, we initialize our encoder and decoder with the model parameters of the CodeBERT and UniXcoder.

From TABLE 4, it can be observed that, in all non-pre-training baselines, SCRIPT and SiT perform the best on JCSD and PCSD datasets in terms of all three metrics, respectively. However, SCRIPT requires complex preprocessing for code snippets and does not release preprocessing code implementation. Thus, we re-run SCRIPT on their preprocessed datasets. Although the preprocessed datasets are derived from JCSD and PCSD datasets, they have different training and test sets. Therefore, we mainly compare our ESALE to SiT in subsequent sections. ESALE built on CodeBERT or UniXcoder is more powerful than SiT and achieves more impressive performance. On the JCSD dataset, compared with SiT, ESALE built on UniXcoder improves by 6.83% in BLEU, 13.62% in METEOR, and 6.39% in ROUGE-L. On the PCSD dataset, ESALE built on UniXcoder also clearly outperforms SiT, improving by 7.73% in BLEU, 12.27% in METEOR, and 6.23% in ROUGE-L. Because ESALE built upon UniXcoder performs the best, unless explicitly stated, ESALE appearing alone refers to "ESALE + UniXcoder" by default.

In addition, it can be observed that, our method consistently improves the performance of the original pre-trained models, i.e., CodeBERT and UniXcoder on both datasets in general. It should be noted that the values in TABLE 4 are the average scores of all test samples. For a more comprehensive comparison, we further compare the distribution of the scores of CodeBERT, UniXcoder and ESALE on all test samples, and the statistical results are shown in Fig. 5.



(a) BLEU Score Distribution (b) METEOR Score Distribution (c) ROUGE-L Score Distribution

(d) BLEU Score Distribution (e) METEOR Score Distribution (f) ROUGE-L Score Distribution

Fig. 5. Score distribution of three metrics. "*" ($0.01 < p < 0.05$), "**" ($0.001 < p < 0.01$), "***" ($0.0001 < p < 0.001$) and "****" ($p < 0.0001$) represent the differences between two groups are Significant, Very significant, Extremely significant and Extremely significant, respectively. And 'ns' ($p \geq 0.05$) means Not significant.

TABLE 5
Effectiveness of ESALE on the deduplicated JCSD and PCSD datasets

Technique	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
SiT	27.98	16.38	41.15	33.76	21.03	48.35
SCRIPT	29.24	17.74	42.44	33.51	20.80	48.09
CodeBERT	44.64	27.60	55.86	34.33	22.02	49.68
UniXcoder	46.85	29.89	57.85	35.99	23.33	50.90
ESALE	**48.05**	**30.69**	**58.89**	**36.34**	**23.61**	**51.33**

TABLE 6
Effectiveness of ESALE on the JCSD and PCSD datasets from [78]

Technique	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
SiT	11.24	9.86	24.30	15.40	9.56	23.08
CodeBERT	27.88	19.00	39.05	16.14	11.37	25.23
UniXcoder	29.00	20.36	40.31	17.10	12.19	26.80
ESALE	**30.52**	**21.63**	**42.63**	**17.94**	**12.82**	**28.14**

TABLE 7
Effectiveness of ESALE on the CPJD dataset

Technique	CPJD		
	BLEU	METEOR	ROUGE-L
SiT	10.35	6.75	14.06
CodeBERT	15.40	10.05	19.29
UniXcoder	20.62	13.11	24.44
ESALE	**21.42**	**13.79**	**25.94**

In Fig. 5, '+' denotes the mean, which is the value filled in TABLE 4. Overall, the score distribution of ESALE is better than that of pre-trained models (i.e., CodeBERT and UniXcoder). To test whether there is a statistically significant difference between ESALE and pre-trained models, 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 [76], [77]. From Fig. 5, it can be observed that, intuitively, in all three metrics, ESALE outperforms pre-trained models CodeBERT and UniXcoder on both the JCSD and PCSD datasets. In summary, the results and observations above demonstrate that under all experimental settings, our ESALE consistently achieves higher performance in all three metrics, which indicates better code summarization performance. Note that our ESALE is non-intrusive, indicating that it can be combined with future state-of-the-art language models to further improve code summarization performance.

Existing research [21] find that there are different degrees of data overlap in the training sets and test sets of the original JCSD and PCSD datasets, respectively. Considering that data leakage may affect the accuracy of performance evaluation, we further remove the duplicate code snippets in the training and test sets of these two datasets, and measure the performance of ESALE and several baselines again. TABLE 5 presents their performance on the deduplicated JCSD and PCSD datasets. It is observed that ESALE consistently outperforms the four baselines in all three metrics.

Shi et al. [78] find that there are some noisy data in the JCSD and PCSD datasets and provide filtered versions of these two datasets after removing the noisy data. Therefore, we also compare ESALE with the three baselines

(SiT, CodeBERT, and UniXcoder) on the JCSD and PCSD datasets released by Shi et al. [78]. SCRIPT requires special data preprocessing, and we failed to reproduce it on these two filtered datasets. The results are shown in TABLE 6, demonstrating that ESALE outperforms the best baseline UniXcoder in all metrics.

As mentioned earlier, we also conduct an experiment on a cross-project dataset called CPJD. TABLE 7 shows the overall performance of the three baselines and our ESALE on the CPJD dataset. SCRIPT requires special data preprocessing, and we failed to reproduce it on CPJD. From TABLE 7, it is observed that our ESALE outperforms all three baselines in terms of all three metrics.

Moreover, we conduct experiments on the CSN dataset to evaluate the effectiveness and generalizability of ESALE on more programming languages. TABLE 8 shows the overall performance of CodeBERT, UniXcoder, and our ESALE on the CSN-PHP, -Go, -JavaScript, and -Ruby datasets. The baselines SiT and SCRIPT require special data preprocessing, which cannot be applied to some programming languages in the CSN dataset (e.g., Go, PHP, and Ruby). Therefore, on the CSN dataset, we only compare ESALE with CodeBERT and UniXcoder. From TABLE 8, it is observed that ESALE consistently outperforms CodeBERT and UniXcoder on the four programming language code summarization tasks.

To accurately reflect whether the performance improvement is attributable to the shared encoder trained with three summary-focused pre-training tasks, we further conduct experiments where the pre-trained encoder remains frozen while only the decoder undergoes fine-tuning. Rows CodeBERT$_{frozen}$, UniXcoder$_{frozen}$, and ESALE$_{frozen}$ of TABLE 9 show the performance of baselines and ESALE when the parameters of the pre-trained shared encoder are frozen during fine-tuning on downstream code summarization tasks. It is observed that compared to code summarization models built on unfrozen encoders, CodeBERT$_{frozen}$, UniXcoder$_{frozen}$, and ESALE$_{frozen}$ employing frozen encoders all exhibit varying degrees of performance degradation. It is reasonable because freezing the parameters of the pre-trained encoder can restrict the model's adaptability and hinder its ability to effectively learn task-specific features. Furthermore, it can be observed that ESALE$_{frozen}$ outperforms CodeBERT$_{frozen}$ and UniXcoder$_{frozen}$ in all metrics, which effectively demonstrates the capability of the three summary-focused pre-training tasks to enhance the encoder's code summarization ability.

4.2.2 *RQ2: Effect of Each Pre-train Tasks (Ablation Study)*

We use three tasks (AWP, MLM, and ULM) to enhance the ability of our model to learn code-summary alignment. We conduct ablation studies to reveal the influence of each task on the performance of ESALE. The study results are shown in TABLE 10, in which "ESALE w/o AWP", "ESALE w/o MLM", and "ESALE w/o ULM" mean that we train ESALE without the AWP, MLM, and ULM tasks, respectively. It is observed that the performance of ESALE degrades when any of the three tasks are ignored. Therefore, it can be concluded that all three tasks play an important role in improving the code summarization performance of ESALE.

In addition, from TABLE 10, it can be observed that the AWP task has the most significant effect on the performance of ESALE. We further delve into the contribution of the AWP task to ESALE, which is a task especially designed for the code summarization [20]. In TABLE 11, the second column shows the total number of samples in the JCSD and PCSD test set, and the "Num." and "Pro." columns show the number and proportion of samples whose action words are included in the top-40 common list, respectively. From this table, it can be observed that the top-40 setting only covers about 61% of samples in test sets. In other words, many action words (in the remaining 39% samples) are still not included. We further compute the AWP accuracy of ESALE, and the results are shown in the "AWP Acc." column of TABLE 11. It can be observed that the average AWP accuracy of ESALE is 64.89%. We also check whether the performance of ESALE can be improved when the action words are correctly predicted. TABLE 12 shows the results of ESALE and ESALE w/o AWP on the samples whose action words are included in the top 40 common list and predicted correctly. In TABLE 12, "ESALE w/o AWP" means that we train ESALE without the AWP task; "# Improved" denotes the number of the samples for which ESALE can generate higher quality summaries (i.e., larger BLEU-4, METEOR, and ROUGE-L) when correctly predicting their action words. From TABLE 12, it can be observed that ESALE can generate higher quality summaries for 914 Java and 2,283 Python code snippets on average when predicting their action words correctly. For each metric, we also perform the paired Wilcoxon-Mann-Whitney signed-rank test on all scores got by ESALE w/o AWP and ESALE at a significance level of 5%. The test results are presented in the "p-value" columns of TABLE 12. We can intuitively observe that in all three metrics, ESALE outperforms ESALE w/o AWP on both the JCSD and PCSD datasets, which means the AWP plays a significant role in facilitating ESALE to generate high-quality summaries, as claimed by [20]. Fig. 6 shows two examples, including a Java example c_2 and a Python example c_3. From these two examples, we can also intuitively observe that compared to ESALE w/o AWP, ESALE can generate higher quality and closer reference summaries, which can be attributed to correctly predicted action words.

4.2.3 *RQ3: Robustness of* ESALE

To analyze the robustness of ESALE, we study two parameters (i.e., code length and comment length) that may influence the embedding representations of code snippets and comments. Fig. 7 shows the length distributions of code snippets and comments on the test sets of the JCSD and PCSD datasets. For a code snippet, its length refers to the lines of the code snippet. For a comment, its length refers to the number of words in the comment. From Fig. 7 (a) and (c), it can be observed that most code snippets are between 20 and 40 lines. From Fig. 7 (b) and (d), it is noticed that almost all comments are less than 20 in length. This also confirms the challenge of capturing the correlation between the long code snippet and its short comment (summary).

Fig. 8 shows the performance of two best baselines (i.e., CodeBERT and UniXcoder) and ESALE based on the BLEU metric with varying parameters. In this figure, the version of

TABLE 8
Effectiveness of ESALE on other programming language datasets, including CSN-PHP, -GO, -JavaScript, and -Ruby.

Technique	CSN-PHP			CSN-Go			CSN-JavaScript			CSN-Ruby		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
CodeBERT	23.11	15.16	38.69	16.59	10.47	31.06	14.06	9.30	25.97	12.70	8.94	20.11
UniXcoder	26.36	16.53	41.75	17.78	11.69	33.88	15.46	9.74	26.77	14.85	9.88	26.57
ESALE	**26.76**	**16.62**	**41.84**	**18.15**	**11.74**	**34.07**	**15.61**	**9.83**	**26.90**	**14.99**	**9.89**	**26.68**

TABLE 9
Performance of ESALE when the pre-trained shared encoder is frozen during fine-tuning on downstream code summarization tasks

Technique	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
CodeBERT$_{frozen}$	18.59	11.13	30.67	18.29	9.18	29.62
UniXcoder$_{frozen}$	20.25	12.38	31.49	18.62	10.04	30.01
ESALE$_{frozen}$	**25.15**	**15.31**	**35.57**	**21.52**	**13.62**	**34.18**

TABLE 10
Effect of three pre-training tasks AWP, MLM, and ULM

Technique	JCSD			PCSD		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
ESALE w/o AWP	47.86	30.54	58.55	36.10	23.39	51.02
ESALE w/o MLM	48.13	30.6	58.67	36.25	23.45	51.18
ESALE w/o ULM	47.92	30.54	58.65	36.14	23.47	51.10

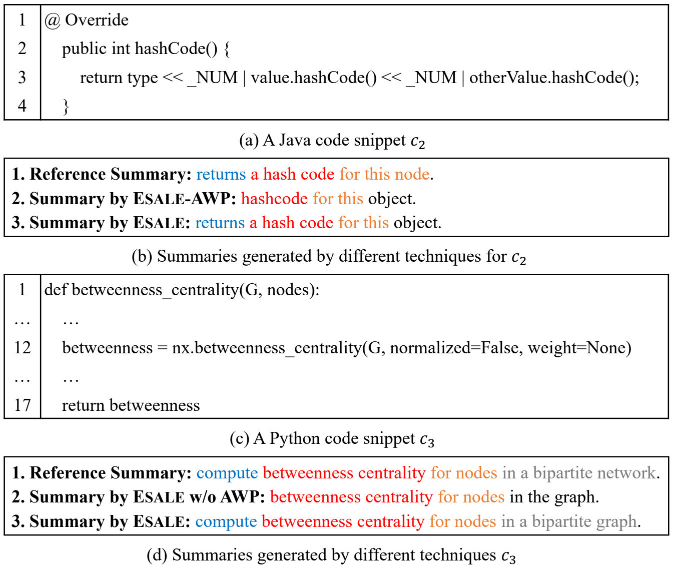
ESALE is the same as "ESALE + UniXcoder". From Fig. 8 (a) and (b), it can be observed that on the JCSD test set, ESALE maintains stable performance even though the code snippet length or comment length increases. On the PCSD test set, from Fig. 8(c) and (d), it can be observed that ESALE also maintains stable performance when the code snippet length increases, however, the performance of ESALE degrades significantly as the length of the comment increases. In addition, it is observed that the same phenomenon occurred in CodeBERT and UniXcoder. It means that as the expected length of the generated summary continues to increase, it will be more challenging to generate. Overall, the results above verify the robustness of our ESALE. Due to the page limit, please refer to our project website [79] for additional results on the METEOR and ROUGE-L metrics, where you can find that the robustness of both the baselines and ESALE on METEOR and ROUGE-L is similar to that on BLEU.

4.2.4 *RQ4: Human Evaluation*

Many works [21], [22], [28], [29], [30], [39], [58], [80] find that it is not enough to use only automatic evaluation because the automatic metrics (BLEU, METEOR, and ROUGE-L) mainly calculate the textual similarity rather than the semantic similarity between the generated summaries and the reference summaries. Hence, we conduct a human evaluation by following the previous works [29], [39], [46], [58], [80] to evaluate the summaries generated by SiT, UniXcoder, and our ESALE. The selection of SiT and UniXcoder as comparison models stems from their status as advanced code summarization techniques representing distinct paradigms: SiT is a non-pre-training-based technique, while UniXcoder is a pre-training-based technique. Specifically, we invite ten volunteers with more than three years of programming experience and excellent English skills to carry out the evaluation. Each volunteer scores the generated summaries from

TABLE 11
The statistic information of samples in test sets whose action words (AW) are included in the pre-defined top 40 common list. "Num.", "Pro.", and "Acc." are the abbreviations of the words "Number", "Proportion", and "Accuracy", respectively.

Dataset	Test Set	With Common AW		AWP Acc.
		Num.	Pro.	
JCSD	8,714	5,351	61.41%	72.42%
PCSD	19,066	11,583	60.75%	57.35%
Average			61.08%	64.89%



```
@ Override
  public int hashCode() {
    return type << _NUM | value.hashCode() << _NUM | otherValue.hashCode();
  }
```

(a) A Java code snippet c_2

1. Reference Summary: returns a hash code for this node.
2. Summary by ESALE-AWP: hashcode for this object.
3. Summary by ESALE: returns a hash code for this object.

(b) Summaries generated by different techniques for c_2

```
def betweenness_centrality(G, nodes):
  …
  betweenness = nx.betweenness_centrality(G, normalized=False, weight=None)
  …
  return betweenness
```

(c) A Python code snippet c_3

1. Reference Summary: compute betweenness centrality for nodes in a bipartite network.
2. Summary by ESALE w/o AWP: betweenness centrality for nodes in the graph.
3. Summary by ESALE: compute betweenness centrality for nodes in a bipartite graph.

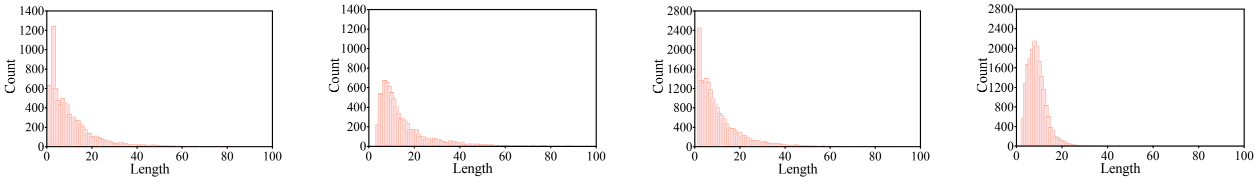
(d) Summaries generated by different techniques c_3

Fig. 6. Examples of AWP contributions

0 to 4 (the higher, the better) from four aspects: similarity (similarity of the generated summaries and the reference summaries), naturalness (grammaticality and fluency), informativeness (the amount of content carried over from the input code snippets to the generated summaries, ignoring fluency) and relevance (the degree to which the generated summaries are relevant with the input code snippets). We randomly select 100 code snippets, including 50 from the JCSD dataset and 50 from the PCSD dataset, the corresponding summaries generated by SiT, UniXcoder, and our ESALE, and the reference summaries (i.e., ground-truth), respectively. We divide the 100 samples into two groups, and each of them includes 50 samples, of which 25 belong to the JCSD dataset and 25 belong to the PCSD dataset. We place all samples to be evaluated in text files. Each participant is provided with two files containing 25 samples randomly selected from the test set of the JCSD and PCSD datasets, respectively. As shown in Fig. 9, for each sample, we present

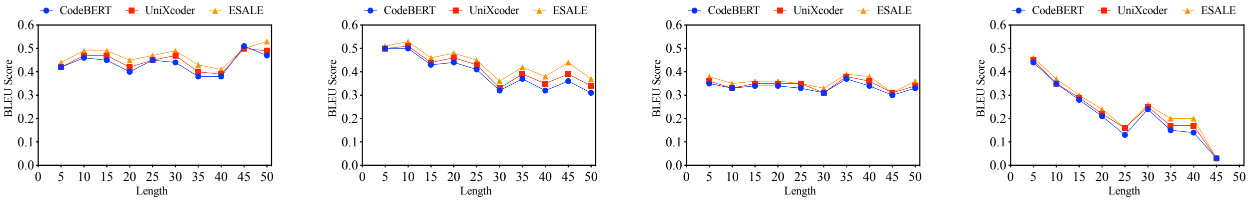
TABLE 12
Results of ESALE and ESALE w/o AWP on the samples whose action words are included in the top 40 and predicted correctly. In the "p-value" columns, the symbol "*" has the same meaning as that in Figure 5. # Improved denotes the number of the samples for which ESALE can generate higher quality summaries when correctly predicting their action words.

Metric	JCSD				PCSD			
	# Improved	ESALE w/o AWP	ESALE	p-value	# Improved	ESALE w/o AWP	ESALE	p-value
BLEU	890	21.53	35.70	****	2,236	23.77	39.66	****
METEOR	940	20.18	33.97	****	2,327	20.75	36.27	****
ROUGE-L	913	39.52	55.52	****	2,286	44.66	62.56	****
Average	914	27.08	41.73	****	2,283	29.73	46.16	****



(a) Code in JCSD Test Set (b) Comments in JCSD Test Set (c) Code in PCSD Test Set (d) Comments in PCSD Test Set

Fig. 7. Length distribution of code snippets and comments in test sets



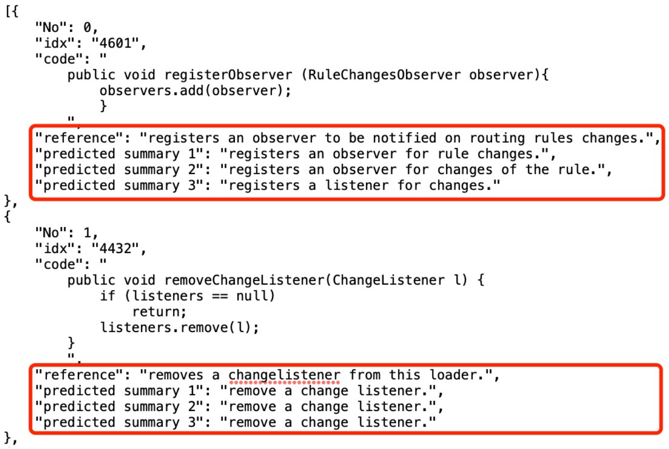
(a) Code in JCSD Test Set (b) Comments in JCSD Test Set (c) Code in PCSD Test Set (d) Comments in PCSD Test Set

Fig. 8. Effect of code snippet and comment length on the robustness of ESALE. The values shown in the figures are averaged results across length intervals (e.g., [0-5], [6-10], . . . , [46-50]) rather than specific lengths.

TABLE 13
Results of human evaluation. The first and second values in parentheses represent standard deviation and significant difference, respectively. The symbol "*" has the same meaning as that in Figure 5.

Dataset	Metrics	SiT	UniXcoder	ESALE
JCSD	Similarity	2.06 (0.51, ****)	2.35 (0.54, *)	**2.50 (0.48)**
	Naturalness	3.06 (0.58, **)	3.21 (0.53, ns)	**3.30 (0.55)**
	Informativeness	2.44 (0.51, ****)	2.79 (0.54, **)	**2.98 (0.53)**
	Relevance	2.34 (0.64, ****)	2.68 (0.63, ***)	**2.89 (0.60)**
	Average	2.48	2.76	**2.92**
PCSD	Similarity	2.01 (0.53, ****)	2.46 (0.49, *)	**2.58 (0.43)**
	Naturalness	3.09 (0.65, ****)	3.37 (0.54, ns)	**3.40 (0.55)**
	Informativeness	2.51 (0.59, ****)	2.92 (0.53, *)	**3.05 (0.55)**
	Relevance	2.24 (0.69, ****)	2.70 (0.61, **)	**2.84 (0.60)**
	Average	2.46	2.89	**2.97**

7 attributes, including Number (No), the index of the sample in the test set (idx), code snippet to be summarized (code), reference summary (reference), and summaries generated by three code summarization techniques (predicted summary 1, predicted summary 2, and predicted summary 3). To prevent response bias, the specific code summarization techniques that predicted summaries 1–3 are intransparent to participants. The three techniques generating predicted summaries 1–3 do not have fixed correspondences with SiT, UniXCoder, and our ESALE. In other words, any summary seen by participants could be generated by any of the three techniques. To reduce the workload of volunteers and ensure the fairness of experimental results, each volunteer randomly evaluates only one group of samples. Each summary is evaluated by five volunteers, and the final score is the average of them.



```
[{
    "No": 0,
    "idx": "4601",
    "code": "
        public void registerObserver (RuleChangesObserver observer){
            observers.add(observer);
            }
        ",
    "reference": "registers an observer to be notified on routing rules changes.",
    "predicted summary 1": "registers an observer for rule changes.",
    "predicted summary 2": "registers an observer for changes of the rule.",
    "predicted summary 3": "registers a listener for changes."
},
{
    "No": 1,
    "idx": "4432",
    "code": "
        public void removeChangeListener(ChangeListener l) {
            if (listeners == null)
                return;
            listeners.remove(l);
        }
        ",
    "reference": "removes a changelistener from this loader.",
    "predicted summary 1": "remove a change listener.",
    "predicted summary 2": "remove a change listener.",
    "predicted summary 3": "remove a change listener."
},
```

Fig. 9. Example of interface of human evaluation

The results of the human evaluation are shown in TABLE 13. The standard deviations of all techniques (the first values in all parentheses) are small, indicating that their scores by humans are about the same degree of concentration [28]. From TABLE 13, it can be observed that overall our ESALE consistently outperforms SiT and UniXcoder in all four aspects. On the JCSD dataset, compared with SiT and UniXcoder, ESALE improves on average by 17.88% and 5.80% in four aspects, respectively, while on the PCSD dataset, ESALE improves on average by 20.51% and 3.67%, respectively.

In addition, we follow [80] and confirm the superiority of our ESALE using Wilcoxon signed-rank tests [76], [77], [81] for the human evaluation. Specifically, for each aspect, we perform the paired Wilcoxon-Mann-Whitney signed-rank test on all scores by humans for ESALE and each baseline (i.e., SiT or UniXcoder) at a significance level of 5%. The test results are presented in the second values of the parentheses in TABLE 13. For example, "****" in the SiT column of the second row indicates that there is an extremely significant difference between scores by humans for ESALE and SiT in terms of the similarity aspect. From TABLE 13, it can be observed that compared with SiT and UniXcoder, the summaries generated by ESALE are more significantly similar to the reference summaries. For the naturalness of generated summaries, ESALE and UniXcoder are significantly better than SiT, which means that both ESALE and UniXcoder can generate more grammatically fluent summaries. For the informativeness of generated summaries, ESALE is better than SiT and UniXcoder, which means that ESALE tends to generate summaries with comprehensive semantics. For the relevance aspect, ESALE significantly outperforms SiT and UniXcoder, which means the summaries generated by ESALE are more relevant to the input code snippets.

5 Case Study

In this section, we provide case studies to understand the generated summaries of ESALE compared to SiT, CodeBERT, and UniXcoder.

Fig. 10 shows two real-world examples c_4 and c_5 from the test sets of the JCSD and PCSD datasets, respectively. Fig. 10(b) shows the reference summary of c_4 (line 1) and summaries generated by SiT, CodeBERT, UniXcoder, and our ESALE for c_4 (lines 2-5). According to the grammar rules in natural language, we can simply divide the reference summary into three parts: "expand" (Blue font), "all paths" (Red), and "in the tree" (Orange font). It can be observed, compared with the reference summary, 1) both SiT and CodeBERT only correctly cover the first part (i.e., "expand"); 2) UniXcoder can cover the last two parts (i.e., "all paths" and "in the tree"); 3) our ESALE can successfully cover all three parts. Similarly, for the python code snippet c_5 in Fig. 10(c), our ESALE can successfully cover all three parts of the reference summary, as shown in Fig. 10(d), while SiT only covers the first part (i.e., "search for") and CodeBERT and UniXcoder can cover the first two parts (i.e., "search for" and "artists"). In summary, based on the above two examples and observations, it can be concluded that our ESALE outperforms SiT, CodeBERT, and UniXcoder in learning the mapping between code snippets and summaries, and has more a powerful code summarization performance. Due to the page limit, please refer to our project website [79] for more case studies.

6 Threats to Validity

There are three main threats to validity.

First, we cannot guarantee that the scores of human evaluation are fair. To mitigate this threat, we evaluate every generated summary by five evaluators and use the average score of the five evaluators as the final result. In addition, the standard deviations of all techniques are small (less than 0.7), indicating that scores by humans are about the same degree of concentration (detailed in Section 4.2.4).

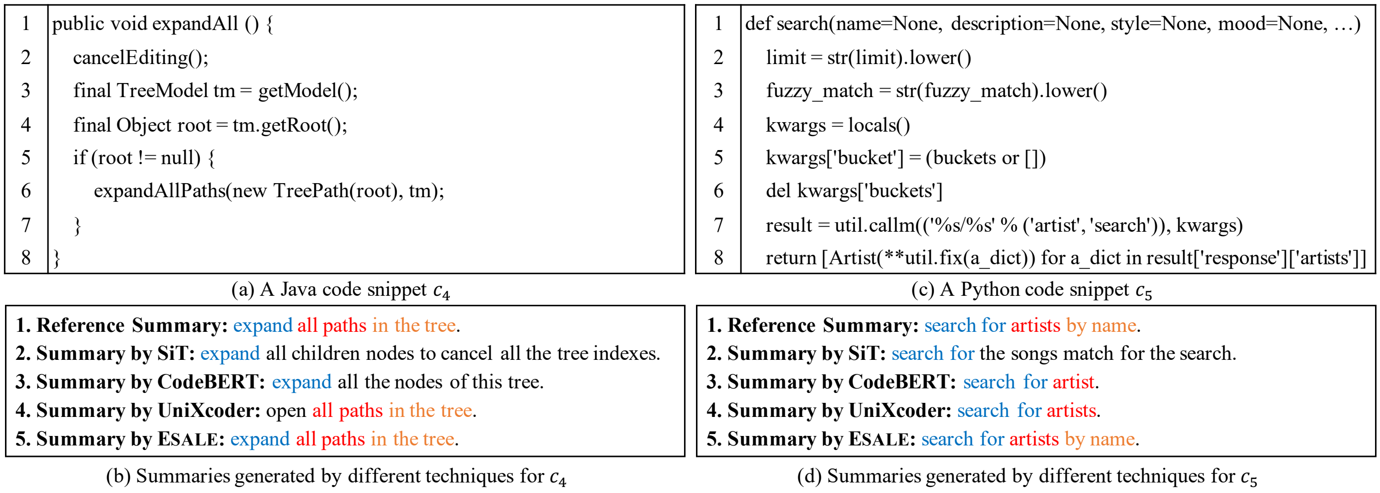
Second, in neural network model design, there are many orthogonal aspects such as different token embeddings, whether to use beam search or teacher forcing. When showing the generality of ESALE, we have done the experiments in a controlled way. A future work is to do all experiments in a more controlled way, and the performance of ESALE could rise further when combined with all other orthogonal techniques.

Third, we use datasets across six programming languages to validate the effectiveness of ESALE we proposed in this paper. Although ESALE only takes token sequences of code snippets as input and does not require other complex code features (e.g., AST and CFG), we do not know whether ESALE is equally applicable to other programming language summarization tasks. Therefore, it is necessary to conduct experiments on more programming language datasets (e.g., C/C++) to verify the reliability of ESALE.

7 Related Work

Code summaries can help developers quickly understand the functionalities of the program. More and more researchers are exploring code summarization techniques, which aim to automatically generate code summaries.

Nowadays, NMT-based models have been widely used to generate summaries for code snippets with encoder-decoder neural networks [4], [26], [30], [37], [39], [46], [61], [73], [82], [83], [84], [85]. For example, Iyer et al. [30] are the first to apply deep learning to automatic code summarization. They adopt LSTM networks [56] with attention to leverage the code vectors and generate natural language sentences that describe C# code snippets and SQL queries. Hu et al. [12] use one additional encoder to encode API sequences and improve the summary generation by learning the API knowledge. Subsequently, various additional information is incorporated to further improve DL-based code summarization performance, such as abstract syntax trees (ASTs) [37], [38], [46], [61], [69], [73], [83], value flows [86], data flow graph [87], code property graphs [88], similar code snippets [28], [29], important code statements [89], [90], file context [91], project-specific knowledge [92], etc. In addition, with the success of the pre-training and fine-tuning paradigm in the field of NLP (e.g., BERT [31] and T5 [32]), many works have introduced this paradigm to further boost neural code summarization, such as CodeBERT [33], CodeT5 [34], and UniXcoder [35]. These works first pre-train a model with general language modeling tasks, such as MLM and ULM. Then, they fine-tune the pre-trained models on code summarization [33], [34], [35]. However, although existing pre-trained models have achieved significant progress in general code feature learning, they are still insufficient in learning the code-summary alignment.



```java
public void expandAll () {
    cancelEditing();
    final TreeModel tm = getModel();
    final Object root = tm.getRoot();
    if (root != null) {
        expandAllPaths(new TreePath(root), tm);
    }
}
```

(a) A Java code snippet c_4

1. Reference Summary: expand all paths in the tree.
2. Summary by SiT: expand all children nodes to cancel all the tree indexes.
3. Summary by CodeBERT: expand all the nodes of this tree.
4. Summary by UniXcoder: open all paths in the tree.
5. Summary by ESALE: expand all paths in the tree.

(b) Summaries generated by different techniques for c_4

```python
def search(name=None, description=None, style=None, mood=None, …)
    limit = str(limit).lower()
    fuzzy_match = str(fuzzy_match).lower()
    kwargs = locals()
    kwargs['bucket'] = (buckets or [])
    del kwargs['buckets']
    result = util.callm(('%s/%s' % ('artist', 'search')), kwargs)
    return [Artist(**util.fix(a_dict)) for a_dict in result['response']['artists']]
```

(c) A Python code snippet c_5

1. Reference Summary: search for artists by name.
2. Summary by SiT: search for the songs match for the search.
3. Summary by CodeBERT: search for artist.
4. Summary by UniXcoder: search for artists.
5. Summary by ESALE: search for artists by name.

(d) Summaries generated by different techniques for c_5

Fig. 10. c_4 is from the test set of the JCSD with sample number 793. c_5 is from the test set of the PCSD with sample number 9580.

Recently, with the success of large language models (LLMs) in NLP [93], [94], an increasing number of SE researchers have started integrating them into the resolution process of various SE tasks [95], [96], including code summarization tasks. For example, Ahmed et al. [97] investigate the effectiveness of few-shot training in adapting LLMs to code summarization and find that it can make Codex significantly outperform fine-tuned small pre-trained language models (PLMs) (e.g., CodeT5). Given the concern of potential code asset leakage when using commercial LLMs (e.g., GPT-3.5), Su et al. [98] utilize knowledge distillation technology to distill small models from LLMs (e.g., GPT-3.5). Their experimental findings reveal that the distilled small models can achieve comparable code summarization performance to LLMs. Gao et al. [99] investigate the optimal settings for in-context learning, including few-shot example selection methods, few-shot example order, and the number of few-shot examples. Their experimental results demonstrate that carefully designed few-shot examples can significantly improve LLMs' performance. Geng et al. [100] investigate the ability of LLMs to address multi-intent comment generation. Ahmed et al. [101] propose to enhance few-shot samples with semantic facts automatically extracted from the source code. Sun et al. [102] design some heuristic questions to collect the feedback of ChatGPT, thereby finding an appropriate prompt to guide ChatGPT to generate in-distribution code summaries. Some studies [103], [104], [105] have also investigated the applicability of Parameter-Efficient Fine-Tuning (PEFT) techniques in code summarization tasks.

Although LLMs have been widely researched and applied due to their powerful content generation capabilities, it is important to note that there is no free lunch. Firstly, extensively utilizing commercial LLMs (e.g., GPT-3.5 and GPT-4) for code summarization tasks is costly. It also poses a risk of data leakage, as sensitive code might need to be transmitted to external servers. In contrast, our model can be deployed locally, ensuring that sensitive code remains within the organization's secure environment. Secondly, deploying open-source LLMs independently is also expensive for users or organizations due to their significant computational requirements. The more advanced LLMs typically have more parameters, and to ensure these models can efficiently generate useful outputs, extensive and costly hardware (such as GPUs) is essential. Our model offers a cost-effective alternative that can be deployed with less expensive hardware. Furthermore, most LLMs are designed to be general-purpose (to support multiple downstream tasks) and may not perform optimally on specific downstream tasks without extensive fine-tuning. Despite the advancements in prompt engineering techniques, the performance of LLMs on code summarization tasks may not significantly surpass that of smaller specialized models trained using supervised learning. For example, the work by Ahmed et al. [97] demonstrates that the performance of the adapted LLM Codex on Python and PHP code summarization tasks is not significantly better than that of the smaller model CodeT5 [34]. Our model, however, is optimized specifically for code summarization, ensuring more reliable and accurate performance for this task. Finally, our small model allows for greater flexibility and customizability for specific scenario/domain requirements. This adaptability is particularly beneficial for users or organizations with unique needs that may not be fully met by general-purpose LLMs.

8 Conclusion

In this paper, we propose an approach for code summarization, namely ESALE, which improves the code summarization performance by enhancing the encoder to code-summary alignment. ESALE is first trained using a multi-task learning paradigm with three summary-focused tasks, and then fine-tuned on the code summarization task. We conduct quantitative comprehensive experiments and qualitative human evaluations to verify the effectiveness of ESALE. And all results show that our ESALE is significantly better than state-of-the-art baselines.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. The authors thank Guanhong Tao (taog@purdue.edu) for his constructive comments and discussions on the manuscript. This work is supported partially by National Natural Science Foundation of China

(61932012, 62372228), and the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*. San Diego, California, USA: IEEE Computer Society, March 9-12 1981, pp. 215–223.

[2] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.

[3] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: a large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.

[4] 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*. Melbourne, Australia: IEEE, September 21-25 2020, pp. 746–757.

[5] 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*. Toronto, Ontario, Canada: IBM, October 24-28 1993, pp. 138–170.

[6] 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*. Coventry, UK: ACM, September 21-23 2005, pp. 68–75.

[7] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in PostgreSQL," in *Proceedings of the 3rd International Workshop on Mining Software Repositories*. Shanghai, China: ACM, May 22-23 2006, pp. 179–180.

[8] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "CPC: Automatically classifying and propagating natural language comments via program analysis," in *Proceedings of the 42nd International Conference on Software Engineering*. Seoul, South Korea: ACM, 27 June - 19 July 2020, pp. 1359–1371.

[9] 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," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 2, pp. 25:1–25:29, 2021.

[10] T. Tenny, "Procedures and comments vs. the banker's algorithm," *ACM SIGCSE Bulletin*, vol. 17, no. 3, pp. 44–53, 1985.

[11] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[12] 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*. Stockholm, Sweden: ijcai.org, July 13-19 2018, pp. 2269–2275.

[13] 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*. Beverly, MA, USA: IEEE Computer Society, 13-16 October 2010, pp. 35–44.

[14] 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*. San Francisco, CA, USA: IEEE Computer Society, 20-21 May 2013, pp. 13–22.

[15] P. Jayavardhan Reddy, Y. Ziyu, W. Zhen, and S. Huan, "A comprehensive study of staqc for deep code summarization," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. London, UK: ACM, August 19-23 2018, pp. 1–8.

[16] Y. Zhu and M. Pan, "Automatic code summarization: a systematic literature review," *CoRR*, vol. abs/1909.04352, pp. 1–12, 2019.

[17] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 23th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Minneapolis, MN, USA: Association for Computational Linguistics, June 2-7 2019, pp. 3931–3937.

[18] N. J. Abid, J. I. Maletic, and B. Sharif, "Using developer eye movements to externalize the mental model used in code summarization tasks," in *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*. Denver , CO, USA: ACM, June 25-28 2019, pp. 13:1–13:9.

[19] 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*. Seoul, Republic of Korea: ACM, July 13-15 2020, pp. 2–13.

[20] S. Haque, A. Bansal, L. Wu, and C. McMillan, "Action word prediction for neural source code summarization," in *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. Honolulu, HI, USA: IEEE, March 9-12 2021, pp. 330–341.

[21] 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*. Pittsburgh, USA: ACM, May 21–29 2022, pp. 1–12.

[22] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: ACM, August 23-28 2021, pp. 1105–1116.

[23] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. C. Gall, "Adversarial robustness of deep code comment generation," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 1–30, 2021.

[24] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the 25th International Conference on Automated Software Engineering*. Antwerp, Belgium: ACM, September 20-24 2010, pp. 43–52.

[25] 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*. Montreal, QC, Canada: IEEE Computer Society, March 2-6 2015, pp. 380–389.

[26] 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*. Gothenburg, Sweden: ACM, May 27-28 2018, pp. 200–210.

[27] E. Wong, J. Yang, and L. Tan, "AutoComment: mining question and answer sites for automatic comment generation," in *Proceedings of the 28th International Conference on Automated Software Engineering*. Silicon Valley, CA, USA: IEEE, November 11-15 2013, pp. 562–567.

[28] 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*. Melbourne, Australia: IEEE, November 15-19 2021, pp. 155–166.

[29] 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*. Melbourne, Australia: IEEE, September 21-25 2020, pp. 349–360.

[30] 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*. Berlin, Germany: The Association for Computer Linguistics, August 7-12 2016, pp. 2073–2083.

[31] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 23th Conference of the North Ameri-*

can Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis, MN, USA: Association for Computational Linguistics, June 2-7 2019, pp. 4171–4186.

[32] 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.

[33] 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*. Online Event: Association for Computational Linguistics, 16-20 November 2020, pp. 1536–1547.

[34] 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*. Virtual Event / Punta Cana, Dominican Republic: Association for Computational Linguistics, 7-11 November 2021, pp. 8696–8708.

[35] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: unified cross-modal pre-training for code representation," *CoRR*, vol. abs/2203.03850, pp. 1–14, 2022.

[36] R. Alec, N. Karthik, S. Tim, and S. Ilya, "Improving language understanding by generative pre-training," *OpenAI Tech Report*, pp. 1–12, 2018.

[37] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*. Montreal, QC, Canada: IEEE / ACM, May 25-31 2019, pp. 795–806.

[38] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-lstm," in *Proceedings of the 18th International Joint Conference on Neural Networks*. Budapest, Hungary: IEEE, July 14-19 2019, pp. 1–8.

[39] 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.

[40] M. Crawshaw, "Multi-task learning with deep neural networks: a survey," *CoRR*, vol. abs/2009.09796, pp. 1–43, 2020.

[41] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," in *Proceedings of the 57th Conference of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, 2019, pp. 4487–4496.

[42] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.

[43] T. W. W. Aung, Y. Wan, H. Huo, and Y. Sui, "Multi-triage: A multi-task learning framework for bug triage," *Journal of Systems and Software*, vol. 184, no. 111133, pp. 1–20, 2022.

[44] D. Wang, Y. Yu, S. Li, W. Dong, J. Wang, and Q. Liao, "MulCode: a multi-task learning approach for source code understanding," in *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. Honolulu, HI, USA: IEEE, March 9-12 2021, pp. 48–59.

[45] C. Fang, W. Sun, Y. Chen, X. Chen, Z. Wei, Q. Zhang, Y. You, B. Luo, Y. Liu, and Z. Chen, "Implementation code of ESALE." site: https://github.com/wssun/ESALE, 2024.

[46] 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*. Online Event: Association for Computational Linguistics, August 1-6 2021, pp. 1078–1090.

[47] 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*. Long Beach, CA, USA: Curran Associates Inc., December 4-9 2017, pp. 5998–6008.

[48] J. Vig and Y. Belinkov, "Analyzing the structure of attention in a transformer language model," in *Proceedings of the 4th ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Florence, Italy: Association for Computational Linguistics, August 1 2019, pp. 63–76.

[49] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the 3rd International Conference on Learning Representations*. San Diego, CA, USA: OpenReview.net, May 7-9 2015, pp. 1–15.

[50] Y. Belinkov and J. R. Glass, "Analysis methods in neural language processing: a survey," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 49–72, 2019.

[51] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: pre-training code representations with data flow," in *Proceedings of the 9th International Conference on Learning Representations*. Virtual Event, Austria: OpenReview.net, May 3-7 2021, pp. 1–12.

[52] 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.

[53] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th International Conference on Automated Software Engineering*. Melbourne, Australia: IEEE, 2020, pp. 473–485.

[54] 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*, Vancouver, BC, Canada, December 8-14 2019, pp. 13 042–13 054.

[55] R. Caruana, "Multitask learning," *Machine Learning*, vol. 28, no. 1, pp. 41–75, 1997.

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

[57] A. V. M. Barone and R. Sennrich, "A parallel corpus of Python functions and documentation strings for automated code documentation and code generation," in *Proceedings of the 8th International Joint Conference on Natural Language Processing*. Taipei, Taiwan: Asian Federation of Natural Language Processing, November 27 - December 1 2017, pp. 314–319.

[58] 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*. Seoul, South Korea: ACM, 27 June - 19 July 2020, pp. 1385–1397.

[59] 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*, Vancouver, BC, Canada, December 8-14 2019, pp. 6559–6569.

[60] 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*. Online: Association for Computational Linguistics, July 5-10 2020, pp. 4998–5007.

[61] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source code summarization with structural relative position guided transformer," *CoRR*, vol. abs/2202.06521, pp. 1–12, 2022.

[62] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 9th International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Athens, Greece: ACM, October 23-24 2019, pp. 143–153.

[63] P. Nie, J. Zhang, J. J. Li, R. J. Mooney, and M. Gligoric, "Impact of evaluation methodologies on code summarization," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*. Dublin, Ireland: Association for Computational Linguistics, May 22-27 2022, pp. 4936–4960.

[64] 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.

[65] 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*, virtual, December 2021, pp. 1–14.

[66] 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*. Philadelphia, PA, USA: ACL, July 6-12 2002, pp. 311–318.

[67] 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*. Ann Arbor, Michigan, USA: Association for Computational Linguistics, June 29 2005, pp. 65–72.

[68] 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*. Barcelona, Spain: Association for Computational Linguistics, July 21-26 2004, pp. 74–81.

[69] 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*. Montpellier, France: ACM/IEEE, September 3-7 2018, pp. 397–407.

[70] 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*. Madrid, Spain: IEEE, May 20-21 2021, pp. 1–12.

[71] 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," *IEEE Transactions on Software Engineering (Early Access)*, pp. 1–19, 2020.

[72] 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*. Madrid, Spain: IEEE, May 20-21 2021, pp. 253–264.

[73] 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*. Madrid, Spain: IEEE, May 20-21 2021, pp. 184–195.

[74] R. Shahbazi, R. Sharma, and F. H. Fard, "API2Com: on the improvement of automatically generated code comments using API documentations," in *Proceedings of the 29th International Conference on Program Comprehension*. Madrid, Spain: IEEE, May 20-21 2021, pp. 411–421.

[75] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," in *Proceedings of the 3th International Conference on Learning Representations – Poster*. San Diego, CA, USA: OpenReview.net, May 2015, pp. 1–15.

[76] 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.

[77] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*. Baltimore, MD, USA: ACM, July 12-17 2015, pp. 211–222.

[78] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore: ACM, November 14-18 2022, pp. 107–119.

[79] C. Fang, W. Sun, Y. Chen, X. Chen, Z. Wei, Q. Zhang, Y. You, B. Luo, Y. Liu, and Z. Chen, "ESALE," site: https://sites.google.com/view/esale4cs/home, 2024.

[80] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," in *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing*. Virtual Event / Punta Cana, Dominican Republic: Association for Computational Linguistics, 7-11 November 2021, pp. 4053–4062.

[81] F. Wilcoxon, S. Katti, and R. A. Wilcox, *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*. American Cyanamid Company, 1963.

[82] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "CoCoSum: contextual code summarization with multi-relational graph neural network," *CoRR*, vol. abs/2107.01933, pp. 1–24, 2021.

[83] Y. Gao and C. Lyu, "M2TS: multi-scale multi-modal approach based on transformer for source code summarization," *CoRR*, vol. abs/2203.09707, pp. 1–12, 2022.

[84] Y. Wang, Y. Dong, X. Lu, and A. Zhou, "GypSum: learning hybrid representations for code summarization," *CoRR*, vol. abs/2204.12916, pp. 1–12, 2022.

[85] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," *CoRR*, vol. abs/2201.05222, pp. 1–12, 2022.

[86] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2Vec: value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 233:1–233:27, 2020.

[87] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. R. Lyu, "Code structure-guided transformer for source code summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 23:1–23:32, 2023.

[88] 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, pp. 1–12, 2020.

[89] W. Sun, Y. Hu, Y. Xu, Y. Chen, and C. Fang, "Integrating extractive and abstractive models for code comment generation," in *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security*. Chiang Mai, Thailand: IEEE, October 22-26 2023, pp. 184–195.

[90] W. Sun, C. Fang, Y. Chen, Q. Zhang, G. Tao, Y. You, T. Han, Y. Ge, Y. Hu, B. Luo, and Z. Chen, "An extractive-and-abstractive framework for source code summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 75:1–75:39, 2024.

[91] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Proceedings of the 17th International Conference on Mining Software Repositories*. Seoul, Republic of Korea: ACM, 29-30 June 2020, pp. 300–310.

[92] R. Xie, T. Hu, W. Ye, and S. Zhang, "Low-resources project-specific code summarization," in *Proceedings of the 37th International Conference on Automated Software Engineering*. Rochester, MI, USA: ACM, October 10-14 2022, pp. 68:1–68:12.

[93] M. Du, F. He, N. Zou, D. Tao, and X. Hu, "Shortcut learning of large language models in natural language understanding," *Communications of the ACM*, vol. 67, no. 1, pp. 110–120, 2023.

[94] C. Qin, A. Zhang, Z. Zhang, J. Chen, M. Yasunaga, and D. Yang, "Is ChatGPT a general-purpose natural language processing task solver?" *arXiv preprint arXiv:2302.06476*, 2023.

[95] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering," *CoRR*, vol. abs/2312.15223, no. 1, pp. 1–57, 2023.

[96] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[97] T. Ahmed and P. T. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th International Conference on Automated Software Engineering*. Rochester, MI, USA: ACM, October 10-14 2022, pp. 177:1–177:5.

[98] C. Su and C. McMillan, "Distilled GPT for source code summarization," *Automated Software Engineering*, vol. 31, no. 1, p. 22, 2024.

[99] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *Proceedings of the 38th International Conference on Automated Software Engineering*. Luxembourg: IEEE, September 11-15 2023, pp. 761–773.

[100] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the 46th International Conference on Software Engineering*. Lisbon, Portugal: ACM, April 14-20 2024, pp. 39:1–39:13.

[101] T. A. andKunal Suresh Pai, P. Devanbu, and E. T. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the 46th International Conference on Software Engineering*. Lisbon, Portugal: ACM, April 14–20 2024, pp. 1–13.

[102] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, "Automatic code summarization via chatgpt: How far are we?" *CoRR*, vol. abs/2305.12865, pp. 1–13, 2023.

[103] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning

in code intelligence," in *Proceedings of the 30th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore, Singapore: ACM, November 14-18 2022, pp. 382–394.

[104] Y. Choi and J. Lee, "Codeprompt: Task-agnostic prefix tuning for program and language generation," in *Proceedings of the Findings of the 61st Association for Computational Linguistics*. Toronto, Canada: Association for Computational Linguistics, July 9-14 2023, pp. 5282–5297.

[105] W. Sun, C. Fang, Y. You, Y. Chen, Y. Liu, C. Wang, J. Zhang, Q. Zhang, H. Qian, W. Zhao *et al.*, "A prompt learning framework for source code summarization," *arXiv preprint arXiv:2312.16066*, 2023.

Chunrong Fang (Member, IEEE) received the B.E. and Ph.D. degrees in software engineering from Software Institute, Nanjing University, Jiangsu, China. He is currently an associate professor with the Software Institute of Nanjing University. His research interests lie in intelligent software engineering, e.g. BigCode and AITesting.

Weisong Sun is currently a research fellow at the College of Computing and Data Science, Nanyang Technological University, Singapore. He received a Ph.D. degree in Software Engineering from Nanjing University, China in 2023. His research interests include intelligent software engineering, trustworthy artificial intelligence (especially AI model security), and research spanning both fields (e.g., trustworthy intelligent software engineering). He has more than 30 high-quality publications including TDSC, TSE, TOSEM, ICSE, ESEC/FSE, ASE, ACL, etc. He served as the reviewer of TSE, TOSEM, ACL, NeurIPS, TR, IJHC, QRS, etc. In addition, he served as the co-chair of the International Workshop on AI Reliability and Security (AIRS 2024).

Yuchen Chen is currently working the Ph.D. degree in the Software Institute at Nanjing University, China. His current research interests include intelligent software engineering and code model security.

Xiao Chen is currently working at Huawei, China. He received an M.S. degree in Software Engineering from Nanjing University, China in 2023. His current research interests lie in intelligent software engineering.

Zhao Wei is currently an R&D efficiency expert and head of Code Intelligence working at Tencent. His research interests are in AI for software engineering, including LLM for code, code generation, code search & navigation, code review, etc.

Quanjun Zhang is currently working toward the Ph.D. degree in Software Institute at Nanjing University, Nanjing, China. His current research interests include intelligent software testing and automated program repair.

Yudu You is currently working at Meituan, China. He received an M.S. degree in Software Engineering from Nanjing University, China in 2024. His current research interests lie in intelligent software engineering.

Bin Luo is a full professor with the Software Institute, Nanjing University. He is also a member of the National Key Laboratory for Novel Software Technology (Nanjing University). His main research interests include cloud computing, computer network, decentralized computing and edge computing, services computing, natural language processing and intelligent software engineering, machine learning and deep learning. His research results have been published in more than 90 papers in international journals and conference proceedings including IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Mobile Computing, ACM Transactions on Knowledge Discovery from Data, IEEE Transactions on Services Computing, Computer Networks, Journal of Parallel and Distributed Computing, Future Generation Computer Systems, Journal of Systems and Software, Informatic Science, Journal of Network and Computer Applications, Expert Systems with Applications, ICSE, ESEC/FSE, ASE, EMNLP, GlobeCom etc.

Yang Liu is a full professor and University Leadership Forum Chair, College of Computing and Data Science, Nanyang Technological University. His current research interests are related to Cybersecurity, Software Engineering, and Artificial Intelligence. He is also the Programme Director for HP-NTU Digital Manufacturing Corp Lab, Deputy Director of the National Satellite of Excellence of Singapore, and Cluster Director in Cybersecurity, Energy Research Institute @NTU.

Zhenyu Chen (Member, IEEE) is currently a full professor with Software Institute, Nanjing University, China. He is an Associate Editor of IEEE Transactions on Reliability. He is also the Contest Co-Chair at QRS 2018, ICST 2019, and ISSTA 2019. He is the Industrial Track Co-Chair of SANER 2019. He specializes in software testing. His research interests include collective intelligence, deep learning testing and optimization, big data quality, and intelligent software engineering.