IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 25, 2019, accepted July 13, 2019, date of publication July 29, 2019, date of current version August 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931579

A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques

XIAOTAO SONG[1], HAILONG SUN[2,3], XU WANG[2,3], AND JIAFEI YAN[2,3,4,5]

[1]School of Software, Taiyuan University of Technology, Taiyuan 030024, China
[2]SKLSDE Laboratory, School of Computer Science and Engineering, Beihang University, Beijing 100191, China
[3]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100191, China
[4]Beijing Aeronautical Science and Technology Research Institute, Beijing 102211, China
[5]Beijing Key Laboratory of Civil Aircraft Design and Simulation Technology, Beijing 102211, China

Corresponding author: Hailong Sun (sunhl@buaa.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB1000804, and in part by the National Natural Science Foundation under Grant 61702024 and Grant 61421003.

ABSTRACT As an integral part of source code files, code comments help improve program readability and comprehension. However, developers sometimes do not comment their program code adequately due to the incurred extra efforts, lack of relevant knowledge, unawareness of the importance of code commenting or some other factors. As a result, code comments can be inadequate, absent or even mismatched with source code, which affects the understanding, reusing and the maintenance of software. To solve these problems of code comments, researchers have been concerned with generating code comments automatically. In this work, we aim at conducting a survey of automatic code commenting researches. First, we generally analyze the challenges and research framework of automatic generation of program comments. Second, we present the classification of representative algorithms, the design principles, strengths and weaknesses of each category of algorithms. Meanwhile, we also provide an overview of the quality assessment of the generated comments. Finally, we summarize some future directions for advancing the techniques of automatic generation of code comments and the quality assessment of comments.

INDEX TERMS Code comment, deep learning, information retrieval, machine learning, program annotation.

I. INTRODUCTION

Code comments, also called program annotations, are human-readable explanations or annotations of the source code of a computer program [1], which mainly describe the functions and intentions of source code. Good comments can improve the readability of programs [74], [75], [84], thus helping people comprehend programs. For instance, an early study [74] shows that comments can improve the readability of the banker algorithm used in operating systems. As a result, it has been widely acknowledged that comments play an important role in software development and maintenance [6], [65], [73], [75], [84].

However, writing high quality comments in practice during development is laborious and time-consuming for developers [27], [35]. To deal with this issue, many efforts [31], [61], [67], [82], [83] have been made towards automatically generating code comments. At the same time, researchers propose other approaches to improving the readability of programs too. For example, some researchers have tried to define identifiers with a long descriptive name in order to implement self-commented code [7], [23]. However, it makes code comprehension more difficult [14], [42]. In general, automatic code commenting has become an important and challenging research direction in software engineering area.

Studies on code comments and readability of programs can be traced back to the 1980s [74], [75], [84] while the history of autocommenting just started in the last decade. Existing methods are mainly based on machine learning or information retrieval techniques to generate comments for programs. The generation framework of code commenting is mainly composed of three parts. The first one is data preparation which prepares data for the commenting system. The second part is the representation of source code, which aims at capturing the structure and semantics of source code, such as information of structure, lexis, grammar, semantics, contexts, invocation relation and data dependency of source code. The third part is text generation, which is responsible for

The associate editor coordinating the review of this article and approving it for publication was Xiao Liu.

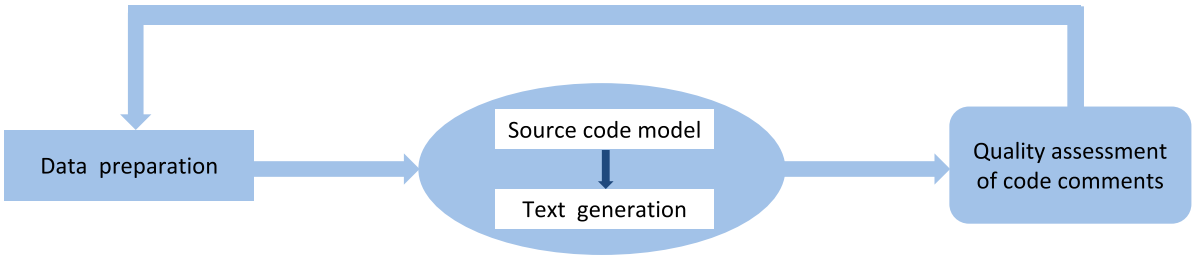


FIGURE 1. The general process of the automatic generation of code comments.

generating natural language sentences based on the information extracted from source code.

Along with the study of code comment researchers have found that the assessment of the quality of code comments is another important research problem, as the quality of generated comments is an important indicator for evaluating whether a commenting algorithm is efficient and effective. Thus designing appropriate criteria for code comment quality assessment is another challenge faced by automatic comment generation [13], [20], [38], [58], [87]. The task of assessment for quality of code comments involves the comparison and verification of various algorithms.

There have been a lot of research efforts on automatic code commenting, especially from the overlapping community of software engineering and artificial intelligence. As a result, many papers have been published in top software engineering and artificial intelligence venues including IEEE ICSE, IEEE FSE, IEEE/ACM ASE, IEEE TSE, ACM TOSEM, EMSE, AAAI, IJCAI and so on. To our best knowledge, there are few efforts on the survey of studies in this field. In [85], Yang *et al.* summarize the work on code comments from four aspects: code comment generation, classification of comments, the consistency of code and comments, and quality assessment of code comments. However, the paper does not discuss the principle of each algorithm in detail and fails to analyze the future research direction. Nazar *et al.* [53] survey the studies on summarizing software artifacts, which include bug reports, mailing list, source code and developer discussions, where the task of summarizing source code is similar to code commenting because code summary can be viewed as a special type of comments. In light of this, we aim at giving a comprehensive survey of the work on automatic code commenting for the following objectives: (1) giving researchers access to a catalogue of representative algorithms for automatic comment generation and providing new researchers with a good understanding of the state-of-the-art algorithms of automatic code commenting; (2) summarizing the main challenges and limitations of existing studies.

In this work, we present the motivation of automatic code comment generation first, analyze the main challenges, and describe the workflow of code commenting automatically. Next we discuss the three mainstream categories of algorithms of automatic comment generation, and show the potential trends in automatic comment techniques. This paper also summarizes the work on quality assessment of comments and presents the future direction accordingly. Note that in our survey we also investigate the work on generating code summaries, a short brief description of the code that is often viewed as a special type of comments.

The rest of the paper is organized as follows. Section II provides the motivation of automatic comment generation, and discusses the technical challenges. Section III discusses the core ideas for comment generation techniques, and gives a summarization of all kinds of techniques. In Section IV, we discuss the problem of quality assessments of comments, with datasets used in different studies as our focus, and summarize the quality assessment criteria of code comments. The future directions on the automatic code comment generation will be discussed in Section V. In Section VI, we conclude the paper.

II. OVERVIEW OF AUTOMATIC GENERATION OF CODE COMMENTS

A. PROBLEM STATEMENT

Automatic code comment generation concerns the production of some textual descriptions of source code. The essential task is to translate the code written in programming languages into textual comments written in natural languages. Meanwhile, comments may describe not only the functions, but also the design intents of developers behind source code. In brief, automatic code commenting is to generate textual description written in natural languages automatically for source code by means of source code analysis, which can reveal the design intents, program logic, functionality of programs and the meanings of the related parameters, etc.

B. CHALLENGES OF AUTOMATIC CODE COMMENTING AND RESEARCH FRAMEWORK

Although the processes of different code commenting algorithms are not completely the same, the fundamental workflows are roughly similar, as shown in Figure 1.

The processing of automatic code commenting is usually performed in three steps. First, data collection to construct datasets for comment generation systems. These data are used for training, validating and testing models, extracting code and the corresponding comments, or extracting particular rules needed by a comment generation system. In order to collect these data, researchers often crawl or download

them from open source communities or websites, e.g. Stack Overflow. Accordingly, the specific tasks in this step vary from algorithm to algorithm [17], [82], [83] to some extent. For example, it is necessary for deep neural network based comment generation system [9], [29], [31], [33], [51], [80], [89] to build high quality datasets (i.e. source files) which contain code and the corresponding comments, so as to provide data for training, testing and verification of commenting algorithms.

Second, comment generation through certain algorithms. This step can be divided into two subtasks, i.e. representation of source code and text generation. It involves varying processes depending on different algorithms of automatic code comment generation, which will be described in Section III.

Third, the assessment of the generated comments in terms of their quality. Designing practical and objective quality assessment criteria of comments directly affects the comparative results for different algorithms in performance and quality assessment. There are two popular evaluation methods including human assessment and automatic assessment, which will be discussed in Section IV.

After assessment of code comments, commenting systems will take different actions depending on the assessment results of comments. If the amount and quality of comments generated by the commenting system is satisfactory, the process of commenting will stop. Otherwise, the commenting system will go back to the first step: preparing more and suitable data, and/or adjusting source code models, generating text and assessing the quality of code comments again, repeating this process till the need of code commenting is met.

1) CHALLENGES OF AUTOMATIC CODE COMMENTING

As for automatic code commenting, the first thing is to build the source code model to express the structural, lexical, grammatical, semantic and context features of source code. Then, source code model is processed to yield the natural language comments. The third step is to evaluate the generated comments. However, generating satisfying code comments remains a challenging issue. The fundamental reason lies in the fact that programming languages are different from natural languages in nature. The difference between code and comments is two-fold: source code contains a large amount of information about classes, methods and parameters of methods, and at the same time has many nested structures and complex invocation relations; meanwhile, comments written in natural languages are unstructured, and expressed freely in form [59]. Consequently, automatic code commenting faces the following two challenges.

a: CHALLENGE 1: AUTOMATIC CODE COMMENTING ALGORITHMS

At present, there exist many kinds of algorithms for automatic or semi-automatic code commenting. We summarize them into three main classes: information retrieval based algorithms, deep neural networks based algorithms and other automatic code comment generation algorithms (see Section III for details).

The two main issues in automatic commenting include source code representation and comment text generation.

SOURCE CODE MODELS

Among automatic comment generation studies, source code model is one of the core problems. There are a number of different source code models including Abstract Syntax Trees (ASTs), parse trees, token contexts, Control Flow Graphs (CFGs), data flow, etc. The source code models that have been used in autocommenting can be classified into three categories [8]. First, the token-based source code models, which mainly extract tokens such as key words and topics from source code, e.g. the models in [26], [54]. In token-based models, source code is viewed as plain text, thus is often modeled as a bag of code tokens (BoT), or characters or bag of words (BoW). Information retrieval (IR) based commenting algorithms mainly adopt these models to represent source code. These models simply represent the lexical information of source code. Second, the syntax-based source code models, which model source code at level of abstract syntax trees (ASTs) [88]. These kinds of models are often used in deep neural network based commenting algorithms. Third, the other source code models, which represent code in forms that are fit for follow-up process. The models used in [29], [67] belong to this category, such as Software Word Usage Model (SWUM). However, a combined model that can represent various information such as lexis, syntax and structure of source code is still missing though it has received some attention [76]. As a result, seeking a comprehensive and effective model is an open research topic for source code commenting.

TEXT GENERATION

Since a natural language is unstructured and its expression form is flexible, the task of generating natural language text is difficult [11]. When it comes to code commenting, information should first be extracted from source code accurately before constructing natural language comments, which makes it more challenging. Existing solutions for text generation can be classified into three categories. First, rule-based text generation solutions, which generate text depending on designed rules or natural language models (templates) [46], [49], [55], [62], [67]–[69], [81]. Second, generative-based methods, which yield text by decoder [9], [31], [32], [51], [89]. Finally, search-based text generation solutions, which produce natural language comments through retrieving existing comment text from corpus [25], [26], [57], [61], [77], [82], [83].

b: CHALLENGE 2: COMMENT QUALITY ASSESSMENT

Quality assessment of code comments is another key problem for code comment generation. There exist two main issues in comment quality assessment: unification of datasets used for

validating and testing commenting algorithms, and selection of evaluation criteria.

UNIFICATION OF DATASETS FOR VERIFYING COMMENTING ALGORITHMS

At present, there exist many algorithms for automatic or semi-automatic code comment generation. These studies exploit different datasets to test their algorithms, which makes it difficult to compare testing results and performance of algorithms. As a result, unifying the datasets for testing is very important. However, because each specific comment generation algorithm has the language dependency, unification of dataset for testing is challenging.

SELECTION OF EVALUATION CRITERIA FOR QUALITY ASSESSMENT OF CODE COMMENTS

The lack of appropriate quality assessment metrics will lead to the absence of a quantitative comparison that highlights the strengths and weaknesses of each commenting algorithm. In existing work, the criteria of quality assessment of code comments are different depending on the category of comments. For example, according to [21], [70], from the perspective of functions, comments can be categorized into descriptive comments, summary comments, conditional comments, comments for debugging and metadata comments, etc. Even in the same category, different automatic comments generation techniques adopt different comment assessment criteria. Thus it is important to design and formulate appropriate quality assessment metrics for comments, which will promote the study of automatic code comment generation.

2) RESEARCH FRAMEWORK

Over years, software quality has always been an important research topic in software engineering. Quality of code comments is one of the important factors for evaluating software quality. As early as the 1980s, researchers began to study code comments. At present, the literature on the study of code comments focuses on relations between comments and the readability of code, relationships between comments and code understandability, algorithms of automatic program annotation and quality assessment of code comments, etc. In general, we summarize the studies of automatic code commenting and the related work from two perspectives:

- Technologies of automatic program annotation.
- Quality assessment of code comments.

The study of automatic code comment generation techniques is a hot research topic in code commenting, and another problem associated with code commenting is the study of comments quality evaluation. According to [87], there exist two kinds of comments: native comments written by code authors, and analytical comments produced by a computer program instead of code authors. We will discuss the quality assessment of analytical comments in this paper. The other studies closely related to comment generation are those on supporting comment decision that aims at guiding developers to choose the locations needed to comment in source code. With appropriate locations, comments could well improve the readability of code. Additionally, several studies are related to source code analysis and processing, such as code suggestion [28], generating natural language summaries for code defect [63], crosscutting source code concerns [62], class diagram [16] and source code commit [34], [44] etc.

In summary, this paper focuses on studies on the algorithms of comment generation and the quality assessment of comment. These two lines of work are interdependent on each other, and their relationship can be shown in Figure 2.

3) TRENDS OF THE DEVELOPMENT OF CODE COMMENTING TECHNIQUES

The research on code commenting techniques has received much attention in the last decade, fostered by the rapid spread of information retrieval, machine learning, neural networks and other related techniques. Our survey indicates that most of code commenting systems developed from 2010 to 2014 exploit information retrieval techniques, and most of code commenting systems developed in the last five years mainly adopt deep neural network techniques.

To provide a comprehensive survey of the emerging trends in code comment generation automatically, we systematically reviewed the literature from 2010 to 2018 and select 32 representative papers from 59 papers that were published in the last ten years. These papers focus on the main code commenting algorithms, and reflect the changing of research interests in the area of code commenting algorithms.

Note that in the process of collecting papers, we first performed two types of searches for related papers: (1) Online library search for papers containing keywords including "code + comment", "comment", "code + summary" and "summary" in the fields of title, abstract and index terms of the papers from ACM Digital Library, IEEE Xplore Digital Library, DBLP, Google Scholar and arXiv.org. (2) Specific search of major conference proceedings and journals in software engineering and artificial intelligence, including IEEE ICSE, IEEE FSE, IEEE/ACM ASE, IEEE TSE, ACM TOSEM, EMSE, AAAI and IJCAI. Then we refined the list of the returned papers manually and read them one by one. To further collect more relevant papers on code comment generation and avoid missing important research efforts, we further performed a citation analysis on the selected papers from keywords search. A citation analysis is a manual process of reading title and abstract of candidate papers. To sum up, we selected most of the papers through keywords search, and complemented the results further by means of manual citation search.

Figure 3 shows the distribution of papers over the years according to the types of algorithms used in the papers. The figure indicates a relevant increase of interest and results: almost half of the papers have been published in 2015-2018, and more than 60% of papers appeared after 2014. The earliest technique used in studies on automatic code

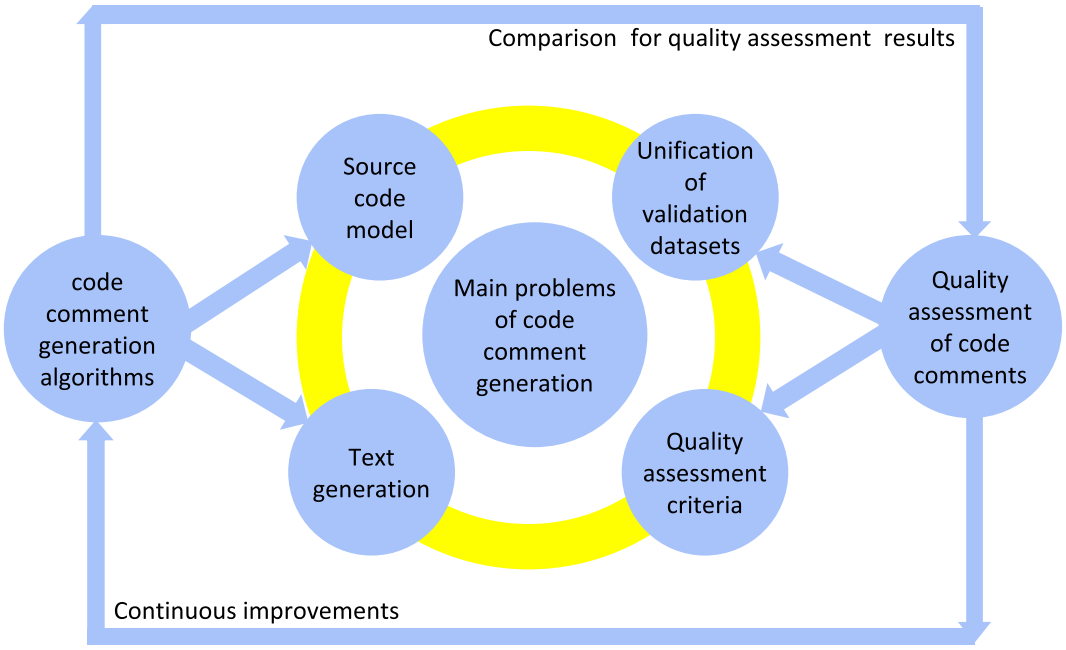


FIGURE 2. The framework of code commenting research.

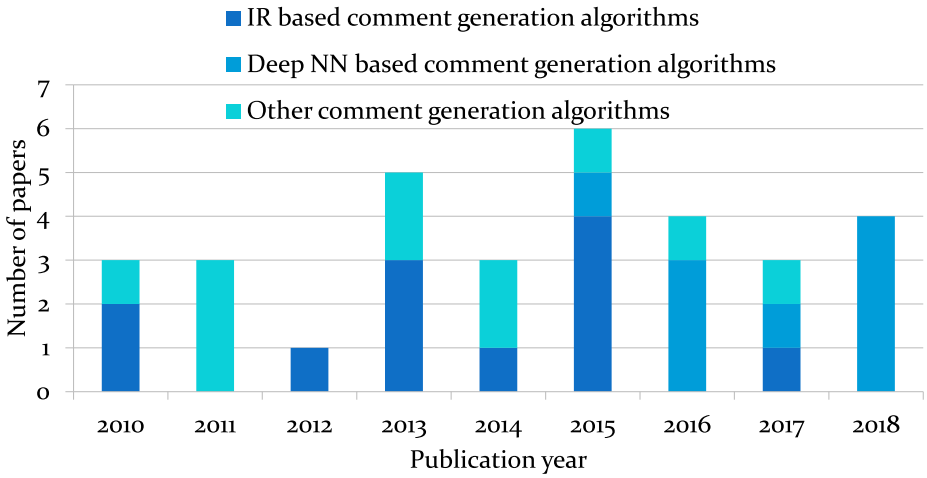


FIGURE 3. Distribution of the selected 32 research papers over publication years.

comment generation is information retrieval. In 2015, with the emergence and development of neural network techniques, deep neural network models was first applied to automatic generation of code comments. Afterwards, the interests of research in deep neural network based comment generation have been increasing dramatically over the years. At the same time, Figure 3 indicates that recent researches mainly focus on deep neural network based commenting techniques.

III. THE ALGORITHMS OF AUTOMATIC GENERATION OF CODE COMMENTS

This section first presents the classification of code commenting algorithms, then gives a thorough analysis of principles of each type of algorithms. Finally, we summarize the characteristics of the existing algorithms.

A. CLASSIFICATION OF AUTOMATIC CODE COMMENTING

As shown in Figure 4, existing algorithms mainly fall into three categories: information retrieval based algorithms, deep neural networks based algorithms and other automatic code comment generation algorithms.

In the following, we will summarize and analyze these three categories one by one in detail.

B. INFORMATION RETRIEVAL BASED COMMENT GENERATION ALGORITHMS

In general, given a piece of source code that lacks comments and a dataset of source code with comments, information retrieval algorithms first compute the relevance between the target code and other source code in the dataset. Then

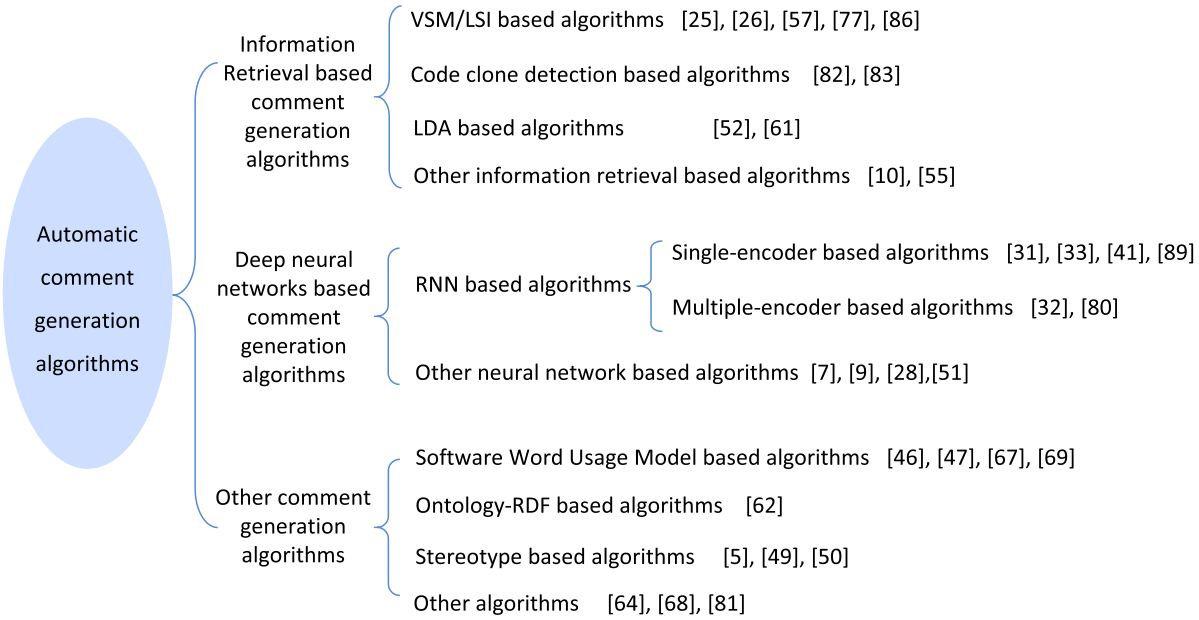


FIGURE 4. The classification of code comment generation algorithms and related literature.

one or multiple pieces of source code that well match the target code will be returned, and their comments will be used to generate the comments for the target program. The commenting algorithms based on the information retrieval techniques generally exploit the techniques based on Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Latent Dirichlet Allocation (LDA) or other related techniques like code clone detection.

One of the early applications of information retrieval (IR) techniques in the area of software engineering is about the traceability between code and comments. In 2003, Marcus and Maletic [45] employed the latent semantic index (LSI) technique, to analyze source code and the external documents for extracting semantic information from program and documents, and they further recovered links between documents and source code. Although the study itself is not on the problem of automatic comment generation, the proposed method can be applied to code commenting. Kuhn *et al.* [39] use Latent Semantic Indexing (LSI) technique to find the linguistic topics which refer to the informal semantics information contained in the identifiers names and comments in source code. And these linguistic topics reflect the intention of the code, and cluster source code according to topics.

In existing literature similarity comparison is not performed directly in the form of source code text. Source code usually is converted into varying representation forms that is fit for follow-up comparison. Most commenting systems convert source code into the form of parse tree [83] or abstract syntax tree (AST) [82], then, compare the target code with other source code from datasets. According to the comparison results, the matched code is returned. The corresponding comments of the matched code are filtered with some heuristic rules. Finally the most relevant text description is recommended as the comments or summaries for target source code. All in all, these kinds of algorithms generally generate comment texts by searching or designed rules.

Some commenting systems [82], [83] require high quality datasets which contain source code and the corresponding comments pairs in order to generate comments for programs. Some other researches make use of code clone detection techniques to find matched source code for target code.

1) VSM/LSI BASED COMMENT GENERATION ALGORITHMS

These kinds of algorithms refer to leverage Vector Space Model (VSM) [4], Latent Semantic Indexing (LSI) [2], or combination of both to generate comments for classes and methods.

VSM and LSI both belong to the techniques of information retrieval. They initially developed for the tasks of natural language processing. When we employ VSM and/or LSI to generate comments for source code, source code text or query text are usually represented as vectors, matrix or tuples [32]. Each element in the vector denotes the weight of a word in the documents. There are many methods to compute weight of term in VSM, and term frequency-inverse document frequency (tf-idf) is the most widely used weighting methods. Employing singular value decomposition (SVD), LSI recognizes term relevance between terms and concepts, and extracts the conceptual topic of a text. The commenting system determines whether the term should appear in the comments of source code according to the value of weight for each term, or calculates text similarities between the vector of query text and source code text. The term with higher similarity represents higher relevance to the code snippets or the topic of queries. So commenting systems recommend these

terms as the key words to construct comments for target source code.

These techniques are used to mine code text, and find out key words in source code text for constructing natural language description as code comments. These comments are used to describe the functions, characters and parameters of some form of source code, such as classes, methods or code block etc.

Summaries of source code can be treated as the special comments which can help developers and maintenance engineers comprehend programs more accurately and faster, saving their precious time. Several years ago, researchers have tried to summarize source code by exploiting VSM and LSI techniques. The achievements have been presented in the literature [25], [26], [77], [86].

Haiduc *et al.* [26] use VSM and LSI to analyze the source code text, and generate the extractive and abstractive natural language summaries for classes and methods. First, they convert source code documents and packages into a document collection, called corpus. Then they represent the terms which are included in the identifier names and comments from source code and documents in the corpus as a matrix. When generating extractive summaries for source code with VSM, the most relevant terms in the source code document are selected according to the chosen weight. At the same time, they also use LSI techniques to calculate the cosine similarities between the vector of each term in the corpus and the vector of a source code document to be summarized, then generate terms with high similarity that do not appear in the method or class to be summarized, but appear in the corpus. In this way, they analyze the method and class source code in Java project and generate short and accurate textual descriptions for them. In another effort, Haiduc *et al.* [25] exploit LSI only to generate summary comments for the code of Java class in open source repository.

Exploiting the same approach, Vassallo *et al.* [77] use VSM model to represent source code text and developer discussion text from Question and Answer (Q&A) on Stack Overflow as vectors, and calculate the cosine similarity between target source code text and discussion text to find the maps. The paragraph texts with high similarity are recommended as the comments of target source code. As a result, they mine the crowdsourcing knowledge to recommend comments for method.

Similarly, Panichella *et al.* [57] employ heuristic and Vector Space Model to process and analyze developer communications for methods descriptions. The developer communications mainly refer to emails and bug reports that are related to classes, methods and parameters. They extract paragraph texts which can be traced to source code methods and recognize the relevant paragraph by means of computing textual similarities (that is cosine similarity) between text paragraph and the text of each traced method. The relevant paragraph with high similarity is recommend as the method description.

The drawback of this type of technologies is that it only takes into consideration of terms that appear in the corpus or source code documents, and pays no regards to other information that is contained in source code documents, e.g., program invocation, data dependency, words sequence in source code. Therefore, it is difficult for these systems to improve the accuracy of generated comments further.

2) CODE CLONE DETECTION BASED COMMENT GENERATION ALGORITHMS

Code clone detection based comment generation algorithms are concerned with utilizing code clone detection technique to find similar code in a database, and the corresponding comments of the matched code or discussion text are viewed as comments for the target code.

Wong *et al.* [83] propose an approach based on code clone detection techniques, which mines comments from a large programming Question and Answer (Q&A) site. They mine posts from Q&A of Stack Overflow, where developers can post questions and receive the corresponding solutions in Q&A site. The posts from Q&A on Stack Overflow, containing code snippets together with the corresponding textual descriptions, are referred to as code-description mappings. They extract such mappings to build a database. Then they leverage code clone detection technique, i.e. the longest common substring, to discover similar code segments in database, and extract corresponding comments for target code segments. In a different effort, Wong *et al.* [82] mine comments from open source software projects from software repositories in GitHub. With the help of improved code clone detection techniques they find out more matched code snippets. Their new code clone detection tool leverages code parser to build ASTs for code snippets, and tokenize the serialization of AST node of code before comparison. In this way, their clone detection tool takes into consideration of structural information of source code so as to find out more matched code snippets. As a result, the improved code commenting system generates more comments for target code.

The principles of code clone detection based comment generation algorithms are simple, and the quantity and quality of the generated comments heavily depend on the scale and the quality of the dataset built for commenting systems. Consequently, when we require to provide comments for source code written in particular programming languages in certain area, it is important to build a high quality dataset containing code and comment pairs written in the same programming languages in the same domain.

The quality of comments generated by these commenting systems, to a large extent, depends upon the performance of code clone detection algorithms and the quality of comments from datasets. Accordingly, to improve the quality of the generated comments, it demands we collect more high quality open source software projects or more discussion and communication information which contain code snippets and the corresponding natural language comments.

The drawback of this approach is that the quantity of the generated comments is much smaller. The reason is that the quantity of the generated comments heavily depends on the information contained in databases or the open source software projects from GitHub. For example, if a code segment is never discussed in any posts, commenting system will fail to recommend any comment for it at all.

3) LDA BASED COMMENT GENERATION ALGORITHMS

Latent Dirichlet Allocation (LDA) [15] is one of the topic models proposed for automatically extracting topic from text documents. It is one of the probability models. Probability models used in comment generation algorithms include n-gram language models, Latent Dirichlet Allocation (LDA) [15] and the other variants of LDA. N-gram models are widely used in statistical natural language processing. LDA is an IR model that can fit a generative probabilistic model from the term occurrences in a corpus of documents [56]. LDA based comment generation algorithms are concerned with building the source code model with LDA model and generate comments for target source code. In other words, LDA can extract particular features of source code. When N-gram model is used to solve automatic comment generation problems, it is usually used to assist other statistical model to analyze source code, or train source code models. Its model is simple and effective, and it becomes the effective model for semantic mining from source code documents.

In another effort, Movshovitz-Attias and Cohen [52] use topic models, LDA, and n-gram models to predict comments for Java source code. They train n-gram models and LDA models over the same source code documents from multiple training datasets respectively. Then they consider documents as having a mixed member of two entity types, code and text tokens, and train link-LDA models over the documents. Using trained models they compute the posterior probability of document topics and with which they further infer the probability of the comment tokens. Finally the comments tokens with high probability are recommended as comments for source code files [52], [61].

Employing LDA, Rahman *et al.* [61] analyze discussions from Stack Overflow Q&A site to recommend insightful comments for open source project. They exploit a heuristic-based technique, which is different from Wong *et al.* [83], to mine the crowdsourcing knowledge to yield comment for open source project. The generated comments mainly describe the deficiency, quality and scope of source code to improve source code and can help maintenance engineers to perform the maintenance tasks.

4) OTHER INFORMATION RETRIEVAL BASED COMMENT GENERATION ALGORITHMS

Oda *et al.* [55] use phrase-based machine translation (PBML) and tree to string machine translation (T2SMT) to generate pseudo-code from Python source code. PBML and T2SMT are both statistical machine translation frameworks. Pseudo-code is a natural language description, and an informal high-level description of the operating principle of a computer program [3]. Pseudo-code can aid novice developers or inexperienced readers to understand programs, so it can be treated as a special type of comments. First, they construct the source code/pseudo-code parallel corpora (a dataset, which consists of <code, pseudo-code> pairs), that is, they add pseudo-code for existing source code by human labors. Then they exploit two frameworks (PBMT and T2SMT) and some existing, open source tools to train the pseudo-code generator over the parallel corpus. Here, open source tools include: tokenizer of the target natural languge, tokenizer and parser of the source programming language. Through training they acquire the translation rules automatically, and finally, the trained pseudo-code generator can output NL pseudo code for target source code.

In a different effort, Allamanis *et al.* [10] exploit probabilistic models that jointly model short natural language expressions and source code snippets, which represent the mapping relation between source code snippets and the corresponding natural language queries. They train parameters of their model which denote mappings between source code snippets and natural language queries. After building the model, which allows mapping in both directions: from natural language to source code, and from source code to natural language, this model can be used into two retrieval tasks, i.e. retrieving source code snippets for a natural language query, and retrieving natural language descriptions for a source code query. For the second task, matched natural language descriptions can be regarded as comments of a given source code.

To sum up, information retrieval technology is the earliest trial for researches on code comment generation. With the development of artificial intelligence and machine learning technologies, researchers continually apply the emerging techniques to automatic comment generation researches, such as deep learning algorithm. We will summarize and compare this type of technology with others in detail in the following subsection.

C. DEEP NEURAL NETWORKS BASED COMMENT GENERATION ALGORITHMS

Recent years have witnessed deep neural networks' excellent performance on natural language processing, machine translation, image recognition and speech processing [19], [36], [66], [71], [78]. In the field of software engineering, researchers formulate the conversion task from source code to comments as a translation problem between programming languages and natural languages, and they try to exploit deep neural network approaches to solve source code commenting problems [7], [9], [31]–[33], [41], [51], [80], [89].

Deep neural network based comment generation algorithms fall into two main categories: RNN based algorithms and other neural network based algorithms. There are three kinds of deep neural networks: Convolutional Neural Network (CNN), Recurrent Neural Network (RNN) and Recursive Neural Network (RvNN) [60]. Convolutional neural networks are fit for solving the problems of natural

language processing (NLP), image recognition and speech processing [11], [36]. Recurrent neural networks are good at processing and predicting sequential data, and are preferred in NLP and speech processing. So, RNN is fit for generating comments for source code. RvNN is preferred in NLP too, and it can process source code to generate comment for code [41]. When borrowing neural machine translation techniques as the solution of commenting problems, two important structures, encoder-decoder structure and attention mechanism, are worth mentioning. They are usually used in deep neural networks based commenting systems to assist to generate comments for source code.

a: ENCODER-DECODER FRAMEWORK

In deep neural network based comment generation systems encoder-decoder structure, also known as sequence to sequence model, is generally exploited. In the structure of encoder-decoder, the encoder plays the role of encoding source code into a fixed-sized vector; and the decoder is responsible for decoding source code vector and predicting comments for source code. The difference among various encoder-decoder structures lies in the form of inputs and the type of neural network [24]. Generally, the inner structure of encoder-decoder can choose RNN, CNN [24] and the variants of RNN, such as Gated Recurrent Unit (GRU) and Long Short term Memory model (LSTM).

b: ATTENTION MECHANISM

Attention mechanism is usually added to the encoder-decoder framework. It is responsible for dynamically assigning the higher weight values to more relevant tokens of each word in the input sequences of decoders. In this way, it makes the effectiveness of decoders improved. Attention mechanism is proposed by Bahdanau *et al.* [12] and Mnih *et al.* [48]. It is a good solution for the problem of the poor performance in the case of the long sequences.

Since Deep neural network based comment generation algorithms belong to the categories of machine learning, commenting generation systems based on the Deep Neural Networks require high quality datasets containing code and comments to train the neural network. The dataset can meet all data requirements for the system, providing data for training and for validation and test of the commenting algorithm as well.

When training and adjusting the parameters for the neural network model, we could take some preprocessed source code, (such as the AST of the code or the sequences of AST of source code, etc.) as the input data. The corresponding comments of source code can be treated as the output of the commenting system. Trained neural networks can generate natural language descriptions for target programs.

1) RNN BASED COMMENT GENERATION ALGORITHMS

In order to model temporal sequential information with neural network, developers design RNN algorithm. RNN is a popular and widely used algorithm in deep learning [19], especially in natural language processing and speech processing [40]. When RNN algorithm servers as the solutions for comment generation, the encoder-decoder structure and attention mechanism are commonly used so as to improve the accuracy of commenting systems.

Another two important variants of RNN are Long Short term Memory model (LSTM) [66], [72] and Gated Recurrent Unit (GRU) [18]. LSTM is a special kind of RNN, capable of learning the long term dependency information. The characteristics of LSTM are that it has a three-gated-controller structure and construct a controlled memory neuron which solves the gradient descent and gradient explosion in the traditional RNN. Compared with LSTM, GRU is simple in structure, and overcomes the shortcomings of LSTM: complex structure, complicated realization and lower execution efficiency. GRU only uses two gates: one is the update gate, and the other is the reset gate. The characteristics of GRU are that it is easy to realize, having shorter runtime, simpler parameter training and more easily train than LSTM, especially in the case of processing large training data. So GRU is adopted in many applications.

According to the number of RNN used in encoder, we divide the RNN based comment generation algorithms into two categories: single-encoder based commenting algorithms and multiple-encoder based commenting algorithms.

a: SINGLE-ENCODER BASED COMMENT GENERATION ALGORITHMS

In single-encoder based comment generation algorithms, the encoder is composed of one RNN structure. This is a classical encoder-decoder structure used in comment generation tasks.

Iyer *et al.* [33] propose an LSTM based comment generation model, named CODE-NN, which uses Long Short Term Memory (LSTM) networks with attention mechanism to produce natural language summaries for C# code segments and SQL queries. The character of CODE-NN is adding the attention mechanism to LSTM model. CODE-NN is trained in the dataset automatically collected from Stack Overflow which includes title and code segment pairs. During the process, attention mechanism highlights the relevant and important tokens in source code, completes the relevant and important content selection, and LSTM provides the context for words. After training the parameters of embedding matrices and other matrices, Iyer et al. leverage the trained model and an input code snippet to generate the natural language summaries for the code.

GRU is also used in the encoder-decoder framework to generate comments for source code. In a different effort, Zheng *et al.* [89] use the encoder-decoder structure whose basic element is GRU. Their attention module in encoder is a global attention mechanism. They take the embedding symbols (e.g identifiers) in code snippets as learnable prior weights to evaluate the importance of different parts of input sequences. After sorting identifiers in code segments based on the order of appearance, they encode tokens of source code

into an embedded token. In this way, their attention mechanism is able to focus on these specific features in programs. In other words, their attention mechanism can understand the structure of code better. Finally attention mechanism helps to improve the accuracy of generated comments in commenting system.

In another effort, Hu *et al.* [31] suggest that the AST sequences of source code generated by Structure-Based Traversal (SBT) often bring about much structural information from source code. So they adopt sequence to sequence model and attention mechanism to generate comments for source code. In their commenting system, named DeepCom, they use the AST sequences of source code generated by Structure-Based Traversal (SBT) as the input of neural network, and LSTM as the basic element of encoder and decoder. After training, the neural network can learn the structural and semantic information from source code, which finally generate more accurate comments for the code.

Since Recursive Neural Network (shorting for RvNN) can be used to represent parse trees of natural language sentences, Liang and Zhu [41] apply a RvNN over the parse trees of source code to combine the semantic and structural information from the code into representation vectors. Then they leverage a recurrent neural network decoder (Code-GRU) to decode these vectors. The overall framework generates text descriptions for the code with accuracy higher than other learning based approaches such as sequence-to-sequence model. Their algorithm mainly generates summary for code blocks.

b: MULTIPLE-ENCODER BASED COMMENT GENERATION ALGORITHMS

These kinds of comment generation algorithms contain more than one encoder in the encoder-decoder structure. Because each encoder represents and extracts one type of information from source code, these comment generation algorithms can produce code comments with high accuracy with the help of multiple encoders.

Hu *et al.* [32] exploit the transferred knowledge acquired from the process of automatic API summaries to solve the problem of automatic source code comment generation. Thus their commenting system is equipped with two encoders: an API encoder and a code encoder. This system produces higher quality summaries for source code than other code summary generation systems. The core of their commenting algorithm is putting an API encoder into the RNN encoder-decoder model. With the help of learned transferred API knowledge, the RNN decoder integrate attention information collected from both two encoders to produce the summary for target code. Their algorithm outperforms CODE-NN, a state-of-the-art code summarization approach at that time. The study is innovative for adding another API encoder which enhances the performance of code summary generation.

Exploiting reinforcement learning, Wan *et al.* [80] improve automatic code summarization. They employ one LSTM model to represent the sequential information of code, and another AST-based LSTM model to represent the structure of source code. That is, they use two encoders in the classical encoder-decoder framework. Under this framework, they exploit reinforcement learning model to solve two issues: exposure bias and inconsistency between train and test measurement [37]. They leverage an actor network and a critic network to jointly determine the next best word at each time step. Besides, they employ an advantage reward composed of BLEU metric to train both networks. In this way, they directly use the characters from comment assessment to optimize code summarization.

To sum up, RNN can utilize the sequential information in the input data [60]. This property is essential in comment generation where the structure embedded in the source code sequence conveys useful knowledge, such as the structure of program and lexical and syntactic information of source code. RvNN also can make predictions in a hierarchical structure using compositional vectors [60].

2) OTHER NEURAL NETWORK BASED COMMENT GENERATION ALGORITHMS

Although seq2seq model generally adopts RNN based encoder-decoder structure, CNN model also can be used to construct encoder-decoder architecture [24] in the task of natural language translation. The strength of CNN encoder-decoder is that the computation over all elements can be fully parallelized during training to better exploit the GPU hardware. CNN model can be used to represent the syntactic and semantic information of source code. With the assistance of the special attention mechanism or the RNN decoder, CNN can solve the problem of input sequence [24]. When it is applied to sequence problem, the convolution model can represent input sequential data hierarchically. The role of CNN lies in that it can extract hierarchical feature representation from input sequence.

Allamanis *et al.* [9] introduce a convolutional network into attention mechanism, which can produce short and descriptive summaries for source code snippets. The commenting system adopts the encoder-decoder framework and attention mechanism, where the basic element of decoder is GRU. Their convolutional attention module applies convolution on the input source code snippets to detect patterns and determine the important token sequences where attention should be focused. So generally CNN models are not commonly used to model temporal sequences, apart from those mentioned in [24]. We usually select CNN model and other auxiliary model to solve the temporal sequences problems.

In another effort, Mou *et al.* [51] exploit convolutional neural networks to represent the abstract syntax tree of source code as vectors, then classify programs according to functionality. They take the entire AST of a program as input, and design a convolution kernel over AST of source code to capture structural information of source code. Although the goal of their work is not to generate comments for programs, their source code model can be used for generating summary for program in future work.

Although most of the aforementioned approaches adopt encoder and decoder structure, Allamanis *et al.* [7] adopt a log-bilinear model in neural network to suggest method and class names. Their model leverages continuous embedding to represent identifier name. They believe identifiers with similar vector will appear in similar contexts. So they recommend the name of class or method by selecting the name with similar vector, and that means comparing the vector between the function body and candidate identifier name. In a word, they exploit a neural probability model instead of encoder-decoder framework to solve the method naming problem.

In existing literature, RNN and CNN all can be exploited to model source code and generate natural language summaries or comments for source code [9], [51]. RNN makes use of GRU or LSTM to represent the long distance features between long input sequences. CNN exploits a convolutional attention or convolution layers to collect and represent the features and position model of source code.

All in all, in the deep learning based commenting algorithms, RNN, LSTM, GRU or CNN are generally adopted to model source code and perform various SE tasks. In comment generation models, RNN and CNN models are equipped with attention mechanism to improve the accuracy and efficiency. Although deep neural network based comment generation algorithms receive successes in comment generation, they still have difficulties in modeling multiple complex information from source code at the same time. Combined model is a popular direction in future researches.

D. OTHER COMMENT GENERATION ALGORITHMS

We further present some other technologies for generating code comments. These technologies mostly either adopt some existing models from other research areas to represent source code or adopt models that exclude the aforementioned models, such as Software Word Usage Model (SWUM) [29], Ontology based Resource Description Framework (RDF) [62] and stereotype identification [22] etc. These models can represent the structure and semantics of source code. Many commenting systems [46], [47], [67], [69] apply software word usage model (SWUM) to represent structural, semantic and lexical information in source code; some systems [62] exploit ontology based Resource Description Framework (RDF) to depict semantic information of source code, and use heuristic method to find out key facts in the source code, etc. Several efforts [5], [49], [50] leverage stereotype identification techniques to generate summaries for Java classes and methods. Finally these solutions improve the accuracy of comments and performance in commenting systems to some extent.

1) SOFTWARE WORD USAGE MODEL BASED COMMENT GENERATION ALGORITHMS

Software Word Usage Model (SWUM) was proposed by Emily Hill in 2009. She designs this model in [29], [30]. This model is a new representation model for source code. Compared with software BOW (short for Bag Of Word), which is the commonly used model for software in earlier time, SWUM represents much textual and structural information in source code to improve software searching and program exploration. SWUM combines textual and structural information of source code into one model.

Hill proposes SWUM model to represent the facts in source code. An SWUM model is composed of three layers [29]: SWUM_{word} which models program words, $\text{SWUM}_{program}$ which models program structural information, and SWUM_{core} which models phrase structure in source code, and bridge edges connect different layers. Thus SWUM can not only extract the lexical and structural information from source code, but also links between the linguistic information and structural information. With this model, developers have designed different score functions in order to complete the different application tasks. According to score values calculated by score functions, the focal phrases and key structural information are selected from source code. Finally with these focal phrase and key words, comments are produced with the assistance of designed language templates.

Sridhara *et al.* [67] utilize SWUM to represent source code and generate comments automatically for Java methods and the focal parameters of java classes. Through traditional program analysis and natural language analysis they first construct SWUM, then select the content to be included in the summary comment, and leverage the focal terms and keywords to construct the natural language text from templates for method. After combining and smoothing the generated text, the summary comments are yielded. Similarly, Sridhara *et al.* [69] combine SWUM and heuristics to produce parameter comments for method and add them to the method summaries. They first leverage structural and linguistic information of source code to determine the content to be included in summary of parameter of method, then generate succinct description phrases with the assistance of templates. These description phrases are used to describe the intent of method.

Based on the same model, McBurney and McMillan [47] combine PageRank algorithm and SWUM to generate summary for Java method. They use SWUM to represent source code and extract keywords about the behavior of important methods. The role of PageRank is to discover the most important methods in given context. Their summary is of higher quality owing to the addition of context information of method. Here, the context information of method is obtained by the analysis of method calls. These commenting systems all adopt SWUM to represent source code models.

2) ONTOLOGY-RDF BASED COMMENT GENERATION ALGORITHMS

Rastkar *et al.* [62] propose an approach that automatically produces a natural language summary for crosscutting source code concerns.[1] They extract the structural and lexical information from source code and exploit ontology instance

[1]Crosscutting source code concerns: are concerned with source code that crosscut the modules defined in the code.

to store and describe the extracted semantic information, and they manipulate an ontology instance through an RDF graph which is used to depict resources originally. In their approaches, triples (resource, attribute type, attribute value) are used to represent one attribute of a particular class. Then they use a set of heuristics to find salient code elements and patterns from the related code concern. Finally, the information extracted in the previous steps is used to construct summary comments according to templates. Their generated summaries mainly describe what is the code concern and how it is implemented.

3) STEREOTYPE IDENTIFICATION BASED COMMENT GENERATION ALGORITHMS

Stereotypes are a simple abstraction of a class's role and responsibility in a system's design, for instance, an accessor is one of the method stereotype that returns information [22].

Moreno *et al.* [49], [50] use stereotype information of Java classes and methods to select content to be included into the summary and generate summary for Java class. First, they identify stereotype information of Java classes and methods. Next they exploit heuristic rules to determine which information in source code to be extracted and added into the summaries. Their summaries focus on the responsibilities and content of the classes instead of their relationships with other classes. In another effort, Abid *et al.* [5] differentiate the type of stereotype of C++ methods using stereotype identification, and employ static analysis to extract the main components of the method, so as to generate summaries according to predesigned templates for every type of stereotype of methods.

4) OTHER ALGORITHMS

Employing heuristics approach only, Sridhara *et al.* [68] generate concise comments for the high-level action in Java methods. In their commenting system, they design and implement the rule sets by which code snippets of statement sequences, conditionals and loops are identified. Next they develop the corresponding summary templates for generating comments for Java methods. Finally they generate summaries for Java methods according to these templates.

Among many literature on the study of commenting generation, there are several studies which adopt approaches initially designed for non-commenting problems. In a different effort, Rodeghero *et al.* [64] introduce an eye-tracking technique into comment generation researches. Basing on the related studies of eye-tracking in program comprehension, they detect eye movements and the amount of gaze time which programmers spend in scanning source code. According to these information they adjust the weight value of words in source code, and identify keywords, which makes generated summaries more accurate. In another effort, Wang *et al.* [81] use some rules mined from open source projects to identify the object-related action unit within a method, and generate natural language phrases for action unit. They exploit data-driven approach to obtain a set of rules to identify the focal statements and arguments in source code. In the end, they employ the information obtained in the previous step to construct natural language description for action unit according to the predefined templates.

E. SUMMARIZATION OF COMMENT GENERATION ALGORITHMS

We have described three main categories of comment generation algorithms, and we further summarize the characteristics of those algorithms.

The characteristics of Information Retrieval (IR) based algorithms are salient on three aspects. First, these kinds of commenting algorithms view source code as plain text, and construct comments by searching keywords or tokens from source code or searching comments from similar code. They attach great importance to lexical semantics of source code, and neglect the structural information, data dependency and invoke information of source code. Second, the effectiveness of IR-based algorithms, to a large extent, depends on the similar code from datasets. Without similar code in datasets, IR-based algorithms will fail to generate accurate comments or any comment at all for classes. If the source code contains poorly named identifiers, they still fail to recommend accurate comments. Third, for IR-based algorithms, the quality and quantity of generated comments both depend on the quality and quantity of source code contained in datasets.

Among deep neural network based comment generation algorithms, most of them employ RNN encoder-decoder model with the assistance of attention mechanism to represent the features of input code as the vector. They usually formulate comments generation from source code as machine translation in natural language processing. The role of attention mechanism lies in adjusting weights of the tokens in source code, which makes the accuracy of the generated comments better. When choosing CNN to build comment generation system, it is necessary to equip it with attention mechanism. The deep neural network based comment generation algorithms are one type of the supervised learning. So, these kinds of algorithms require high quality datasets for training parameters of neural network so as to acquire the generative model for source code.

Besides information retrieval based algorithms and deep neural network based algorithms, other comment generation algorithms mostly either adopt some existing models from other research areas to represent source code, such as eye-tracking and semantic ontology based RDF etc., or adopt models that exclude the aforementioned models, such as SWUM model to represent source code and yield satisfactory comments for given source code.

IV. QUALITY EVALUATION CRITERIA OF CODE COMMENTS

As discussed above, most existing commenting algorithms are evaluated based on different datasets, which causes the experimental results to be noncomparable. The fundamental

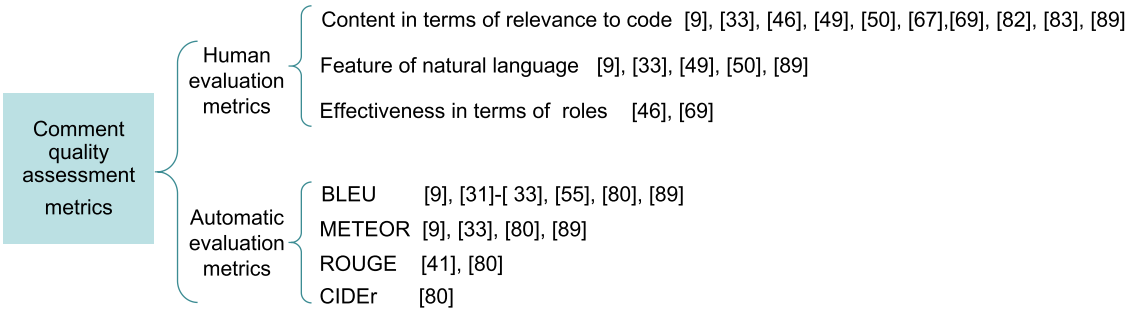


FIGURE 5. The classification of code comment quality assessment metrics and referenced literature.

reasons lie in the lack of an unified dataset for algorithm evaluation. In addition, appropriate comment evaluation criteria in terms of the efficiency and effectiveness of algorithms are still missing.

In recent years, some researchers have conducted many studies on comment evaluation models. The studies on comment quality assessment and reference literature are shown in Figure 5. Khamis *et al.* [38] present JavadocMiner, a tool that can analyze the quality of Javadoc comments automatically. Using a set of heuristics, they evaluate the quality of comments in terms of quality of language and consistency between source code and its comments. Steidl *et al.* [70] perform the study on quality evaluation for native comments with a quality assessment model and a set of comprehensive and systematic metrics. They classify comments into seven categories: copyright comments, header comments, member comments, inline comments, section comments, code comments and task comments. Then they propose the corresponding evaluation metrics and a general quality model for each type of comments according to the code-comment relevance and the length of comments. Their work mainly focuses on the quality of inline comments and member comments. In 2016, Yu *et al.* [87] propose a source code quality assessment approach based on aggregation of classification algorithms, which evaluates comments in terms of their format, language form, contents and correlation degree of code. This method is characterized by introducing machine learning and natural language processing technologies into comment quality assessment. They improve the evaluation results by using the aggregation of the basic classification algorithms. In general, there are a few studies on quality assessment for source code comments in existing literature, however, there is a lack of studies on the quality assessment of analytical comments.

There are two classes of assessment methods available: manual evaluation and automatic evaluation. Manual evaluation relies on experienced developers to analyze and rate the generated comments one by one according to predefined quality metrics, such as conciseness, readability, accuracy, etc. Although the manual assessment methods avoid the complex technological design of feature selection and evaluation algorithms, it is usually time-consuming and costly, thus it is not fit for a vast number of analytical comments. However, there are no automatic quality assessment tool available exclusively for code comments, and researchers in software engineering field generally borrow natural language assessment criteria and tools. The commonly used metrics and tools in automatic assessment are BLEU [58] and METEOR [13]. Sometimes, ROUGE and CIDER are also used. To sum up, the comment quality assessment generally involves the following three perspectives:

- **Experimental datasets**, which refers to datasets that are used for evaluating code commenting algorithms.
- **Evaluation methods**, which generally include manual assessment and automatic assessment.
- **Evaluation metrics**, which refer to the criteria for evaluating the quality of comments.

A. DATASETS FOR VALIDATION

The datasets for validation in existing studies usually varies across commenting systems. These datasets come from three sources: open source projects from GitHub software repositories, Q&A sites in Stack Overflow, and communication and discussion information, like emails, among developers.

Although there only exist three kinds of datasets, every commenting system collects different projects written in different programming languages as datasets. Even some systems collect information from the same source such as Q&A sites in Stack Overflow, there still exist differences in tags and time segments. As a whole, there is not a uniform public validation dataset for use.

B. AUTOMATIC EVALUATION

Since comments are sentences written in natural languages, the comment quality assessment criteria mostly come from the corresponding evaluation metrics in the field of natural language processing. These metrics originally are designed for measuring the quality of sentences generated by machine translation system, and have been justified to well reflect the accuracy of test results, and the test results are highly coherent with human evaluation. Existing automatic evaluation metrics for comment quality mainly include BLEU, METEOR, ROUGE and CIDEr.

1) BLEU

BLEU [58], standing for Bilingual Evaluation Understudy, is a method for automatic evaluation of machine translation, proposed by Papineni et al. in 2002. It can be used to analyze automatically the degree of common appearance of candidate texts and reference translations. Since human evaluations of machine translation are extensive and expensive, BLEU is designed as a quick, inexpensive and language-independent automatic evaluation method for machine translation. In terms of the reliability of evaluation results, the results of BLEU highly cohere with human evaluation, so it is extensively applied to the evaluation of machine translation and the evaluation understudy of automatic commenting systems [31], [32], [89]. The key assessment contents of BLEU is n-gram precision, which refers to the proportion of the matched n-grams out of the total number of n-grams in the evaluated translation. Precision is calculated separately for each n-gram order, and the final precision is computed as weighted geometric mean for each precision. For the candidate texts, the more similar they are to the natural language reference written by human, the higher scores they get. For n-gram precision, n can be 1, 2, 3, 4. When $n = 1$, BLEU has good performance at the sentences with well matched in corpus level, but the matches in sentence-level become poorer as the length of the sentences grow longer. BLEU does not take direct recall into account. That lack of recall and no consideration about explicit word-matches between translation and reference makes the degree of coherence between human evaluation and BLEU evaluation advance difficultly.

2) METEOR

METEOR [13], short for Metric for Evaluation of Translation with Explicit Ordering, is proposed by Lavie in 2005. It is a supplementary evaluation metric to BLEU, which incorporates the recall to reflect how much the translated results cover the entire contents of the source sentences. In consideration of the weakness of BLEU method without reflecting the recall, METEOR takes the recall into account, and computes a score based on explicit word-to-word matches between translation and a reference translation and evaluates the translation according to the score. The higher the score, the closer it is to the reference translation, which denotes the higher quality of the translation. Experiments show [13] that METEOR has significantly improved coherence between human judgments in comparison of BLEU. So, METEOR metrics usually are used to evaluate the quality of machine-generated comments of code as the complement to BLEU.

3) ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) proposed by Lin [43], is an automatic summary assessment method, and now it is extensively adopted in the tasks of summary assessment. ROUGE includes several different metrics, among which four commonly used ones are ROUGE-N, ROUGE-L, ROUGE-W and ROUGE-S. They are recall-based methods which measure similarity between a summary and the ideal summaries written by human. ROUGE-N and ROUGE-L are generally used as the metrics for measuring the summary comment quality assessment [41], [80]. ROUGE-L denotes the longest common subsequence-based precision and recall statistics.

4) CIDER

CIDEr [79], standing for Consensus-based Image Description Evaluation, is proposed by Vedantm in 2015 in the computational vision and mode recognition conference. Researchers believe that multiple existing evaluation metrics have high relevance to human evaluation, but there is no way to unify them into an uniform metric which can determine the similarity between a candidate sentence and reference sentences. In order to solve this issue, a consensus-based metric, named CIDEr, is proposed. Its primary principle is to measure the similarity between a test sentence and the majority of reference sentences. The experiment results demonstrate that CIDERr metric coheres with human consensus-based matches better than the existing aforementioned metrics.

Accordingly, researchers generally exploit BLEU to evaluate machine-produced translations automatically firstly, then METEOR is used as a supplementary to final results of evaluation, which makes assessment results more objective and closer to human evaluation results. And some code summary generation systems adopt ROUGE and CIDEr metrics to evaluate generated comments [80], but ROUGE and CIDEr are not used extensively.

In summary, because of similarities between code commenting and machine translation, researchers usually adopt BLEU, METEOR, ROUGE and CIDEr methods to measure the quality of computer-produced comments. Among these metrics BLEU and METEOR are the common used metrics.

C. HUMAN EVALUATION

In the commenting systems summarized in Section III, researchers ask human evaluators to evaluate the generated comments on assessment metrics.

Though the scores from human evaluation are subjective and manual scoring is subject to low efficiency, it is still one of the important methods for assessing the performance of many commenting algorithms. There are several manual assessment metrics for measuring the quality of generated comments. Although the names of the quality assessment metrics in different researches are not completely the same, we group them into three categories according to their characteristics as follows.

First, measuring code comments with their contents: evaluating the generated comments on contents, such as adequacy, accuracy [46], [67], [69], conciseness, informativeness [33] and interpretability [89], etc. The meanings of the features in this group are described as follows.

- **Accuracy** measures to what extent the generated comments reflect the semantics of the relevant source code.

- **Content adequacy** is used to evaluate how much information the comments miss with regard to the information contained in source code.
- **Conciseness** is used to evaluate to what extent the comments contains the unnecessary information.
- **Informativeness** measures the amount of contents carried over from the input code to the NL comment, ignoring fluency in language.
- **Interpretability** measures to what extent the generated comments convey the meaning of the source code.

Second, measuring code comments with the features of natural languages: evaluate the generated comments on grammaticality and fluency, ignoring contents of comments. These modalities are, namely, expressiveness [50], naturalness [33] and understandability [89] etc. The definition of each feature in this group is described as follows.

- **Naturalness** measures the grammaticality and fluency of comments.
- **Expressiveness** measures the comments' readability and understandability in the respect of their way of description.
- **Understandability** is used to evaluate comments according to their fluency and grammar.

Naturalness and understandability metrics almost have the same meaning in different names in different researches.

Third, measuring code comments in terms of their effectiveness: judging whether the generated comments are useful [46] and necessary [69], or evaluating to what extent can the generated comments help developers understand the programs, namely code understandability. The meanings of features in this group are described as follows.

- **Usefulness** measures to what extent the generated comments are useful for developers to understand code.
- **Code understandability** is used to evaluate to what degree does the generated comments help developers understand the programs.
- **Necessity** measures to what extent the generated comments are necessary.
- **Utility** measures to what extent the generated comment can help developers comprehend code.

All modalities discussed above can be rated in different scale. Some researchers rate comments on a scale between 1 and 5; some rate comments on a scale between 1 and 4, or on a scale between 1 and 3. For accuracy metric, evaluators can rate comments by answering multiple choice questions, and each question could be answered as ''Strongly Agree'', ''Agree'', ''Disagree'', ''Strongly Disagree'', four items in total; or answered as ''Strongly Agree'', ''Agree'', ''Neural'', ''Disagree'', ''Strongly Disagree'', five items in total; or ''Accurate'', ''Slightly Inaccurate'', ''Very Inaccurate'', three items in total. The answers are assigned on a scale between 1 and 5, or between 1 and 4, or between 1 and 3. For other metrics, the same approach can be adopted to rate comments.

In addition, there are two assessment metrics worth mentioning: precision and recall. These two metrics can be used to evaluate the usefulness of generated comments. Precision and recall metrics both come from information retrieval metrics [53]. Precision [13] is the proportion of the correct n-grams in the evaluated comments. Recall [13] is the proportion of the correctly predicted n-grams in reference comments. Among aforementioned metrics, BLEU is based on precision metric, and METEOR is based on recall metric.

Evaluators are usually the programmers with five-year experiences in the corresponding programming language development or Masters/PhDs in the corresponding field.

Human evaluation is characterized by high accuracy and convincing assessment results, while it is also subject to high costs, subjective influence from evaluators and being time-consuming. However, human evaluation is still a very important quality assessment methods for code comments. And it is not replaced by automatic evaluation. All in all, automatic evaluation has its own advantages, it can supplement the weakness of manual evaluation. Till now, there is no mature, efficient and inexpensive comment quality assessment tools, which is an important problem to solve in the field of code comment generation fields.

To sum up, when designing evaluation experiments and selecting evaluation criteria, we should design and select the suitable assessment criteria according to the algorithms that are adopted in commenting systems so as to make the designed experiments and assessment results more convincing.

V. FUTURE DIRECTIONS

As an important research direction in software engineering field, the source code commenting technologies have received much attention from the academics since the last decade. But it remains a challenge research topic due to its internal complexity and the limitations of existing technologies. In recent years, one of the representative efforts is to utilize deep neural networks to solve the problem of automatic comment generation, and some promising results have been obtained. However, there still exist some drawbacks such as low accuracy of generated comments and insufficient generated comments on the whole. In the following discussion, we summarize the future research opportunities for automatic comment generation.

- **Exploring the synergy between deep neural network and other models.** At present, the deep neural networks technology as the emerging technology is adopted to solve the problem of automatic generation of code comments, and obtains better results than previous methods. The reason is that the structure of deep neural network technology is fit for solving the problem of sequences to sequences. However, the problem of automatic generation of code comments is not simple translation problems within the natural language, it is the conversion problem from structured source code to natural language

sentences. As a result, exploring the synergy between deep neural network and other models to represent the source code remains an open research topic.

- **Fusion of different source code models.** The combination of multiple models for representing source code is well suitable for the solution of code comment generation problems. Because one model is fitter for describing one feature of source code. Token-based model can be used to describe lexical information in source code, which is the words and tokens hidden in identifiers name or comments of source code, such as BoW model etc. And statistical language models describe the probabilities for the words to appear in sequences, such as n-gram model etc. Besides, SWUM model is suitable for representing the semantic, structural information and phrases information in source code, but it cannot represent word sequences in source code. Accordingly, appropriate fusion of different source code models may be open for future research.
- **Designing a customized, intelligent automatic comment generation system to meet various scenarios.** To solve the problem of automatic code comment generation, we should perform mainly two tasks: source code representation and text generation. The goal of code commenting is to improve readability of source code, which is to help software developers and maintenance engineers comprehend programs much faster and better, and is beneficial for them to perform other tasks; On the other hand, code commenting can free developers from writing comments manually, which is laborious and tedious for developers. So it is one of the important directions for automatic code comment generation to design a customized, intelligent automatic comment generation system to meet the requirements of different developers in specific application scenarios.
- **Unification of test datasets and comment quality assessment model.** In terms of code comment assessment, the unification of multiple test datasets opens the future direction first, because if there were no unified test dataset, there would be no means to compare the advantages and disadvantages of commenting generation algorithms, which will certainly affect the development of algorithms. So designing and building up a unified, universal test dataset is an urgent problem needed to be solved. Second, designing and setting up an appropriate, objective comment quality assessment model is another vital research direction. Accordingly, the study on comment quality assessment metrics is also an active research direction of comment assessment.

VI. CONCLUSION

This paper provides a survey of the recent development of automatic code comment generation technology. The research in this field has explored information retrieval based code commenting techniques, neural network based code commenting techniques and other code comment generation techniques. We introduce the features of code commenting problem, the research framework and the workflow of automatic code comment generation. By summarizing and analyzing three main classes of automatic code comment generation techniques, we present the future research opportunities. Specifically, exploring the neural network, combination of multiple technologies and flexible switching in multi-application scenarios remain open research topics.

In the field of quality assessment research on code comments, we summarize four kinds of automatic assessment metrics including BLEU, METEOR, ROUGE and CIDER, which represent the strengths and weaknesses of each comment metric; and we outline the commonly used metrics in manual evaluation from three aspects: natural language features, contents of comments and effectiveness of comments. Choosing reasonable automatic assessment criteria and manual assessment criteria, building the universal validation datasets are open for future research on comment quality assessment.

REFERENCES

[1] *Comment (Computer Programming)*. Accessed: Aug. 12, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Comment_(computer_programming)
[2] *Latent Semantic Analysis*. Accessed: Aug. 12, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Latent_semantic_analysis
[3] *Pseudocode*. Accessed: Aug. 12, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Pseudocode
[4] V. S. Model. [Online]. Available: https://en.wikipedia.org/wiki/Vector_space_model
[5] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using stereotypes in the automatic generation of natural language summaries for C++ methods," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2015, pp. 561–565.
[6] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Proc. Annu. Rel. Maintainability Symp.*, Jan. 2002, pp. 235–241.
[7] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Suggesting accurate method and class names," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 38–49.
[8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, p. 81, Jul. 2018.
[9] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.
[10] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2123–2132.
[11] G. Angeli, P. Liang, and D. Klein, "A simple domain-independent probabilistic approach to generation," in *Proc. Conf. Empirical Methods Natural Lang. Process. Assoc. Comput. Linguistics*, Oct. 2010, pp. 502–512.
[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*. [Online]. Available: https://arxiv.org/abs/1409.0473
[13] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.
[14] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Impact of limited memory resources," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 83–92.
[15] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Jan. 2003.
[16] H. Burden and R. Heldal, "Natural language generation from class diagrams," in *Proc. 8th Int. Workshop Model-Driven Eng., Verification Validation*, Oct. 2011, p. 8.

[17] P. Chatterjee, B. Gause, H. Hedinger, and L. Pollock, "Extracting code segments and their descriptions from research articles," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 91–101.

[18] K. Cho, B. V. Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014, *arXiv:1409.1259*. [Online]. Available: https://arxiv.org/abs/1409.1259

[19] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*. [Online]. Available: https://arxiv.org/abs/1406.1078

[20] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proc. 9th Workshop Stat. Mach. Transl.*, 2014, pp. 376–380.

[21] L. I. D. Technischen and D. Steidl, "Quality analysis and assessment of source code comments," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 83–92.

[22] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[23] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 2018.

[24] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 1243–1252.

[25] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. 32Nd ACM/IEEE Int. Conf. Softw. Eng.*, vol. 2, May 2010, pp. 223–226.

[26] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Working Conf. Reverse Eng.*, Oct. 2010, pp. 35–44.

[27] D. Haouari, H. Sahraoui, and P. Langlais, "How good is your comment? A study of comments in java programs," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2011, pp. 137–146.

[28] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. 11th Joint Meeting Found. Softw. Eng.*, Aug. 2017, pp. 763–773.

[29] E. Hill, *Integrating Natural Language Program Structure Information to Improve Software Search Exploration*. Newark, DE, USA: University of Delaware, 2010.

[30] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 232–242.

[31] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, May 2018, pp. 200–210.

[32] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proc. IJCAI*, 2018, pp. 2269–2275.

[33] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. Annu. Meeting Assoc. Comput. Linguistics*, vol. 1, 2016, pp. 2073–2083.

[34] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2017, pp. 135–146.

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

[36] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, *arXiv:1404.2188*. [Online]. Available: https://arxiv.org/abs/1404.2188

[37] Y. Keneshloo, T. Shi, C. K. Reddy, and N. Ramakrishnan, "Deep reinforcement learning for sequence to sequence models," 2018, *arXiv:1805.09461*. [Online]. Available: https://arxiv.org/abs/1805.09461

[38] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The javadocminer," in *Proc. Int. Conf. Appl. Natural Lang. Inf. Syst.*, 2010, pp. 68–79.

[39] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[40] X. Li and X. Wu, "Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 4520–4524.

[41] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proc. 32nd AAAI Conf. Artif. Intell.*, Apr. 2018, pp. 1–27.

[42] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annu. Psychol. Program. Workshop*, 2006, pp. 53–67.

[43] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Workshop Text Summarization Branches Out, Post-Conf. Workshop ACL*, Barcelona, Spain, Jul. 2004. [Online]. Available: https://www.microsoft.com/en-us/research/publication/rouge-a-package-for-automatic-evaluation-of-summaries/

[44] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 373–384.

[45] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 125–135.

[46] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proc. 22nd Int. Conf. Program Comprehension*, Jun. 2014, pp. 279–290.

[47] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 103–119, Feb. 2016.

[48] V. Mnih, N. Heess, and A. Graves, "Recurrent models of visual attention," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 2204–2212.

[49] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, pp. 23–32, May 2013.

[50] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 230–232.

[51] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. AAAI*, vol. 2, Jan. 2016, p. 4.

[52] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *Proc. 51st Annu. Meeting Assoc. Comput. Linguistics*, vol. 2, 2013, pp. 35–40.

[53] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 883–909, Sep. 2016.

[54] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, Aug. 2013, pp. 532–542.

[55] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 574–584.

[56] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? An approach based on Genetic Algorithms," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 522–531.

[57] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Proc. 20th IEEE Int. Conf. Program Comprehension (ICPC)*, Jun. 2012, pp. 63–72.

[58] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. linguistics. Assoc. Comput. Linguistics*, Jul. 2002, pp. 311–318.

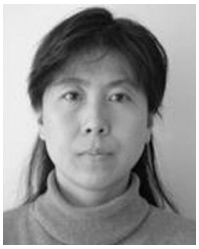
[59] S. Pinker, *The Language Instinct: The New Science of Language and Mind* (Penguin Books: Language and Linguistics). Penguin Adult, 1995. [Online]. Available: https://books.google.com/books?id=6KQ4ENWvEuAC

[60] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P . Reyes, M. L. Shyu, S. C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, p. 92, 2018.

[61] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *Proc. IEEE 15th Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2015, pp. 81–90.

[62] S. Rastkar, G. C. Murphy, and A. W. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2011, pp. 103–112.

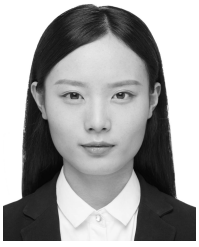
[63] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, May 2010, pp. 505–514.
[64] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and D. Sidney, "Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 390–401.
[65] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 255–265.
[66] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," 2015, *arXiv:1509.00685*. [Online]. Available: https://arxiv.org/abs/1509.00685
[67] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2010, pp. 43–52.
[68] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 101–110.
[69] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 71–80.
[70] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int. Conf. Program Comprehension (ICPC)*, May 2013, pp. 83–92.
[71] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[72] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," 2015, *arXiv:1503.00075*. [Online]. Available: https://arxiv.org/abs/1503.00075
[73] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: An experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, Sep. 1996.
[74] T. Tenny, "Procedures and comments vs. The banker's algorithm," *ACM SIGCSE Bull.*, vol. 17, no. 3, pp. 44–53, 1985.
[75] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.
[76] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories (MSR)*, Jun. 2018, pp. 542–553.
[77] C. Vassallo, S. Panichella, M. D. Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proc. 22nd Int. Conf. Program Comprehension*, Jun. 2014, pp. 106–109.
[78] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[79] R. Vedantam, C. L. Zitnick, and D. Parikh, "CIDEr: Consensus-based image description evaluation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 4566–4575.
[80] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 397–407.
[81] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 205–216.
[82] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2015, pp. 380–389.
[83] E. Wong, J. Yang, and L. Tan, "AutoComment: Mining question and answer sites for automatic comment generation," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 562–567.
[84] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proc. 5th Int. Conf. Softw. Eng.*, Mar. 1981, pp. 215–223.
[85] B. Yang, Z. Liping, and Z. Fengrong, "A survey on research of code comment," in *Proc. 3rd Int. Conf. Manage. Eng., Softw. Eng. Service Sci.*, Jan. 2019, pp. 45–51.
[86] A. T. Ying and M. P. Robillard, "Code fragment summarization," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 655–658.
[87] H. Yu, B. Li, P. Wang, D. Jia, and Y. Wang, "Source code comments quality assessment method based on aggregation of classification algorithms," *J. Comput. Appl.*, vol. 36, no. 12, pp. 3448–3453, 2016.
[88] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. 41st Int. Conf. Softw. Eng.*, May 2019, pp. 783–794.
[89] W. Zheng, H.-Y. Zhou, M. Li, and J. Wu, "Code attention: Translating code to comments by exploiting domain features," 2017, *arXiv:1709.07642*. [Online]. Available: https://arxiv.org/abs/1709.07642



XIAOTAO SONG received the B.S. degree in electronic information from the Taiyuan University of Technology, Taiyuan, China, in 1993, and the master's degree in computer technology and application from Shanxi University, in 2006. She is currently an Assistant Professor with the School of Software, Taiyuan University of Technology. Her main research interest includes intelligent software engineering.

HAILONG SUN received the B.S. degree in computer science from Beijing Jiaotong University, in 2001, and the Ph.D. degree in computer software and theory from Beihang University, Beijing, China, in 2008, where he is currently an Associate Professor with the School of Computer Science and Engineering. His research interests include intelligent software engineering, crowd intelligence/crowdsourcing, and distributed systems. He is a member of the ACM.

XU WANG received the B.Eng. and Ph.D. degrees in computer science from Beihang University, in 2008 and 2015, respectively, where he is currently an Assistant Professor with the School of Computer Science and Engineering. He was a Visiting Scholar with the Department of Computer Science, University of Chicago, from 2016 to 2017. His research interests focus on how to improve software development efficiency and software quality through AI techniques, program analysis, and algorithm optimization.



JIAFEI YAN received the B.Eng. degree from Xidian University, in 2016, and the M.Eng. degree from Beihang University, in 2019, both in computer science. She is currently an Research and Development Engineer with the Beijing Aeronautical Science and Technology Research Institute and the Beijing Key Laboratory of Civil Aircraft Design and Simulation Technology. Her research interests include profiling software developer for improving software development efficiency and software quality.