Empirical Software Engineering (2024) 29:162
https://doi.org/10.1007/s10664-024-10553-6

A review of automatic source code summarization

Xuejun Zhang[1] · Xia Hou[1] · Xiuming Qiao[1] · Wenfeng Song[1]

Accepted: 23 September 2024 / Published online: 7 October 2024

Abstract
Code summarization plays a pivotal role in the field of software engineering by offering developers a concise natural language comprehension of source code semantics. As software complexity continues to escalate, code summarization confronts various challenges, including discrepancies between source code and summarization, the absence of crucial or up-to-date information, and the inefficiency and resource demands of manual summarization. To address these challenges, Automatic Source Code Summarization (ASCS) has garnered widespread attention. This paper presents a comprehensive review and synthesis of ASCS research. It aims to provide an in-depth understanding of the core issues and challenges inherent in each phase of ASCS, illustrated with specific examples and application scenarios. Around of the core phases of ASCS including data collection, source code modeling, the generation of code summaries, and the assessment of their quality, the paper thoroughly compiles and assesses existing datasets, categorizes and examines prevalent source code modeling techniques, and delves into the methods for generating and evaluating the quality of code summaries. Concluding with an exploration of future research avenues and emerging trends, this paper serves as a guide for readers to grasp the cutting-edge developments in this field, enriched by the analysis of pivotal research contributions.

Keywords Automatic source code summarization · Data collection · Code modeling · Quality evaluation

1 Introduction

Code summarization are concise textual descriptions in natural language that elucidate the functionality and logic of source code. They play a pivotal role in the software development

Communicated by: Xia Hou

✉ Xuejun Zhang
zhangxuejun0828@bistu.edu.cn

Xia Hou
houxia@bistu.edu.cn

Xiuming Qiao
qiaoxiuming@bistu.edu.cn

Wenfeng Song
songwenfenga@163.com

[1] Beijing Information Science and Technology University, Beijing, China

process, aiding developers in comprehending, maintaining, and collaborating on project code (Xia et al. 2018). High-quality code summarization can significantly enhance program comprehension efficiency, reduce maintenance costs, and mitigate errors during development (Panichella 2018). However, in real-world scenarios, many projects lack detailed and meaningful code summarization or comments. Manual writing of code comments is not only costly but also time-consuming, posing a significant challenge for developers (Hu et al. 2018b). This is an issue that demands our attention and resolution.

To address this issue, automatic source code summarization (ASCS) technology has emerged. This technology can autonomously generate supplementary text for code, providing clear explanations of code functionality, logic, and purpose. Consequently, it elevates the quality and efficiency of software development. ASCS technology plays a pivotal role in the following aspects:

Enhancing collaboration efficiency: In multi-person collaborative software development projects, code summarization facilitates team members in swiftly comprehending and integrating each other's work, thereby reducing communication costs and development time.

Lowering the learning curve: Newly onboarded developers can expedite their familiarization with the codebase and understand its purpose and design more rapidly through code summarization, expediting their workflow.

Augmenting code maintenance efficiency: As codebases evolve and expand over time, high-quality code summarization assists maintainers to identify and resolve issues, thereby reducing the risk of errors.

Automating documentation generation: ASCS can be used for automated documentation generation, producing clear and consistent project documentation, thereby relieving the burden of manual documentation creation.

Mitigating the risk of knowledge loss: When original developers depart from a project or company, there's a risk of knowledge loss. Code summarization help preserve critical information, thereby reducing this risk.

Hence, ASCS not only elevates software quality but also contributes to more efficient software development, maintenance, and project management. It addresses the challenges posed by the absence of detailed code summarization and holds significant practical application potential.

ASCS is a task that converts source code into natural language descriptions (Gros et al. 2020). The research on this task started with the information retrieval-based method by Haiduc et al. (2010a) in 2010. Early methods used template-based or information retrieval-based approaches, which extracted key information from code by using heuristic rules and synthesized code summarization. In recent years, deep learning-based methods have improved the quality of code summarization and become a mainstream direction for this task, thanks to the rapid development of deep learning technology and the availability of code comment corpora. Iyer et al. (2016) were the first to propose the CODE-NN method based on the sequence-to-sequence model in 2016. Later research mostly followed the advances in neural machine translation, focusing on how to learn the structural information within code and the implicit relationship with natural language descriptions, and achieved good results.

ASCS is an emerging field of study with the potential to enhance software development, program comprehension, and maintenance. This research is a multidisciplinary, cross-domain challenge that encompasses various fields, including software engineering, program analysis, deep learning, information retrieval, natural language processing, and more. Despite nearly a decade of investigation, it continues to grapple with several significant challenges:

Source code modeling: Source code differs from regular text; it contains intricate syntax and structural information, and each programming language adheres to its specific rules and

conventions. Consequently, unique analytical methods and modeling standards are essential for effective source code processing. Moreover, diverse coding logic and naming conventions among different developers contribute to variations in code style and vocabulary, thereby amplifying the challenges in generating code summarization

Code summarization generation: Early approaches relied on information retrieval-based techniques to extract keywords from source code or locate similar code summarization. The effectiveness of these methods often hinges on the quality of the dataset. With advancements in artificial intelligence, researchers have increasingly adopted natural language processing techniques, such as sequence-to-sequence models, which have made significant strides and become the primary approach to addressing this problem. Nonetheless, these methods still grapple with issues like long-term dependencies, data scarcity, and the absence of high-quality training datasets.

Code summarization evaluation: Current evaluation methods primarily rely on machine translation evaluation metrics like BLEU and ROUGE. However, these metrics fall short in providing a comprehensive evaluation of code summarization quality due to the substantial differences between source code and natural language text. Additionally, the lack of standardized datasets and evaluation criteria complicates comparisons across different research efforts. Therefore, there is a pressing need for an efficient, cost-effective code summarization evaluation method to accurately gauge the quality of generated code summarization and drive advancements in research and practical applications.

To comprehensively and deeply understand the research field of automatic code summarization, this paper adopts a systematic literature retrieval strategy aimed at broadly covering mainstream academic resources in computer science and software engineering. Specifically, this paper conducted thorough literature searches in authoritative databases such as Google Scholar, IEEE Xplore, and ACM Digital Library, carefully selecting research outcomes closely related to the task of automatic code summarization. In the retrieval process, keywords such as "automatic code summarization" and "source code summarization" were employed, and Boolean operators were utilized to optimize the search scope, ensuring that the retrieval results were both targeted and sufficiently broad.

In terms of the selection of the time range, to focus on the latest research trends in this field, this paper limits the search scope to literature published within the past ten years (up to June 2024). Meanwhile, to ensure the comprehensiveness and depth of the review, we have also prudently incorporated some classic early research works to provide historical background and perspectives on research evolution. After the initial retrieval, this paper further conducted rigorous secondary screening. The main criteria for screening included the relevance of the literature title and abstract to the task of automatic code summarization, whether it had undergone peer review to ensure academic quality, the number of citations and its influence within the field, as well as the clarity and credibility of research methods and experimental results.

Through this series of rigorous screening processes, this paper ultimately identified a set of high-quality core literature. This literature provides a solid foundation for subsequent analysis and discussion, aiming to better elaborate and explain the research status, challenges faced, and future development trends in the field of automatic code summarization, while also providing references and insights for future research. The related work introduced in the main body of this paper is summarized in Table 12 in the appendix. This table provides a detailed list of code summarization-related work mentioned in the main text, systematically categorized and organized according to source code modeling methods, code summarization methods, and evaluation methods.

This paper conducted a cumulative annual statistics on the selected papers and classified them in detail according to different research methods, as shown in Fig. 1. Since 2009, research work on the topic of code summarization has gradually increased, and this trend has become more pronounced over time. In the early stages, code summarization methods mainly focused on retrieval-based and template-based approaches. However, since 2016, thanks to the rapid development of deep learning technology, deep learning methods have begun to emerge and gradually become the mainstream in the field of code summarization, continuously growing and evolving along with technological advancements. This visual analysis provides strong support for us to intuitively understand the development dynamics of different research methods over time.

Currently, there is still a relative scarcity of comprehensive reviews regarding automatic code summarization tasks. Nazar et al. (2016a) conducted an in-depth exploration of methods for summarizing software artifacts. They analyzed the working content across different software artifacts, unveiling the challenges and opportunities faced in the current landscape of software artifact summarization. Meanwhile, Bai et al. (2019) focused on investigating issues related to the generation of code comments. Through exhaustive investigation and analysis, they elucidated the significant role that code comment generation plays in enhancing code readability and maintainability. Furthermore, Chen and Monperrus (2019) conducted a systematic study on the types of source code embeddings, providing an important theoretical foundation for the representation and learning of source code. In the realm of algorithms and techniques for code summarization, the research conducted by Song et al. (2019) and Zhang et al. (2022a) is particularly noteworthy. They not only investigated various existing code summarization algorithms but also comprehensively evaluated the performance and applicability of these algorithms, offering valuable insights for future research endeavors.

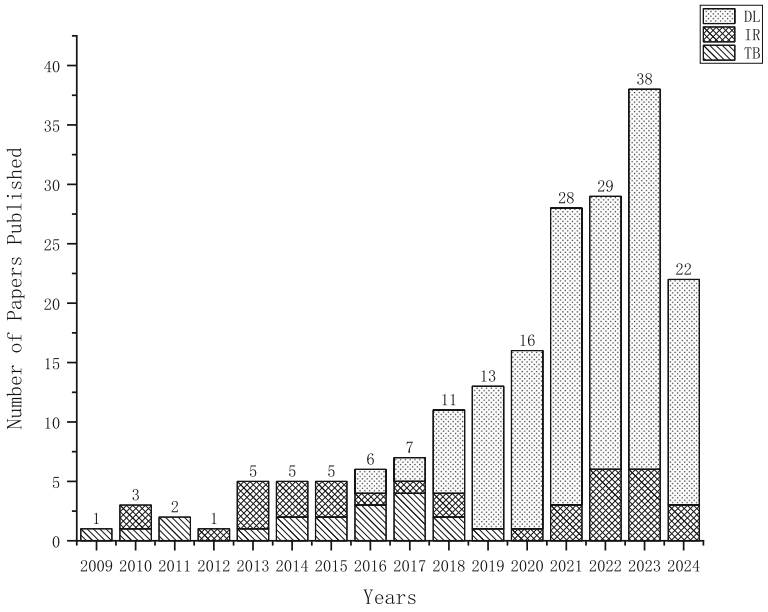

Fig. 1 Number of relevant papers published per year. TB represents template-based methods, IR represents information retrieval-based methods, and DL represents deep learning-based methods

However, despite providing valuable insights, these studies have not fully captured the latest advancements in the field of automatic code summarization, particularly the breakthroughs in deep learning technology. Compared to existing reviews, this paper exhibits several unique strengths. Firstly, it not only comprehensively covers the four key dimensions of dataset analysis, source code modeling, automatic code summarization algorithms, and quality assessment, but also emphasizes the intrinsic connections between these dimensions. This multi-dimensional and systematic analysis offers a broader perspective for comprehensively understanding the overall development of the code summarization task. Secondly, in terms of dataset analysis, this paper conducts a thorough exploration, detailing the evolution process of datasets, expansion situations, and coverage of support for multiple programming languages, systematically summarizing the development trends of these datasets, which is rare in current literature. Furthermore, this paper incorporates the latest research findings in this field in recent years, making the review content more cutting-edge and timely. Lastly, this paper also provides a detailed analysis of the key challenges currently faced in the field of code summarization and proposes several potential research directions for future studies, offering guiding suggestions for the further development of this field.

The paper is structured as follows: Section 2 introduces the general workflow and key steps of automatic code summarization algorithms. In Section 3, we gather and analyze datasets available for code summarization tasks. Section 4 provides a comprehensive summarization of source code modeling methods, including their advantages and disadvantages. Section 5 reviews representative code summarization generation algorithms, highlighting their limitations and potential improvements. Section 6 introduces quality evaluation metrics for code summarization. Finally, in Section 7, we discuss future research directions in the field of code summarization.

2 Overview of Automated Code Summarization Tasks

Code is the carrier of software functionality, and code summarization is an effective way to understand software-related content quickly. ASCS is a task that converts source code into natural language descriptions, which can help program developers better understand and maintain code. This task can be divided into two subtasks: method-level summarization generation and class-level summarization generation. Method-level summarization generation is to generate a short description for a method, explaining its function and usage (Rai et al. 2017). Class-level summarization generation is to generate a short description for a class, explaining its role and attributes (Li et al. 2023c).

Generally speaking, code comments can be divided into three types according to the position of the comment: document comments, block comments, and line comments (Fluri et al. 2007). Document comments are used to explain the role and usage of classes, methods, or attributes; block comments are used to explain the function and logic of a piece of code, located before the commented code; line comments are used to describe the meaning and function of a single line of code, located after the commented code. Most of the research focuses on document comments and block comments, while class-level summarization generation is relatively less common (Li et al. 2023c).

As shown in Fig. 2, the overall process of an ASCS task generally consists of four steps: data collection, source code modeling, code summarization generation, and quality evaluation.

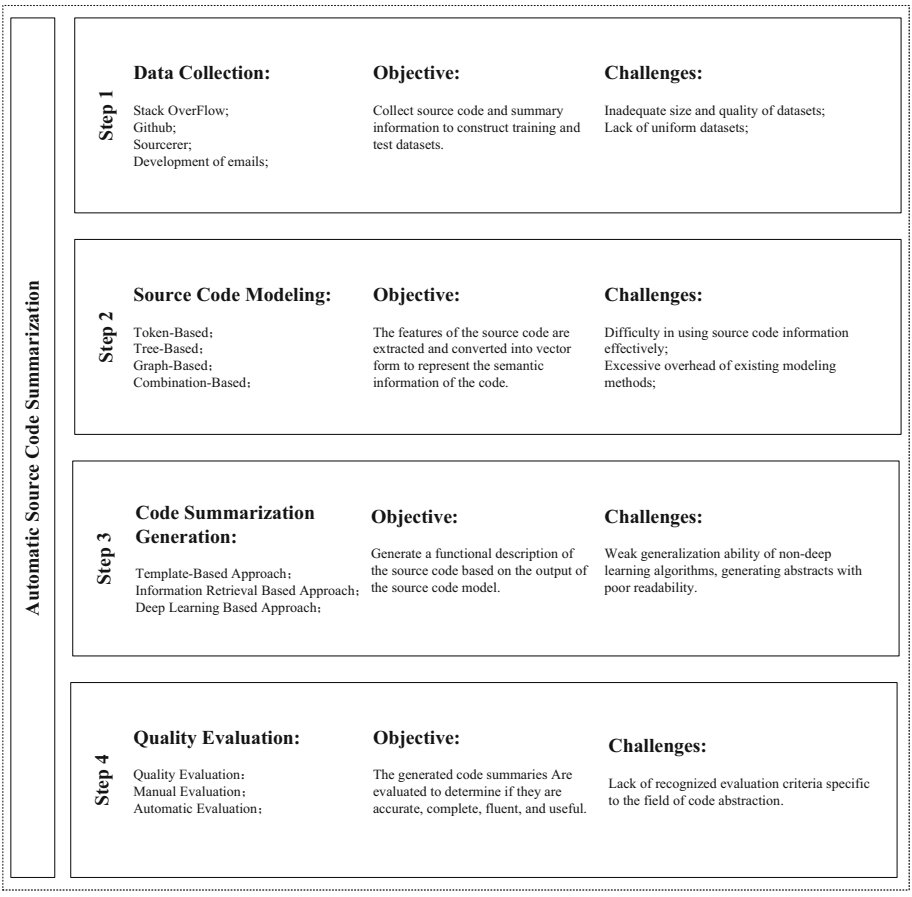

Fig. 2 The basic process of the code summarization task, which consists of four steps: data collection, source code modeling, code summarization generation algorithm, and quality evaluation. Each step has its own goals and challenges

2.1 Data Collection

Data collection stands as a pivotal process involving the aggregation of source code and pertinent summarization data from diverse sources and channels to create datasets for training and testing purposes. The success and performance enhancements of code summarization methods in contemporary research are intrinsically linked to the quality of these datasets. Beyond serving as essential material for model training and evaluation, these datasets mirror the real-world demand for code summarization. They offer valuable insights into the intricacies of different programming languages, domains, and annotation styles, aiding in the more effective handling of these challenges.

Particularly in the realm of deep learning-based methodologies, annotation data is deemed crucial for boosting model performance. Presently, it's a common consensus among researchers that there's a pressing need for more high-quality, diverse, and representative datasets to bolster the continuous research in code summarization. This is a universally acknowledged and urgent issue in the research community (Song et al. 2023).

2.2 Source Code Modeling

Source code modeling is a critical step, tasked with extracting key features from source code and transforming them into vector form to effectively convey the semantic information of the code, thus providing input data for code summarization generation. In the area of code summarization tasks, choosing the appropriate source code modeling method is paramount because it directly impacts our ability to fully harness the embedded information within the source code (Tufano et al. 2018).

For instance, a common source code modeling approach involves the use of word embedding techniques, mapping words and identifiers from the source code into high-dimensional vector spaces (Zheng et al. 2017). This approach enables the model to comprehend the semantic meaning of the code, such as understanding relationships between variables, functions, and classes. Additionally, graph neural networks serve as a potent source code modeling tool, representing source code as graphical structures to better capture dependencies and hierarchical structures within the code (LeClair et al. 2020).

However, it's worth emphasizing that source code modeling encompasses not only technical choices but also how to handle the diversity and complexity of source code. Different programming languages, domains, and styles may necessitate varying modeling methods. Furthermore, external knowledge repositories like API documentation (Hu et al. 2018b) and developer communities can be utilized to enhance source code representation, thereby improving the quality and accuracy of code summarization .

In conclusion, source code modeling methods play a pivotal role in code summarization tasks. They assist in better comprehending and leveraging the semantic information embedded within source code, addressing the complexities and diversities within code summarization, and providing essential support for the successful execution of the task.

2.3 Code Summarization Generation

The core of the code summarization task lies in its generation algorithms. These algorithms generate functional descriptions of the code based on the output of the source code model. In the early stages of deep learning technology, researchers primarily used rule-based and template-based methods to generate code summarization (Rai et al. 2017). However, these methods have limitations in handling complex source code and generating accurate summarization.

With the continuous advancement of deep learning technology, models based on neural networks have gradually become the main method. These methods typically use an encoder-decoder framework to facilitate the conversion from source code vectors to natural language descriptions. For example, the sequence-to-sequence (Seq2Seq) model is a common encoder-decoder structure that can handle input and output sequences of various lengths, and is therefore widely used in code summarization tasks (Zeng et al. 2018).

It's worth noting that models based on neural networks are increasingly dominating the field of code summarization. However, these methods also encounter a series of challenges, such as effectively handling long-distance dependencies, dealing with sparse tokens, and handling unknown tokens (Ahmad et al. 2020).

In essence, generation algorithms hold a pivotal role in code summarization tasks, acting as a crucial element in converting source code into meaningful descriptions. Gaining a profound understanding of the operational principles and challenges associated with these algorithms

can lead to significant improvements in the quality and precision of code summarization. This, in turn, can add substantial value to the software development process.

2.4 Quality Evaluation

Quality evaluation involves evaluating generated code summarization to determine their accuracy, completeness, fluency, and utility. This evaluation process is crucial for researchers as it aids in model selection, identifying strengths and weaknesses, and suggesting areas for improvement. However, the field of quality evaluation for code summarization is currently relatively underdeveloped and faces challenges such as how to precisely define and measure quality, how to collect and annotate reference summarization , and how to incorporate human evaluation (Nie et al. 2022).

Presently, there are several methods for assessing the quality of generated code summarization. These include automated evaluation metrics such as BLEU (Papineni et al. 2002), ROUGE (Lin 2004), and METEOR (Banerjee and Lavie 2005), which use automated methods to compare the similarity between generated summarization and reference summarization . These metrics provide quick assessments but may sometimes overlook subtle differences in semantics and fluency. Therefore, human evaluation is also a common evaluation method, where human reviewers judge whether the generated summarization meet expected quality standards (Shen et al. 2021). However, human evaluation can be time-consuming, labor-intensive, and subjective.

In conclusion, quality evaluation is a critical aspect of the code summarization task, but it still requires further research to address related challenges. Automated evaluation metrics and human evaluation each have their advantages and disadvantages (Roy et al. 2021). Therefore, a comprehensive approach that combines these methods may currently be the best evaluation strategy to ensure that generated code summarization achieve high-quality standards in terms of accuracy and fluency, among other aspects.

In this paper, we provide a detailed description of the four basic tasks of ASCS, and analyze the strengths, weaknesses, and challenges of existing approaches, as well as look at possible future research directions.

3 Datasets Analysis

In order to train and evaluate ASCS models, large-scale and high-quality training datasets are required. Most of the currently available datasets are built from programming quiz sites (e.g Stack Overflow), code contests, and code hosting sites (e.g., GitHub). This section analyzes these datasets to compare and evaluate them in terms of both time and content dimensions. Figure 3 illustrates the developmental lineage of the dataset for the code summarization task.

3.1 Evolution of Code Summarization Datasets Over Time

From a chronological perspective, the development and evolution of these datasets are closely intertwined with the advancement of ASCS.

The development of code summarization datasets can be traced back to 2014 when (McBurney and McMillan 2014) created the first dataset for evaluating information retrieval-based code summarization methods, marking the beginning of this research field. With the rapid advancement of deep learning technology, neural network-based code summarization

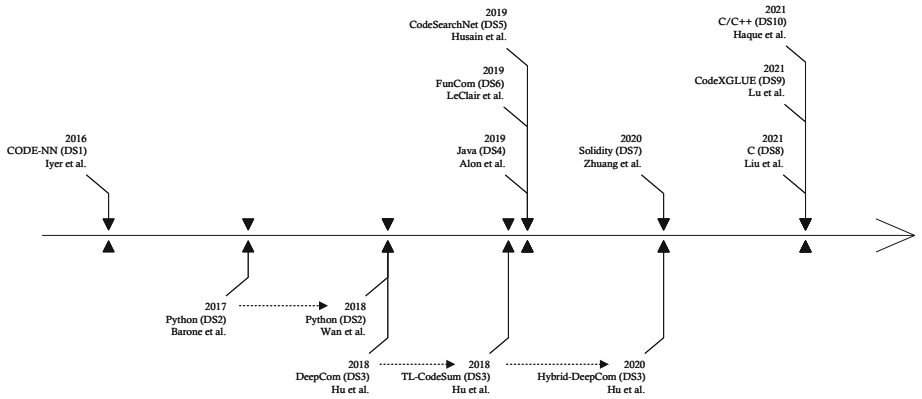

Fig. 3 The developmental lineage of the code summarization task datasets

models began to gain widespread attention. In 2016, Iyer et al. (2016) released the CODE-NN dataset, which includes C# and SQL code snippets from Stack Overflow and their corresponding question titles, providing valuable resources for training neural network models.

By 2018, Hu et al. (2018a) had created the DeepCom dataset, constructed using Java project methods and their Javadoc comments from GitHub. In the same year, Hu et al. (2018b) expanded the dataset by introducing API sequences, laying a solid foundation for subsequent research on automatic code summarization. Two years later, Hu et al. (2020) updated the DeepCom dataset, increasing its size and manually cleaning it, significantly improving the dataset's quality. Additionally, in 2019, LeClair et al. (2019) released the FunCom dataset, which covers more brief Java methods and comments through function and project partitioning, further enriching research resources. Alon et al. (2019a) also constructed three Java datasets of varying sizes based on the number of projects and examples to evaluate model performance under different scales.

To cover more programming languages, especially widely used languages with rich docstring documentation like Python, researchers began exploring possibilities from diverse data sources and task formats. In 2017, Barone and Sennrich (2017) introduced a neural network-based Python code generation model, accompanied by a dataset consisting of Python files and their docstring documentation from GitHub. Subsequently, in 2018, Wan et al. (2018) further processed the data by converting the code into continuous text and parsing it into an Abstract Syntax Tree (AST), providing more structured data for model training. In 2019, Husain et al. (2019) created the CodeSearchNet dataset, which covers functions in six programming languages and their corresponding docstring documentation or comments, greatly broadening the research horizon. Shortly after, in 2020, Zhuang et al. (2020) constructed a Solidity dataset using Ethereum and the VNT chain, supporting automatic code summarization research for this language. By 2021, Lu et al. (2021) proposed the CodeXGLUE dataset, which integrates and extends CodeSearchNet, further advancing research in the field of code intelligence. In the same year, Liu et al. (2021a) extracted C language programs from GitHub and performed deduplication processing to construct a specialized C language code summarization dataset. Meanwhile, Haque et al. (2021) extracted C/C++ projects from GitHub and used Eberhart et al. (2020)'s model to generate summaries, creating a C/C++ code summarization dataset.

In recent years, the issue of dataset quality has gradually emerged. In 2022, Shi et al. (2022c) introduced a classification system containing 12 types of data noise and developed an automated cleaning tool called CAT (Code-comment cleAning Tool) to detect and eliminate

noisy data. In the same year, Shi et al. (2022b) reclassified existing datasets and delved into the impact of factors such as corpus size, data splitting methods, and repetition ratios on model performance. By 2023, Song et al. (2023) addressed the challenges of insufficient training data and data distribution bias by proposing a method to cluster functionally similar codes and precisely replace non-key words in summaries, effectively tackling these issues. In 2024, Zhang et al. (2024) presented a comprehensive classification method for inline comments. They statistically analyzed the dataset from three perspectives: "what," "how," and "why," and conducted more refined noise cleaning. This innovation not only improved the dataset's quality but also enhanced the model's ability to understand and generate code comments.

In summary, code summarization datasets have undergone rapid development over the past decade, evolving from initial datasets focused on a single programming language to diversified datasets covering multiple programming languages, and further to recent innovations in quality improvement and data cleaning methods. These advancements have provided a solid foundation for research on code summarization tasks. In the future, with the continuous improvement of dataset quality and scale, as well as the introduction of new dataset construction methods, the field of code summarization is expected to achieve greater breakthroughs, providing more intelligent solutions for program understanding and maintenance.

3.2 Content Analysis of Code Summarization Datasets

At the content level, these datasets exhibit significant differences in multiple dimensions, including programming languages, size, quality assessment, and features. Each dataset possesses unique and notable characteristics, along with certain limitations.

From the perspective of programming languages, the currently available datasets extensively cover mainstream programming languages such as Java (DS3,DS4,DS5,DS6), C# (including SQL) (DS1), and Python (DS2,DS5). Notably, Java datasets occupy a substantial proportion. This could be attributed to factors like the widespread use of Java, its rich historical background, extensive annotation specifications, and strong community support, which provide researchers with convenient access to high-quality data resources. However, with the popularity of emerging programming languages like Python, related datasets have gradually emerged, fully demonstrating the diversity and practicality of these languages in programming styles, application scenarios, and community support.

Regarding dataset size, there are significant differences due to variations in data sources, data selection criteria, and preprocessing methods. This diversity in size reflects the richness of dataset samples, ranging from large datasets containing numerous samples (DS5,DS9) to small datasets focused on specific domains or scenarios (DS7).

In terms of quality assessment, existing research has adopted various standards to evaluate the quality of datasets, such as compliance with annotation specifications like Javadoc (DS3,DS4,DS6), the presence of redundant or duplicated information (DS2), and the ease of data prediction (DS10). These assessment standards have a profound impact on the usability, applicability, and research value of the datasets.

Furthermore, the partitioning methods of datasets also demonstrate diversity and flexibility. Some datasets are partitioned based on functionality or modules, which aids models in learning code features specific to certain functions (DS1,DS2,DS3,DS5,DS10). Others are partitioned according to code sources or projects, facilitating the assessment of model performance across projects or libraries (DS4,DS6,DS7,DS8). This diverse partitioning approach provides multiple perspectives and challenges for model training and evaluation.

Table 1 Comparison of key information in ASCS datasets

ID	Dataset	Year	Source	Language	Granularity	Size/pairs
DS1	Iyer et al. (2016)	2016	StackOverFlow	C# SQL	Method	C#: 66015 SQL: 32337
DS2	Barone and Sennrich (2017)	2017	Github	Python	Method	Python: 161630
DS3	Hu et al. (2018b)	2018	Github	Java	Method	Java: 69708
DS4	Alon et al. (2019a)	2019	Github	Java	Project	Over 20 million
DS5	Husain et al. (2019)	2019	Github	Go Java JavaScript PHP Python Ruby	Method	Over 2 million
DS6	LeClair and McMillan (2019)	2019	Sourcerer	Java	Project	2.1 million
DS7	Zhuang et al. (2020)	2020	Github	Solidity	Project	ESC: 307396 VSC: 13761
DS8	Liu et al. (2021b)	2021	Github	C	Project	C:95k+
DS9	Lu et al. (2021)	2021	Github	Various	Mixture	Over 2 million
DS10	Haque et al. (2021)	2021	Github	C/C++	Method	C/C++: 1.1million

At the feature level, each dataset has its unique strengths and limitations. Some datasets incorporate additional features such as API call sequences (DS3) and code context information (DS4), providing strong support for models to gain a deeper understanding of code logic. At the same time, different datasets may be more suitable for specific coding styles, development scenarios, or application requirements, which are their unique aspects.

To enhance understanding and facilitate comparisons between different datasets, we have carefully compiled Table 1. This comprehensive table provides detailed information on key aspects of various datasets, including data sources, programming languages used, dataset sizes, and data granularity. With the help of this table, researchers can quickly grasp the core characteristics of each dataset, laying a solid foundation for further investigation. It should be noted that Table 1 highlights only some commonly used datasets and does not comprehensively list all available data resources in this field. Additionally, the DS2 dataset was initially proposed by Barone and Sennrich (2017) and later processed by Wan et al. (2018), gaining widespread application. On the other hand, the DS3 dataset was first published by Hu et al. (2018a) and was subsequently optimized in 2018 Hu et al. (2018b) and 2020 Hu et al. (2020). To ensure coherence in the discussion, this paper focuses only on the initial versions of these two datasets.

Figure 4 clearly illustrates the usage frequency of different datasets in the field of code summarization research. In this chart, each bar represents a specific dataset, with the x-axis labeling the names of various datasets and the y-axis displaying their usage counts. This visual chart intuitively reflects the popularity and contribution of each dataset in the research field.

Among them, DS3 as a dataset focused on the Java language, ranks at the top in usage frequency, which is closely related to the high-quality Java code summarization data it provides. DS2 is a commonly used dataset in the Python language domain, and its frequent usage fully reflects the widespread attention received by Python code summarization research. Furthermore, the large-scale dataset DS6, containing over 2 million entries, has also garnered

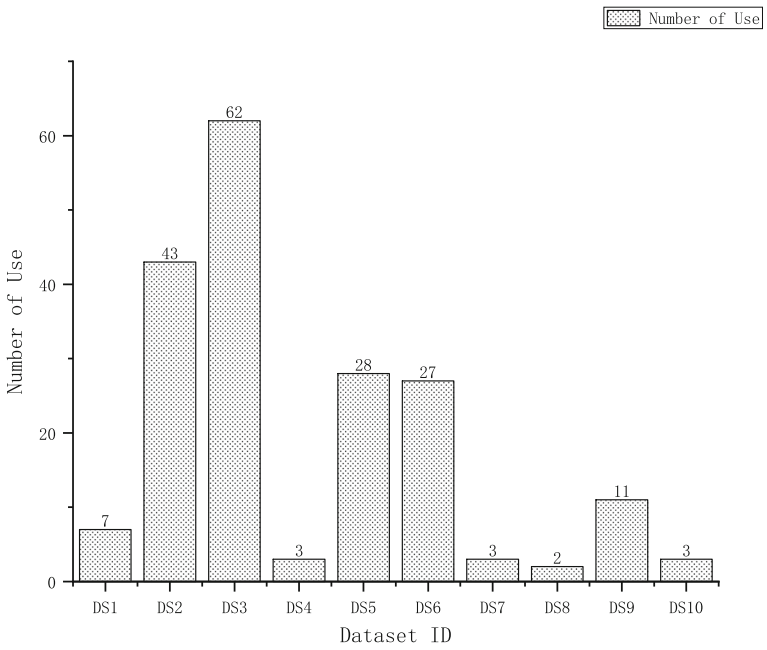

Fig. 4 Cumulative usage frequency of datasets in the field of code summarization

significant attention from the research community. Its extensive data coverage provides valuable support for researchers to conduct more comprehensive model training. DS5 and DS9 are widely favored due to their support for multiple programming languages, further promoting the development of cross-language code summarization research. In comparison, other datasets have lower usage frequencies than the aforementioned datasets due to limitations in programming languages and issues such as data quality.

To ensure comprehensiveness and representativeness of the data, we systematically collected research papers on code summarization tasks published between 2015 and 2024. The data sources cover mainstream academic databases such as IEEE Xplore, ACM Digital Library, Google Scholar, as well as top academic conferences and journals. We carefully selected research papers to ensure the diversity and representativeness of the selected papers, including research work from different research teams, research institutions, and different regions. This helps comprehensively reflect the actual usage of datasets in the field of code summarization research.

During the statistical process, we rigorously excluded self-citations by dataset authors and eliminated citation data from review articles, focusing only on research papers specifically targeting code summarization tasks for statistics. This refined statistical method effectively ensures the objectivity and accuracy of the results.

3.3 Challenges and Future Directions

In this section, we delved into an extensive analysis of existing code summarization datasets. Despite some advancements, we identified significant issues and potential shortcomings that could affect research and applications in this field.

Dataset Scale: The size of a dataset is a critical factor affecting the performance of automatic code summarization models. Generally, an effective code summarization dataset should contain at least hundreds of thousands of samples of code and corresponding comments (Hu et al. 2018a). Research by Song et al. (2022) indicates that as the number of code-comment samples in the dataset increases, the generalization ability of the model improves significantly. However, when dealing with datasets of programming languages other than Java, the size of the dataset is often far from sufficient, which limits the model's performance in these scenarios. Smaller datasets may not cover enough syntactic and semantic variations, thereby restricting the model's performance in practical applications.

Dataset Quality: High-quality datasets are crucial for generating accurate and readable code summaries. Typically, high-quality datasets should possess the following characteristics: Firstly, comments should accurately describe the functionality of the code and adhere to consistent documentation standards, such as Javadoc or Python's docstring style. Secondly, there should be a close semantic correlation between the code and the comments to ensure that the model can learn accurate mapping relationships (Zhang et al. 2024). Additionally, to ensure the high quality of the dataset, rigorous data cleaning and filtering processes are usually conducted to eliminate noisy data and irrelevant samples. According to Shi et al. (2022c), systematically identifying and removing noisy data can significantly enhance model performance.

Dataset Diversity: The diversity of a dataset plays a significant role in ensuring the adaptability and generalization ability of the model. A diverse dataset should include samples from different programming languages, coding styles, and application domains. For example, a multilingual code summarization dataset should typically cover multiple mainstream programming languages (such as Java, Python, C++, JavaScript, and Ruby) to ensure that the model can generate consistent and accurate summaries in a multilingual environment (Lu et al. 2021). Furthermore, the dataset should include code snippets of various complexities, ranging from simple function definitions to complex class structures and cross-file call relationships. Such diversity can help the model better adapt to a wide range of practical application scenarios.

Future Directions and Challenges: Looking ahead, building larger-scale, higher-quality, and more diverse datasets remains a significant direction in the field of automatic code summarization. Based on the experience of current research, the size of future datasets should be expanded to at least a million samples to meet the demands of more complex models. Simultaneously, data cleaning and quality control will become even more crucial, and automated tools and methods are expected to help maintain the high standards of the dataset. Lastly, in terms of diversity, the expansion of datasets will cover more programming languages and application domains to ensure that the model can excel in diverse practical applications.

4 Source Code Modeling

Source code modeling is the process of analyzing and modeling code to extract syntactic and semantic information within it, thus helping to generate concise and accurate code summarization. Code modeling is one of the key issues in the code summarization task because it determines the ability of the model to understand and model the code. Based on different representations of code, in this paper we classify code modeling methods into four types: token-based, tree-based, graph-based, and combination-based source code model.

4.1 Token-based Source Code Model

Token-based code modeling views code as a sequence of lexical units (tokens). It utilizes natural language processing techniques for analysis and representation. This approach uses identifiers, variable names, method names, and other elements in the code as vocabulary and spaces, line breaks, indentation, and other symbols in the code as separators to tokenize the code into a sequence similar to natural language text. An example is shown in Fig. 5. The top is the source code and the bottom is the token representation of the source code.

Haiduc et al. (2010b) treated source code as ordinary text for analysis and used text summarization techniques to generate functional descriptions. Wang et al. (2015) used lexical annotation to extract keywords for source code features and selected the words with the highest weights as summarization . Some studies use the characteristics of source code itself to introduce auxiliary information. Hu et al. (2018b) found that API knowledge contains important information about source code functionality. Ahmad et al. (2020) proposed to use transformer model to learn code representation by modeling pairwise relationships between code tokens. Hussain et al. (2020) capture the context of the source code by utilizing the tag types of the source code and use a code specifier to encode the source code based on the tag types.

Token-based code modeling has the following advantages: first, it's simple and effective for lexical and syntactic analyses; second, it's able to utilize a large amount of open-source code data for pre-training and fine-tuning to improve the model's generalization ability; and third, it's able to adapt to different programming languages and styles to improve the model's generality. However, token-based code modeling also has the following limitations: first, it relies on identifiers in code to accurately reflect the function, which may be ambiguous, lack of specification, or inconsistent with natural language; second, it focuses only on the surface information of code and ignores important details and contextual information, such as data types, control flow, data flow, and so on; third, it's difficult to handle long dependencies and complex structures such as nested loops, recursive calls, and so on.

4.2 Tree-based Source Code Model

Abstract Syntax Trees (ASTs) are an important way to model code information, usually using a sequence of nodes of an AST or its subtrees, subsequences, and a collection of paths extracted from the AST to characterize the structural and semantic information of the code (Alon et al. 2019a; Zhang et al. 2020). An AST is a tree structure generated according to the syntactic rules of a programming language, which models each element (such as a variable, an operator, a function, etc.) of the code as a node and models the relationship between them through edges. Tree-based code modeling is one of the approaches that has received much

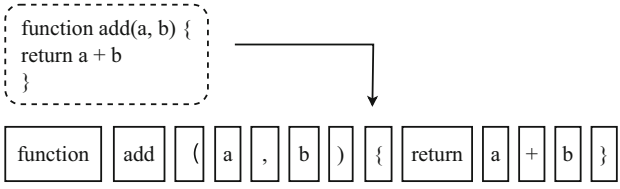

Fig. 5 An example of token-based source code representation. The top is the source code and the bottom is the token representation of the source code

attention and research in recent years, and using AST to model code has significant results. An example is shown in Fig. 6. The left side of the arrow is source code, and the right side is the tree-based representation of source code. This AST consists of two types of edges and nodes. AST edges represent the parent-child relationship in the code, that is, a node is another node's child or parent node. AST nodes represent different elements in the code, such as variables, constants, operators, functions, expressions, statements, etc. Each node has a type and a value, indicating its syntactic category and specific content.

Code2Vec (Alon et al. 2019b) employs AST paths to represent source code. This method is suitable for handling large-scale codebases with simple structures but may struggle when dealing with code with complex structures or long-distance dependencies. In contrast, Code2Seq (Alon et al. 2019a) enhances the model's ability to capture diversity by randomly sampling AST paths, making it more suitable for code summarization tasks. However, this randomness introduces uncertainty and computational efficiency issues. The Structural-Based Traversal (SBT) method (Hu et al. 2018a) uses pairs of parentheses to represent the AST structure, which effectively preserves the complete hierarchical information of the code but may generate excessively long sequences, increasing memory overhead and introducing noise.

Compared to the aforementioned methods, the approach proposed by Zhang et al. (2019) splits the AST into subtrees and performs recursive encoding, making it more suitable for handling code with complex local structures. However, it has higher computational costs and performs poorly in capturing global dependencies. Finally, Guo et al. (2021) proposes a strategy that utilizes partial AST information and data flow structure to reduce computational overhead, which is more suitable for handling large-scale and complex codebases. It should be noted, however, that this approach may lose fine-grained syntactic information and has limited performance on code lacking explicit data flow.

In the task of code summarization, tree-based source code modeling methods deeply explore the structured information of source code, with the core objective of capturing the syntactic and semantic features of programs. This approach employs AST to model the structure and semantic information of code, and dynamically assigns weights to relevant nodes through attention mechanisms, effectively handling long-range dependencies in the code. Compared to token-based methods, this approach better retains and utilizes the inherent

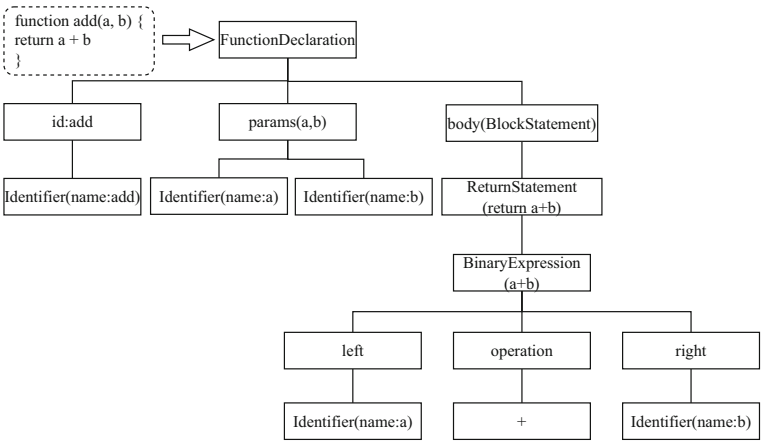

Fig. 6 An example of tree-based source code representation. The left side of the arrow is source code, and the right side is the tree-based representation of source code

hierarchical structural information in the code, significantly enhancing the model's ability to understand the semantic and logical information of the code.

Tree-based code representations can be divided into linearization and non-linearization. Linearizing the AST simplifies data processing but increases sequence length, leading to higher memory usage and reduced training efficiency. It also requires tailored design for different programming languages, increasing implementation complexity. Non-linearized representations retain the tree's original structure, reflecting the source code's syntax and preserving parent-child relationships. This effectively captures program structure, crucial for understanding complex logic. While non-linearized representations excel in retaining structured information and capturing logic, their limitations in computational complexity, cross-function dependency representation, scalability, and handling deep structures must be considered in practical use.

4.3 Graph-based Source Code Model

A graph structure is a flexible data structure used to describe information such as an AST, parse tree, or control flow graph (CFG) for code. Graph-based code modeling is an approach to modeling code information using graph structures that capture the elements and relationships in the code. Graph structures can capture complex and dynamic semantic information in the code such as data flow, control flow, data dependencies, and so on. Graph-based code modeling is one of the approaches that has received extensive attention and research in recent years. Common graph structures include control flow graph, data flow graph (DFG), program dependency graph (PDG), and code property graph (CPG), which capture code from different perspectives. An example is shown in Fig. 7. Above the arrow is a piece of Java code, and below the arrow is the code property graph (CPG) corresponding to the code.

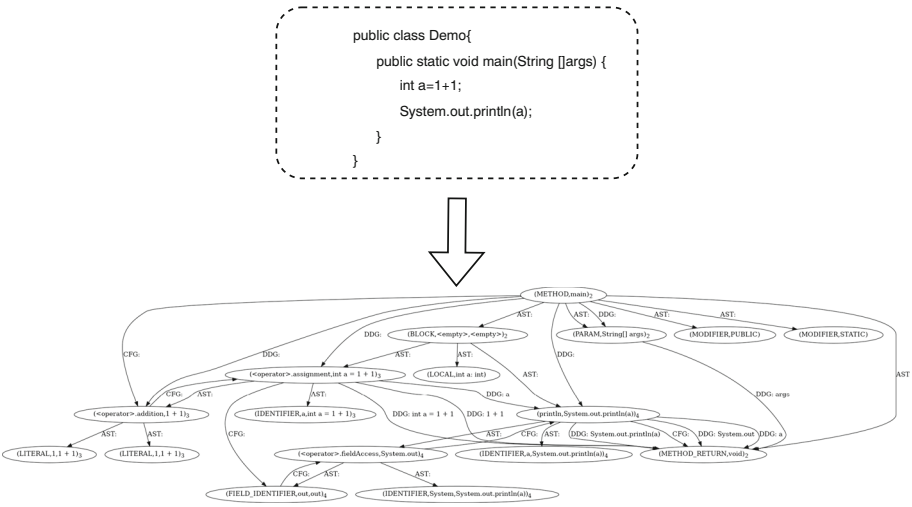

Fig. 7 An example of graph-based source code representation. Above the arrow is a piece of Java code, and below the arrow is the code property graph (CPG) corresponding to the code. This code property graph integrates three types of edges and nodes, which represent different aspects of the code information. AST edges and nodes reflect the syntactic structure of the code, such as variables, functions, expressions, statements, etc. CFG edges and nodes reflect the control flow logic of the code, such as conditions, loops, jumps, etc. DDG edges and nodes reflect the data dependency relationships of the code, such as variable assignments, function calls, parameter passing, etc

Allamanis et al. (2018) proposes a graph representation method based on data flow and control flow dependencies, which can effectively analyze complex program behavior. However, its computational complexity is relatively high when dealing with large-scale codebases. On the other hand, Tufano et al. (2018) adopts multiple representation forms, including identifiers, AST, Control Flow Graph (CFG), and bytecode. While this multi-level representation allows for deep analysis of code from multiple perspectives, it may also increase model complexity and computational resource consumption. Zhou et al. (2019) and Wu et al. (2021) propose a quadruple view based on AST, CFG, Data Flow Graph (DFG), and Natural Code Sequences (NCS), as well as a triple view based on abstract syntax, control flow, and data dependencies. These methods enhance code understanding through rich grammatical and semantic representations but also increase computational overhead. Cheng et al. (2021) and Wang et al. (2022b) integrate code snippet sequences and AST representations, with Cheng et al. (2021) introducing Syntactic Code Graph (SCG) and Wang et al. (2022b) adding semantic edges to reflect information flow. These methods can better combine structural and semantic information in code but may face performance bottlenecks when dealing with large-scale code.

Graph-based code modeling methods typically employ a triple graph structure (Fernandes et al. 2019) to model the elements and relationships within the code, thereby effectively preserving and utilizing the structural information of the code. Compared to token-based or tree-based methods, these approaches excel in handling long-range dependencies and complex structures, such as nested loops and recursive calls, while also enhancing the model's ability to understand semantic and logical information within the code. While non-linearized tree-based representations preserve the hierarchical structure of the code and can be more effective than linearized methods in certain scenarios, they have limitations in directly representing long-distance dependencies across multiple parent-child node levels. In contrast, graph-based modeling can more naturally capture these complex dependencies, making it more suitable for tasks that require capturing intricate interactions. Despite the advantages of non-linearized tree-based code representations, they still exhibit certain limitations in representing complex cross-node dependencies when compared to graph-based code modeling. However, graph-based methods also have their own limitations. Firstly, they require converting the source code into a graph structure, which increases complexity and time requirements. Secondly, they require additional domain knowledge and also have higher data requirements. Furthermore, since different programming languages may have different syntactic rules and graph structures, they often need to be customized and adjusted to accommodate these differences. In conclusion, graph-based code modeling methods have great potential in improving code comprehension. However, they also need to overcome some technical and domain-specific challenges.

4.4 Combination-based Source Code Model

Combination-based code modeling is an approach that combines different abstraction levels of code into a unified vector to leverage the multifaceted information of the code and improve the effectiveness of code representation. This approach can use tokens, trees, graphs, or other forms to model different levels of code, such as lexical, syntax, semantic, and so on, and integrate them into a single code vector using different combinatorial strategies, such as splicing, weighting, and fusion. Combination-based code modeling is one of the approaches that has received extensive attention and research in recent years.

McBurney and McMillan (2014) proposed a technique to include code contextual information by analyzing the invocation of Java methods through SWUM (software word usage model). Some studies have used API knowledge mapping to represent the code model of APIs (Hu et al. 2018b) and supplemented the code representation with external knowledge. Zhou et al. (2022b) extracted token-level and statement-level code features using hierarchical representation of code. Ma et al. (2022) proposed a fusion method to accurately align and fuse semantic and syntactic structure information for token and AST fine-grained fusion.

Combination-based code modeling captures multiple aspects of code and uses the complementary nature of different levels of code to handle the limitations of a single level of code. This approach provides a more comprehensive understanding and modeling of the syntactic and semantic code, and improves the model's ability to describe the code's functionality and behavior than token-based, tree-based, or graph-based approaches. However, there are some challenges within this approach: it requires complex combinatorial strategies, which increase the model's parameters and computation; it also requires interconverting different levels of code to each other, which increases the difficulty and overhead of data processing.

5 Code Summarization Generation

Code summarization is the technique of using natural language to describe the functionality and purpose of source code files or snippets. Code summarization generation algorithms are the core component of ASCS. Haiduc et al. (2010b) classified code digests into extractive and generative categories according to the way of digest generation. Extractive summarization consists of keywords or phrases extracted from the source code, while generative summarization produces complete natural sentences based on the semantics and structure of

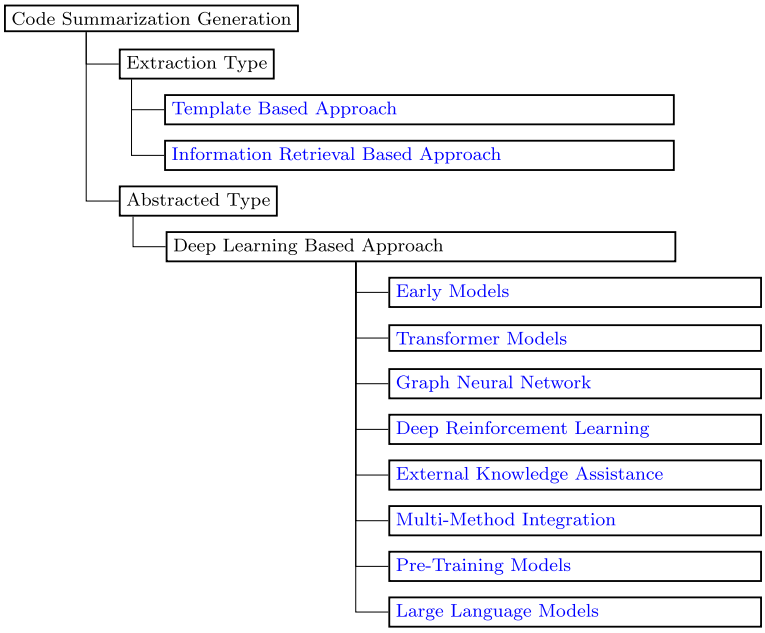

Fig. 8 The common code summarization generation algorithms and their technical routes

the source code. In this paper, we follow the previous classification criteria and categorize code summarization generation methods into three categories: template-based, information retrieval-based, and deep learning-based methods. These methods have different characteristics, strengths, and weaknesses, and we will provide a detailed introduction and comparison of them in this paper. The common code summarization generation algorithms and their technical routes are shown in Fig. 8:

5.1 Template-based Approach

The template-based approach uses manually customized rules and templates to generate summarization based on specific information in the source code, filling in the template. The basic idea of this approach is to treat the code summarization generation task as a natural language generation task, that is, to extract information from the source code that is suitable for generating summarization, and then to transform this information into natural language text using natural language generation techniques. The relevant information about the template-based approach is detailed in Table 2 of this paper.

Table 2 Comparison of template-based ASCS methods

ID	Language	Main features
PS1	Java	Contextual search, phrase hierarchy, natural language query enhancement.
PS4	Java	Automatic Java method summarization, leveraging structural and linguistic cues.
PS5	Java	Identifies high-level actions, synthesizes descriptions.
PS6	Java	Automatic Java parameter commenting with method intent integration.
PS8	Java	Stereotype-based, heuristic-driven, lexicalization tools.
PS12	Java	Automatic Java method context summarization, PageRank, SWUM, NLG integration.
PS13	Java	Automated Java commit messaging, code change summarization, impact set analysis.
PS17	C++	Automatic C++ method summarization using stereotypes and srcML analysis.
PS22	Java	Crowdsourcing-based feature extraction, SVM, and NB classifiers for Java code summarization.
PS23	Java	Automatic commit message generation, what & why summarization, rule-based constraints.
PS24	Java	Java method summarization, automated documentation, PageRank analysis, NLG system.
PS27	Java	Java method summarization, nano-patterns identification, template-based generation.
PS28	Python	Python function-docstring corpus, neural machine translation, code documentation.
PS29	Java	Automatic Java action unit identification and natural language summarization.
PS45	Java	Automatic Java source code summarization, context-aware, static analysis-based.

The basic process is illustrated in Fig. 9: First, the source code undergoes syntax analysis to extract the code's structural and semantic information. Next, based on the code's type and function, appropriate summarization templates are selected. These summarization templates are natural language sentences containing placeholders to describe the main purpose of the code. Finally, key information from the code is inserted into the summarization templates to generate the final code summarization.

Hill et al. (2009) generated natural language based code annotations based on SWUM by analyzing the identifiers of Java methods. Sridhara et al. (2010) further considered the code information inside the method body in the generation of code summarization . Based on this, Sridhara et al. (2011b) further generated code summarization for parameters in Java methods (Sridhara et al. 2011a) and high-level actions inside the code. Wang et al. (2017) focused on sequences of consecutive statements with interrelated object references to optimize the generated summarization . Moreno et al. (2013) used existing heuristic rules and classifiers to determine the prototypes of classes and methods , and select appropriate templates to generate summarization based on different prototypes.

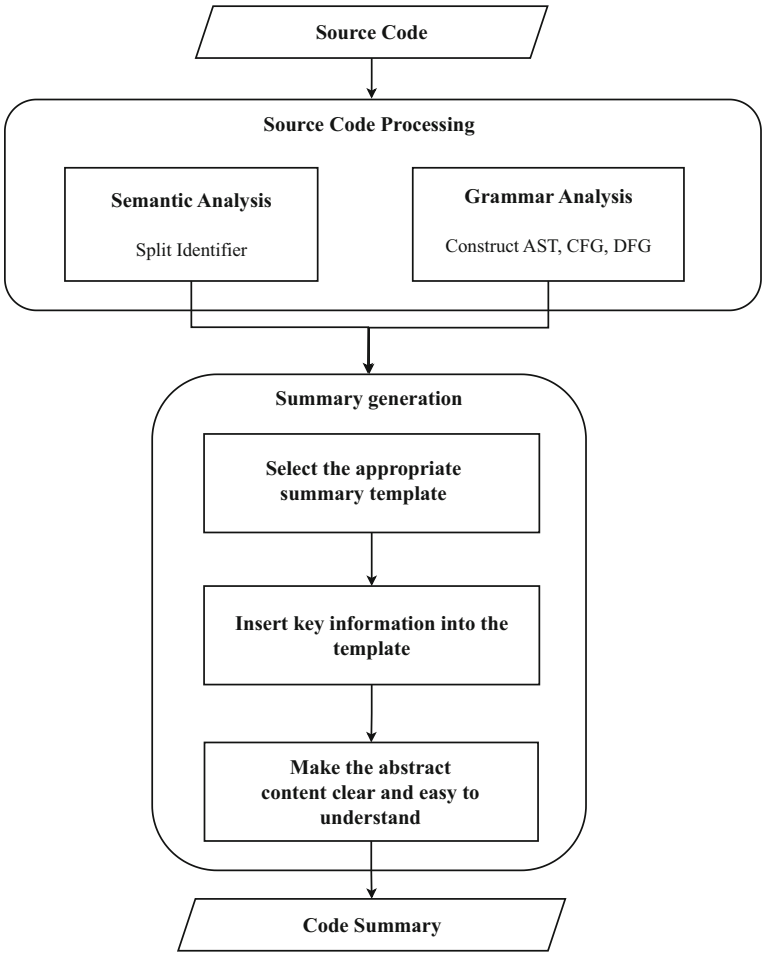

Fig. 9 Basic flowchart of the template-based automatic code summarization approach

The advantage of template-based methods is that they can generate concise, accurate, and consistent summarization. These methods use human-designed rules and templates to ensure the quality and consistency of the summarization, and they can also customize different rules and templates according to different programming languages and domains. These methods can also be combined with other types of methods, such as information retrieval-based or deep learning-based methods, to improve the diversity and flexibility of the summarization. However, the disadvantage of these methods is that they require a lot of template design and maintenance work, which increases the workload and complexity. These methods also have difficulty adapting to different styles of code and diverse summarization requirements. Moreover, these methods require a thorough analysis and understanding of the source code to extract the information suitable for generating summarization, which may require complex code analysis techniques.

Table 3 Comparison of ASCS methods based on information retrieval

ID	Language	Main features
PS2	Java	Automatic summarization leveraging lexical and structural code analysis.
PS3	Java	Automated Java code summarization using text retrieval techniques.
PS7	Java	Automatic extraction of Java method descriptions from developer communications.
PS9	Java	Automatic comment generation from Q&A sites using NLP and heuristics.
PS10	Java	Predictive commenting using Java n-grams, LDA, and link-LDA models.
PS11	Java	Hierarchical PAM-based, developer evaluation study.
PS14	Java	Eye-tracking study, keyword selection enhancement.
PS15	Java	Hierarchical topic modeling, Java-centric.
PS16	Java	Java method documentation mining, social approach, StackOverflow integration.
PS18	Java	Code clone detection, context-sensitive text similarity.
PS19	Various	Crowdsourcing insights, heuristic mining, Stack Overflow comments.
PS20	Java	Eye-tracking study, keyword extraction enhancement.
PS26	Java	Autofolding via AST parsing, salience optimization, language-agnostic approach.
PS30	Java	Reuse of existing comments, syntax and semantic similarity analysis.
PS31	Java	Autofolding, AST-based, scoped topic model, unsupervised summarization.
PS39	Java	Neural machine translation-based, nearest neighbor enhancement.
PS42	Java	Bimodal VAE framework, semantic code-natural language mapping, efficient retrieval.

5.2 Information Retrieval Based Approach

Code summarization generation based on information retrieval is a method that uses a statistical language model to find the best matching summarization from a database of existing summarization. It involves manually selecting features to extract information from the code that can be used for summarizing. Then, it calculates the similarity between the target code and the code in the database, and returns the most similar summarization. The main idea of this approach is to treat code summarization generation as a natural language generation task, i.e., extracting information from the source code that is suitable for generating summarization, and then transforming this information into natural language text using natural language generation techniques. This paper provides a detailed enumeration of the relevant information regarding the information retrieval-based approach in Table 3.

The basic process, as shown in Fig. 10, begins with the construction of a corpus containing source code and relevant documents. Users provide a query or description, and then relevant

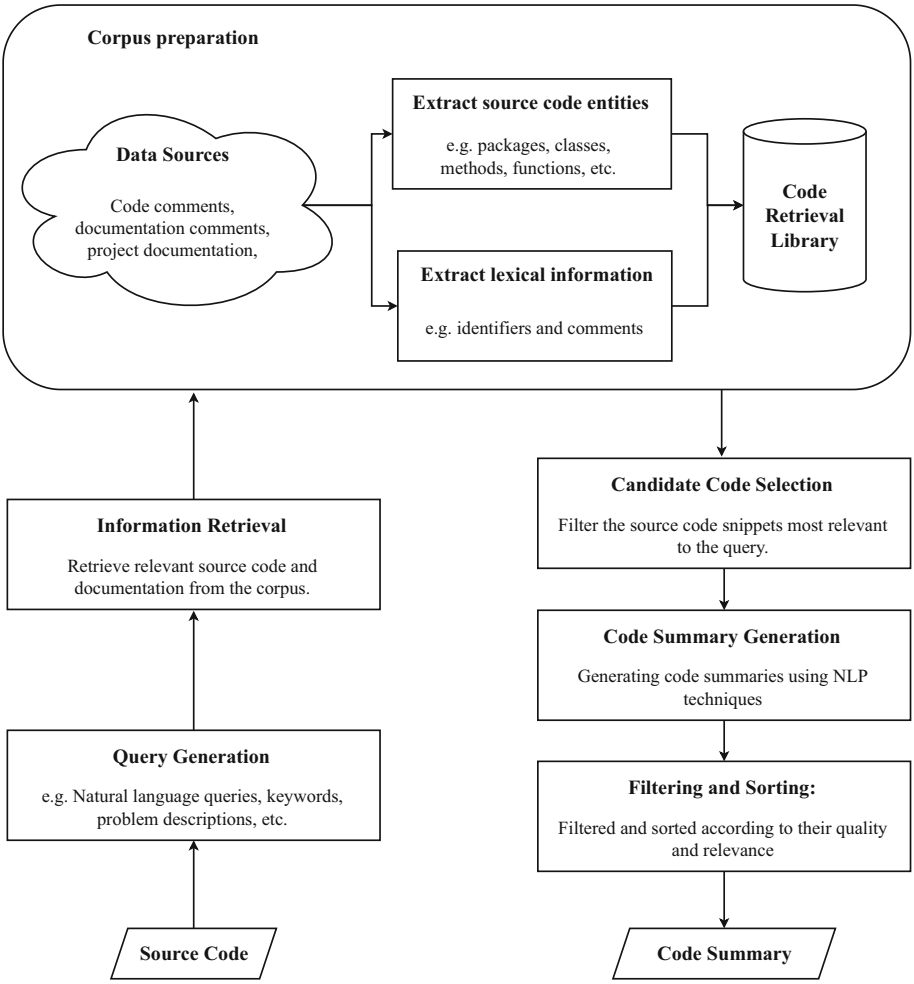

Fig. 10 Basic flowchart of the information retrieval based automatic code summarization approach

code and documents are retrieved from the corpus. Next, code snippets related to the query are selected and transformed into natural language summarization. Finally, summarization is filtered and ranked based on quality and relevance, and presented in a user-friendly format.

Haiduc et al. (2010b) was the first to propose an information retrieval-based code summarization generation technique that analyzes code with two retrieval models, VSM (Vector Space Model) and LSI (Latent Semantic Indexing), to generate class and method summarization. Panichella et al. (2012) used regular expression matching methods to automatically extract method descriptions from bug tracking systems and mailing lists. Rahman et al. (2015) proposed to retrieve useful annotations from Stack Overflow question postings and discussions to obtain useful annotations. Wong et al. (2015) used code cloning techniques to discover similar code segments from open source projects in GitHub and used their annotations to describe other similar code segments. Liu et al. (2018) obtained information about the code changes in the training set that are most similar to the test code by calculating the similarity and considered it as the code summarization of the test code.

The information retrieval-based approach is a data-driven method that uses a large number of existing codes and annotations as a database to find the most suitable or relevant summarization for the target code. Its advantage is that it can use existing high-quality annotations to create new ones, and it can adjust the results based on different retrieval models and similarity measures. This method is simple and easy to implement, does not need a lot of training data, and works for different programming languages. However, this method depends on the quality and quantity of the database, and the summarization may be incomplete or inaccurate if there are no similar or related codes or annotations.

5.3 Deep Learning Based Approach

Deep learning-based approaches are a hot research topic for automatic code digest generation. Such approaches generally use a sequence-to-sequence framework, encode source code with a neural network model, and generate digests by decoding with an attention mechanism.

The basic process is shown in Fig. 11: First, the source code dataset is prepared and preprocessed, which includes data collection and text preprocessing. Then, after choosing an appropriate deep learning model architecture, the model is trained to learn the mapping relationship between the source code and summarization based on the dataset, while hyperparameter tuning is performed. Next, the trained model is evaluated using automated evaluation metrics. After that, the source code is input into the trained model to generate code summarization, and optionally post-processed, such as removing redundant information. Finally, the generated code summarization is presented in a user-friendly format.

Early models in code summarization: In 2016, Iyer et al. (2016) proposed the CODE-NN model, which uses LSTM and attention mechanism to generate code summarization . This model became the baseline for many subsequent studies. The CODE-NN model takes code and natural language as input and output, respectively. It uses an LSTM encoder and decoder to learn the alignment between code and natural language, and an attention mechanism to focus on the important parts of the encoder output. Liang and Zhu (2018) used a recurrent neural network based on Code-RNN to extract code features and a Code-GRU to generate code summarization. Hu et al. (2018a) used AST to analyze the structure and semantics of Java methods, and proposed a new linearization method for traversing ASTs called SBT. LeClair et al. (2019) treated code as plain text, and considered both word-based and AST-based representations, allowing the model to learn code structure information independently.

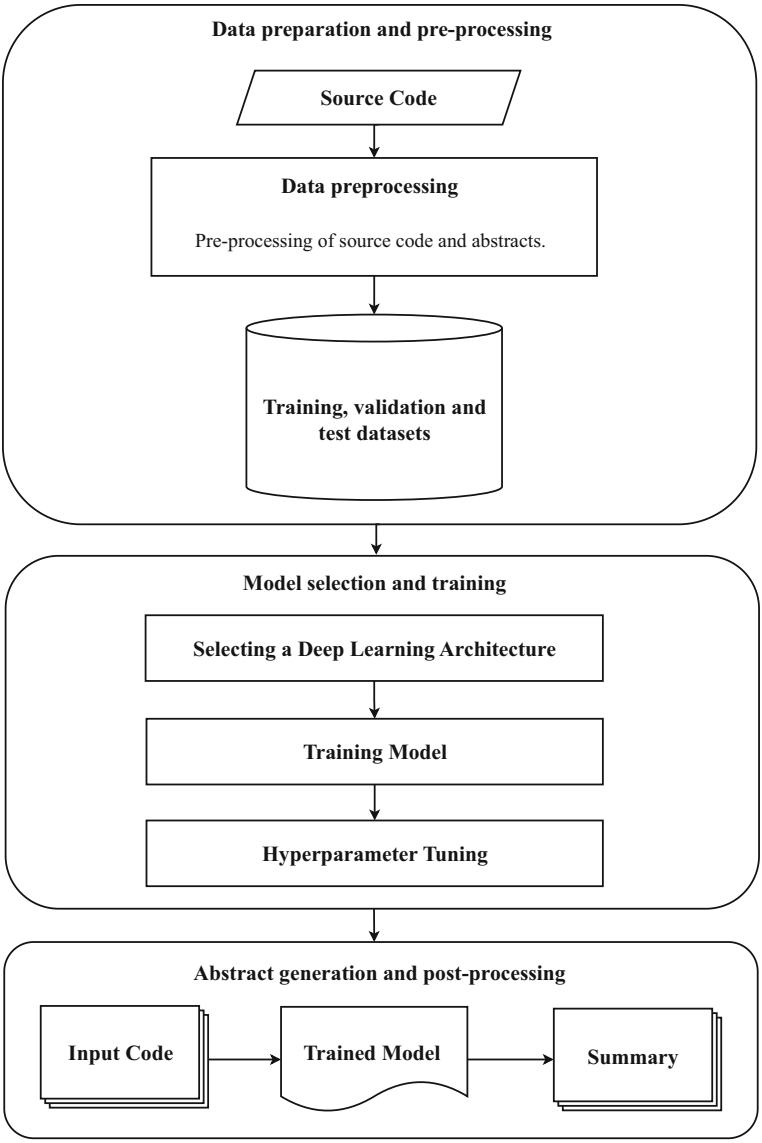

Fig. 11 Basic flowchart of the deep learning based automatic code summarization approach

The relevant research work on code summarization tasks using early deep learning models in recent years is enumerated in Table 4.

Transformer model for code summarization: The Transformer model is a neural network model proposed by Vaswani et al. (2017). It uses a self-attention mechanism to handle long-distance dependency problems. Unlike RNN models, the Transformer model does not need to process the input sequence sequentially, but can parallelly compute the global dependency relationship of each element in the input sequence. Ahmad et al. (2020) applied the Transformer model to generate code summarization . They used a Transformer encoder to encode

Table 4 Comparison of earlier ASCS methods

ID	Language	Main features
PS21	C#,SQL	Neural attention model, LSTM-based, code summarization, data-driven approach.
PS25	Java	Convolutional attention for code summarization.
PS32	Java	Neural Machine Translation for commit message generation, V-DO pattern filtering.
PS33	Java	API knowledge, deep learning, Java method summarization, Seq2Seq model.
PS34	Java	Combines deep learning with code structure for Java summarization.
PS35	Java	Neural encoder-decoder, Java context modeling, two-step attention mechanism.
PS36	Java	Java method AST, structure-based traversal, attention mechanism.
PS38	C#	Graph-based deep learning, program understanding, semantic reasoning.
PS40	Java	Class-level summarization, micro patterns, design patterns, dependencies analysis.
PS41	Java	Recursive Neural Network (Code-RNN), Code-GRU for comment generation.
PS42	C#,SQL	Bimodal VAE framework, semantic code-natural language mapping, efficient retrieval.
PS43	Java	AST-Word dual input, ensemble decoding.
PS44	Java	Code embeddings, path-based attention, AST syntax, semantic prediction.
PS46	Java	AST-based code representation, encoder-decoder model, LSTM with Attention.
PS47	Java,Python	Dual training framework, attention mechanism, cross-task regularization.
PS48	Java	Language-agnostic encoder, open-vocabulary decoder.
PS49	Java	Attention-based code-comment translation, leveraging code constructs.
PS50	Java	Extended Tree-LSTM, structural properties utilization.
PS51	Java	Call dependency integration, Seq2Seq model.
PS57	Java	Lexical-syntactical fusion, CNN for AST nodes, Switch Network for adaptive combination.
PS58	Java,C#	Attention-based, two-phase model, keyword prediction, semantic gap reduction.
PS59	Java	Hybrid lexical-syntactical deep learning for Java code comments.
PS79	Java	Hierarchical code representation, attention-based encoder-decoder, ensemble learning.

the code tokens, and a Transformer decoder to generate the summarization tokens. Wu et al. (2021) introduced a multi-view structure attention mechanism to improve the ability of learning code structure information. They used three views to represent code: token view, AST

view, and path view. Guo et al. (2022) introduced a ternary position to model the hierarchy and structure of code, and used a pointer generation network to better generate summarization. Zeng et al. (2023a) solved the out-of-vocabulary (OOV) problem with a byte pair encoding algorithm and a pointer generation network. They also captured the structure information of the code sequences by using a contrastive learning strategy and a dynamic graph attention mechanism. Zeng et al. (2023b) adopted a structure-induced transformer architecture

Table 5 Comparison of ASCS methods based on Transformer

ID	Language	Main features
PS53	Java,Python	Transformer-based, self-attention, relative position encoding, copy mechanism.
PS60	Java,Python	Functional reinforcement, BERT integration, Transformer-based.
PS61	Python	Transformer-based, actor-critic network, code search enhancement.
PS64	Java,Python	Retrieval-augmented framework, dual-encoder retrieval.
PS67	Java,Python	Multi-task BERT-based encoder, API sequence utilization.
PS68	Java,Python	Sequential-Structural Learning, Graph Convolution.
PS70	Java	Graph-structured neural network integration with Transformer-XL.
PS71	Java	Dynamic graph attention, subword sequence modeling, OOV reduction.
PS78	Java,Python	Hybrid representations, graph attention, Transformer decoder.
PS80	Java	Combined retrieval and transformer, AST-augmented, code sequence-augmented.
PS81	Java	Multi-modal transformer-based code summarization, contrastive learning retrieval.
PS82	Java,Python	Triplet position encoding, GraphSAGE, multi-source pointer-generator network.
PS85	Java,Solidity	Multi-modal fusion, Local-ADG, AST integration, similarity network.
PS86	C,Java	PDG-based graph embedding, SBERT semantic evaluation.
PS87	Java,Python	Efficient tree-structured attention, linear complexity, AST encoding innovation.
PS88	Python,Solidity	Transfer learning, discriminator-based head selection, feature space adaptation.
PS89	Java	Graph attention networks, BiGRU, tokenized AST, dual encoders.
PS96	Java	Contrastive learning, BPE algorithm.
PS97	Java,Python,Solidity	Statement semantics leveraging, Graph attention control flow.
PS98	Java,Python	Graph-augmented Transformer, syntax-informed self-attention.
PS99	Java,Python	Structure-sequence aligned, bidirectional decoding, multi-scale fusion strategy.
PS100	Java,Python	Code structure-guided Transformer, hierarchical attention.
PS101	Java,Python	Code structure features, semantic summarization.
PS102	Java,Python	Semantic-structural transformer, local feature extraction.
PS103	Java	Hybrid AST expansion, heterogeneous GNN.
PS104	Java,Python	Multi-scale, multi-modal, enhanced ASTs, semantic fusion.
PS117	Various	Self-supervised pretraining, AST path embedding.
PS119	Java,Python	Integrating non-fourier and AST-structural positional encodings in Transformers.
PS120	Various	Context-based transfer learning, low-resource code summarization.

to process multiple abstract syntax trees simultaneously. The architecture consisted of an encoder, a fusion module, and a decoder. The fusion module used an adaptive weight fusion strategy to combine global, structural, and local information, thereby improving the quality of code summarization. Table 5 enumerates relevant research work on code summarization tasks using the Transformer model in recent years.

Utilizing graph neural networks for code structure information: To utilize the structure information of the code, some researchers tried to use graph neural network methods. Graph neural network methods are neural network methods based on graph structures, which were proposed by Kipf and Welling (Kipf and Welling 2017). They can learn graph structure features by aggregating the information of nodes and their neighbors. LeClair et al. (2020) proposed a graph-based neural architecture that matched the default structure of ASTs, and modeled the AST for each function. Liu et al. (2021a) combined different code representations, and fused static and dynamic graphs. Wang et al. (2022b) used a graph attention neural network to learn a hybrid representation of code, by incorporating control flow edges from code fragments into the graph. To highlight the overall information of the source code, Yang et al. (2023b) used the API usage knowledge to construct an API context graph that reflects the relationships between APIs. Then, they designed a neural network based on graph attention mechanism to encode the API context graph, thereby extracting the key features of the source code. Table 6 enumerates relevant research work on code summarization tasks using graph neural networks in recent years.

Enhancing code summarization with deep reinforcement learning: To improve the quality of code summarization generation, some researchers used deep reinforcement learning methods. Deep reinforcement learning methods are neural network optimization methods based on a reward mechanism. They can learn the optimal generation strategy by interacting with the environment continuously. Wan et al. (2018) integrated the abstract syntax tree structure and the sequential content of the code snippets into a deep reinforcement learning framework. Based on this, Wang et al. (2022a) proposed a hierarchical attention network that captured the structure information of the code through type-enhanced AST sequences, and fed it into the reinforcement learning framework. Huang et al. (2020) used the ASTs of code snippets to generate discourse-based tagging sequences, which were then combined with the reinforcement learning actor-critic algorithm and the encoder-decoder algorithm to generate block comments. To improve the readability and informativeness of the generated code summarization, Zhang et al. (2023a) adopted a reinforcement learning training strategy. This training strategy used the BLEU score as the reward function in the training process, maximizing the semantic similarity between the generated code summarization and the reference summarization. Table 7 enumerates relevant research work on code summarization tasks using deep reinforcement learning in recent years.

Leveraging external information for code summarization: Besides the information in the code itself, some researchers also utilized other sources of information to help generate code summarization. For example, Hu et al. (2018b) used API knowledge to build a mapping relationship between APIs and functional descriptions. They used an API knowledge base that contains information such as functional descriptions, parameters, and return values for each API method. They matched this information with the API calls in the code snippets, which improved the accuracy and readability of the generated summarization. Son et al. (2022) employed dependency information within the code to enhance the model's ability to capture code structure. By embedding the program dependency graph (PDG) into the encoder, they improved the model's understanding of code structure. Gao et al. (2022) used multimodal information from native API dependency graphs and AST fusion methods for semantic modeling by calculating the similarity of the two graphs. They used a native API

Table 6 Comparison of GNN-based ASCS methods

ID	Language	Main features
PS38	C#	Graph-based deep learning, program understanding, semantic reasoning.
PS52	Java	Graph-based neural architecture, AST encoding, sequence-structure integration.
PS68	Java,Python	Sequential-Structural Learning, Graph Convolution.
PS70	Java	Graph-structured neural network integration with Transformer-XL.
PS71	Java	Dynamic graph attention, subword sequence modeling, OOV reduction.
PS77	C	Hybrid GNN, Retrieval-Augmented, C Language Dataset.
PS78	Java,Python	Hybrid representations, graph attention, Transformer decoder.
PS82	Java,Python	Triplet position encoding, GraphSAGE, multi-source pointer-generator network
PS84	Java,Python	Retrieval-augmented framework, dense retrieval extension.
PS85	Java,Solidity	Multi-modal fusion, Local-ADG, AST integration, similarity network.
PS86	C,Java	PDG-based graph embedding, SBERT semantic evaluation.
PS89	Java	Graph attention networks, BiGRU, tokenized AST, dual encoders.
PS97	Java,Python,Solidity	Statement semantics leveraging, Graph attention control flow
PS100	Java,Python	Code structure-guided Transformer, hierarchical attention.
PS102	Java,Python	Semantic-structural transformer, local feature extraction.
PS103	Java	Hybrid AST expansion, heterogeneous GNN.
PS104	Java,Python	Multi-scale, multi-modal, enhanced ASTs, semantic fusion.
PS105	Java,Python	Multi-modal code representation, graph attention, holistic API usage modeling.

Table 7 Comparison of reinforcement learning-based ASCS methods

ID	Language	Main features
PS37	Python	Hybrid code representation, AST-based LSTM, actor-critic network.
PS61	Python	Transformer-based, actor-critic network, code search enhancement.
PS62	Java	Reinforcement learning-based, encoder-decoder model, block comment generation.
PS83	Python	Hierarchical attention, reinforcement learning, multi-feature representation.
PS98	Java,Python	Graph-augmented Transformer, syntax-informed self-attention.

dependency graph generator to generate native API dependency graphs based on API calls and control flow in code snippets. They fused these graphs with ASTs to enhance the structural information of the generated summarization. Table 8 enumerates relevant research work on code summarization tasks that have leveraged external information in recent years.

Integration of multiple approaches for code summarization: Besides using a single approach, some researchers also tried to combine different approaches. Zhang et al. (2020) proposed a hybrid approach based on retrieval and generation, where they built a corpus search database to find semantically similar codes, and used a deep learning model to generate summarization if there were no similar codes in the database. This approach could utilize existing code summarization as references, while also adapting to novel code snippets. LeClair et al. (2021) proposed a synthesis model that exploited the orthogonal performance of different deep neural network models. They used three different models: an RNN-based model, a Transformer-based model, and a graph neural network-based model, and combined their outputs using a weighted average or a voting decision. This approach could leverage the advantages of the different models, while avoiding the drawbacks of a single model. Sun et al. (2024) fused both extraction and abstraction methods to make the generated summarization both factually detailed and readable. They used an extraction model and an abstraction model, and concatenated or inserted their outputs. This approach could preserve the impor-

Table 8 Comparison of externally information-aided ASCS methods

ID	Language	Main features
PS27	Java	Java method summarization, nano-patterns identification, template-based generation.
PS33	Java	API knowledge, deep learning, Java method summarization, Seq2Seq model.
PS45	Java	Automatic Java source code summarization, context-aware, static analysis-based.
PS51	Java	Call dependency integration, Seq2Seq model.
PS54	Java	Context-aware GRU, token type leveraging, variable-size context learning.
PS67	Java,Python	Multi-task BERT-based encoder, API sequence utilization.
PS72	Java	Category-aware code summarization, hybrid model ensemble.
PS78	Java,Python	Hybrid representations, graph attention, Transformer decoder.
PS84	Java,Python	Retrieval-augmented framework, dense retrieval extension.
PS85	Java,Solidity	Multi-modal fusion, Local-ADG, AST integration, similarity network.
PS86	C,Java	PDG-based graph embedding, SBERT semantic evaluation.
PS104	Java,Python	Multi-scale, multi-modal, enhanced ASTs, semantic fusion.
PS105	Java,Python	Multi-modal code representation, graph attention, holistic API usage modeling.
PS110	Java	API context incorporation, ranking mechanism for comment generation.

tant information in the code snippets, and also increase the natural language expression in the summarization. Choi et al. (2023) combined the abstract and extract methods, which not only considered the structure and sequence information of the input code, but also used the retrieved similar code summarization to increase the frequency of keywords when generating code summarization . To improve the quality of code generation, the model fused the original code and the retrieved similar code with a multi-head self-attention mechanism, designed an adaptive attention network and a fusion network to enhance the quality of code generation, and used the extract method to extract important keywords, thereby generating high-quality code summarization. Table 9 enumerates relevant research work on code summarization tasks that have integrated multiple methods in recent years.

Table 9 Comparison of ASCS methods that integrate multiple methods

ID	Language	Main features
PS47	Java,Python	Dual training framework, attention mechanism, cross-task regularization.
PS55	Java,Python	Retrieval-augmented neural summarization, syntactic and semantic code retrieval.
PS58	Java,C#	Attention-based, two-phase model, keyword prediction, semantic gap reduction.
PS63	SQL,Python	End-to-end model, dual learning, multi-task learning, code retrieval, and summarization.
PS64	Java,Python	Retrieval-augmented framework, dual-encoder retrieval.
PS65	Java	Retrieve-and-edit framework, pattern learning, semantic-aware code summarization.
PS66	Java	Ensemble neural models, orthogonality exploitation, performance enhancement.
PS73	Java	Ensemble approach, automated quality assurance.
PS77	C	Hybrid GNN, Retrieval-Augmented, C Language Dataset.
PS80	Java	Combined retrieval and transformer, AST-augmented, code sequence-augmented.
PS81	Java	Multi-modal transformer-based code summarization, contrastive learning retrieval.
PS92	Solidity	Retrieval-enhanced, Solidity-focused, pre-trained model-based comment generation.
PS93	Java,Python	Retrieval-augmented, structural-sequential learning, dual copy mechanism.
PS94	Java,Python	Retrieval-based, meta-learning approach, few-shot learning.
PS95	Java	IR-DL integration, comparative analysis, hybrid summarization strategy.
PS106	Java,Python	Two-stage summarization, model interpretation, code focus reinforcement.
PS114	Various	Extractive-abstractive framework, multi-dataset, cross-language.
PS117	Java,C/C++	Multi-task learning, action word prediction, code summarization enhancement.
PS121	Various	Pipeline framework, redundancy reduction, code summarization enhancement.

Advancements in pre-trained models for programming languages: In recent years, researchers have proposed pre-trained models for multiple programming languages. A pre-trained model is a neural network model that uses a large-scale code dataset to learn a generic code representation that can be adapted to different programming languages or tasks by fine-tuning or zero-shot learning. Feng et al. (2020) proposed a transformer-based bimodal pre-trained model, CodeBERT. The CodeBERT model used a large-scale NL-PL aligned dataset, CodeSearchNet, to pre-train bidirectional encoders between code and natural language, and fine-tuned or applied zero-shot learning on multiple downstream tasks. Wang et al. (2021) designed the CodeT5 model using a large-scale code dataset, CodeXGLUE (Lu et al. 2021), to pre-train text encoders for code, and fine-tuned or applied zero-shot learning on multiple cross-language and cross-domain code tasks. Guo et al. (2021) constructed the GraphCodeBert model using data flows as training objectives to avoid the overhead of deep ASTs. The GraphCodeBert model used a large-scale code dataset, CodeSearchNet, to pre-train a graph encoder for code, and fine-tuned or applied zero-shot learning on multiple code analysis and generation tasks. Table 10 enumerates relevant research work on code summarization tasks that have employed pre-trained models in recent years.

Exploration of large language models (LLMs) in the domain of code summarization: Recently, LLMs have become key tools in text generation, drawing widespread attention. Using deep learning on extensive text data, these models excel at creating coherent and meaningful content. Their impressive performance extends across different fields, including code summarization, attracting significant interest (Zhang et al. 2023). Ahmed and Devanbu (2022a) conducted a study on the Codex (Chen et al. 2021b) model using a few-shot learning approach. They explored the feasibility of LLMs learning with limited data, and whether this method is applicable to the task of code summarization. Sun et al. (2023) examined the performance of using ChatGPT for code summarization generation. They employed carefully crafted heuristic questions and feedback to guide ChatGPT in generating comments that align

Table 10 Comparison of ASCS methods based on pre-trained models

ID	Language	Main features
PS56	Various	Bimodal NL-PL pre-training, Transformer-based, hybrid objective,.
PS69	Java	Self-supervised contrastive learning, code retrieval, semantic-preserving transformations.
PS74	Various	Identifier-aware pre-training, unified NL-PL model, multi-task learning.
PS76	Various	Data flow-based pre-training, structure-aware tasks, Transformer architecture.
PS90	Java,Python	Same-project training, hybrid model, sample efficiency.
PS107	Java,Python	Graph-augmented PL-NL pre-trained model, FA-AST integration.
PS108	Python	Pseudo-language-based summarization, NLP model fine-tuning.
PS1095	Java,Python	Graph-augmented PL-NL model, fine-tuning with FA-AST structure.
PS111	Various	Flexible architecture, diverse pretraining objectives, instruction tuning.

with the distribution. Su and McMillan (2024) investigated the use of samples generated by GPT-3.5 in the knowledge distillation process to train open-source models. They tailored the model to be compact for efficient operation on a single GPU, while sufficiently emulating the performance of GPT-3.5 in code summarization tasks. Table 11 enumerates relevant research work on code summarization tasks that have utilized large language models in recent years.

Advancements and challenges in deep learning based code summarization: Deep learning based methods have become a hot research topic in ASCS, which use multi-layer neural network models to automatically extract and generate summarization from source code. These methods have many advantages, such as being able to handle complex code structures and semantic information, reflecting the functionality and logic of the code; being able to use data-driven methods to automatically learn the features and styles of the summarization, improving the readability and information content of the summarization; being able to use end-to-end methods to convert from source code to natural language, adapting to different programming languages and styles. However, these methods also face a series of challenges. Firstly, they significantly rely on large-scale parallel computing resources, which leads to considerable time and computational costs. Secondly, the presence of low-frequency and out-of-vocabulary words in source code, such as proper nouns, abbreviations, and spelling errors, may threaten the model's generalization ability and output quality. Additionally, the "black box" nature of neural network models results in a lack of interpretability, making it difficult to understand their internal mechanisms and complicating the debugging and improvement of outcomes. At the same time, the content generated by the models has not yet achieved the fluency comparable to natural language (Ferretti and Saletta 2023).

To address these challenges, future research should focus on several key aspects. **First, enhancing the quality of summarization can be achieved by leveraging auxiliary information.** This includes applying information retrieval techniques to identify the most suitable summarization from existing code and annotations, as well as utilizing knowledge graphs and other external sources to bolster the semantic understanding of the code. **Second, there's a need to develop more efficient neural network structures and training strategies.** This can be achieved by employing lighter models or faster optimization algorithms to reduce computational costs. Additionally, advanced pre-training models, such as LLMs, and transfer learning techniques can be utilized to expedite the training process. **Lastly, improving the interpretability and credibility of the summarization is vital.** This can be achieved through

Table 11 Comparison of ASCS methods based on large language models

ID	Language	Main features
PS75	Python	Fine-tuned GPT, functional correctness evaluation, multiple sampling strategy.
PS91	Various	Few-shot learning, LLMs code summarization, project-specific training.
PS112	Python	ChatGPT-based, zero-shot code summarization evaluation.
PS113	Python	Federated LLM, parameter-efficient fine-tuning, privacy-preserving collaboration.
PS115	Java	Knowledge distillation, model efficiency, GPT-3.5 mimicking.
PS116	Various	Automatic semantic augmentation, few-shot learning, multi-language code summarization.

techniques such as attention visualization, which provides insight into the model's attention distribution and generation process. Furthermore, the integration of adversarial generation networks and other error detection and correction techniques can enhance the reliability of the generated results.

6 Quality Evaluation

ASCS as a formidable task in the field of natural language processing, poses challenges not only due to the technical complexity but also because of the subtlety and difficulty in the evaluation process. This is primarily because the industry has not yet established a unified and explicit standard for what constitutes the best code comments. When confronted with the same piece of code, different developers, based on their varying levels of understanding, programming habits, and even personal styles, may produce vastly different code comments. The discrepancies between these comments undoubtedly pose significant challenges for the evaluation process.

In the current research endeavors, two major evaluation methods have been primarily adopted to measure the quality of code summaries: manual evaluation and automatic evaluation. Manual evaluation, as the name suggests, involves inviting experienced developers to act as judges and conduct detailed scoring or ranking of code comments. This method relies on the intuition and experience of professionals, and while it is highly subjective, it still serves as a valuable reference in the absence of objective standards. On the other hand, automatic evaluation borrows assessment metrics from the field of machine translation, attempting to provide a relatively objective judgment of the quality of code comments through quantification. The advantage of this method lies in its ability to quickly process large amounts of data, providing consistent evaluation results and reducing interference from human factors. However, its limitations cannot be overlooked: whether machine translation metrics are fully applicable to the evaluation of code summaries remains a question worthy of further exploration.

Next, this paper will delve into the two aforementioned evaluation methods, analyzing their specific applications and respective advantages and disadvantages, with the aim of providing useful references and insights for future code summary evaluation work.

6.1 Manual Evaluation

In the task of automatic code summarization, human evaluation is a crucial means to assess the quality of code summaries generated by systems. In the early stages, due to the immature nature of automatic evaluation techniques, it was difficult to comprehensively capture the semantics and readability of summaries. Therefore, human evaluation became the primary method for measuring summary quality. Experts and users provided valuable feedback for the initial development of code summarization technology through thorough and detailed assessments. Even with technological advancements, human evaluation remains indispensable. Although automatic evaluation methods, such as BLEU (Papineni et al. 2002) and ROUGE (Lin 2004), can provide quantitative metrics, they often struggle to fully capture the semantics and readability of summaries. Therefore, human evaluation, with its unique depth and granularity, serves as a supplementary means that complements automatic evaluation, ensuring the comprehensiveness and accuracy of evaluation results. This paper will delve into the application of human evaluation in automatic code summarization tasks from several

aspects, including evaluation metrics, evaluation methods, the combination of automatic and human evaluation, as well as common challenges and solutions.

Evaluation Metrics: Evaluation metrics constitute the core elements of human evaluation, determining the focus and direction of the assessment. In the task of code summarization, the primary evaluation metrics encompass readability (Moreno et al. 2013), simplicity (Wong et al. 2013), relevance (Eddy et al. 2013), completeness (McBurney and McMillan 2014), and correctness (McBurney et al. 2014). Readability assesses the fluency and ease of understanding of the generated summary, ensuring that the summary content is easily comprehensible and widely acceptable (Moreno et al. 2013). Simplicity focuses on examining whether the summary can effectively summarize the main functions of the code in a concise text, avoiding lengthy and cumbersome descriptions (Wong et al. 2013). Relevance evaluates the degree of match between the summary content and the actual functionality of the code, ensuring that the summary accurately reflects the intent and logical structure of the code (Eddy et al. 2013). Completeness emphasizes assessing whether the summary comprehensively covers the main functions and logic of the code, rather than being limited to partial descriptions (McBurney and McMillan 2014). Finally, correctness rigorously evaluates the accuracy of the summary content to ensure that there are no erroneous or misleading descriptions (McBurney et al. 2014).

Evaluation Methods: In the early research of code summarization technology, due to the limitations of automatic evaluation methods, the importance of human evaluation became increasingly prominent. Researchers primarily relied on two methods: expert evaluation (Sridhara et al. 2011b; McBurney and McMillan 2014; Rahman et al. 2015; McBurney and McMillan 2016) and user evaluation (Panichella et al. 2012; Wong et al. 2013; Rodeghero et al. 2014; Iyer et al. 2016), to gain a deeper understanding and improve code summarization technology. Expert evaluation is typically conducted by developers or researchers with relevant knowledge and experience, who provide thorough assessments of summaries based on their professional backgrounds. This method emphasizes the accuracy and professionalism of summaries (Sridhara et al. 2011b). However, it may be subject to the subjectivity of experts, leading to deviations in evaluation results. On the other hand, user evaluation is conducted by actual users of the code, and this method better reflects the effectiveness and user satisfaction of summaries in practical applications, focusing on practicality and user feedback (Panichella et al. 2012). Nevertheless, due to differences in user backgrounds, user evaluation may exhibit some bias. With technological advancements, rating scales (Jiang et al. 2017; Liu et al. 2018; Zheng et al. 2019; Wang et al. 2020a) and open-ended feedback (Chen et al. 2021b; Jiang et al. 2023) have gradually been introduced into human evaluation, providing richer and more comprehensive perspectives for the assessment of code summaries. Rating scales quantify summaries based on predefined criteria, helping to reduce subjectivity and enhance evaluation consistency (Jiang et al. 2017). Open-ended feedback allows reviewers to provide detailed written comments, further supplementing the quantitative scores, revealing potential issues in summaries, and suggesting improvements (Chen et al. 2021b). These methods work together, significantly enhancing the comprehensiveness and accuracy of human evaluation.

Combination of Automatic Evaluation and Human Evaluation: Although automatic evaluation methods can quickly provide quantitative evaluations, they often struggle to fully capture the semantic and readability characteristics of summaries. Therefore, researchers typically adopt a combined approach of automatic and human evaluation to compensate for the limitations of each method (Mayer et al. 2023a). Automatic evaluation provides an effective means for rapid screening and initial evaluation, while human evaluation offers a more

detailed and comprehensive quality assessment. This mixed evaluation method enables a more thorough analysis of the quality of generated summaries. Specifically, automatic evaluation can be used initially to filter out low-quality summaries, followed by human evaluation of the higher-quality summaries, thereby enhancing the efficiency and effectiveness of the evaluation process (Li et al. 2024a). This approach not only leverages the efficiency advantages of automatic evaluation but also supplements the details and semantic information that quantitative methods may miss through human evaluation, ensuring that the generated code summaries meet high-quality standards across multiple dimensions. By combining these two methods, researchers can more comprehensively assess the performance of code summarization systems and provide guidance for further system improvements.

Challenges and Solutions: There are several common challenges in the process of human evaluation, including subjectivity, high cost, and consistency issues. Subjectivity refers to the possibility that different reviewers may have different evaluation criteria and preferences, leading to variations in evaluation results. To mitigate this subjectivity, multiple reviewers can be invited to conduct evaluations, and statistical methods (such as Kappa) can be used to measure the consistency among reviewers (Lyu et al. 2021). If necessary, training can be provided to enhance consistency. The high cost refers to the fact that human evaluation requires more time and human resources compared to automatic evaluation, especially for the evaluation of large-scale datasets. Solutions to this issue include the rational selection of review samples and the use of automatic evaluation for initial screening (Roy et al. 2021). Consistency issues refer to ensuring consistent evaluations among different reviewers. This can be addressed through methods such as consistency analysis, discussion, and coordination to ensure the reliability of evaluation results (Wang et al. 2017).

The widespread use of manual evaluation in early code summarization techniques not only highlights its importance in technological development but also lays the foundation for the emergence and evolution of subsequent automatic evaluation methods. With the continuous advancement of technology, it is foreseeable that manual evaluation will continue to be combined with automatic evaluation methods, jointly driving the further development and refinement of code summarization techniques.

6.2 Automatic Evaluation

Automatic evaluation draws on approaches from natural language processing fields such as machine translation and text summarization, and focuses on evaluating the text similarity of generated summarization to reference summarization (Haque et al. 2022). There are two types of evaluation metrics for automatic evaluation: machine translation evaluation metrics and statistical evaluation metrics. Machine translation evaluation metrics such as BLEU (Papineni et al. 2002), METEOR (Banerjee and Lavie 2005), ROUGE (Lin 2004) and CIDER (Vedantam et al. 2015) are commonly used for abstract quality assessment, while statistical evaluation metrics such as precision and recall are less frequently used (Nie et al. 2022; Zhang et al. 2021; Nazar et al. 2016b).

BLEU (Bilingual Evaluation Understudy) (Papineni et al. 2002) is a popular metric for evaluating machine translation tasks, which measures the word choice and fluency of the generated text. It is based on precision, which assumes that the reference text is the best translation possible, and therefore demands high quality word choice from the generated text. It calculates the score by comparing the overlap of n-grams between the candidate text and the reference text. Among them, BLEU-1/2/3/4 indicate the overlap of unigrams, bigrams, trigrams, and four-grams respectively. BLEU-1 reflects how well the words are

translated, and the larger the n value, the more BLEU captures the fluency of the text. BLEU is calculated as:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} \omega_n \log P_n\right) \tag{1}$$

Where n denotes the n-gram, ω_n denotes the weight of the n-gram, P_n denotes the coverage of the n-gram; BP denotes the short sentence penalty factor (brevity penalty), with r denoting the length of the shortest reference translation, and c denoting the length of the candidate translations, then BP is specifically calculated as:

$$BP = \begin{cases} 1, & c > r \\ e^{1-\frac{r}{c}}, & c \leq r \end{cases} \tag{2}$$

Generally speaking, the higher the BLEU score, the better the quality of the candidate text. In practice, BLEU has different variants (Gros et al. 2020). However, BLEU also has some limitations. First, it does not account for semantic and grammatical correctness, and is easily influenced by common words, resulting in some nonsensical or repetitive sentences getting high scores. Second, it gives low scores when the semantics are similar but the words are different. Finally, it favors short sentences, because short sentences are more likely to match with reference texts, and the length penalty factor (brevity penalty) is not enough to counter this bias.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) (Banerjee and Lavie 2005) is a word-based metric that measures how well the generated content covers the reference content. It uses recall to capture the coverage of the reference content by the generated content. It computes the score by matching words between the generated summary and the reference summary. The higher the score, the more similar the generated summary is to the reference summary, indicating better quality. METEOR is calculated as:

$$METEOR = (1 - Pen)F_{means} \tag{3}$$

$$F_{means} = \frac{PR}{\alpha P + (1-\alpha)R} \tag{4}$$

$$P = \frac{m}{c} \tag{5}$$

$$R = \frac{m}{r} \tag{6}$$

Where α is a tunable parameter, m is the number of monads in the candidate translation that can be matched, c is the length of the candidate translation, and r is the length of the reference summary. Pen is a penalty factor that penalizes the word order in the candidate translation for being different from the word order in the reference translation, which is calculated as follows:

$$Pen = \gamma \cdot \left(\frac{ch}{m}\right)^{\beta} \tag{7}$$

where γ, β are tunable parameters, ch is the number of tokens.

Unlike BLEU and ROUGE, METEOR uses WordNet and other knowledge sources to expand the synonym set, which allows it to evaluate the fluency and semantic consistency of the generated content more accurately. It is a widely used metric for assessing the quality of code summarization.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) (Lin 2004) is a metric similar to BLEU, but it focuses on recall instead of precision, which means how much of the information in the reference text is covered by the generated text. This metric is often used to evaluate text summarization.

ROUGE has several variants, such as ROUGE-N, ROUGE-L, ROUGE-W, and ROUGE-S. In code summarization tasks, the most common one is ROUGE-L, which computes the score based on the precision and recall of the longest common subsequence (LCS) between the generated summary and the reference summary. Suppose M and N are generated and reference sentences of lengths a and b, then:

$$P_L = \frac{LCS(M, N)}{a} \tag{8}$$

$$R_L = \frac{LCS(M, N)}{b} \tag{9}$$

$$F_{ROUGE_L} = \frac{(1 + \beta^2) P_L \cdot R_L}{R_L + \beta^2 P_L} \tag{10}$$

Where $\beta_1 = P_L / R_L$, and the F_{ROUGE_L} is the value of ROUGE-L. However, ROUGE-L does not take into account the fluency of the generated summary.

CIDER (Consensus-based Image Description Evaluation) (Vedantam et al. 2015) is an evaluation metric for image captioning tasks. It measures the similarity between the generated summary and the reference summary by using the TF-IDF weights and cosine similarity of n-grams. It then averages the scores of different n-grams to obtain the final evaluation result. The CIDER calculation formula is as follows:

$$CIDER(c, s) = \sum_{n=1}^{N} w_n CIDER_n(c, s) \tag{11}$$

Where the c and s are the generated and reference sentences, n is set from 1 to 4 and the $CIDER_n(c, s)$ score for n-gram is computed using the average cosine similarity between c and s.

Unlike other metrics, CIDER emphasizes the importance of words, rather than matching all words equally. Therefore, CIDER is not only suitable for image captioning, but also for evaluating the quality of code summarization. Some researchers have used CIDER to assess the relevance and informativeness of the generated code summarization.

Evaluating automatic code summarization is a crucial issue that influences the development and enhancement of summarization models. However, there are some limitations in the current automatic evaluation methods, which can be summarized in two main points. First, most of the existing automatic evaluation metrics rely on the matching degree of n-grams, such as BLEU, which overlook the semantics and fluency of the generated summarization, and differ significantly from the subjective evaluation of humans. Stapleton et al. (2020) found that existing automatic evaluation metrics do not truly reflect the quality of generated summarization . Secondly, the BLEU metrics used in previous work have not been consistent. Some work used BLEU-1, some work used BLEU-4, and some work even used BLEU-N. These different BLEU metrics may yield quite different results for the same generated summary, because they have different weights for the matching degree of n-grams. Moreover, some work did not accurately describe or cite the BLEU metrics they used, resulting in the inability of subsequent research to reproduce or verify their results.

To address these issues, some researchers proposed new evaluation metrics for ASCS. For example, Ren et al. (2020) proposed an automatic evaluation metric CodeBLEU, which

combines code syntax and code semantics, making it closer to human evaluation. Shi et al. (2022b) found that BLEU-DC is the best BLEU variant that reflects human perception, which can be used to evaluate neural code summarization models. Vassallo et al. (2014) used BERTScore, a metric, to complement traditional evaluation metrics. BERTScore relies on pre-trained BERT contextual embeddings instead of string matching, which correlates highly with human judgment. However, when evaluating the task of automatic code summarization, we should not merely focus on the differences between the generated summaries and the reference summaries, but also pay more attention to assessing the connection between the generated summaries and their corresponding code. This approach can prevent us from solely catering to the reference summaries and neglecting the actual content of the generated summaries themselves, thereby enhancing the comprehensiveness and practicality of the evaluation (Rani et al. 2023). A comprehensive evaluation method can better reflect the actual value of the generated summaries and prompt the model to generate more accurate and useful summaries.

The code summarization task mostly uses a combination of automated and manual assessments, both of which complement each other to improve the credibility, effectiveness, efficiency, and consistency of the assessment. However, neither of these two evaluation approaches is the most ideal method (Mayer et al. 2023b). Therefore, future research needs to explore more effective rubrics to assess the quality of generated summarization.

7 Discussion and Outlook

ASCS is an important application scenario in the field of program understanding. It aims to generate natural language descriptions of the main functionality and features of source code. This technique has many advantages, such as improving code readability and maintainability, saving developers' time and effort, and facilitating team collaboration. However, it also faces some challenges and limitations, such as relying on data quality, dealing with code diversity, and evaluating summarization quality. Some of the specific challenges and limitations are: semantic understanding of source code, especially for cases involving domain knowledge and context; multi-granularity problem, which requires choosing the appropriate level of information to meet different task requirements; and data bias, which may cause the generated summarization to be too inclined to the patterns in the training data and difficult to generalize to new code.

This paper provides a systematic review of this technique, mainly covering the following aspects: First, it analyzes the challenges faced by ASCS, such as the complexity, diversity and dynamism of source code, and how to extract suitable information from source code for generating summarization. Second, it introduces different types of ASCS methods, such as template-based methods, information retrieval-based methods, deep learning-based methods, etc., and their respective advantages, disadvantages and application scenarios. Third, it summarizes the commonly used datasets and their processing methods, and compares the differences and characteristics of the datasets, as well as how to select appropriate methods and evaluation metrics according to different datasets. Finally, it analyzes the quality evaluation methods of code summarization at different levels, and discusses the pros and cons and applicability of the evaluation methods, as well as how to improve the reliability and comprehensiveness of the evaluation methods.

Although researchers have achieved a number of high-quality research results for the task of automatic code abstract generation, there are still many research points for this task that deserve further attention from researchers:

(1) Enhancing the utilization of source code information to improve code summarization quality. Source code information refers to other information related to the source code, such as comments, documentation, test cases, API calls and so on. This information can provide more semantic information for the generation of code summarization and improve the quality and coverage of code summarization. However, how to utilize source code information more effectively is still a challenging task. For example, how to extract useful information from a large number of cluttered comments, and how to utilize documentation and API calls to enhance the readability and maintainability of code summarization.

(2) Investigating the impact of multi-granularity information on code summarization generation methods. Different code summarization tasks may require different information granularity, e.g., function-level code summarization may require more detailed information, while module-level or project-level code summarization may require more general information. Therefore, it is valuable research to investigate how to choose the appropriate information granularity according to different task requirements, and how to effectively merge and transform between different information granularities. For example, how to utilize different levels of code representations such as abstract syntax tree, control flow graph, data flow graph to generate code summarization, and how to utilize techniques such as attention mechanism, hierarchical encoder, and multi-task learning to achieve the integration and switching of multi-granularity information.

(3) Exploring code summarization generation algorithms that integrate deep and non-deep models. Most of the current researches focus on single methods. However, combining these methods to generate code summarization proves to be an effective strategy. This approach leverages their respective strengths in feature extraction, semantic understanding, and structured representation, while also compensating for deficiencies in generalization ability, robustness, and interpretability (Zhu et al. 2023). There are several research directions: on the one hand, balancing the collaboration and competition between the two to achieve the best results. For example, how to design suitable deep models to assist non-deep models for feature selection and rule learning, and how to design suitable non-deep models to constrain deep models for structure generation and anomaly detection. On the other hand, assess and optimize the impact and contribution of both to improve quality and efficiency. For example, how to determine the proportion and role of the depth model and non-depth model in the overall algorithm, how to use techniques such as meta-learning or multi-task learning to adjust the parameters and weights between the two, etc.

(4) Explore the potential of large language models (LLMs) for applications in the field of code summarization. LLMs are able to leverage their powerful language models and knowledge bases to produce smooth, accurate, and detailed code summarization. At the same time, they have strong transfer learning capabilities that allow them to adapt to the needs of different programming languages and domains. As modeling techniques continue to advance, they are expected to handle more complex code structures and patterns, resulting in more accurate summarization content. In the future, these advanced models may be seamlessly integrated into various development environments to enable real-time generation of code summarization, providing developers with assistance in making it easier to understand and maintain their code. As artificial intelligence continues to evolve, LLMs may become smarter, capable of generating more targeted and useful summarization information based on the context of the code and the needs of the developer. This trend will hopefully lead to more efficient and intelligent aids to the coding process.

(5) Establishing a unified corpus and appropriate evaluation criteria. The currently available corpus has problems such as small scale, low quality and narrow coverage, and the lack of a standard corpus makes it difficult to compare between different methods. At the same time, evaluation criteria are a key factor in measuring the performance of methods. Considering the semantic similarity between generated summarization and reference summarization, developing more effective and reliable automatic evaluation criteria can combine the advantages of manual and automatic evaluation. For this purpose, some methods are needed to build a unified corpus, such as collecting a large amount of high-quality code and comments from open-source communities, extracting useful information from software documentation, and selecting representative samples from different domains and languages. At the same time, some criteria need to be adopted to evaluate the quality of generated summarization, such as indicators based on vocabulary, syntax, semantics, etc., indicators based on manual and automatic evaluation, indicators based on different levels and dimensions, etc.

With the introduction of this survey, we hope that this paper can provide a valuable reference for researchers in related fields, and promote the progress and innovation of this field. At the same time, we also look forward to proposing effective solutions for these challenges, and providing guidance and help for subsequent research work.

Appendix

Table 12 Summary table of related papers

ID	Ref	Source Code Model			Model	Dataset	Evaluation	
		Token	AST	Graph			Automatic	Manual
PS1	Hill et al. (2009)	✓			TB	Others	✓	
PS2	Haiduc et al. (2010a)	✓			IR	Others	✓	✓
PS3	Haiduc et al. (2010b)	✓			IR	Others		✓
PS4	Sridhara et al. (2010)	✓			TB	Others		✓
PS5	Sridhara et al. (2011a)	✓			TB	Others		✓
PS6	Sridhara et al. (2011b)	✓		✓	TB	Others	✓	✓
PS7	Panichella et al. (2012)	✓			IR	Others	✓	✓
PS8	Moreno et al. (2013)	✓	✓		TB	Others		✓
PS9	Wong et al. (2013)	✓			IR	Others	✓	✓
PS10	Movshovitz-Attias and Cohen (2013)	✓			IR	Others	✓	
PS11	Eddy et al. (2013)	✓			IR	Others		✓
PS12	McBurney and McMillan (2014)	✓			TB	Others		✓
PS13	Cortes-Coy et al. (2014)	✓			TB	Others		✓
PS14	Rodeghero et al. (2014)	✓			IR	Others	✓	✓
PS15	McBurney et al. (2014)	✓			IR	Others		✓
PS16	Vassallo et al. (2014)	✓			IR	Others		✓
PS17	Abid et al. (2015)	✓	✓		TB	Others		✓
PS18	Wong et al. (2015)	✓	✓		IR	Others	✓	✓
PS19	Rahman et al. (2015)	✓			IR	Others	✓	✓
PS20	Rodeghero et al. (2015)	✓			IR	Others	✓	✓

Table 12 continued

ID	Ref	Source Code Model			Model	Dataset	Evaluation	
		Token	AST	Graph			Automatic	Manual
PS21	Iyer et al. (2016)	✓			ED	DS1	✓	✓
PS22	Nazar et al. (2016b)	✓			TB	Others	✓	
PS23	Shen et al. (2016)	✓			TB	Others		✓
PS24	McBurney and McMillan (2016)	✓			TB	Others	✓	✓
PS25	Allamanis et al. (2016)	✓			ED	Others	✓	
PS26	Fowkes et al. (2016)	✓	✓		IR	Others		✓
PS27	Rai et al. (2017)	✓	✓		TB,EI	Others		✓
PS28	Barone and Sennrich (2017)	✓			TB	DS2	✓	
PS29	Wang et al. (2017)	✓	✓		TB	Others	✓	✓
PS30	Huang et al. (2017)	✓			IR	Others	✓	✓
PS31	Fowkes et al. (2017)	✓	✓		IR	Others	✓	✓
PS32	Jiang et al. (2017)	✓			ED	Others	✓	✓
PS33	Hu et al. (2018b)	✓			ED,EI	DS3	✓	
PS34	Zeng et al. (2018)	✓			ED	Others	✓	
PS35	Iyer et al. (2018)	✓	✓		ED	Others	✓	
PS36	Hu et al. (2018a)	✓	✓		ED	DS1,DS3	✓	
PS37	Wan et al. (2018)	✓	✓		RL	DS2	✓	
PS38	Allamanis et al. (2018)	✓		✓	ED,GNN	Others	✓	
PS39	Liu et al. (2018)	✓			IR	Others	✓	✓
PS40	Malhotra and Chhabra (2018)	✓			ED	Others	✓	✓
PS41	Liang and Zhu (2018)	✓	✓		ED	Others	✓	
PS42	Chen and Zhou (2018)	✓			IR,ED	DS1	✓	
PS43	LeClair et al. (2019)	✓	✓		ED	DS6	✓	
PS44	Alon et al. (2019b)	✓	✓		ED	Others	✓	
PS45	Al-Msie'deen and Blasi (2019)	✓			TB,EI	Others	✓	
PS46	Chen et al. (2019)	✓	✓		ED	Others	✓	
PS47	Wei et al. (2019)	✓			ED,MA	DS2,DS3	✓	
PS48	Moore et al. (2019)	✓			ED	DS3	✓	
PS49	Zheng et al. (2019)	✓			ED	DS1	✓	✓
PS50	Shido et al. (2019)	✓	✓		ED	DS3	✓	
PS51	Liu et al. (2019)	✓	✓		ED,EI	Others	✓	
PS52	LeClair et al. (2020)	✓		✓	GNN	DS6	✓	
PS53	Ahmad et al. (2020)	✓			TR	DS2,DS3	✓	
PS54	Hussain et al. (2020)	✓			EI	Others	✓	
PS55	Zhang et al. (2020)	✓	✓		MA	DS2,DS3	✓	✓
PS56	Feng et al. (2020)	✓			PT	DS1,DS5	✓	
PS57	Zhou et al. (2020)	✓	✓		ED	DS3	✓	
PS58	Choi et al. (2020)	✓			ED,MA	DS1	✓	
PS59	Hu et al. (2020)	✓	✓		ED	DS3	✓	✓
PS60	Wang et al. (2020a)	✓			TR	DS2,DS3	✓	✓
PS61	Wang et al. (2020b)	✓	✓		TR,RL	DS2	✓	✓

Table 12 continued

ID	Ref	Source Code Model			Model	Dataset	Evaluation	
		Token	AST	Graph			Automatic	Manual
PS62	Huang et al. (2020)	✓	✓		RL	Others	✓	
PS63	Ye et al. (2020)	✓			MA	Others	✓	
PS64	Parvez et al. (2021)	✓	✓	✓	TR,MA	DS9	✓	✓
PS65	Li et al. (2021)	✓			MA	DS6	✓	✓
PS66	LeClair et al. (2021)	✓	✓		MA	DS6	✓	
PS67	Chen et al. (2021a)	✓			TR,EI	DS2,DS3	✓	✓
PS68	Choi et al. (2021)	✓	✓	✓	TR,GNN	DS2,DS3,DS4	✓	✓
PS69	Bui et al. (2021)	✓	✓		PT	Others	✓	
PS70	Zhang et al. (2021)	✓		✓	TR,GNN	DS6	✓	
PS71	Zeng et al. (2021)	✓	✓	✓	TR,GNN	DS6	✓	✓
PS72	Chen et al. (2021c)	✓	✓		EI	DS3	✓	
PS73	Hu et al. (2021)	✓	✓		MA	DS3	✓	✓
PS74	Wang et al. (2021)	✓			PT	DS5,DS9	✓	
PS75	Chen et al. (2021b)	✓			LLM	DS9	✓	✓
PS76	Guo et al. (2021)	✓		✓	PT	DS5	✓	
PS77	Liu et al. (2021a)	✓		✓	MA,GNN	DS8	✓	✓
PS78	Wang et al. (2022b)	✓		✓	TR,EI,GNN	DS2,DS3	✓	✓
PS79	Zhou et al. (2022b)	✓	✓		ED	DS3	✓	✓
PS80	Zhang et al. (2022b)	✓	✓	✓	TR,MA	DS6	✓	
PS81	Lin et al. (2022)	✓	✓		TR,MA	DS6	✓	
PS82	Guo et al. (2022)	✓	✓	✓	TR,GNN	DS2,DS3	✓	
PS83	Wang et al. (2022a)	✓	✓	✓	RL	DS2,DS3	✓	✓
PS84	Cheng et al. (2022)	✓		✓	EI,GNN	DS6,DS10	✓	✓
PS85	Gao et al. (2022)	✓		✓	TR,EI,GNN	DS7	✓	✓
PS86	Son et al. (2022)	✓		✓	TR,EI,GNN	DS3,DS8	✓	
PS87	Tang et al. (2022)	✓	✓		TR	DS2,DS3	✓	
PS88	Shi et al. (2022a)	✓			TR	DS5	✓	
PS89	Zhou et al. (2022a)	✓		✓	TR,GNN	DS3,DS5,DS6	✓	
PS90	Ahmed and Devanbu (2022b)	✓		✓	PT	DS5,DS9	✓	
PS91	Ahmed and Devanbu (2022a)	✓			LLM	DS5,DS9	✓	
PS92	Zhang et al. (2023c)	✓			MA	DS7	✓	✓
PS93	Choi et al. (2023)	✓	✓		MA	DS2,DS3	✓	
PS94	Zhou et al. (2023)	✓			MA	DS2,DS3	✓	✓
PS95	Zhu et al. (2023)	✓	✓		MA	DS3,DS5,DS6	✓	
PS96	Zeng et al. (2023a)	✓	✓	✓	TR	DS5,DS6	✓	✓
PS97	Shi et al. (2023)	✓		✓	TR,GNN	DS2,DS5,DS8	✓	✓
PS98	Zhang et al. (2023a)	✓	✓	✓	TR,RL	DS2,DS3	✓	✓
PS99	Zeng et al. (2023b)	✓	✓		TR	DS2,DS3	✓	
PS100	Gao et al. (2023a)	✓		✓	TR,GNN	DS2,DS3	✓	✓
PS101	Yang et al. (2023a)	✓	✓	✓	TR	DS2,DS3	✓	
PS102	Ji et al. (2023)	✓	✓		TR,GNN	DS2,DS3	✓	

Table 12 continued

ID	Ref	Source Code Model			Model	Dataset	Evaluation	
		Token	AST	Graph			Automatic	Manual
PS103	Lu and Niu (2023)	✓	✓		TR,GNN	DS3	✓	
PS104	Gao et al. (2023b)	✓	✓	✓	TR,EI,GNN	DS2,DS3,DS9	✓	
PS105	Yang et al. (2023b)	✓	✓	✓	EI,GNN	DS2,DS3	✓	
PS106	Geng et al. (2023)	✓			MA	DS5	✓	✓
PS107	Li et al. (2023b)	✓		✓	PT	DS2,DS3	✓	✓
PS108	Ferretti and Saletta (2023)	✓			PT	DS5,DS9	✓	
PS109	Li et al. (2023a)	✓		✓	PT	DS2,DS3	✓	
PS110	Shahbazi and Fard (2023)	✓	✓		EI	DS5	✓	✓
PS111	Wang et al. (2023)	✓			PT	DS5,DS5,DS9	✓	
PS112	Sun et al. (2023)	✓			LLM	DS5	✓	
PS113	Kumar and Chimalakonda (2024)	✓			LLM	DS2	✓	
PS114	Sun et al. (2024)	✓			MA	DS2,DS3,DS5	✓	✓
PS115	Su and McMillan (2024)	✓			LLM	DS6	✓	✓
PS116	Ahmed et al. (2024)	✓	✓	✓	LLM	DS5,DS9	✓	
PS117	Niu et al. (2024)	✓	✓		TR	DS2,DS3,DS5	✓	✓
PS118	Li et al. (2024b)	✓	✓		MA	DS5,DS6,DS10	✓	
PS119	Liang and Huang (2024)	✓	✓		TR	DS2,DS3	✓	
PS120	Guo et al. (2024)	✓	✓		TR	DS5	✓	
PS121	Hu et al. (2024)	✓			MA	DS5	✓	✓

It should be particularly noted that in the "Model" column of the table, the abbreviations represent the following meanings: TB stands for template-based method, IR represents information retrieval-based method, ED denotes early encoder-decoder method, TR signifies Transformer-based method, GNN indicates graph neural network-based method, RL represents deep reinforcement learning-based method, EI denotes the method that utilizes extra information, MA stands for multi-method integration approach, PT represents pre-training method, and LLM signifies large language model-based method

Acknowledgements This work is supported by the National Natural Science Foundation of China (No.62102036) and the Beijing Municipal Natural Science Foundation for Youths (No.4224090).

Data Availability Statement Data availability is not applicable to this article as no new data were created or analyzed in this study.

Declarations

Conflicts of Interests/Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Abid NJ, Dragan N, Collard ML, Maletic JI (2015) Using stereotypes in the automatic generation of natural language summaries for C++ methods. In: Koschke R, Krinke J, Robillard MP (eds) 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, IEEE Computer Society, pp 561–565, https://doi.org/10.1109/ICSM.2015.7332514

Ahmad WU, Chakraborty S, Ray B, Chang K (2020) A transformer-based approach for source code summarization. In: Jurafsky D, Chai J, Schluter N, Tetreault JR (eds) Proceedings of the 58th Annual Meeting

of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, Association for Computational Linguistics, pp 4998–5007, https://doi.org/10.18653/V1/2020.ACL-MAIN.449

Ahmed T, Devanbu PT (2022a) Few-shot training llms for project-specific code-summarization. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022, ACM, pp 177:1–177:5, https://doi.org/10.1145/3551349.3559555

Ahmed T, Devanbu PT (2022b) Learning code summarization from a small and local dataset. https://doi.org/10.48550/ARXIV.2206.00804, arXiv:2206.00804

Ahmed T, Pai KS, Devanbu PT, Barr ET (2024) Automatic semantic augmentation of language model prompts (for code summarization). In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, ACM, pp 220:1–220:13, https://doi.org/10.1145/3597503.3639183

Al-Msie'deen R, Blasi AH (2019) Supporting software documentation with source code summarization. arXiv:1901.01186

Allamanis M, Peng H, Sutton C (2016) A convolutional attention network for extreme summarization of source code. In: Balcan M, Weinberger KQ (eds) Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, JMLR.org, JMLR Workshop and Conference Proceedings, vol 48, pp 2091–2100, http://proceedings.mlr.press/v48/allamanis16.html

Allamanis M, Brockschmidt M, Khademi M (2018) Learning to represent programs with graphs. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, OpenReview.net, https://openreview.net/forum?id=BJOFETxR-

Alon U, Brody S, Levy O, Yahav E (2019a) code2seq: Generating sequences from structured representations of code. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, OpenReview.net, https://openreview.net/forum?id=H1gKYo09tX

Alon U, Zilberstein M, Levy O, Yahav E (2019b) code2vec: learning distributed representations of code. Proc ACM Program Lang 3(POPL):40:1–40:29, https://doi.org/10.1145/3290353

Bai Y, Zhang L, Zhao F (2019) A survey on research of code comment. In: Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, ICMSS 2019, Wuhan, China, January 12-14, 2019, ACM, pp 45–51, https://doi.org/10.1145/3312662.3312710

Banerjee S, Lavie A (2005) METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Goldstein J, Lavie A, Lin C, Voss CR (eds) Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005, Association for Computational Linguistics, pp 65–72, https://aclanthology.org/W05-0909/

Barone AVM, Sennrich R (2017) A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In: Kondrak G, Watanabe T (eds) Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017, Volume 2: Short Papers, Asian Federation of Natural Language Processing, pp 314–319, https://aclanthology.org/I17-2053/

Bui NDQ, Yu Y, Jiang L (2021) Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Diaz F, Shah C, Suel T, Castells P, Jones R, Sakai T (eds) SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021, ACM, pp 511–521, https://doi.org/10.1145/3404835.3462840

Chen F, Kim M, Choo J (2021a) Novel natural language summarization of program code via leveraging multiple input representations. In: Moens M, Huang X, Specia L, Yih SW (eds) Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021, Association for Computational Linguistics, pp 2510–2520, https://doi.org/10.18653/v1/2021.findings-emnlp.214

Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021b) Evaluating large language models trained on code. arXiv:2107.03374

Chen Q, Zhou M (2018) A neural framework for retrieval and summarization of source code. In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, pp 826–831, https://doi.org/10.1145/3238147.3240471

Chen Q, Hu H, Liu Z (2019) Code summarization with abstract syntax tree. In: Gedeon T, Wong KW, Lee M (eds) Neural Information Processing - 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12-15, 2019, Proceedings, Part V, Springer, Communications in Computer and Information Science, vol 1143, pp 652–660, https://doi.org/10.1007/978-3-030-36802-9_69

Chen Q, Xia X, Hu H, Lo D, Li S (2021c) Why my code summarization model does not work: Code comment improvement with category prediction. ACM Trans Softw Eng Methodol 30(2):25:1–25:29, https://doi.org/10.1145/3434280

Chen Z, Monperrus M (2019) A literature study of embeddings on source code. arXiv:1904.03061

Cheng J, Fostiropoulos I, Boehm BW (2021) Gn-transformer: Fusing sequence and graph representation for improved code summarization. arXiv:2111.08874

Cheng W, Hu P, Wei S, Mo R (2022) Keyword-guided abstractive code summarization via incorporating structural and contextual information. Inf Softw Technol 150:106987. https://doi.org/10.1016/J.INFSOF.2022.106987

Choi Y, Kim S, Lee J (2020) Source code summarization using attention-based keyword memory networks. In: Lee W, Chen L, Moon Y, Bourgeois J, Bennis M, Li Y, Ha Y, Kwon H, Cuzzocrea A (eds) 2020 IEEE International Conference on Big Data and Smart Computing, BigComp 2020, Busan, Korea (South), February 19-22, 2020, IEEE, pp 564–570. https://doi.org/10.1109/BigComp48618.2020.00011

Choi Y, Bak J, Na C, Lee J (2021) Learning sequential and structural information for source code summarization. In: Zong C, Xia F, Li W, Navigli R (eds) Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021, Association for Computational Linguistics, Findings of ACL, vol ACL/IJCNLP 2021, pp 2842–2851, https://doi.org/10.18653/v1/2021.findings-acl.251

Choi Y, Na C, Kim H, Lee J (2023) READSUM: retrieval-augmented adaptive transformer for source code summarization. IEEE Access 11:51155–51165. https://doi.org/10.1109/ACCESS.2023.3271992

Cortes-Coy LF, Vásquez ML, Aponte J, Poshyvanyk D (2014) On automatically generating commit messages via summarization of source code changes. In: 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014, IEEE Computer Society, pp 275–284, https://doi.org/10.1109/SCAM.2014.14

Eberhart Z, LeClair A, McMillan C (2020) Automatically extracting subroutine summary descriptions from unstructured comments. In: Kontogiannis K, Khomh F, Chatzigeorgiou A, Fokaefs M, Zhou M (eds) 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020, IEEE, pp 35–46, https://doi.org/10.1109/SANER48275.2020.9054789

Eddy BP, Robinson JA, Kraft NA, Carver JC (2013) Evaluating source code summarization techniques: Replication and expansion. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013, IEEE Computer Society, pp 13–22, https://doi.org/10.1109/ICPC.2013.6613829,

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: A pre-trained model for programming and natural languages. In: Cohn T, He Y, Liu Y (eds) Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020, Association for Computational Linguistics, Findings of ACL, vol EMNLP 2020, pp 1536–1547, https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fernandes P, Allamanis M, Brockschmidt M (2019) Structured neural summarization. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, OpenReview.net, https://openreview.net/forum?id=H1ersoRqtm

Ferretti C, Saletta M (2023) Naturalness in source code summarization. how significant is it? In: 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023, IEEE, pp 125–134, https://doi.org/10.1109/ICPC58990.2023.00027

Fluri B, Würsch M, Gall HC (2007) Do code and comments co-evolve? on the relation between source code and comment changes. In: 14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada, IEEE Computer Society, pp 70–79, https://doi.org/10.1109/WCRE.2007.21

Fowkes JM, Chanthirasegaran P, Ranca R, Allamanis M, Lapata M, Sutton C (2016) TASSAL: autofolding for source code summarization. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, ACM, pp 649–652, https://doi.org/10.1145/2889160.2889171

Fowkes JM, Chanthirasegaran P, Ranca R, Allamanis M, Lapata M, Sutton C (2017) Autofolding for source code summarization. IEEE Trans Software Eng 43(12):1095–1109. https://doi.org/10.1109/TSE.2017.2664836

Gao S, Gao C, He Y, Zeng J, Nie L, Xia X, Lyu MR (2023a) Code structure-guided transformer for source code summarization. ACM Trans Softw Eng Methodol 32(1):23:1–23:32, https://doi.org/10.1145/3522674

Gao X, Jiang X, Wu Q, Wang X, Lyu C, Lyu L (2022) Gt-simnet: Improving code automatic summarization via multi-modal similarity networks. J Syst Softw 194:111495. https://doi.org/10.1016/j.jss.2022.111495

Gao Y, Zhang H, Lyu C (2023) Encosum: enhanced semantic features for multi-scale multi-modal source code summarization. Empir Softw Eng 28(5):126. https://doi.org/10.1007/s10664-023-10384-x

Geng M, Wang S, Dong D, Wang H, Cao S, Zhang K, Jin Z (2023) Interpretation-based code summarization. In: 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023, IEEE, pp 113–124, https://doi.org/10.1109/ICPC58990.2023.00026

Gros D, Sezhiyan H, Devanbu P, Yu Z (2020) Code to comment "translation": Data, metrics, baselining & evaluation. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020, IEEE, pp 746–757, https://doi.org/10.1145/3324884.3416546

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: Pre-training code representations with data flow. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, https://openreview.net/forum?id=jLoC4ez43PZ

Guo J, Liu J, Wan Y, Li L, Zhou P (2022) Modeling hierarchical syntax structure with triplet position for source code summarization. In: Muresan S, Nakov P, Villavicencio A (eds) Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, Association for Computational Linguistics, pp 486–500, https://doi.org/10.18653/v1/2022.acl-long.37

Guo Y, Chai Y, Zhang L, Li H, Luo M, Guo S (2024) Context-based transfer learning for low resource code summarization. Softw Pract Exp 54(3):465–482. https://doi.org/10.1002/spe.3288

Haiduc S, Aponte J, Marcus A (2010a) Supporting program comprehension with source code summarization. In: Kramer J, Bishop J, Devanbu PT, Uchitel S (eds) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, ACM, pp 223–226, https://doi.org/10.1145/1810295.1810335

Haiduc S, Aponte J, Moreno L, Marcus A (2010b) On the use of automated text summarization techniques for summarizing source code. In: Antoniol G, Pinzger M, Chikofsky EJ (eds) 17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA, IEEE Computer Society, pp 35–44, https://doi.org/10.1109/WCRE.2010.13

Haque S, Bansal A, Wu L, McMillan C (2021) Action word prediction for neural source code summarization. In: 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021, IEEE, pp 330–341, https://doi.org/10.1109/SANER50967.2021.00038

Haque S, Eberhart Z, Bansal A, McMillan C (2022) Semantic similarity metrics for evaluating source code summarization. In: Rastogi A, Tufano R, Bavota G, Arnaoudova V, Haiduc S (eds) Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022, ACM, pp 36–47, https://doi.org/10.1145/3524610.3527909

Hill E, Pollock LL, Vijay-Shanker K (2009) Automatically capturing source code context of nl-queries for software maintenance and reuse. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, pp 232–242, https://doi.org/10.1109/ICSE.2009.5070524

Hu X, Li G, Xia X, Lo D, Jin Z (2018a) Deep code comment generation. In: Khomh F, Roy CK, Siegmund J (eds) Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, ACM, pp 200–210, https://doi.org/10.1145/3196321.3196334

Hu X, Li G, Xia X, Lo D, Lu S, Jin Z (2018b) Summarizing source code with transferred API knowledge. In: Lang J (ed) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, ijcai.org, pp 2269–2275, https://doi.org/10.24963/ijcai.2018/314

Hu X, Li G, Xia X, Lo D, Jin Z (2020) Deep code comment generation with hybrid lexical and syntactical information. Empir Softw Eng 25(3):2179–2217. https://doi.org/10.1007/s10664-019-09730-9

Hu X, Zhang X, Lin Z, Zhou D (2024) Reduce redundancy then rerank: Enhancing code summarization with a novel pipeline framework. In: Calzolari N, Kan M, Hoste V, Lenci A, Sakti S, Xue N (eds) Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy, ELRA and ICCL, pp 13722–13733, https://aclanthology.org/2024.lrec-main.1198

Hu Y, Yan M, Liu Z, Chen Q, Wang B (2021) Improving code summarization through automated quality assurance. In: Jin Z, Li X, Xiang J, Mariani L, Liu T, Yu X, Ivaki N (eds) 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021, IEEE, pp 486–497, https://doi.org/10.1109/ISSRE52982.2021.00057

Huang Y, Zheng Q, Chen X, Xiong Y, Liu Z, Luo X (2017) Mining version control system for automatically generating commit comment. In: Bener A, Turhan B, Biffl S (eds) 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017, IEEE Computer Society, pp 414–423, https://doi.org/10.1109/ESEM.2017.56

Huang Y, Huang S, Chen H, Chen X, Zheng Z, Luo X, Jia N, Hu X, Zhou X (2020) Towards automatically generating block comments for code snippets. Inf Softw Technol 127:106373. https://doi.org/10.1016/j.infsof.2020.106373

Husain H, Wu H, Gazit T, Allamanis M, Brockschmidt M (2019) Codesearchnet challenge: Evaluating the state of semantic code search. arXiv:1909.09436

Hussain Y, Huang Z, Zhou Y, Wang S (2020) Codegru: Context-aware deep learning with gated recurrent unit for source code modeling. Inf Softw Technol 125:106309. https://doi.org/10.1016/j.infsof.2020.106309

Iyer S, Konstas I, Cheung A, Zettlemoyer L (2016) Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers, The Association for Computer Linguistics, https://doi.org/10.18653/v1/p16-1195

Iyer S, Konstas I, Cheung A, Zettlemoyer L (2018) Mapping language to code in programmatic context. In: Riloff E, Chiang D, Hockenmaier J, Tsujii J (eds) Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018, Association for Computational Linguistics, pp 1643–1652, https://doi.org/10.18653/v1/d18-1192

Ji R, Tong Z, Luo T, Liu J, Zhang L (2023) A semantic and structural transformer for code summarization generation. In: International Joint Conference on Neural Networks, IJCNN 2023, Gold Coast, Australia, June 18-23, 2023, IEEE, pp 1–9, https://doi.org/10.1109/IJCNN54540.2023.10191735

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: Rosu G, Penta MD, Nguyen TN (eds) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society, pp 135–146, https://doi.org/10.1109/ASE.2017.8115626

Jiang S, Shen J, Wu S, Cai Y, Yu Y, Zhou Y (2023) Towards usable neural comment generation via code-comment linkage interpretation: Method and empirical study. IEEE Trans Software Eng 49(4):2239–2254. https://doi.org/10.1109/TSE.2022.3214859

Kipf TN, Welling M (2017) Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net, https://openreview.net/forum?id=SJU4ayYgl

Kumar J, Chimalakonda S (2024) Code summarization without direct access to code - towards exploring federated llms for software engineering. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024, ACM, pp 100–109, https://doi.org/10.1145/3661167.3661210

LeClair A, McMillan C (2019) Recommendations for datasets for source code summarization. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), Association for Computational Linguistics, pp 3931–3937, https://doi.org/10.18653/v1/n19-1394

LeClair A, Jiang S, McMillan C (2019) A neural model for generating natural language summaries of program subroutines. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp 795–806, https://doi.org/10.1109/ICSE.2019.00087

LeClair A, Haque S, Wu L, McMillan C (2020) Improved code summarization via a graph neural network. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, ACM, pp 184–195, https://doi.org/10.1145/3387904.3389268

LeClair A, Bansal A, McMillan C (2021) Ensemble models for neural source code summarization of subroutines. In: IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021, IEEE, pp 286–297, https://doi.org/10.1109/ICSME52107.2021.00032

Li J, Li Y, Li G, Hu X, Xia X, Jin Z (2021) Editsum: A retrieve-and-edit framework for source code summarization. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE, pp 155–166, https://doi.org/10.1109/ASE51524.2021.9678724

Li J, Li L, Zhu H, Zhang X (2023a) Graphplbart: Code summarization based on graph embedding and pre-trained model. In: Chang S (ed) The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1-10, 2023, KSI Research Inc., pp 304–309, https://doi.org/10.18293/SEKE2023-192

Li J, Zhang Y, Karas Z, McMillan C, Leach K, Huang Y (2024a) Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In: Steinmacher I, Linares-Vásquez M, Moran KP, Baysal O (eds) Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15-16, 2024, ACM, pp 47–51, https://doi.org/10.1145/3643916.3644434

Li L, Li J, Xu Y, Zhu H, Zhang X (2023) Enhancing code summarization with graph embedding and pre-trained model. Int J Softw Eng Knowl Eng 33(11&12):1765–1786. https://doi.org/10.1142/S0218194023410024

Li M, Yu H, Fan G, Zhou Z, Huang J (2023) Classsum: a deep learning model for class-level code summarization. Neural Comput Appl 35(4):3373–3393. https://doi.org/10.1007/S00521-022-07877-Z

Li M, Yu H, Fan G, Zhou Z, Huang Z (2024) Enhancing code summarization with action word prediction. Neurocomputing 563:126777. https://doi.org/10.1016/j.neucom.2023.126777

Liang H, Huang C (2024) Integrating non-fourier and ast-structural relative position representations into transformer-based model for source code summarization. IEEE Access 12:9871–9889. https://doi.org/10.1109/ACCESS.2024.3354390

Liang Y, Zhu KQ (2018) Automatic generation of text descriptive comments for code blocks. In: McIlraith SA, Weinberger KQ (eds) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, AAAI Press, pp 5229–5236, https://doi.org/10.1609/aaai.v32i1.11963

Lin CY (2004) Rouge: A package for automatic evaluation of summaries. In: Text summarization branches out, pp 74–81

Lin L, Huang Z, Yu Y, Liu Y (2022) Multi-modal code summarization with retrieved summary. In: 22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022, IEEE, pp 132–142, https://doi.org/10.1109/SCAM55253.2022.00020

Liu B, Wang T, Zhang X, Fan Q, Yin G, Deng J (2019) A neural-network based code summarization approach by using source code and its call dependencies. In: Internetware '19: The 11th Asia-Pacific Symposium on Internetware, Fukuoka, Japan, October 28-29, 2019, ACM, pp 12:1–12:10, https://doi.org/10.1145/3361242.3362774

Liu S, Chen Y, Xie X, Siow JK, Liu Y (2021a) Retrieval-augmented generation for code summarization via hybrid GNN. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, https://openreview.net/forum?id=zv-typ1gPxA

Liu S, Chen Y, Xie X, Siow JK, Liu Y (2021b) Retrieval-augmented generation for code summarization via hybrid GNN. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, https://openreview.net/forum?id=zv-typ1gPxA

Liu Z, Xia X, Hassan AE, Lo D, Xing Z, Wang X (2018) Neural-machine-translation-based commit message generation: how far are we? In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, pp 373–384, https://doi.org/10.1145/3238147.3238190

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement CB, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. In: Vanschoren J, Yeung S (eds) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual, https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

Lu X, Niu J (2023) Enhancing source code summarization from structure and semantics. In: International Joint Conference on Neural Networks, IJCNN 2023, Gold Coast, Australia, June 18-23, 2023, IEEE, pp 1–7, https://doi.org/10.1109/IJCNN54540.2023.10191872

Lyu C, Wang R, Zhang H, Zhang H, Hu S (2021) Embedding API dependency graph for neural code generation. Empir Softw Eng 26(4):61. https://doi.org/10.1007/S10664-021-09968-2

Ma Z, Gao Y, Lyu L, Lyu C (2022) MMF3: neural code summarization based on multi-modal fine-grained feature fusion. In: Madeiral F, Lassenius C, Conte T, Männistö T (eds) ESEM '22: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki Finland, September 19 - 23, 2022, ACM, pp 171–182, https://doi.org/10.1145/3544902.3546251

Malhotra M, Chhabra JK (2018) Micro level source code summarization of optimal set of object oriented classes. Webology 15(2), http://www.webology.org/2018/v15n2/a175.pdf

Mayer R, Moser M, Geist V (2023a) Leveraging and evaluating automatic code summarization for JPA program comprehension. In: Zhang T, Xia X, Novielli N (eds) IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 768–772, https://doi.org/10.1109/SANER56733.2023.00088

Mayer R, Moser M, Geist V (2023b) Leveraging and evaluating automatic code summarization for JPA program comprehension. In: Zhang T, Xia X, Novielli N (eds) IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 768–772, https://doi.org/10.1109/SANER56733.2023.00088

McBurney PW, McMillan C (2014) Automatic documentation generation via source code summarization of method context. In: Roy CK, Begel A, Moonen L (eds) 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014, ACM, pp 279–290, https://doi.org/10.1145/2597008.2597149

McBurney PW, McMillan C (2016) Automatic source code summarization of context for java methods. IEEE Trans Software Eng 42(2):103–119. https://doi.org/10.1109/TSE.2015.2465386

McBurney PW, Liu C, McMillan C, Weninger T (2014) Improving topic model source code summarization. In: Roy CK, Begel A, Moonen L (eds) 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014, ACM, pp 291–294, https://doi.org/10.1145/2597008.2597793

Moore J, Gelman B, Slater D (2019) A convolutional neural network for language-agnostic source code summarization. In: Damiani E, Spanoudakis G, Maciaszek LA (eds) Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019, SciTePress, pp 15–26, https://doi.org/10.5220/0007678100150026

Moreno L, Aponte J, Sridhara G, Marcus A, Pollock LL, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013, IEEE Computer Society, pp 23–32, https://doi.org/10.1109/ICPC.2013.6613830

Movshovitz-Attias D, Cohen WW (2013) Natural language models for predicting programming comments. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers, The Association for Computer Linguistics, pp 35–40, https://aclanthology.org/P13-2007/

Nazar N, Hu Y, Jiang H (2016) Summarizing software artifacts: A literature review. J Comput Sci Technol 31(5):883–909. https://doi.org/10.1007/s11390-016-1671-1

Nazar N, Jiang H, Gao G, Zhang T, Li X, Ren Z (2016) Source code fragment summarization with small-scale crowdsourcing based features. Frontiers Comput Sci 10(3):504–517. https://doi.org/10.1007/s11704-015-4409-2

Nie P, Zhang J, Li JJ, Mooney RJ, Gligoric M (2022) Impact of evaluation methodologies on code summarization. In: Muresan S, Nakov P, Villavicencio A (eds) Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, Association for Computational Linguistics, pp 4936–4960, https://doi.org/10.18653/v1/2022.acl-long.339

Niu C, Li C, Ng V, Ge J, Huang L, Luo B (2024) Passsum: Leveraging paths of abstract syntax trees and self-supervision for code summarization. J Softw Evol Process 36(6), https://doi.org/10.1002/smr.2620

Panichella S (2018) Summarization techniques for code, change, testing, and user feedback (invited paper). In: Artho C, Ramler R (eds) 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018, IEEE, pp 1–5, https://doi.org/10.1109/VST.2018.8327148

Panichella S, Aponte J, Penta MD, Marcus A, Canfora G (2012) Mining source code descriptions from developer communications. In: Beyer D, van Deursen A, Godfrey MW (eds) IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012, IEEE Computer Society, pp 63–72, https://doi.org/10.1109/ICPC.2012.6240510

Papineni K, Roukos S, Ward T, Zhu W (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA, ACL, pp 311–318, https://doi.org/10.3115/1073083.1073135, https://aclanthology.org/P02-1040/

Parvez MR, Ahmad WU, Chakraborty S, Ray B, Chang K (2021) Retrieval augmented code generation and summarization. In: Moens M, Huang X, Specia L, Yih SW (eds) Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021, Association for Computational Linguistics, pp 2719–2734, https://doi.org/10.18653/v1/2021.findings-emnlp.232

Rahman MM, Roy CK, Keivanloo I (2015) Recommending insightful comments for source code using crowdsourced knowledge. In: Godfrey MW, Lo D, Khomh F (eds) 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015, IEEE Computer Society, pp 81–90, https://doi.org/10.1109/SCAM.2015.7335404

Rai S, Gaikwad T, Jain S, Gupta A (2017) Method level text summarization for java code using nano-patterns. In: Lv J, Zhang HJ, Hinchey M, Liu X (eds) 24th Asia-Pacific Software Engineering Conference, APSEC

2017, Nanjing, China, December 4-8, 2017, IEEE Computer Society, pp 199–208, https://doi.org/10.1109/APSEC.2017.26

Rani P, Blasi A, Stulova N, Panichella S, Gorla A, Nierstrasz O (2023) A decade of code comment quality assessment: A systematic literature review. J Syst Softw 195:111515. https://doi.org/10.1016/j.jss.2022.111515

Ren S, Guo D, Lu S, Zhou L, Liu S, Tang D, Sundaresan N, Zhou M, Blanco A, Ma S (2020) Codebleu: a method for automatic evaluation of code synthesis. arXiv:2009.10297

Rodeghero P, McMillan C, McBurney PW, Bosch N, D'Mello SK (2014) Improving automated source code summarization via an eye-tracking study of programmers. In: Jalote P, Briand LC, van der Hoek A (eds) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, pp 390–401, https://doi.org/10.1145/2568225.2568247

Rodeghero P, Liu C, McBurney PW, McMillan C (2015) An eye-tracking study of java programmers and application to source code summarization. IEEE Trans Software Eng 41(11):1038–1054. https://doi.org/10.1109/TSE.2015.2442238

Roy D, Fakhoury S, Arnaoudova V (2021) Reassessing automatic evaluation metrics for code summarization tasks. In: Spinellis D, Gousios G, Chechik M, Penta MD (eds) ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, ACM, pp 1105–1116, https://doi.org/10.1145/3468264.3468588

Shahbazi R, Fard FH (2023) Apicontext2com: Code comment generation by incorporating pre-defined API documentation. In: 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023, IEEE, pp 13–24, https://doi.org/10.1109/ICPC58990.2023.00012

Shen J, Sun X, Li B, Yang H, Hu J (2016) On automatic summarization of what and why information in source code changes. In: 40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016, IEEE Computer Society, pp 103–112, https://doi.org/10.1109/COMPSAC.2016.162

Shen J, Zhou Y, Wang Y, Chen X, Han T, Chen T (2021) Evaluating code summarization with improved correlation with human assessment. In: 21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021, Hainan, China, December 6-10, 2021, IEEE, pp 990–1001, https://doi.org/10.1109/QRS54544.2021.00108

Shi C, Xiang Y, Yu J, Gao L (2022a) Towards accurate knowledge transfer between transformer-based models for code summarization. In: Peng R, Pantoja CE, Kamthan P (eds) The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022, KSI Research Inc., pp 91–94, https://doi.org/10.18293/SEKE2022-111

Shi C, Cai B, Zhao Y, Gao L, Sood K, Xiang Y (2023) Coss: Leveraging statement semantics for code summarization. IEEE Trans Software Eng 49(6):3472–3486. https://doi.org/10.1109/TSE.2023.3256362

Shi E, Wang Y, Du L, Chen J, Han S, Zhang H, Zhang D, Sun H (2022b) On the evaluation of neural code summarization. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, ACM, pp 1597–1608, https://doi.org/10.1145/3510003.3510060

Shi L, Mu F, Chen X, Wang S, Wang J, Yang Y, Li G, Xia X, Wang Q (2022c) Are we building on the rock? on the importance of data preprocessing for code summarization. In: Roychoudhury A, Cadar C, Kim M (eds) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM, pp 107–119, https://doi.org/10.1145/3540250.3549145

Shido Y, Kobayashi Y, Yamamoto A, Miyamoto A, Matsumura T (2019) Automatic source code summarization with extended tree-lstm. In: International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019, IEEE, pp 1–8, https://doi.org/10.1109/IJCNN.2019.8851751

Son J, Hahn J, Seo H, Han Y (2022) Boosting code summarization by embedding code structures. In: Calzolari N, Huang C, Kim H, Pustejovsky J, Wanner L, Choi K, Ryu P, Chen H, Donatelli L, Ji H, Kurohashi S, Paggio P, Xue N, Kim S, Hahm Y, He Z, Lee TK, Santus E, Bond F, Na S (eds) Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022, International Committee on Computational Linguistics, pp 5966–5977, https://aclanthology.org/2022.coling-1.521

Song X, Sun H, Wang X, Yan J (2019) A survey of automatic generation of source code comments: Algorithms and techniques. IEEE Access 7:111411–111428. https://doi.org/10.1109/ACCESS.2019.2931579

Song Z, Shang X, Li M, Chen R, Li H, Guo S (2022) Do not have enough data? an easy data augmentation for code summarization. In: 13th IEEE International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2022, Beijing, China, November 25-27, 2022, IEEE, pp 1–6, https://doi.org/10.1109/PAAP56126.2022.10010698

Song Z, Zeng H, Shang X, Li G, Li H, Guo S (2023) An data augmentation method for source code summarization. Neurocomputing 549:126385. https://doi.org/10.1016/j.neucom.2023.126385

Sridhara G, Hill E, Muppaneni D, Pollock LL, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Pecheur C, Andrews J, Nitto ED (eds) ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010, ACM, pp 43–52, https://doi.org/10.1145/1858996.1859006

Sridhara G, Pollock LL, Vijay-Shanker K (2011a) Automatically detecting and describing high level actions within methods. In: Taylor RN, Gall HC, Medvidovic N (eds) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, ACM, pp 101–110, https://doi.org/10.1145/1985793.1985808

Sridhara G, Pollock LL, Vijay-Shanker K (2011b) Generating parameter comments and integrating with method summaries. In: The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011, IEEE Computer Society, pp 71–80, https://doi.org/10.1109/ICPC.2011.28

Stapleton S, Gambhir Y, LeClair A, Eberhart Z, Weimer W, Leach K, Huang Y (2020) A human study of comprehension and code summarization. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, ACM, pp 2–13, https://doi.org/10.1145/3387904.3389258

Su C, McMillan C (2024) Distilled GPT for source code summarization. Autom Softw Eng 31(1):22. https://doi.org/10.1007/s10515-024-00421-4

Sun W, Fang C, You Y, Miao Y, Liu Y, Li Y, Deng G, Huang S, Chen Y, Zhang Q, Qian H, Liu Y, Chen Z (2023) Automatic code summarization via chatgpt: How far are we? CoRR abs/2305.12865, https://doi.org/10.48550/ARXIV.2305.12865, arXiv:2305.12865

Sun W, Fang C, Chen Y, Zhang Q, Tao G, You Y, Han T, Ge Y, Hu Y, Luo B, Chen Z (2024) An extractive-and-abstractive framework for source code summarization. ACM Trans Softw Eng Methodol 33(3):75:1–75:39, https://doi.org/10.1145/3632742

Tang Z, Shen X, Li C, Ge J, Huang L, Zhu Z, Luo B (2022) Ast-trans: Code summarization with efficient tree-structured attention. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, ACM, pp 150–162, https://doi.org/10.1145/3510003.3510224

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2018) Deep learning similarities from different representations of source code. In: Zaidman A, Kamei Y, Hill E (eds) Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, ACM, pp 542–553, https://doi.org/10.1145/3196398.3196431

Vassallo C, Panichella S, Penta MD, Canfora G (2014) CODES: mining source code descriptions from developers discussions. In: Roy CK, Begel A, Moonen L (eds) 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014, ACM, pp 106–109, https://doi.org/10.1145/2597008.2597799

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: Guyon I, von Luxburg U, Bengio S, Wallach HM, Fergus R, Vishwanathan SVN, Garnett R (eds) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp 5998–6008, https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Vedantam R, Zitnick CL, Parikh D (2015) Cider: Consensus-based image description evaluation. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015, IEEE Computer Society, pp 4566–4575, https://doi.org/10.1109/CVPR.2015.7299087

Wan Y, Zhao Z, Yang M, Xu G, Ying H, Wu J, Yu PS (2018) Improving automatic source code summarization via deep reinforcement learning. In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, pp 397–407, https://doi.org/10.1145/3238147.3238206

WANG J, XUE X, WENG W (2015) Source code summarization technology based on syntactic analysis. J Comput Appl 35(7):1999

Wang R, Zhang H, Lu G, Lyu L, Lyu C (2020) Fret: Functional reinforced transformer with BERT for code summarization. IEEE Access 8:135591–135604. https://doi.org/10.1109/ACCESS.2020.3011744

Wang W, Zhang Y, Zeng Z, Xu G (2020b) Trans ^ 3: A transformer-based framework for unifying code summarization and code search. arXiv:2003.03238

Wang W, Zhang Y, Sui Y, Wan Y, Zhao Z, Wu J, Yu PS, Xu G (2022) Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Trans Software Eng 48(2):102–119. https://doi.org/10.1109/TSE.2020.2979701

Wang X, Pollock LL, Vijay-Shanker K (2017) Automatically generating natural language descriptions for object-related statement sequences. In: Pinzger M, Bavota G, Marcus A (eds) IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, IEEE Computer Society, pp 205–216, https://doi.org/10.1109/SANER.2017.7884622

Wang Y, Wang W, Joty SR, Hoi SCH (2021) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens M, Huang X, Specia L, Yih SW (eds) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics, pp 8696–8708, https://doi.org/10.18653/v1/2021.emnlp-main.685

Wang Y, Dong Y, Lu X, Zhou A (2022b) Gypsum: learning hybrid representations for code summarization. In: Rastogi A, Tufano R, Bavota G, Arnaoudova V, Haiduc S (eds) Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022, ACM, pp 12–23, https://doi.org/10.1145/3524610.3527903

Wang Y, Le H, Gotmare A, Bui NDQ, Li J, Hoi SCH (2023) Codet5+: Open code large language models for code understanding and generation. In: Bouamor H, Pino J, Bali K (eds) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023, Association for Computational Linguistics, pp 1069–1088, https://doi.org/10.18653/v1/2023.emnlp-main.68

Wei B, Li G, Xia X, Fu Z, Jin Z (2019) Code generation as a dual task of code summarization. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R (eds) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp 6559–6569, https://proceedings.neurips.cc/paper/2019/hash/e52ad5c9f751f599492b4f087ed7ecfc-Abstract.html

Wong E, Yang J, Tan L (2013) Autocomment: Mining question and answer sites for automatic comment generation. In: Denney E, Bultan T, Zeller A (eds) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE, pp 562–567, https://doi.org/10.1109/ASE.2013.6693113

Wong E, Liu T, Tan L (2015) Clocom: Mining existing source code for automatic comment generation. In: Guéhéneuc Y, Adams B, Serebrenik A (eds) 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, IEEE Computer Society, pp 380–389, https://doi.org/10.1109/SANER.2015.7081848

Wu H, Zhao H, Zhang M (2021) Code summarization with structure-induced transformer. In: Zong C, Xia F, Li W, Navigli R (eds) Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021, Association for Computational Linguistics, Findings of ACL, vol ACL/IJCNLP 2021, pp 1078–1090, https://doi.org/10.18653/v1/2021.findings-acl.93

Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2018) Measuring program comprehension: A large-scale field study with professionals. IEEE Trans Software Eng 44(10):951–976. https://doi.org/10.1109/TSE.2017.2734091

Yang K, Mao X, Wang S, Qin Y, Zhang T, Lu Y, Al-Sabahi K (2023a) An extensive study of the structure features in transformer-based code semantic summarization. In: 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023, IEEE, pp 89–100, https://doi.org/10.1109/ICPC58990.2023.00024

Yang K, Wang J, Song Z (2023) Learning a holistic and comprehensive code representation for code summarization. J Syst Softw 203:111746. https://doi.org/10.1016/j.jss.2023.111746

Ye W, Xie R, Zhang J, Hu T, Wang X, Zhang S (2020) Leveraging code generation to improve code retrieval and summarization via dual learning. In: Huang Y, King I, Liu T, van Steen M (eds) WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020, ACM / IW3C2, pp 2309–2319, https://doi.org/10.1145/3366423.3380295

Zeng J, Zhang T, Xu Z (2021) Dg-trans: Automatic code summarization via dynamic graph attention-based transformer. In: 21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021, Hainan, China, December 6-10, 2021, IEEE, pp 786–795, https://doi.org/10.1109/QRS54544.2021.00088

Zeng J, He Y, Zhang T, Xu Z, Han Q (2023) Clg-trans: Contrastive learning for code summarization via graph attention-based transformer. Sci Comput Program 226:102925. https://doi.org/10.1016/j.scico.2023.102925

Zeng J, Qu Z, Cai B (2023) Structure and sequence aligned code summarization with prefix and suffix balanced strategy. Entropy 25(4):570. https://doi.org/10.3390/e25040570

Zeng L, Zhang X, Wang T, Li X, Yu J, Wang H (2018) Improving code summarization by combining deep learning and empirical knowledge (S). In: Pereira ÓM (ed) The 30th International Conference on Software

Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018, KSI Research Inc. and Knowledge Systems Institute Graduate School, pp 566–565, https://doi.org/10.18293/SEKE2018-191

Zhang C, Wang J, Zhou Q, Xu T, Tang K, Gui H, Liu F (2022) A survey of automatic source code summarization. Symmetry 14(3):471. https://doi.org/10.3390/sym14030471

Zhang C, Zhou Q, Qiao M, Tang K, Xu L, Liu F (2022) Re_trans: Combined retrieval and transformer model for source code summarization. Entropy 24(10):1372

Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019) A novel neural source code representation based on abstract syntax tree. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp 783–794, https://doi.org/10.1109/ICSE.2019.00086

Zhang J, Wang X, Zhang H, Sun H, Liu X (2020) Retrieval-based neural source code summarization. In: Rothermel G, Bae D (eds) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, ACM, pp 1385–1397, https://doi.org/10.1145/3377811.3380383

Zhang M, Zhou G, Yu W, Huang N, Liu W (2023a) GA-SCS: graph-augmented source code summarization. ACM Trans Asian Low Resour Lang Inf Process 22(2):53:1–53:19, https://doi.org/10.1145/3554820

Zhang X, Yang S, Duan L, Lang Z, Shi Z, Sun L (2021) Transformer-xl with graph neural network for source code summarization. In: 2021 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2021, Melbourne, Australia, October 17-20, 2021, IEEE, pp 3436–3441, https://doi.org/10.1109/SMC52423.2021.9658619

Zhang X, Chen L, Zou W, Cao Y, Ren H, Wang Z, Li Y, Zhou Y (2024) ICG: A machine learning benchmark dataset and baselines for inline code comments generation task. Int J Softw Eng Knowl Eng 34(2):331–356. https://doi.org/10.1142/S0218194023500547

Zhang Z, Chen C, Liu B, Liao C, Gong Z, Yu H, Li J, Wang R (2023b) A survey on language models for code. arXiv:2311.07989

Zhang Z, Chen S, Fan G, Yang G, Feng Z (2023c) CCGRA: smart contract code comment generation with retrieval-enhanced approach. In: Chang S (ed) The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1-10, 2023, KSI Research Inc., pp 212–217, https://doi.org/10.18293/SEKE2023-090

Zheng W, Zhou H, Li M, Wu J (2017) Code attention: Translating code to comments by exploiting domain features. arXiv:1709.07642

Zheng W, Zhou H, Li M, Wu J (2019) Codeattention: translating source code to comments by exploiting the code constructs. Front Comput Sci 13(3):565–578. https://doi.org/10.1007/s11704-018-7457-6

Zhou Y, Liu S, Siow JK, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R (eds) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp 10197–10207, https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

Zhou Y, Shen J, Zhang X, Yang W, Han T, Chen T (2022) Automatic source code summarization with graph attention networks. J Syst Softw 188:111257. https://doi.org/10.1016/j.jss.2022.111257

Zhou Z, Yu H, Fan G (2020) Effective approaches to combining lexical and syntactical information for code summarization. Softw Pract Exp 50(12):2313–2336. https://doi.org/10.1002/spe.2893

Zhou Z, Yu H, Fan G, Huang Z, Yang X (2022) Summarizing source code with hierarchical code representation. Inf Softw Technol 143:106761. https://doi.org/10.1016/j.infsof.2021.106761

Zhou Z, Yu H, Fan G, Huang Z, Yang K (2023) Towards retrieval-based neural code summarization: A meta-learning approach. IEEE Trans Software Eng 49(4):3008–3031. https://doi.org/10.1109/TSE.2023.3238161

Zhu T, Li Z, Pan M, Shi C, Zhang T, Pei Y, Li X (2023) Revisiting information retrieval and deep learning approaches for code summarization. In: 45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023, IEEE, pp 328–329, https://doi.org/10.1109/ICSE-Companion58688.2023.00091

Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q (2020) Smart contract vulnerability detection using graph neural network. In: Bessiere C (ed) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, pp 3283–3290, https://doi.org/10.24963/ijcai.2020/454