

A Survey of Automatic Source Code Summarization

Chunyan Zhang ¹, Junchao Wang ¹, Qinglei Zhou ², Ting Xu ², Ke Tang ¹, Hairen Gui ¹ and Fudong Liu ^{1,*}

¹ State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China; iecyzhang@163.com (C.Z.); wangjunchao11@126.com (J.W.); tuck3r@foxmail.com (K.T.); guihairen@163.com (H.G.)

² School of Information Engineering, Zhengzhou University, Zhengzhou 450001, China; ieqlzhou@zzu.edu.cn (Q.Z.); ietxu@zzu.edu.cn (T.X.)

* Correspondence: lwfydy@126.com

Abstract: Source code summarization refers to the natural language description of the source code's function. It can help developers easily understand the semantics of the source code. We can think of the source code and the corresponding summarization as being symmetric. However, the existing source code summarization is mismatched with the source code, missing, or out of date. Manual source code summarization is inefficient and requires a lot of human efforts. To overcome such situations, many studies have been conducted on Automatic Source Code Summarization (ASCS). Given a set of source code, the ASCS techniques can automatically generate a summary described with natural language. In this paper, we give a review of the development of ASCS technology. Almost all ASCS technology involves the following stages: source code modeling, code summarization generation, and quality evaluation. We further categorize the existing ASCS techniques based on the above stages and analyze their advantages and shortcomings. We also draw a clear map on the development of the existing algorithms.

Keywords: source code summarization; deep learning; program analysis; neural machine translation



Citation: Zhang, C.; Wang, J.; Zhou, Q.; Xu, T.; Tang, K.; Gui, H.; Liu, F. A Survey of Automatic Source Code Summarization. *Symmetry* **2022**, *14*, 471. <https://doi.org/10.3390/sym14030471>

Academic Editor: Theodore E. Simos

Received: 27 January 2022

Accepted: 20 February 2022

Published: 25 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Code summarization, also called code comment, is a text description for the function and purpose of special identifiers in computer programs. In other words, code summarization explains the logic and functions of source code in natural language, in order to make people understand the program more easily [1]. As we all know, program maintenance is the most expensive and time-consuming stage in the software life cycle [2]. A high-quality code summarization is essential to program comprehension and maintenance [3–5]. For example, it can reduce the time needed for developers to understand the source code and improve code search efficiency. Unfortunately, with the rapid update of software, most code comments are mismatched, outdated, and missing, so the code comments need to be improved and updated continuously.

Given the importance and urgency, significant achievements have been made in ASCS [6–27]. To the best of our knowledge, there are only a few works on the survey of code summarization generation. Nazar et al. [28] mainly summarized four software artifacts, which include bug reports, mailing list, source code, and developer discussions. Yang et al. [29] focused on four aspects: the code comment generation, the consistency of code and comments, the classification of code comments, and the quality evaluation of code comments, but they did not explain the relevant algorithms of code comments in detail. In addition, there exist a few studies on the technical achievements in the last five years. Chen et al. [30] summarized the types of source code embedding, including tokens, methods, sequences, binary codes, and other granularities. Moreover, they also provided an available code embedding list to other researchers. Song et al. [31] ran a survey on the algorithms and techniques of the code comments generation. They provided

the classification, design principles, and quality evaluation of automatic code comments algorithms. However, with the rapid development of deep learning technology, there are still high-quality works in recent years that have not been summarized.

As an emerging research, ASCS is inspired by machine translation modeling. Although there has only been just over 10 years of research [31], it has brought software development to a new level, especially in program comprehension and maintenance. However, ASCS faces the following challenges.

1. The source code modeling used to extract features from source code, such as lexical information, syntax information, and semantics information. Here are the main difficulties:
 - Different from plain text, the source code contains rich syntax and structure information;
 - Faced with different programming languages, both the analysis methods and standards are different;
 - Each developer has his own code logic and naming conventions, which makes the source code irregular;
 - Various programming languages and identifier names lead to huge vocabularies of source code.
2. In terms of ASCS methods, the Information Retrieval (IR)-based algorithms extract keywords from the source code, or look for summarization of similar codes. Distinctly, the effect of these algorithms depends on the quality of datasets. With the development of artificial intelligence technology, researchers have applied the methods of Natural Language Processing (NLP) for ASCS, and have achieved significant results. However, there are still some problems, such as long-term dependence, the limitations of ASCS algorithms, and the lack of high-quality datasets.
3. The quality evaluation is another challenge. The existing ASCS algorithms are mostly evaluated on different datasets, which makes it difficult to compare the effect of the algorithms. Besides, the quality evaluation methods of NLP are used for code summarization, but the source code is different from natural language text. Thus, an efficient and low-cost ASCS evaluation method is an important issue that needs to be solved urgently. Here are the main challenges of quality evaluation:
 - Lacking unified datasets;
 - Lacking recognized and reasonable benchmarks used as the baseline;
 - Lacking professional evaluation indicators.

We select representative papers in the past decade that have published in IEEE ICSE, IEEE/ACM ASE, IEEE TSE, IEEE FSE, ACM TOSEM, ICPC, IJCAI, EMSE, AAAI, or other software engineering and artificial intelligence venues. According to the technique development, we summarize the work from three aspects: source code modeling, automatic code summarization algorithms, and the summarization quality evaluation.

This survey makes the following contributions to the field:

- It discusses the origin of ASCS technology, its evolution over the past decade;
- It concludes the advantages and disadvantages of the above representative methods, and the work with improved relationships between these approaches;
- It provides a synthesized summary of the challenges and future research that requires the attention of the researchers;
- It collates a comprehensive list of available datasets and codes, which are conducive to further research by scholars.

This paper is organized as follows. Section 2 provides the general overall flowchart of ASCS algorithms, and analyzes the key steps in flowchart. Section 3 systematizes various representations of source code, including their pros and cons. Section 4 summarizes the methods of ASCS, even the limitations and interconnections of these methods. Section 5

evaluates the different quality metrics. Section 6 draws the conclusion and the future trends of ASCS.

2. Overview of ASCS

Since all the functions of software are included in the code, code summarization is the most intuitive and effective way to understand the software, especially for software maintenance. Generally, comments can be divided into three types: document comment, block comment, and line comment [32]. Among them, document comments are used to describe a class, a method, or an attribute; block comments are used to describe one or more lines of code snippets, which are located on the previous line of the commented code block; line comments refer to following the code line to provide a description for the current code line. At present, almost all researchers are devoted to document comments and block comments.

The overall flowchart of ASCS generally contain three parts: source code modeling, code summarization generation, and quality evaluation, as shown in Figure 1.

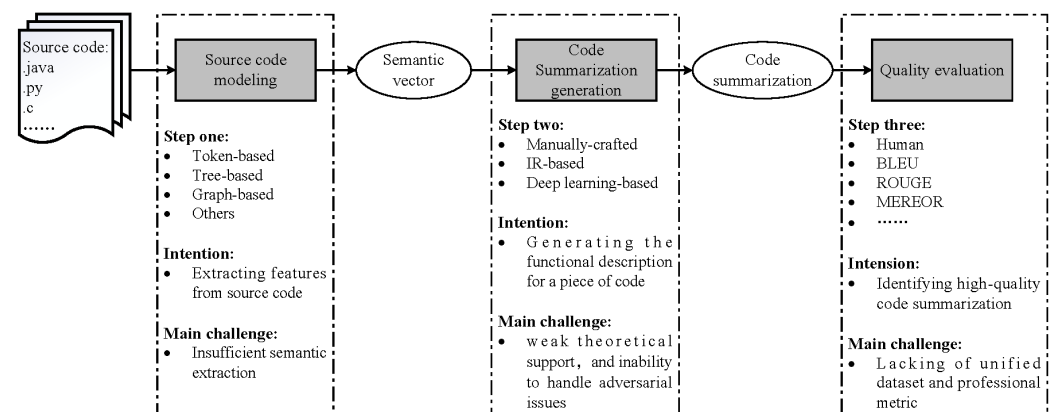


Figure 1. The overall flowchart of automatic source code summarization.

The source code modeling converts the source code into semantic vectors by extracting its features, and the semantic vectors are the input of ASCS generation. In software engineering, building a high-quality source code model is the first step of many tasks, such as code classification [33–35], code search [25,36,37], code clone detection [38–40] and code comment [41–44]. In recent years, almost all source code modeling uses machine learning, and paper [45] can be used as a reference. It carried out an extensive literature search and identified 364 primary studies published between 2002 and 2021, aiming to summarize the current knowledge in the area of applied machine learning for source code analysis. As a key step in ASCS, source code modeling adopts different granularities and methods to represent the source code adequately, which are detailed in Section 3.

The ASCS generation algorithms take the outputs of the source code model as an input, then generate the functional description of the source code. Initially, the code summarization has been generated by extracting keywords from the source code, building bag-of-words model, and using similarity matching. With the development of technology, models based on deep learning (DL) have been widely used, such as seq2seq model, transformer model, and other language models. Among them, the seq2seq model usually applies recurrent neural network (RNN) and its variants (e.g., Long Short Term Memory (LSTM), Gate Recurrent Unit (GRU)), convolution neural network (CNN), and its variants (e.g., Graph Convolutional Networks (GCN), Graph Neural Network (GNN)). Furthermore, attention mechanism is often used as a powerful aid to the model. Section 4 presents the analysis of different excellent algorithms of the ASCS generation.

The quality evaluation measures the pros and cons of ASCS algorithms through the generated code summarization. We discuss the quality evaluation from evaluation objects and standards. The evaluation objects of ASCS are often datasets and ASCS methods.

The evaluation standards are often automatic metrics (such as BLEU, METEOR, CIDER, ROUGE, Recall and Predicate) and human evaluation, which are detailed in Section 5.

3. Source Code Modeling

High quality source code modeling can better extract syntactic and semantic information from the source code. We also think that the source code before and after source code modeling is also symmetric. There exists a lot of excellent literature on source code analysis [46–58], and we have shown the representative source code models from the past five years in Table 1. From Table 1, we know that scholars have considered structural information (e.g., AST, API) in recent years, instead of just treating the source code as pure text. Besides, ML-based methods are widely used in source code analysis. According to the different representations of source code, we divide the source code modeling into four types: Token-based, Tree-based, Graph-based, and other source code modeling.

Table 1. Representative source code models from the last five years. Seq means sequence feature. Stc means structure feature. ML means machine learning. IST means Inf Softw Technol. Src means source code. SCS means source code summarization. CR means code retrieval. SCC means source code classification. PF means predict function. CCD means code clone detection. SCG means source code generation. PR means program repair. ECM means error-code misuse. CP means classifying programs. ✓ indicates used, × indicates unused.

Models	Year	Venue	Seq	Stc	ML	Src Representation	Tasks
Attention [15]	2016	ICML	✓	×	✓	Subtokens	SCS
TBCNN [34]	2016	AAAI	✓	✓	✓	AST	CP, CCD
LSTM [59]	2016	ACL	✓	×	✓	tokens	SCS, CR
SWUM [60]	2016	TSE	✓	×	✓	tokens	SCS
VSM [61]	2016	ICSE	✓	✓	×	AST	SCS
LSTM [40]	2017	IJCAI	✓	✓	✓	AST	CCD
LSTM [6]	2018	ICPC	✓	✓	✓	AST	SCS
Seq2seq [17]	2018	IJCAJ	✓	×	✓	(API, comments) (API, code, comments)	SCS
LSTM [19]	2018	ASE	✓	✓	✓	AST, sourcecode	SCS
Bi-LSTM [43]	2018	ICLR	✓	✓	✓	AST, (token, path, token)	PF, SCS
HOPE [50]	2018	MSR	✓	✓	✓	identifier, AST, CFG, Bytecode	CCD
RNN [51]	2018	ICLR	✓	✓	✓	variable/statetrace	PR
Word2vec [53]	2018	ESEC	✓	✓	✓	abstractedsymbolictraces	ECM
GGNN [55]	2018	ICLR	✓	✓	✓	AST, PDG	PF
MLP [62]	2018	ASE	✓	×	✓	tokens	SCS
RNN [63]	2018	AAAI	✓	✓	✓	AST	SCS, SCC
GRU [22]	2019	ICSE	✓	✓	✓	text, ASTnodetokens	SCS
Bi-LSTM [46]	2019	ICSE	✓	✓	✓	AST, ST-trees	SCC, CCD
Bi-LSTM [64]	2019	POPL	✓	✓	✓	AST, (token, path, token)	PF
Bi-LSTM [65]	2019	ASE	✓	✓	✓	text	SCS
Bi-LSTM [23]	2020	ICSE	×	✓	✓	AST, codesequence	SCS
BERT [24]	2020	Access	✓	✓	✓	functionalkeywords	SCS
Transformer [25]	2020	arXiv	✓	×	✓	comments, code	SCS, CR
Transformer [26]	2020	ACL	×	✓	✓	AST	SCS
GRU [44]	2020	ESE	✓	✓	✓	tokens, AST	SCS
Regularizer [56]	2020	IST	✓	✓	✓	AST	SCG
GRU [66]	2020	JCRD	✓	✓	✓	(code, API, comments), (function, comments)	SCS
GRU [67]	2020	ACL	✓	✓	✓	text, ASTnodetokens	SCS
GNN [68]	2021	arXiv	✓	✓	✓	AST, context	SCS
Seq2Seq [69]	2021	ICPC	✓	×	✓	(seq, comment), (context, comment)	SCS
API2Com [70]	2021	ICPC	✓	✓	✓	AST, API, seq	SCS


3.1. Token-Based Source Code Model

The ASCS generation is inspired by the machine translation model in the natural language processing (NLP) field. Therefore, the early source code analysis methods fully borrowed the methods of NLP. The source code is regarded as pure text sequences [7,9–13,15,21,37,52,58–60,62,66]. This token-based model processes the source code as shown in Figure 2.

```

1. int min(int firstNumber, int secondNumber)
2. {
3.   if(firstNumber > secondNumber){
4.     return secondNumber;
5.   }
6.   else{
7.     return firstNumber;
8.   }
9. }

```



(int, min, if, firstNumber, secondNumber, else, return)

Figure 2. An example of token-based source code representation. The left side of the blue arrow is source code, and the right side is the token-based representation of source code.

To the best of our knowledge, the first code comment generation work came from Haiduc et al. [8]. They analyzed source code as text to generate natural language description of classes or methods of source code described with objective-oriented programming languages. Then, Moreno et al. [11] and Wang et al. [58] used part-of-speech tagging to identify the keywords that best represented the source code features. Efstathiou et al. [52] expressed the source code by mining the identifier information of multiple programming languages, and utilized fastText model to learn word representation.

However, the above methods of source code representation all extract keywords or topics from the source code. Besides, there are a variety of auxiliaries used to extract the source code information, such as API knowledge, document description, identifier naming rules, and so on. For example, Hu et al. [17] discovered that the function of code was related to the API call sequences, then used API information to assist the modeling of generating source code summarization. Different from using API calls, Chen et al. [62] constructed a source code model by using two variational autoencoders (VAE) to complete source code analysis. Among them, C-VAE was used to extract the token sequences of source code, and L-VAE was used to extract the token sequences of the auxiliary documents. Especially, Zheng et al. [21] proposed an attention mechanism, CodeAttention, extracting keywords, symbols, and identifiers information from the source code. They also provided an available and large dataset, C2CGit, which includes java methods and the corresponding comments.

The above methods mostly regard the source code as natural language text, then select tokens to form the bag-of-words model. Although the traditional analysis is simple and effective, obvious drawbacks exist. On the one hand, it requires the identifier naming to reflect the purpose of the function; on the other hand, it ignores the potential information of source code, such as data dependence, control flow, and semantic information.

3.2. Tree-Based Source Code Model

Abstract Syntax Tree (AST) is an important way to express the structure information of the source code. An example is shown in Figure 3. The left side of the arrow is a function, and the right side is the AST parsed by the tools. Scholars usually characterize source code semantic information through different processing of AST, such as converting AST into sequences, randomly extracting AST paths, and dividing AST into multiple sub-ASTs. In short, as an efficient structure representation, AST with DL technology has achieved remarkable results [6,22,23,34,40,43,56,61,63,64].

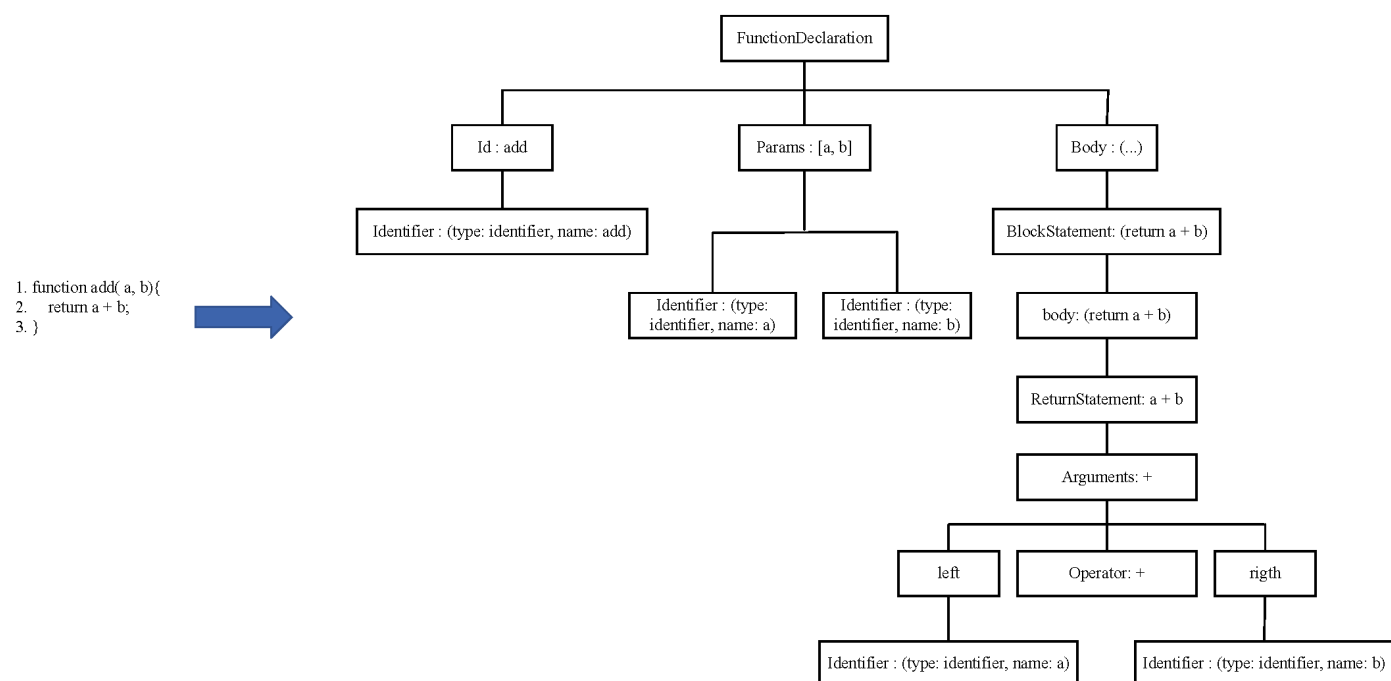


Figure 3. An example of tree-based source code representation. The left side of the blue arrow is source code, and the right side is the tree-based representation of source code.

The earlier typical work using AST to represent the structure of source code is code2vec [64]. Code2vec represented the source code with AST path sets, and each path consisted of two leaf nodes and their intermediate path, (token1, path, token2). Subsequently, Alon et al. [43] proposed the code2seq method on the basis of code2vec. They improved the work of [64] in the following aspects: (1) the latter did sub-token processing on the token; (2) the latter randomly chose several paths from all the paths of AST tree; (3) the latter was suitable for the code summarization in multiple programming languages. They utilize random AST path sets to represent the source code, instead of using the AST or the sub-ASTs directly. In addition, this method can be directly applied on code summarization tasks. Thus, it is often used as a baseline in other works, and it may be applicable to other code intelligence tasks as well.

In 2018, the structure-based traversal (SBT) method [6] made a major breakthrough in structure information extraction. Using SBT, a subtree under a given node is included into a pair of brackets. Furthermore, the brackets represent the structure of AST and can be restored as a tree unambiguously from a sequence generated using SBT. Compared with the previous methods, this new AST traversal method has higher accuracy. Subsequently, Hu et al. [44] extended the above work. When building an ASCS model, they combined the source code vocabulary information and the SBT sequences. Especially, they used camel case naming to solve out-of-vocabulary identifiers problem. Inspired by SBT, some innovative work has emerged. For example, LeClair et al. [22] represented the source code in sequences and AST in two forms, and used a modified SBT method [6] to traverse the AST.

In 2019, ASTNN [46] was proposed, which was of great significance to source code analysis, mainly reflected in the analysis granularity and the skillful construction of statement trees. ASTNN has overcome the following limitations: the paper [34] combined AST with CNN to capture the structural information of source code. However, the AST utilized a window sliding algorithm, which lost a certain part of the source code information. The paper [39] utilized the combination of RNN [71] and AST to represent the source code. The paper [40] combined AST with LSTM to mine the vocabulary and syntax information of the source code. However, the paper [39,40] both performed bottom-up traversal on AST, which made it prone to gradient disappearance when computing large AST structures.

The ASTNN worked as follows: (1) it split the whole AST into small statement trees; (2) it designed a recursive encoder on multi-way statement trees to capture the lexical and syntactical information; (3) it used bidirectional Gated Recurrent Unit (Bi-GRU) to obtain the vector representation of source code. This method solves the huge problem of AST structure, and the batch processing algorithm has greatly improved the computing performance. In addition, the dataset and model code are publicly available, which facilitates the researchers to complete other code intelligence tasks, such as code summarization, code search, and code copyright infringement.

Recently, Hussain et al. [56] proposed a CodeGRU method, which was an effective source code embedding representation method proved by experiments. The CodeGRU selected noise-free source code data and parsed them into AST, then encoded the context, syntax, and structure information of source code. Therefore, scholars can do further research based on this effective model. In addition, blending the AST into a pre-training model is a valuable research direction. For example, Guo et al. [37] combined AST with Data Flow to represent the source code and achieved great results.

3.3. Graph-Based Source Code Model

Graph structure is one of the most flexible data structures and is widely used in many fields. The source code modeling with graph has yielded many significant achievements [50,55,67,72–75]. The common graph structures include Control Flow Graph (CFG), Data Flow Graph (DFG), Program Dependency Graph (PDG), and Code Property Graph (CPG). Different graph structures describe code from different feature perspectives. For example, CFG describes the execution path and control dependency of codes, PDG describes the control dependency and data dependency within codes. We give an example of source code representation by CPG in Figure 4.

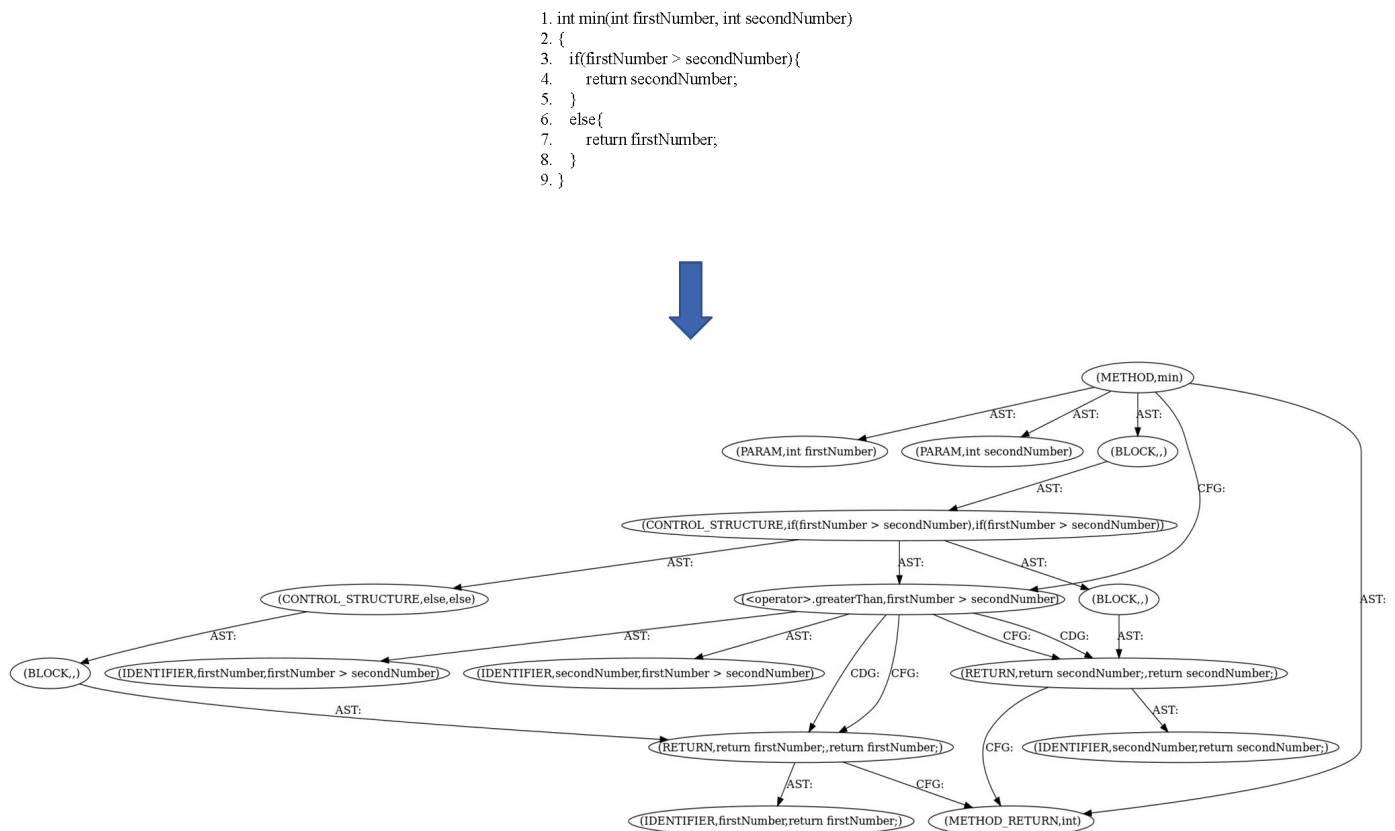


Figure 4. An example of Graph-based (CPG) source code representation. The upper side of the blue arrow is the source code, and the lower side is the CPG of source code.

In recent years, the representative achievements of source code analysis based on PDG are as follows. Tufano et al. [50] expressed the source code from four different levels of identifier, AST, CFG, and Bytecode. Notably, HOPE graph embedding technology [73] was used to train the CFG, and, in the end, different source code representations expressed excellent results in the code cloning task. Allamanis et al. [55] parsed source code into AST. On this basis, the control flow dependence and data flow dependence of the program are captured by additional edges. The paper [72] used GGNN to train the source code graph. Finally, these two tasks of predicting variable name and judging whether the variable is used correctly have achieved high accuracy.

Zhou et al. [75] built a graph neural network model, Devign, which learned rich semantic representations of source code. It represented various subgraphs into one joint graph, CPG, including AST, CFG, DFG, and Natural Code Sequence (NCS). The experiments proved the effectiveness of Devign on vulnerability detection task. The method has achieved a new state of the art on machine-learning-based vulnerability detection, and the CPG can learn rich semantic information of source code. However, the model can only process C/C++ code, if researchers make some improvement to it, which will be a good research direction to achieve other tasks or to process other programming languages.

3.4. Other Source Code Models

Apart from the above methods, there exist other methods of source code modeling, such as Software Word Usage Model (SWUM) [76], data-driven methods, and dynamic methods.

SWUM captures program word relationships and links them with the program structure. Sridhara et al. [9,10] and Moreno et al. [11] extracted the keywords of source code to describe the function of Java methods. However, this type of code summarization cannot explain why this method exists or what role it plays in software. Later, McBurney et al. [13] made important extensions to their work, which was analyzing the call ways of Java methods and including context through SWUM. Subsequently, McBurney et al. [60] also utilized SWUM to propose a method of automatic source code summarization, in which the source code representation is call functions and keywords. However, the SWUM model relies heavily on the naming of function and identifier; if the naming is not precise, the summarization will be inaccurate.

As we all know, the dynamic methods are more accurate than the static methods. For example, Wang et al. [51] learned source code representation through the execution path of the program, and experiments proved that this dynamic code semantic embedding is more effective in error type classification than code syntax embedding. However, the dynamic method is neither comprehensive analysis nor suitable for the large dataset analysis.

4. Code Summarization Generation

The ASCS generation is a hot field that has emerged in the past decade. We divide its methods into three categories: manually-crafted templates, IR-based, and Deep Learning (DL)-based. In recent years, the representative methods and the relation between them have been shown in Figure 5. We can see that human evaluation is mostly used before 2016. The evaluation metrics gradually changed, which further illustrates the evolution of ASCS technology.

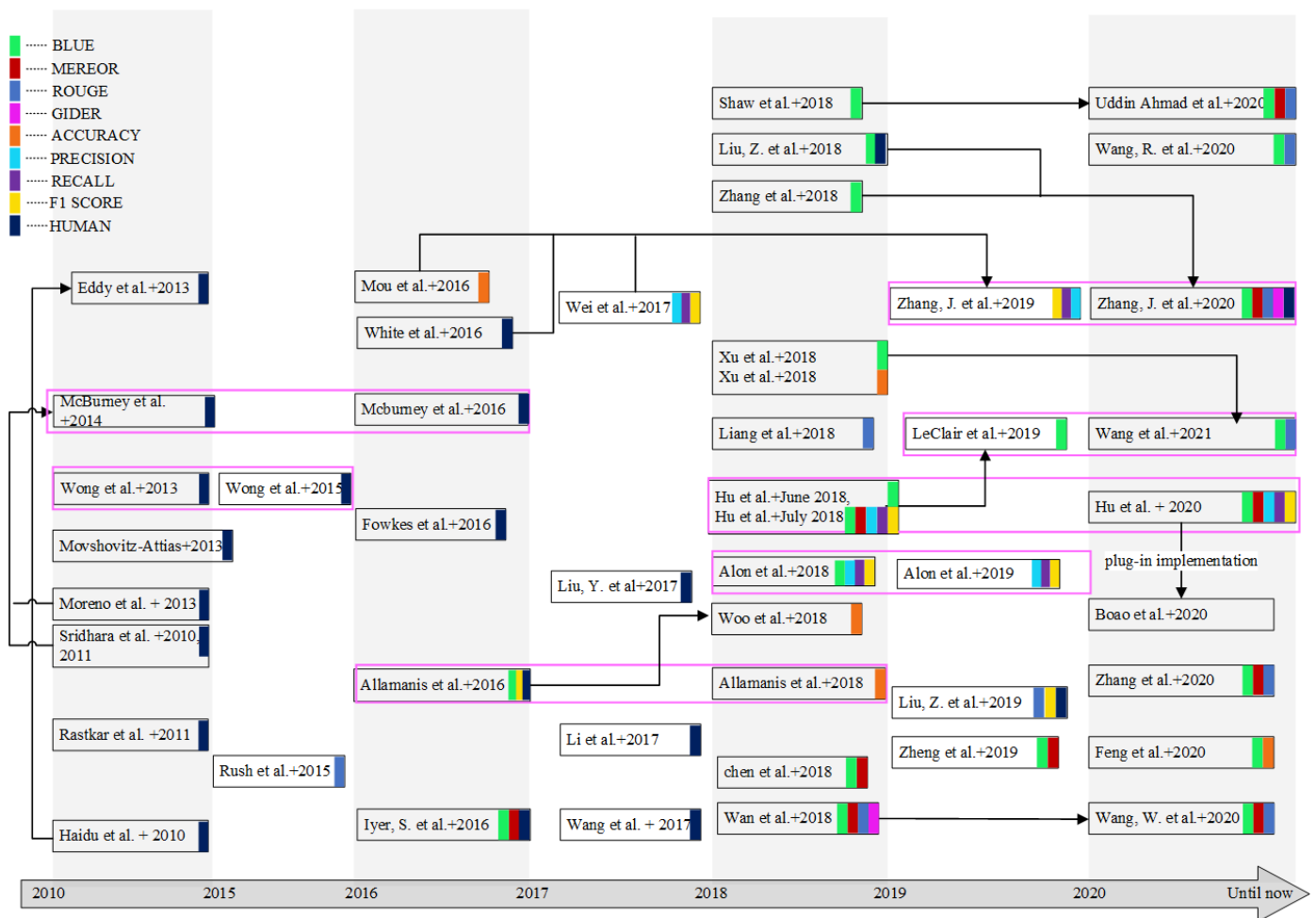


Figure 5. The representative methods [6–17,19,21–26,34,39,40,43,44,46,55,58–66,74,77–89] and their relationship of automatic code summarization. Each node in the same column represents achievements that appeared in the same year. A solid arrow from X to Y indicates that Y cites, references, or otherwise uses techniques from X. Each rectangle represents the works of the same author.

From Figure 5, we can also clearly see which work has been improved and innovated based on the previous work. For example, the work in paper [23] is an improvement over the papers [85,86]. Furthermore, papers [23,43] were done by the same authors.

4.1. Manually-Crafted Templates-Based ASCS Generation

The manually-crafted template is an early method used in ASCS generation. It is mainly based on some custom rules to generate code summarization [9–11,13]. For example, Moreno et al. [11] automatically generated summarization for Java classes through existing text generation tools, in which they utilized the information of the building class with the proposed heuristic rules. Wang et al. [16] utilized high-quality source code projects to train templates for learning the behavior of related objects, and automatically generated a summarization for the related objects with the source code method.

However, the manual template methods usually extract keywords from source code to generate summarization, which ignores a lot of potential information of the source code. In addition, the summarization quality depends on whether the identifier-naming of the source code is standardized. If the naming cannot reflect the function purpose, the summarization is inaccurate. In the current source code information extraction, this method is mostly used as an auxiliary or is simply discarded.

4.2. IR-Based ASCS Generation

After manual template technology, IR-base modeling is widely used in ASCS generation. The IR can be abstracted as a statistical language model. It needs to calculate the correlation between target code and code in the dataset, then returns the corresponding summarization of the code that best matches the target code. The common approaches of IR-based modeling are Latent Semantic Indexing (LSI), Vector Space Model (VSM), theme-based, and so on [7,8,11,58,77–79].

Initially, Haiduc et al. [7,8] analyzed source text using VSM and LSI methods, producing natural language description of source classes or methods. Subsequently, Eddy et al. [78] improved the above work by topic model. Li et al. [80] used Latent Dirichlet Allocation (LDA) technology to conduct topic mining on resources such as code, documentation, question and answer information, and automatically generated code topic summarization. Liu et al. [81] utilized latent semantic analysis and clustering algorithms to extract the semantic information of source code. Finally, the package summarization of the Java project was generated. Movshovitz-Attias et al. [77] adopted the combination of topic models and n-grams to predict Java method summarization.

Wang et al. [58] used part-of-speech tagging to identify keywords that best represented the features of the source code. Through noise reduction, a number of keywords with the highest weights were selected to form the code summarization. However, the method has some disadvantages. The quality of code summarization heavily depends on the source code and cannot be guaranteed, especially for the obfuscated source code.

The code cloning-based method is a common IR method to generate source code summarization. The principle is to search for the most similar code segment with the given source code, then extract its comments as the summarization of the given source code [11,41,79]. Obviously, the quality and quantity of summarization depend on the quality and quantity of the code snippets and the corresponding comments contained in the dataset.

Wong et al. [41] collected a large-scale Q&A dataset from the source code in Stack Overflow. They calculated the similarity between the input code and the codes in Stack Overflow. The description of code in Stack Overflow is used as the summarization of the input code. Later, Wong et al. [79] proposed another method of automatic code summarization by mining the existing software code bases. It used code clone detection technology to find code snippets with similar syntax from the code bases, and applied the summarization of these code snippets to other codes with similar syntax.

The above methods generate code summarization usually by searching for keywords of source code or comments of similar code. The drawbacks of IR-based methods include: First, it overly relies on the standardization of naming. If the naming of identifiers and function are irrelevant to the code function, the extracted keywords will not represent the source code accurately; second, it depends on similar codes in the dataset. If the dataset has no code similar to the given code, no summarization is generated correctly. In summary, relying only on lexical information to represent the source code ignores the rich structure and data dependence of the source code.

4.3. DL-Based ASCS Generation

In recent years, the research of ASCS generation with DL technology has made breakthroughs [6,17,19,21,22,43,44,59,62–64,67,85]. The common DL techniques include RNN and its variant models, CNN and its variant models, transformer model, and large-scale language training model (e.g., BERT and GPT), etc. The Attention mechanism is usually used as a key auxiliary to the above methods.

In 2016, CODE-NN was declared the most advanced model of ASCS generation [59], which used LSTM and Attention mechanism to generate a description of C# codes and SQL sequences. Later, a lot of code summarization papers have taken CODE-NN as the benchmark, such as these papers [6,17,44,63,64]. In 2018, Woo et al. [82] improved the method of paper [15], and proposed CBAM, a new method to improve the representation

ability of CNN networks. CBAM was applied to code generation by adding convolution operations on input symbols, extracting the local translation invariant features of the input sequences. The paper also provides opensource code (<https://github.com/Jongchan/attention-module>, accessed on 18 August 2019), which is a good resource for the research of code intelligence tasks. Deep-Com [6] is a seq2seq model proposed to generate Java method summarization based on Attention mechanism. The semantic information learning of the source code was detailed in Section 3.2. Deep-Com has higher accuracy than previous methods, which has been proven by many experiments. In 2020, the Hybrid-DeepCom [44] is a new ASCS method extending the work of [6]. The improvements have been described in Section 3.2. At the same year, Boao Li et al. [89] implemented a tool plug-in for this method, which helped researchers to understand and apply the method intuitively. Furthermore, it also proved the feasibility and practicability of Hybrid-DeepCom. In 2020, LeClair et al. [67] improved the method of processing source code AST information [83,84] on the basis of ast-attendgru [22], and the accuracy of source code summarization was increased. In 2021, LeClair et al. [90] explored the orthogonal nature of different neural code summarization approaches and proposed ensemble models to exploit this orthogonality for better overall performance. The combination of search and generative methods in ASCS is a promising idea. To the best of our knowledge, there is only one work similar to this, which is the paper by Rencos [23]. It found that the IR-based method could better obtain source code low-frequency words [86], and the NMT-based method was more flexible and semantically correct [85].

The Transformer model can be computed in parallel, and can better capture the long-term dependencies of the sequence, even with strong comprehensive feature extraction capabilities. It can compensate for the shortcomings of RNN parallelization and the difficulty of CNN. Therefore, Transformer has been widely used in NMT field and other various tasks, including code summarization generation [24,26,91]. For example, Uddin Ahmad et al. [26] and Wang et al. [24] utilized Transformer to complete the code summarization task. Compared with existing methods, they improved the effectiveness and accuracy of code summarization. Among them, Uddin Ahmad et al. optimized the model of Shaw et al. [87], in which an attention layer was added to the decoder for copying rare tokens in the source code. The experiments proved that the relative paths performs better than absolute paths. Wang et al. combined the Transformer with BERT and proposed Fret, a new method for generating code summarization. However, Transformer processes a long text by cutting it into multiple fixed-length fields, which may cause information to go missing. In 2019, Dai et al. [91] proposed the Transformer-XL method after optimizing the Transformer model. Transformer-XL could model dependencies exceeding a fixed length, and Dai et al. opened the source model code (<https://github.com/kimiyoung/transformer-xl>, accessed on 11 April 2020).

In 2020, based on paper [43,64], Alon et al. [54] proposed a structure-based language model, SLM, which could solve the code generation problem of various programming languages, and it performed better than seq2seq and other methods of generating Java codes and C# codes. Alon et al. highlighted the importance of structural language modeling and its wide application of code intelligence tasks. They also made the datasets and codes public (<https://github.com/tech-srl/slm-code-generation>, accessed on 28 September 2021).

To solve the long-term dependency problem, reinforcement learning is a popular method. Wan et al. [19] incorporated an AST structure as well as sequential code snippets into a deep reinforcement learning framework (i.e. actor-critic network), which improved the performance of ASCS generation. Based on the above work, Wang and Wan et al. [88] combined hierarchical Attention-based learning with actor-critic reinforcement learning, and adopted type-enhanced AST sequences and CFG instead of the pure AST to capture the correlation between comments and programs. In addition, the use of hierarchical attention network (HAN) fully took into account the hierarchical structure of the source code, and the experimental effect was better than the papers [6,43,59,92].

CodeCMR [93] is a cross-modal retrieval method for binary source code matching on NeurIPS2020. They adopted DPCNN [49] to extract source code features and Graph Neural Network (GNN) for binary code feature extraction. It captured code literals, including strings and integers. The results showed that compared with paper [46], using the source code directly as a input not only retained the integrity but also saved time when dividing AST into statement trees. To the best of our knowledge, this is the first binary source code matching method on function-level, and its dataset is an available resource (<https://github.com/binarya>, accessed on 15 July 2020).

5. Quality Evaluation

The quality evaluation of source code summarization is divided into evaluation objects and evaluation methods. The evaluation objects are mainly datasets and summarization generation algorithms introduced in Section 4. In this section, we introduce the datasets and two types of evaluation methods: automatic evaluation and human evaluation.

5.1. Datasets

Most of the datasets come from Stack Overflow [41,57,59], GitHub [6,15,17,21,44,52,55,60], and code contests [34,40,46,63]. There are also some datasets constructed by researchers [94–98]. The common available datasets have been shown in Table 2.

Stack Overflow (SO) is a Q&A website in the field of programming, which consists of code snippets and the corresponding purpose description. The scholars often take these Q&A pairs as their dataset. The advantage is that the datasets are easy to extract, and the disadvantage is that the quality of Q&A pairs are uneven, because the platform does not guarantee the quality of questions and answers.

GitHub (GH) is a software source code hosting service platform, which is also the largest code site and opensource community [99]. As a popular source of datasets, the code quality in GitHub can be reflected by the number of five-pointed stars left in the reviews. More stars means better quality. However, GH only provides the source code. If you choose the ASCS dataset, you need to manually find the corresponding code summarization.

Google Code contests Jam and Online Judge (OJ), as the dataset format is similar to the SO. The quality of OJ is relatively higher, however, the scale of the dataset in OJ is limited, which may result in poor general adaptation of the training model.

CodeXGLUE [100], a benchmark dataset and open challenge for code intelligence, contains 14 datasets for 10 diversified code intelligence tasks covering the following scenarios: Code-Code, Text-Code, Code-Text, and Text-Text. The ASCS belongs to Code-Text, and the public dataset is CodeSearchNet [101], which includes six programming language sub-datasets of Python, Java, PHP, JavaScript, Ruby, and Go. Furthermore, each sub-dataset provides source code token tags and corresponding comments. The CodeSearchNet is a milestone for the technology. Besides, paper [102] researched how the different characteristics of the dataset affect the summarization performance. It evaluated the performance of five typical methods on three widely used datasets, which were different in three attributes: corpus size, data splitting methods, and duplication ratio. The paper [102] helps researchers choose and process datasets correctly.

Table 2. The public datasets of typical ASCS (* represents closed source; SO represents Stack Overflow; GH represents GitHub; OJ represents Online Judge).

Methods	Venue	Datasets	Opened-Source
AutoComment [41]	ASE	Java(SO)	*
TASSAL [61]	ICSE	Java(GH)	https://github.com/mast-group/tassal (accessed on 25 November 2019)
conv_attention [15]	ICML	Java(GH)	http://groups.inf.ed.ac.uk/cup/codeattention (accessed on 25 February 2020)
Allamanis et al. [55]	ICLR	C#(GH)	https://aka.ms/iclr18-prog-graphs-dataset (accessed on 2 March 2020)
McBurney et al. [60]	TSE	Java(GH)	*
CODENN [59]	ACL	C#, SQL(SO)	https://stackoverflow.com/ (accessed on 2 March 2020)
DeepCom [6]	ICPC	Java(GH)	https://github.com/xing-hu/DeepCom (accessed on 3 March 2021)
TL-CodeSum [17]	IJCAJ	Java(GH)	https://github.com/xing-hu/TL-CodeSum (accessed on 3 March 2021)
Hybrid-DeepCom [44]	ESE	Java [6]	https://github.com/xing-hu/DeepCom (accessed on 5 March 2021)
BVAE [62]	ASE	C#, SQL [59]	https://stackoverflow.com/ (accessed on 28 November 2019)
HybridDRL [19]	ASE	Python [94]	https://github.com/wanyao1992/code_summarization_public (accessed on 7 February 2021)
CodeRNN [63]	AAAI	Java(GH)	https://adapt.seiee.sjtu.edu.cn/CodeComment/ (accessed on 29 November 2020)
Attn+PG+RL [65]	ASE	Java(GH)	https://tinyurl.com/y3yk6oey (accessed on 28 November 2019)
Code2vec [64]	POPL	Java(GH)	https://github.com/tech-srl/code2vec (accessed on 15 December 2019)
Code2seq [43]	ICLR	Java(GH),C# [59]	https://github.com/tech-srl/code2seq (accessed on 15 December 2019)
Ast-attendgru [22]	ICSE	Java [95]	http://www.ics.uci.edu/~lopes/datasets/ (accessed on 18 December 2020)
LeClair et al. [67]	ACL	Java [96]	http://leclair.tech/data/funcom/ (accessed on 15 February 2020)
ASTNN [46]	ICSE	C(OJ), Java [93]	https://github.com/zhangj1994/astnn (accessed on 15 January 2021)
Rencos [23]	ICSE	Java [17], Python [94]	https://github.com/xing-hu/TL-CodeSum (accessed on 23 February 2021)
CodeBERT [25]	arXiv	Go, Java, JS, PHP, Python, Ruby	https://github.com/microsoft/CodeBERT (accessed on 23 February 2021)
TBCNN [34]	AAAI	C(OJ)	http://programming.grids.cn (accessed on 28 February 2021)
CDLH [40]	IJCAI	Java [93], C(OJ)	http://programming.grids.cn (accessed on 5 May 2021)

Table 2. Cont.

Methods	Venue	Datesets	Opened-Source
DL-based [50]	MSR	Java(GH)	https://archive.apache.org/dist/commons (accessed on 23 May 2021)
Code-GRU [56]	IST	Java(GH)	https://github.com/yaxirhuxxain/Source-Code-Suggestion (accessed on 23 May 2021)
CodeAttention [21]	FCS	Java(GH)	https://github.com/wenhaozheng-nju/CodeAttention (accessed on 25 May 2021)
Transformer-based [26]	ACL	Java [17], Python [94]	https://github.com/wasiahmad/NeuralCodeSum (accessed on 23 February 2021)
Fret [24]	Access	Java(GH), Python [94]	https://github.com/xing-hu/EMSE-DeepCom (accessed on 23 March 2020)
fc-pc [68]	ICPC	Java(GH), Python	https://github.com/aakashba/projcon (accessed on 5 November 2021)
KBCoS [66]	JCRD	Java [6], Python [94]	https://github.com/xing-hu/TL-CodeSum (accessed on 26 February 2021) https://github.com/EdinburghNLP/code-docstring-corpora (accessed on 26 February 2021)

5.2. Automatic Evaluation Mechanism

Evaluation criteria are used to measure the quality of the ASCS. As an emerging research hotspot, the quality evaluation of summarization usually uses automatic evaluation indicators and tools of NMT. The use of evaluation methods from the recent years are shown in Figure 5, and we will provide a brief introduction to them.

BLEU [103], Bilingual Evaluation Understudy, was proposed by Papineni et al. in 2002. It is used to count the proportion of n-grams in candidate texts that appear in reference translations, where n can be 1, 2, 3, or 4. Obviously, the higher the BLEU value, the higher the quality of code summarization. BLEU is the earliest automatic evaluation method of machine translation and has been used in many code summarization works [6,15,17,19,21–24,43,59,67]. Moreover, the granularity considered by BLEU method is n-grams instead of word, which can match longer information at a time. However, the drawbacks of BLEU are that it only focuses on accuracy and ignores recall (that is, how many phrases in the reference translations appear in the candidate texts).

ROUGE [104], Recall-Oriented Understudy for Gisting Evaluation, was proposed by Lin. It is similar to the calculation of the BLEU method, but n-gram includes several different metrics, among which ROUGE-N, ROUGE-L, ROUGE-W, and ROUGE-S are commonly used. This metric has been used in many ASCS tasks [19,24,63,67]. However, ROUGE only considers the recall but ignores fluency.

METEOR [105] is a Metric for Evaluation of Translation with Explicit Ordering. It was proposed by Lavie et al. after discovering the significance of recall in evaluation. The method of adding recall to METEOR calculation and expanding the synset by WordNet has solved the defects in BLEU. It achieves similar results with human evaluation. It is more reasonable than BLEU and ROUGE, so the METEOR method is often used in ASCS algorithms [17,21,22,26,59,62,66]. However, there are also some weaknesses: for example, it can be only used in Java language and the parameters need to be debugged according to different datasets.

CIDER [106], Consensus-based Image Description Evaluation, is proposed by Vedantam et al. CIDER is used to measure the similarity between a test sentence and the majority of reference sentences. Among them, TF-IDF [107] calculates the weights of each n-gram. The TF-IDF avoids matching all words equally and makes the important words more prominent. Some researchers use this method to measure the quality of code summarization [19,23,44].

Besides, there are other evaluation methods of ASCS, such as Accuracy [25,34,50,53], Precision [17], Recall [43,46,64], and F1 score [15]. Among them, Accuracy is widely used, but it is not the best evaluation method. Recall expresses the ability to find relevant instances in a dataset, and Precision expresses the ability to find the actual relative proportion of the data points. F1 score takes the harmonic average of Recall and Precision to achieve the best precision-recall balance model, which can better evaluate the classification model and solve the imbalance problem. Although many quality assessment tools can provide automatic evaluation for ASCS, the assessment results of these tools are often unstable, which is an important problem that needs to be solved in ASCS.

Some researchers have also summarized the related evaluation methods of ASCS [1,108–112]. Khamis et al. [108] used heuristic algorithms to propose an automatic evaluation method for ASCS. The principle is based on the consistency between the source code and its corresponding comments. The more similar the comments are, the higher is its quality. Obviously, this method relies on the quality of the dataset. Steidl et al. [1] analyzed the quality of source code summarization from four aspects of consistency, validity, completeness, and relevance. Sun et al. [110] proposed a method for automatically evaluating the quality of code summarization. It not only gives advice to improve the quality of code summarization, but also enhances the accuracy of summarization. However, this method can analyze fewer types of comments, and it is dependent on the vocabularies and comments of source code, ignoring the semantics.

Recently, CodeBLUE [113], a code summarization evaluation mechanism, provided three benchmark models including CodeBERT, CodeGPT, and encoder-decoder. Among them, the CodeBERT model was used for code understanding [25], and was the first known large scale Natural Language-Programming Language (NL-PL) pre-training model. In the task of Code-Text, CodeBERT had pre-trained the six programming languages of CodeSearchNet [101], and plenty of experiments proved that the CodeBERT model is highly effective in both code search and ASCS tasks.

5.3. Human Evaluation

Human evaluation generally selects experienced developers or people to join the project. It mainly measures the accuracy, fluency, and effectiveness of the code summarization. Among them, the accuracy measures the expression degree of code summarization for source code feature information. The fluency measures whether the code summarization is grammatical and understandable. The effectiveness measures the usefulness and necessity of code summarization. Usually, the researchers rate comments on a scale between 1 and 5, and the higher the score, the higher the accuracy of the comments.

Obviously, human evaluation has high accuracy, and it avoids the complicated design of the evaluation algorithm. However, drawbacks exist. On one hand, it is not suitable for enormous quantities of code summarization evaluation work, because human evaluation is completed by many experienced developers, which is costly and ineffective. On the other hand, it is easy to have deviations occur, caused by personal factors such as high pressure, fatigue, and inexperience.

6. Discussion and Conclusions

ASCS is a hot research topic at present. In this paper, we conducted an in-depth analysis of ASCS: (1) We outlined the core of the paper, which consists of the current challenges, and systematized the ASCS based on three dimensions: source code analysis, code summarization generation algorithms, and the evaluation methodologies used to evaluate them. (2) We discussed the advantages and limitations of different algorithms, their implementation, and their evaluation. (3) We summarized the effective evaluation mechanism of ASCS (automatic evaluation mechanism and human evaluation), and analyzed the recent evaluation methods. Thus, it can help scholars to choose reasonable assessment criteria for source code summarization, creating universal validation datasets that are open for future research on summarization performance.

Furthermore, although scholars have achieved a series of high-quality research results on ASCS, there are still many possible future research directions that deserve further attention:

- The current research is almost generative or search-based, but combining the two to generate code summaries is a promising research direction;
- Large-scale language training model will be an inevitable requirement with the increasing daily data. In view of the complex structure and semantic information of source code, it will be a meaningful research direction to combine the graph neural network to represent source code in large-scale language training model;
- The CPG [114] is a good multi-feature integrated extraction tool. However, the existing works can only be applied in C/C++. In the future works, we can use it for many other programming languages;
- The challenges concluded in the paper are urgent issues to be solved in ASCS. If they were to be solved, it would greatly enhance the recognition and research significance.

Author Contributions: Conceptualization, C.Z. and J.W.; investigation and methodology analysis, C.Z.; original draft preparation, C.Z.; review and editing, C.Z., Q.Z., T.X. and K.T.; supervision, H.G. and F.L.; project administration and funding acquisition, F.L. All authors have read and agreed to the published version of the manuscript.

Funding: The work was partially supported by the Foundation of National Natural Science Foundation of China (No. 61802435, No. 61472447).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We thank Junchao Wang for his suggestions and guidance on the paper. We thank the associate editor and the reviewers for their useful feedback that improved this paper. This paper is supported the Foundation of National Natural Science Foundation of China (NSFC) under grant No. 61802435 and No. 61472447.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Steidl, D.; Hummel, B.; Juergens, E. Quality analysis of source code comments. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013.
2. Yau, S.S.; Collofello, J.S. Some stability measures for software maintenance. *IEEE Trans. Softw. Eng.* **1980**, *SE-6*, 545–552. [[CrossRef](#)]
3. Woodfield, S.N.; Dunsmore, H.E.; Shen, V.Y. The effect of modularization and comments on program comprehension. In Proceedings of the 5th International Conference on Software Engineering, San Diego, CA, USA, 9–12 March 1981; pp. 215–223.
4. Tenny, T. Program readability: Procedures versus comments. *IEEE Trans. Softw. Eng.* **1988**, *14*, 1271–1279. [[CrossRef](#)]
5. Xia, X.; Bao, L.; Lo, D.; Xing, Z.; Hassan, A.E.; Li, S. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Softw. Eng.* **2017**, *44*, 951–976. [[CrossRef](#)]
6. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 27 May–3 June 2018; pp. 200–210.
7. Haiduc, S.; Aponte, J.; Marcus, A. Supporting program comprehension with source code summarization. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 2–8 May 2010; pp. 223–226.
8. Haiduc, S.; Aponte, J.; Moreno, L.; Marcus, A. On the use of automated text summarization techniques for summarizing source code. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 35–44.
9. Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; Vijay-Shanker, K. Towards automatically generating summary comments for Java methods. In Proceedings of the ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010.
10. Sridhara, G.; Pollock, L.L.; Vijay-Shanker, K. Automatically detecting and describing high level actions within methods. In Proceedings of the International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011.
11. Moreno, L.; Aponte, J.; Sridhara, G.; Marcus, A.; Pollock, L.; Vijay-Shanker, K. Automatic generation of natural language summaries for Java classes. In Proceedings of the IEEE International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013.
12. Rastkar, S.; Murphy, G.C.; Bradley, A.W.J. Generating natural language summaries for crosscutting source code concerns. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, 25–30 September 2011.
13. McBurney, P.W.; McMillan, C. Automatic documentation generation via source code summarization of method context. In Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, 2–3 June 2014; pp. 279–290.
14. Rush, A.M.; Chopra, S.; Weston, J. A neural attention model for abstractive sentence summarization. *arXiv* **2015**, arXiv:1509.00685.
15. Allamanis, M.; Peng, H.; Sutton, C. A convolutional Attention network for extreme summarization of source code. In Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 20–22 June 2016; pp. 2091–2100.
16. Wang, X.; Pollock, L.; Vijay-Shanker, K. Automatically generating natural language descriptions for object-related statement sequences. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 205–216.
17. Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; Jin, Z. Summarizing source code with transferred api knowledge. In Proceedings of the TwentySeventh International Joint Conference on Artificial Intelligence, IJCAI-18, Stockholm, Sweden, 13–19 July 2018; pp. 2269–2275.
18. Huang, Y.; Jia, N.; Zhou, Q.; Chen, X.P.; Xiong, Y.F.; Luo, X.N. Method combining structural and semantic features to support code commenting decision. *Ruan Jian Xue Bao/J. Softw.* **2018**, *29*, 2226–2242. (In Chinese with English abstract)
19. Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; Yu, P.S. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 397–407.
20. Al-Msie'deen, R.F.; Blasi, A.H. Supporting software documentation with source code summarization. *arXiv* **2018**, arXiv:1901.01186.
21. Zheng, W.; Zhou, H.; Li, M.; Wu, J. CodeAttention: Translating source code to comments by exploiting the code constructs. *Front. Comput. Sci.* **2019**, *13*, 565–578. [[CrossRef](#)]

22. LeClair, A.; Jiang, S.; McMillan, C. A neural model for generating natural language summaries of program subroutines. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 795–806.
23. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Liu, X. Retrieval-based neural source code summarization. In Proceedings of the 42nd International Conference on Software Engineering, Seoul, Korea, 5–11 October 2020.
24. Wang, R.; Zhang, H.; Lu, G.; Lyu, L.; Lyu, C. Fret: Functional Reinforced Transformer with BERT for Code Summarization. *IEEE Access* **2020**, *8*, 135591–135604. [\[CrossRef\]](#)
25. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Zhou, M. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
26. Uddin Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. A Transformer-based Approach for Source Code Summarization. *arXiv* **2020**, arXiv:2005.00653.
27. Panichella, S.; Aponte, J.; Penta, M.D.; Marcus, A.; Canfora, G. Mining source code descriptions from developer communications. In Proceedings of the IEEE International Conference on Program Comprehension, Passau, Germany, 11–13 June 2012.
28. Nazar, N.; Hu, Y.; Jiang, H. Summarizing software artifacts: A literature review. *J. Comput. Sci. Technol.* **2016**, *31*, 883–909. [\[CrossRef\]](#)
29. Yang, B.; Liping, Z.; Fengrong, Z. A Survey on Research of Code Comment. In Proceedings of the the 2019 3rd International Conference, Wuhan, China, 12–14 January 2019.
30. Chen, Z.; Monperrus, M. A literature study of embeddings on source code. *arXiv* **2019**, arXiv:1904.03061.
31. Song, X.; Sun, H.; Wang, X.; Yan, J. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access* **2019**, *7*, 111411–111428. [\[CrossRef\]](#)
32. Fluri, B.; Wursch, M.; Gall, H.C. Do code and comments co-evolve? On the relation between source code and comment changes. In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007), Vancouver, BC, Canada, 28–31 October 2007; pp. 70–79.
33. Frantzeskou, G.; MacDonell, S.; Stamatatos, E.; Gritzalis, S. Examining the significance of high-level programming features in source code author classification. *J. Syst. Softw.* **2008**, *81*, 447–460. [\[CrossRef\]](#)
34. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In Proceedings of the Thirtieth AAAI conference on artificial intelligence, Phoenix, AZ, USA, 12–17 February 2016.
35. LeClair, A.; Eberhart, Z.; McMillan, C. Adapting neural text classification for improved software categorization. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 461–472.
36. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 933–944.
37. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, M. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv* **2020**, arXiv:2009.08366.
38. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [\[CrossRef\]](#)
39. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; pp. 87–98.
40. Wei, H.; Li, M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In Proceedings of the IJCAI, Melbourne, Australia, 19–25 August 2017; pp. 3034–3040.
41. Wong, E.; Yang, J.; Tan, L. Autocomment: Mining question and answer sites for automatic comment generation. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 562–567.
42. Jiang, S.; Armaly, A.; McMillan, C. Automatically generating commit messages from diffs using neural machine translation. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana, IL, USA, 30 October–3 November 2017; pp. 135–146.
43. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv* **2018**, arXiv:1808.01400.
44. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* **2020**, *25*, 2179–2217. [\[CrossRef\]](#)
45. Sharma, T.; Kechagia, M.; Georgiou, S.; Tiwari, R.; Sarro, F. A Survey on Machine Learning Techniques for Source Code Analysis. *arXiv* **2021**, arXiv:2110.09610.
46. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019.
47. Ishio, T.; Etsuda, S.; Inoue, K. A lightweight visualization of interprocedural data-flow paths for source code reading. In Proceedings of the 2012 20th IEEE International Conference on Program Comprehension (ICPC), Passau, Germany, 11–13 June 2012.

48. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 3585–3597.
49. Johnson, R.; Zhang, T. Deep pyramid convolutional neural networks for text categorization. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vancouver, Canada, 30 July–4 August 2017; pp. 562–570.
50. Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshyvanyk, D. Deep learning similarities from different representations of source code. In Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, 27 May–3 June 2018; pp. 542–553.
51. Wang, K.; Singh, R.; Su, Z. Dynamic neural program embedding for program repair. *arXiv* **2017**, arXiv:1711.07163.
52. Efstathiou, V.; Spinellis, D. Semantic source code models using identifier embeddings. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 25–31 May 2019; pp. 29–33.
53. Henkel, J.; Lahiri, S.K.; Liblit, B.; Reps, T. Code vectors: Understanding programs through embedded abstracted symbolic traces. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2018; pp. 163–174.
54. Alon, U.; Sadaka, R.; Levy, O.; Yahav, E. Structural language models of code. In Proceedings of the International Conference on Machine Learning, Virtual, 28–29 November 2020; Volume 119, pp. 245–256.
55. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
56. Hussain, Y.; Huang, Z.; Zhou, Y.; Wang, S. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. *Inf. Softw. Technol.* **2020**, *125*, 106309. [[CrossRef](#)]
57. Chen, F.; Kim, M.; Choo, J. Novel Natural Language Summarization of Program Code via Leveraging Multiple Input Representations. In Proceedings of the Findings of the Association for Computational Linguistics (EMNLP 2021), Punta Cana, Dominican Republic, 7–11 November 2021; pp. 2510–2520.
58. Wang, J.; Xue, X.; Weng, W. Source code summarization technology based on syntactic analysis. *J. Comput. Appl.* **2015**, *35*, 1999.
59. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing Source Code using a Neural Attention Model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics 2016, Berlin, Germany, 7–12 August 2016; pp. 2073–2083.
60. Mcburney, P.W.; Mcmillan, C. Automatic Source Code Summarization of Context for Java Methods. *IEEE Trans. Softw. Eng.* **2016**, *42*, 103–119. [[CrossRef](#)]
61. Fowkes, J.; Chanthirasegaran, P.; Ranca, R.; Allamanis, M.; Lapata, M.; Sutton, C. TASSAL: Autofolding for source code summarization. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Austin, TX, USA, 14–22 May 2016; pp. 649–652.
62. Chen, Q.; Zhou, M. A neural framework for retrieval and summarization of source code. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 826–831.
63. Liang, Y.; Zhu, K.Q. Automatic generation of text descriptive comments for code blocks. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018.
64. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. In Proceedings of the ACM on Programming Languages, Athens, Greece, 21–22 October 2019; Volume 3, pp. 1–29.
65. Liu, Z.; Xia, X.; Treude, C.; Lo, D.; Li, S. Automatic generation of pull request descriptions. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 176–188.
66. Zhang, S.K.; Xie, R.; Ye, W. Keyword G asedSourceCodeSummarization. *J. Comput. Res. Dev.* **2020**, *57*, 1987. (In Chinese with English abstract)
67. LeClair, A.; Haque, S.; Wu, L.; McMillan, C. Improved code summarization via a graph neural network. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Korea, 13–15 July 2020; pp. 184–195.
68. Zügner, D.; Kirschstein, T.; Catasta, M.; Leskovec, J.; Günnemann, S. Language-agnostic representation learning of source code from structure and context. *arXiv* **2021**, arXiv:2103.11318.
69. Bansal, A.; Haque, S.; McMillan, C. Project-level encoding for neural source code summarization of subroutines. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021; pp. 253–264.
70. Shahbazi, R.; Sharma, R.; Fard, F.H. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021; pp. 411–421.
71. Socher, R.; Lin, C.C.; Manning, C.; Ng, A.Y. Parsing natural scenes and natural language with recursive neural networks. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), Bellevue, WA, USA, 28 June–2 July 2011; pp. 129–136.
72. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated graph sequence neural networks. *arXiv* **2015**, arXiv:1511.05493.
73. Ou, M.; Cui, P.; Pei, J.; Zhang, Z.; Zhu, W. Asymmetric transitivity preserving graph embedding. In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.

74. Wang, M.; Tang, Y.; Wang, J.; Deng, J. Premise selection for theorem proving by deep graph embedding. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 2783–2793.
75. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.
76. Hill, E. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*; University of Delaware: Newark, DE, USA, 2010.
77. Movshovitz-Attias, D.; Cohen, W. Natural language models for predicting programming comments. In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, Sofia, Bulgaria, 4–9 August 2013; Volume 2, pp. 35–40.
78. Eddy, B.P.; Robinson, J.A.; Kraft, N.A.; Carver, J.C. Evaluating source code summarization techniques: Replication and expansion. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 13–22.
79. Wong, E.; Liu, T.; Tan, L. Clocom: Mining existing source code for automatic comment generation. In Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2–6 March 2015; pp. 380–389.
80. Li, W.P.; Zhao, J.F.; Xie, B. Summary Extraction Method for Code Topic Based on LDA. *Comput. Sci.* **2017**, *44*, 35–38. (In Chinese with English abstract)
81. Liu, Y.; Sun, X.B.; Li, B. Research on Automatic Summarization for Java Packages. *J. Front. Comput. Sci. Technol.* **2017**, *11*, 46–54. (In Chinese with English abstract)
82. Woo, S.; Park, J.; Lee, J.Y.; Kweon, I.S. Cbam: Convolutional block Attention module. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 10 September 2018; pp. 3–19.
83. Xu, K.; Wu, L.; Wang, Z.; Yu, M.; Chen, L.; Sheinin, V. Exploiting rich syntactic information for semantic parsing with graph-to-sequence model. *arXiv* **2018**, arXiv:1808.07624.
84. Xu, K.; Wu, L.; Wang, Z.; Feng, Y.; Witbrock, M.; Sheinin, V. Graph2seq: Graph to sequence learning with Attention-based neural networks. *arXiv* **2018**, arXiv:1804.00823.
85. Zhang, J.; Utiyama, M.; Sumita, E.; Neubig, G.; Nakamura, S. Guiding neural machine translation with retrieved translation pieces. *arXiv* **2018**, arXiv:1804.02559.
86. Liu, Z.; Xia, X.; Hassan, A.E.; Lo, D.; Xing, Z.; Wang, X. Neural-machine-translation-based commit message generation: How far are we? In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, New York, NY, USA, 3–7 September 2018; pp. 373–384.
87. Shaw, P.; Uszkoreit, J.; Vaswani, A. Self-attention with relative position representations. *arXiv* **2018**, arXiv:1803.02155.
88. Wang, W.; Zhang, Y.; Sui, Y.; Wan, Y.; Zhao, Z.; Wu, J.; Xu, G. Reinforcement-Learning-Guided Source Code Summarization via Hierarchical Attention. *IEEE Trans. Softw. Eng.* **2020**, *48*, 102–119. [[CrossRef](#)]
89. Li, B.; Yan, M.; Xia, X.; Li, G.; Lo, D. DeepCommenter: A deep code comment generation tool with hybrid lexical and syntactical information. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, 8–13 November 2020; pp. 1571–1575.
90. LeClair, A.; Bansal, A.; McMillan, C. Ensemble Models for Neural Source Code Summarization of Subroutines. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 286–297.
91. Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J.; Le, Q.V.; Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv* **2019**, arXiv:1901.02860.
92. Yao, Z.; Peddamail, J.R.; Sun, H. CoaCor: Code annotation for code retrieval with reinforcement learning. In Proceedings of the The World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 2203–2214.
93. Yu, Z.; Zheng, W.; Wang, J.; Tang, Q.; Nie, S.; Wu, S. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 3872–3883.
94. Miceli-Barone, A.V.; Sennrich, R. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. In Proceedings of the Eighth International Joint Conference on Natural Language Processing, Taipei, Taiwan, 27 November–1 December 2017; Volume 48, pp. 1–5.
95. Lopes, C. UCI Source Code Data Sets. Available online: <http://www.ics.uci.edu/~lopes/datasets/> (accessed on 18 November 2019).
96. LeClair, A.; McMillan, C. Recommendations for datasets for source code summarization. *arXiv* **2019**, arXiv:1904.02660.
97. Svajlenko, J.; Islam, J.F.; Keivanloo, I.; Roy, C.K.; Mia, M.M. Towards a big data curated benchmark of inter-project code clones. In Proceedings of the Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, 29 September–3 October 2014; pp. 476–480.
98. Wang, X. Java Opensource Repository. Available online: <https://www.eecis.udel.edu/~xiwang/open-source-projects-data.html> (accessed on 19 October 2019).
99. Gousios, G.; Vasilescu, B.; Serebrenik, A.; Zaidman, A. Lean GHTorrent: GitHub data on demand. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 384–387.
100. Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Liu, S.; et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv* **2021**, arXiv:2102.04664.

101. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv* **2019**, arXiv:1909.09436.
102. Shi, E.; Wang, Y.; Du, L.; Chen, J.; Han, S.; Zhang, H.; Zhang, D.; Sun, H. Neural Code Summarization: How Far Are We? *arXiv* **2021**, arXiv:2107.07112.
103. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, PA, USA, 7–12 July 2002; pp. 311–318.
104. Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries. In Proceedings of the Workshop on Text Summarization of ACL, Barcelona, Spain, 25–26 July 2004; pp. 74–81.
105. Banerjee, S.; Lavie, A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, Ann Arbor, MI, USA, 9 June 2005; pp. 65–72.
106. Vedantam, R.; Zitnick, C.L.; Parikh, D. CIDEr: Consensus-based Image Description Evaluation. *arXiv* **2014**, arXiv:1411.5726.
107. Aizawa, A. An information-theoretic perspective of tf-idf measures. *Inf. Process. Manag.* **2003**, *39*, 45–65. [[CrossRef](#)]
108. Khamis, N.; Witte, R.; Rilling, J. Automatic quality assessment of source code comments: The JavadocMiner. In Proceedings of the International Conference on Application of Natural Language to Information Systems, Cardiff, UK, 23–25 June 2010; pp. 68–79.
109. Gao, X.S.; Du, J.; Wang, Q. Quality Evaluation Framework of Source Code Analysis Comments. *Comput. Syst. Appl.* **2015**, *24*, 1–8. (In Chinese with English abstract)
110. Sun, X.; Geng, Q.; Lo, D.; Duan, Y.; Liu, X.; Li, B. Code comment quality analysis and improvement recommendation: An automated approach. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26*, 981–1000. [[CrossRef](#)]
111. Yu, H.; Li, B.; Wang, P.X. Source code comments quality assessment method based on aggregation of classification algorithms. *J. Comput. Appl.* **2016**, *36*, 3448–3453. (In Chinese with English abstract)
112. Wang, D.; Guo, Y.; Dong, W.; Wang, Z.; Liu, H.; Li, S. Deep Code-Comment Understanding and Assessment. *IEEE Access* **2019**, *7*, 174200–174209. [[CrossRef](#)]
113. Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; Ma, S. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. *arXiv* **2020**, arXiv:2009.10297.
114. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014.