A Survey of Source Code Search: A 3-Dimensional Perspective

WEISONG SUN, Nanjing University, Nanjing, China and Nanyang Technological University, Singapore
CHUNRONG FANG*, Nanjing University, Nanjing, China
YIFEI GE, Nanjing University, Nanjing, China
YULING HU, Nanjing University, Nanjing, China
YUCHEN CHEN, Nanjing University, Nanjing, China
QUANJUN ZHANG, Nanjing University, Nanjing, China
XIUTING GE, Nanjing University, Nanjing, China
YANG LIU, Nanyang Technological University, Singapore
ZHENYU CHEN, Nanjing University, Nanjing, China

(Source) code search is widely concerned by software engineering researchers because it can improve the productivity and quality of software development. Given a functionality requirement usually described in a natural language sentence, a code search system can retrieve code snippets that satisfy the requirement from a large-scale code corpus, e.g., GitHub. To realize effective and efficient code search, many techniques have been proposed successively. These techniques improve code search performance mainly by optimizing three core components, including query understanding component, code understanding component, and query-code matching component. In this paper, we provide a 3-dimensional perspective survey for code search. Specifically, we categorize existing code search studies into query-end optimization techniques, code-end optimization techniques, and match-end optimization techniques according to the specific components they optimize. These optimization techniques are proposed to enhance the performance of specific components, and thus the overall performance of code search. Considering that each end can be optimized independently and contributes to the code search performance, we treat each end as a dimension. Therefore, this survey is 3-dimensional in nature, and it provides a comprehensive summary of each dimension in detail. To understand the research trends of the three dimensions in existing code search studies, we systematically review 68 relevant literatures. Different from existing code search surveys that only focus on the query end or code end or introduce various aspects shallowly (including codebase, evaluation metrics, modeling technique, etc.), our survey provides a more nuanced analysis and review of the evolution and development of the underlying techniques used in the three ends. Based on a systematic review and summary of existing work, we outline several open challenges and opportunities at the three ends that remain to be addressed in future work.

*Chunrong Fang is the corresponding author.

Authors' addresses: Weisong Sun, weisongsun@smail.nju.edu.cn, weisong.sun@ntu.edu.sg, Software Institute, Nanjing University, Nanjing, Jiangsu, China , School of Computer Science and Engineering and Nanyang Technological University, Singapore; Chunrong Fang, fangchunrong@nju.edu.cn, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Yifei Ge, gyf991213@126.com, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Yuling Hu, yulinghu@smail.nju.edu.cn, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Yuchen Chen, yuc.chen@outlook.com, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Quanjun Zhang, quanjun.zhang@smail.nju.edu.cn, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Xiuting Ge, dg20320002@smail.nju.edu.cn, Software Institute, Nanjing University, Nanjing, Jiangsu, China; Yang Liu, yangliu@ntu.edu.sg, School of Computer Science and Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; Zhenyu Chen, zychen@nju.edu.cn, Software Institute, Nanjing University, Nanjing, Jiangsu, China.

ACM 1049-331X/2024/4-ART
https://doi.org/10.1145/3656341

CCS Concepts: • **Software and its engineering → Search-based software engineering**.

Additional Key Words and Phrases: source code search, deep learning, query-end optimization, code-end optimization, match-end optimization

1 INTRODUCTION

Software development is usually a repetitive task, where the same or similar implementations exist in established open-source projects (e.g., GitHub repositories [31]) or online forums (e.g., Stack Overflow [50]). Software developers on average spend about 19 percent of their development time in searching the relative code over a large-scale codebase, because they can reuse or modify previously written code snippets to improve the efficiency of project development [6]. (Source) code search aims to retrieve relevant code in large-scale code corpora for a developer's given query requirements. Reusing the retrieved high-quality code can effectively improve the productivity and quality of software development, which makes code search receive widespread attention from software engineering researchers. Besides, code search provides more opportunities for code reuse, which is an earlier concept than code search [29, 104, 107]. Code reuse aims to modify an existing code into a new code according to the requirements, while code search makes it possible to find the available one. In other words, code reuse can be realized through code search, further proving the considerable research value and significance of code search.

The ideal code search tool can find the closest code snippets based on the arrived queries and return them [37, 71]. In fact, many developers currently rely on Google, Baidu, and other search engines for code search, but the search results are arrestingly unsatisfactory. The search logic or methods used by common search engines are not suitable for matching natural language and code pairs. In most cases, the search results are not available for developers to refer to, which will take them more time to change or choose a better query [9, 52, 82]. Given the importance of code search, there is great interest in implementing better methods for retrieving relevant code snippets from the code corpus, depending on the developer's intent expressed as a search query.

Code search can be narrowly defined as a technique that takes as input a natural language query given by a user, then selects the closest or possible answer from the code corpus and returns it to the developer as an output. Of course, there exist a few individual code search techniques that take some non-natural language queries as input or return the output into other forms such as API interface [76] or I/O examples [105]. In this paper, we mainly study the mainstream code search scenario, where the query is natural language text and the search results are method/function code snippets. By organizing and conducting a statistical analysis of articles on code search that fit our main study, we can get a glimpse of the development history of code search techniques. From the view of technical composition, both early and recent code search techniques are composed of three core components, including a query understanding component, a code understanding component, and a query-code matching component. The query understanding component and the code understanding component are responsible for mining and representing the features of the query and code snippet, respectively. The query-code matching component is responsible for ranking a set of candidate code snippets according to how closely their representations semantically match the query. The technical contributions of different code search studies mainly lie in the optimization of these three components.

In this survey, we propose a novel approach for summarizing code search techniques from a 3-dimensional perspective. Specifically, according to the specific components they optimize, we first classify existing code search techniques into three categories: query-end optimization techniques, code-end optimization techniques, and match-end optimization techniques. Each end exposes a perspective through which we can understand and dissect the essential optimizations/improvements made by existing code search techniques. Therefore, this survey investigates code search techniques from three perspectives, referred to as a 3-dimensional perspective.

Optimization techniques of the three perspectives are instrumental in enhancing the accuracy of query understanding, code understanding, and matching between queries and code snippets, thereby directly impacting the effectiveness and efficiency of code search. Then, we perform a systematic literature review of the optimization techniques proposed in the three ends. We carefully select 68 representative papers from the 1,427 candidates. Among these 44 papers propose solutions for query-end optimization, 50 papers for code-end optimization, and 52 for match-end optimization. Since some papers may propose optimization techniques for multiple ends, there will be some overlapping papers in the three ends. Finally, we tease out the development history of optimization techniques proposed for each end and summarize the development trends. Building on a thorough review of optimization techniques proposed in existing code search papers, we outline persistent challenges that necessitate further attention. Additionally, we present potential research opportunities in the code search field.

The main contributions of this survey include:

- We propose a novel systematic review of 2,191 code search techniques published in journal papers, conference papers, and arxiv papers up to September 30, 2023, as a starting point for future code search research.
- We analyze 68 different code search techniques and conduct analysis from a 3-dimensional perspective to explore these techniques' innovations in query-end optimization, code-end optimization, and match-end optimization. This survey will help subsequent researchers to enumerate their characteristics.
- We divide code search techniques into three categories according to the specific components they optimize, and analyze the evolution of techniques in each category over time as a basis for further comparison and benchmarking.
- We highlight opportunities and challenges in code search research based on our findings to stimulate further research in this field. Some resources, datasets, and code can be found at https://github.com/wssun/SourceCodeSearch.

1.1 Comparison with Existing Code Search Surveys

Around the 1960s, the concept of code search emerged [28, 57, 87]. Subsequently, with the development of open-source platforms, the number of papers related to code search has risen rapidly since 2009, and dozens of related papers appear in the field every year. Over the past nearly sixty years, many techniques have been successfully applied in code search, and there are also surveys and summaries of these techniques [33, 53, 66, 92]. These surveys either only focus on the query or code-end optimization or shallowly introduce various aspects of code search tools, including codebase, evaluation metrics, modeling techniques, etc. For example, Rahman et al. [92] explore the application of automatic composition query technology in code search, mainly reflected in the classification of algorithms, assessment of the quality of results and the future constraints and challenges of query reformulation. Another existing survey conducted by Liu et al. [66] focuses on analyzing the existing code search tools, discusses and learns from the specific search content of the tools, and proposes relevant indicators for evaluating code search tools. Luca et al. [33] discuss and summarize the code search process, including query processing, code indexing, search results sorting, and pruning. While their survey covers natural language queries and lists three techniques used to modify queries, they do not delve into the key optimization techniques for this type of query from different perspectives and their evolution over time, which are explained in detail in our survey. For the process of code, they introduce the artifacts that get indexed and information for indexing code according to the levels of code complexity. Moreover, they enumerate several techniques for comparing queries and code snippets and the ranking of search results, respectively, while we uniformly view comparison and ranking as a matching process. The latest survey completed by Kim et al. [53] provides an encompassing introduction to the task of code search, including the search base (i.e., a repository or dataset), benchmarks for code search, evaluation methods, etc.

However, a systematic review of various optimization techniques involved in different core components in code search techniques is still lacking. From the view of technical composition, most code search techniques are composed of three core components, including a query understanding component, a code understanding component, and a query-code matching component. The lack of a systematic review of the optimization techniques involved in these components hinders developers/researchers from identifying technology trends for each core component, as well as the challenges and opportunities faced in optimizing different components. To fill this gap, our paper focuses on the update and iteration of optimization techniques on these components. This facilitates subsequent researchers to optimize designs for specific components by assisting them in quickly finding comparative baselines.

Structure of the paper: The remainder of this survey is organized as follows. Section 2 briefly introduces the background of code search. Section 3 presents the survey methodology that we follow. Section 4, 5, and 6 summarize the key research questions we investigate and their answers in this study. Section 7 discusses the challenges for the road ahead on code search techniques and presents the potential research opportunities for future work. Section 8 shows the potential threats that may affect the validity of this review. Finally, Section 9 provides a conclusion of this survey.

2 BACKGROUND

In this section, we will introduce the background of code search, including code search definition and code search techniques.

2.1 Code Search

Since the existing works improve and implement the code search task using different techniques, there is no formal problem definition for the code search task. In this paper, we survey a wide range of code search research, including early information retrieval (IR)-based code search research and recent deep learning (DL)-based code search research. To the best of our knowledge, there is no strict definition of general code search. We investigate the widely-studied code search scenario, where the query given by the developer is a short natural language text, and the search result is a code snippet of a method/function [26, 34, 37, 65, 66, 99, 107]. Formally, let $q = \{w_1, w_2, \cdots, w_m\}$ be a query given by the developer, where w_i is the i-th word in q; $S = \{s_1, s_2, \cdots, s_n\}$ be a large-scale code corpus, where s is a code snippet; code search is defined as follows.

Definition 1 (Code Search). *Code search is the task of retrieving a code snippet $s \in S$ for q that satisfies the following conditions:*

- $\forall s' \in S, s' \neq s$
- $\boldsymbol{q} = \Phi(q)$
- $\boldsymbol{s} = \Psi(s), \boldsymbol{s'} = \Psi(s')$
- $sim(\boldsymbol{q}, \boldsymbol{s'}) \leq sim(\boldsymbol{q}, \boldsymbol{s})$ *or* $p(\boldsymbol{q}, \boldsymbol{s'}) \leq p(\boldsymbol{q}, \boldsymbol{s})$

where $\Phi(\cdot)$ and $\Psi(\cdot)$ are functions designed to represent features of query and code snippet, respectively. $sim(\cdot)$ is a function that measures the similarity between two feature representations of the query and code snippet. $p(\cdot)$ is a predictor function that predicts the classification probability that the query semantically matches the code snippet.

We focus on investigating free-text code search where queries given by developers are free-form natural language text and search results are method-level code snippets. This is the most practical, common, and widely researched code search application scenario [9, 66]. Figure 1 shows two examples of code search. In both examples, queries are natural language descriptions and code snippets are methods written in programming languages (e.g., Java and C/C++). From Figure 1(a) and (c), we can observe that the former (q_1) wants to find a code snippet calculating the factorial of a number, while the latter (q_2) aims to retrieve a code snippet implementing the calculate the factorial of a number.

(a) A Query q_1

```
public long factorial (int number) {
    long factorial = 0;
    int i = 1;
    for (; i <= number; i++) {
        factorial = factorial * i;
    }
    return factorial;
}
```

(b) A Java Code Snippet s_1

calculate how many digits an integer has.

(c) A Query q_2

```
unsigned int cnt_digits(unsigned int n) {
    unsigned int count = 0;
    while (n > 0) {
        n /= 10;
        count++;
    }
    return count;
}
```

(d) A C/C++ Code Snippet s_2

Fig. 1. Examples of code search

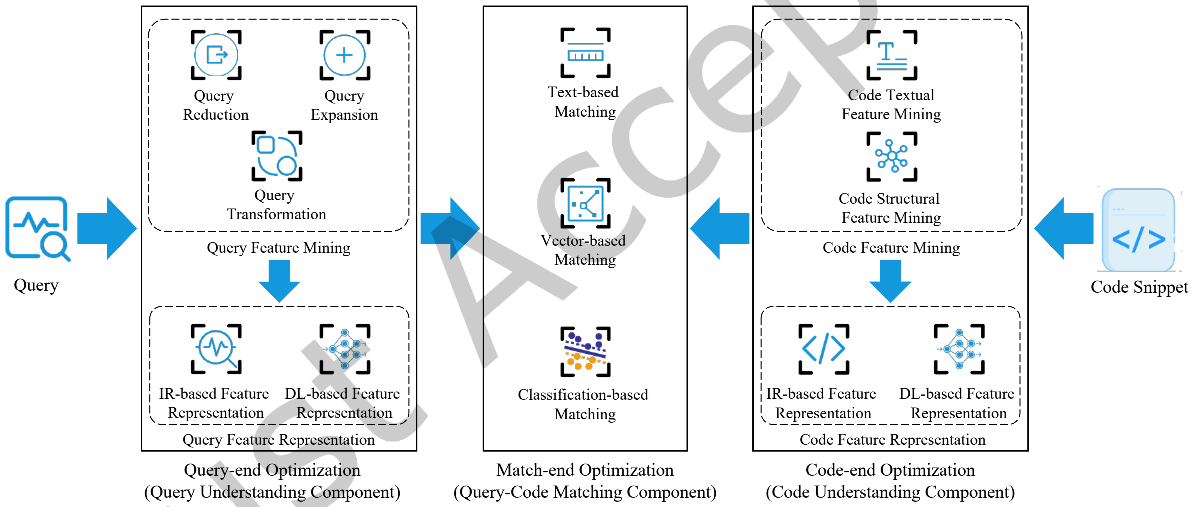

Query
Query Reduction
Query Expansion
Query Transformation
Query Feature Mining
IR-based Feature Representation
DL-based Feature Representation
Query Feature Representation
Query-end Optimization
(Query Understanding Component)
Text-based Matching
Vector-based Matching
Classification-based Matching
Match-end Optimization
(Query-Code Matching Component)
Code Textual Feature Mining
Code Structural Feature Mining
Code Feature Mining
IR-based Feature Representation
DL-based Feature Representation
Code Feature Representation
Code-end Optimization
(Code Understanding Component)
Code Snippet

Fig. 2. Overall framework of the code search technique

functionality of counting the number of digits in an integer value. Obviously, the two queries have different intents or requirements. The query understanding component is responsible for capturing the intents/semantics in the natural language query. Figure 1(b) and (d) show two code snippets. All code search systems require a code understanding component, which is responsible for capturing the semantics in the programming language code snippet. Intuitively, a good code search technique requires understanding the semantics of both the query and the code snippet. Only in this way, can it retrieve the code snippets that satisfy the query intents.

2.2 Code Search Techniques

Figure 2 shows the overall framework of the code search technique. The goal of the code search technique is to retrieve relevant code from a large-scale code corpus according to the intent of the developers' query. A typical code search system/tool contains three components: a query understanding component, a code understanding component, and a query-code matching component. The query understanding component is responsible for processing the natural language queries given by developers, such as mining and representing the key information (also called features/semantics) from the query. The code understanding component is responsible for processing the programming language code snippets in the code corpus, such as mining and representing features from the code snippets. The query-code matching component is responsible for ranking code snippets according to how well they semantically match the query. So, to implement effective and efficient code search, existing works have proposed various techniques to optimize these three components. These optimization techniques enhance the overall performance of code search by improving specific components. In this survey, according to the specific components they optimize, we divide the existing code search studies into the following three categories: *query-end optimization* techniques, *code-end optimization* techniques, and *match-end optimization* techniques.

Query-end Optimization. Given a query, query-end optimization aims to produce a query representation that not only preserves the core semantics of the query but also facilitates the computing of matching components, thereby improving the effectiveness and efficiency of code search. The query understanding component produces such a representation through two sequential steps: query feature mining and query feature representation. The query feature mining step treats the raw query given by the developer as input and extracts the important features from the raw query. Considering that the raw query may be of low quality and contains few useful features, existing works propose a variety of techniques to automatically optimize the quality of the features extracted from the raw queries [72, 73, 82, 92, 116]. In this survey, we investigate four classic techniques for optimizing query quality, including query reduction, query replacement, query expansion, and query transformation (details are described in Section 4.1). The query feature representation step takes in the features extracted by the query feature mining step and produces a semantic-preserving feature representation. Such a representation will be used to rank a large number of code snippets in the query-code matching component. From another perspective, query representation techniques are equally important. They also determine the final performance of the code search techniques, because accurate representation of semantic features in queries can assist code search models in retrieving code snippets correctly. Existing works propose many techniques to optimize the process of feature representation [96]. We divide the query representation methods utilized in existing technologies into two categories: IR-based feature representation and DL-based feature representation. IR-based representation methods always regard queries as vectors or plain texts, which can be utilized easily to optimize the query-end by the researchers. DL-based feature representation methods apply deep neural networks to encode the query. They are frequently adopted by existing code search techniques because they yield better semantic representations of queries. Details of query feature representation are described in Section 4.2.

Code-end Optimization. Given a code snippet, code-end optimization aims to produce a code representation that not only preserves the core semantics of the code but is also convenient for subsequent query-code matching computing, thereby improving the effectiveness and efficiency of code search. Like the query understanding component, the code understanding component produces such code representation through two sequential steps: code feature mining and code feature representation. The code feature mining step treats the raw code snippet in the code corpus as input and extracts its important features. Considering that the raw code snippet may be complex and contains noise features, existing works propose a variety of techniques to automatically optimize the quality of the features extracted from the raw code snippets [2, 9, 37, 64, 115, 121, 127, 128]. It is a common practice to characterize code features from two aspects: textual features and structural features. In this survey, we investigate three textual features and four structural features commonly used in code search works (details are described in Section 5.1). The code feature will be further passed into the code representation step to produce semantic-preserving feature representations. Such a representation will be used to rank a large number of code snippets in the query-code matching component. To make the resulting representation as semantically preserving as possible, existing works propose many techniques to optimize the process of code feature representation [96]. Like query feature representation, we also divide code feature representation techniques into two categories: IR-based feature representation and DL-based feature representation. IR-based feature representation treats the code primarily as text, supplemented by additional information to represent it more comprehensively. DL-based feature representation techniques apply deep neural networks to embed the code features. Details of code feature representation are described in Section 5.2.

Match-end Optimization. Given a query representation and a set of code representations, match-end optimization aims to rank code representations based on their relevance to the query representation. Different techniques adopt different methods to calculate the relevance scores. In this survey, we investigate three widely used matching methods, including text-based matching, vector distance-based matching, and classification-based matching. Text-based matching methods measure the relevance scores by calculating the distance between the keyword-based or IR-based query feature representation and the keyword-based or IR-based code feature representation. Vector-based matching methods measure the relevance scores by calculating the distance between the query feature vectors (including embeddings) and the code feature vectors. Vectors are produced by traditional IR techniques or advanced DL techniques. Classification-based matching methods measure the relevance scores by using neural network classifiers to predict the probability of semantic relevance of the query embeddings and the code embeddings.

3 SURVEY METHODOLOGY

We follow the guidelines for the systematic literature review (SLR) in software engineering [55, 86, 103] to conduct this survey. We start this survey by asking three research questions and then comprehensively analyze the various optimization techniques used in code search.

3.1 Research Questions

The research questions are one of the most important contents of the literature review, which guide us to clarify the research direction and thus conduct a purposeful investigation. In this survey, we want to explore, classify, and summarize various optimization techniques used in code search to date in query-end optimization, code-end optimization, and match-end optimization. These three dimensions are the core components of the code search techniques and the main perspectives for researching code search techniques in this survey. Therefore, we investigate the following three research questions (RQs):

- **RQ1.** What are the query-end optimization techniques in code search studies? The purpose of this RQ is to investigate which techniques are applied to query processing (including query feature mining and query feature representation) and the development trend of query-end optimization techniques.
- **RQ2.** What are the code-end optimization techniques in code search studies? This RQ aims to investigate which code features are used in code-end optimization techniques, how they represent involved code features, and the development trend of code-end optimization techniques.
- **RQ3.** What are the match-end optimization techniques in code search studies? The goal of this RQ is to investigate how match-end optimization techniques match the representations of the query and code features to rank the expected code snippet in a higher rank and the development trend of match-end optimization techniques.

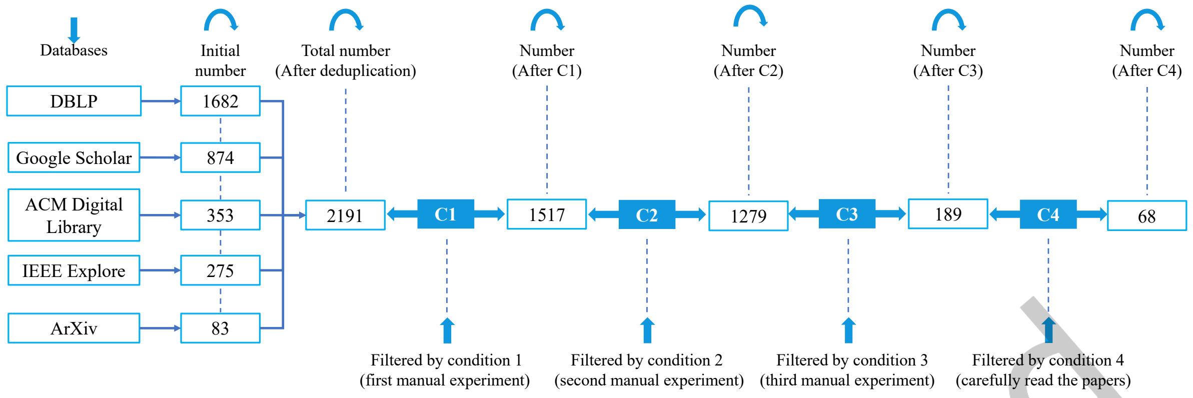

Databases
Initial number
Total number (After deduplication)
Number (After C1)
Number (After C2)
Number (After C3)
Number (After C4)
DBLP
Google Scholar
ACM Digital Library
IEEE Explore
ArXiv
1682
874
353
275
83
2191
C1
1517
C2
1279
C3
189
C4
68
Filtered by condition 1 (first manual experiment)
Filtered by condition 2 (second manual experiment)
Filtered by condition 3 (third manual experiment)
Filtered by condition 4 (carefully read the papers)

Fig. 3. Selection of research studies

In the process of seeking the answers to these questions, we also find the deficiencies and limitations of the existing code search techniques. As a reference, we put forward some suggestions and directions for future research.

3.2 Search Strategy

Collecting published papers requires selecting appropriate publication databases and searching keywords. In this survey, we selected six widely available electronic databases, including DBLP publication database [1], Google scholar database [2], IEEE Explore database [3], ACM Digital Library [4], Web of Science database [5], and ArXiv database [6]. It is worth noting that, considering that some researchers are willing to disclose the latest research techniques on ArXiv in advance, we also collected the code search papers that have been made public on ArXiv but not accepted by any journal or conference. In terms of selecting searching keywords, we referred to the **PIO** (Population + Intervention + Outcome) criteria [56] and used the population terms and intervention terms from code search field to construct keywords, as in previous studies [92]. **Population terms** covered all aspects of the research topic. In this survey, we used "Code Search", "Code Retrieval", "Code Recommendation", and "Code Reuse" as the population terms. **Intervention terms** focus on a specific aspect of the research topic. We used "Query Expansion", "Query Reduction", and "Query Transformation" as the intervention terms for RQ1. Relevant papers up to September 30, 2023, were included in our search results. As Figure 3 shows, we retrieved 2,191 relevant papers from six databases in total as **Outcome** after removing duplicated studies.

3.3 Study Selection

Once those candidate studies were collected, we made a gradual selection according to the filter conditions (**C**) we carefully set up.

- **C1.** eliminating book, thesis, and short papers (our definition of short essay papers: less than five pages); only needing journals and conferences.

[1]https://dblp.uni-trier.de
[2]https://scholar.google.com/
[3]https://ieeexplore.ieee.org/
[4]https://dl.acm.org/
[5]https://www.webofscience.com/
[6]https://arxiv.org/

- **C2.** only selecting technical papers, excluding technical reports, empirical studies, and surveys.
- **C3.** choosing the papers about the field of code search:
  - removing papers in which queries are not in natural language;
  - ruling out papers that propose techniques for local code search (e.g., feature location), as they aim to find relevant locations in a single specific software repository rather than retrieving code snippets from large-scale code corpora;
  - eliminating researches that employ code search engines to improve the performance of other research areas, such as code clone detection.
- **C4.** choosing the papers that meet the requirements of having a technical innovation in one of the ends.
  - query-end: proposing new query processing techniques that help preserve or augment the semantics of queries, including query feature mining and query feature representation techniques;
  - code-end: mining new code features, exploring new methods to extract code features, or proposing innovative approaches to optimize the representation of code features;
  - match-end: introducing novel methods of matching queries and code snippets to sort code snippets more efficiently or optimizing the matching process (e.g., speeding up the matching).

According to the four conditions, manual experiments were carried out to select the final range from the candidate papers. The first manual experiment obtained the page number and published source of the whole paper. We obtained a total of 1,427 papers (including 708 conference papers, 645 journal papers, and 74 arxiv papers) according to the requirements of Condition **C1**. Then, we invited 8 volunteers with strong English proficiency and software engineering research experience to conduct the subsequent manual experiments. In the second manual experiment, they judged whether the remaining papers meet Condition **C2** and retained 1,279 technical papers, excluding 192 empirical studies, 25 surveys, and 21 technical reports. In the third manual experiment, each volunteer checked 178 ~179 papers to determine whether they meet Condition **C3** mainly based on the title and abstract parts. After this step, 189 papers were remained. Finally, we carefully read these papers to assess whether they meet the requirements of **C4** and selected 68 representative papers. Our systematic filter conditions ensure the quality and representativeness of the papers selected. The selected 68 papers have fully covered relevant research on the three ends.

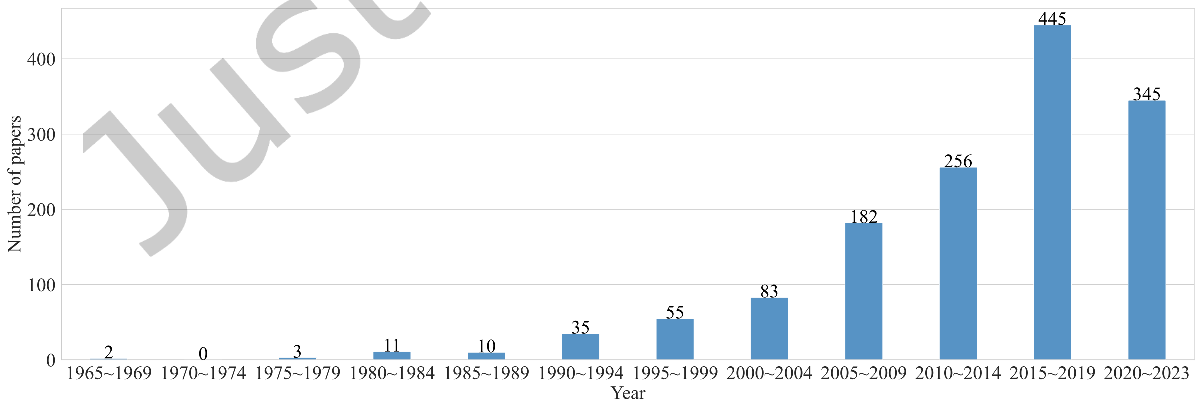
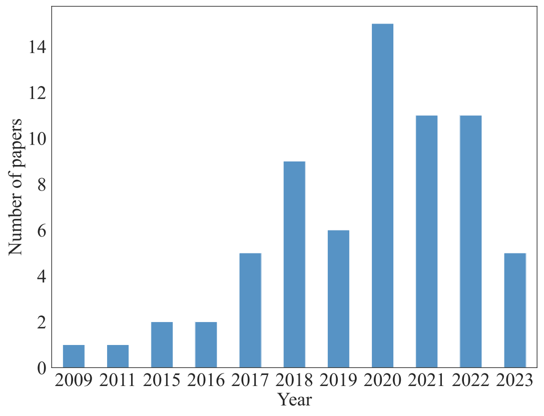
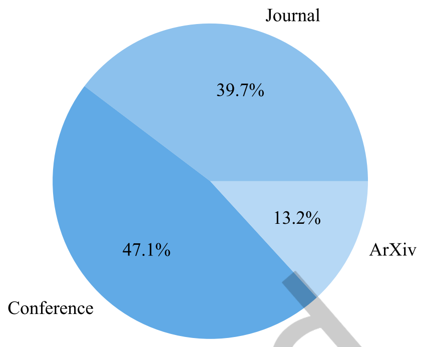


Fig. 4. 1,427 code search papers from 1965 to 2023



(a) Group by Year



(b) Group by Venue

Fig. 5. 68 papers on code search discussed in this article

We grouped the collected 1,427 papers by year of publication, and the statistical results are shown in Figure 4. It is observed that the number of relevant papers has increased significantly since 2005, indicating that the problem of code search has received significant attention.

After screening all the conditions, we finally identified a total of code search studies related to our research. For this study, we carefully selected representative papers from the candidates and conducted a systematic analysis, and finally, 68 studies are selected. According to the three questions we raised above, these papers are divided into three relevant categories, among which 44 papers are related to RQ1; 50 papers are related to RQ2; and 52 papers are related to RQ3. Since some papers may propose optimization techniques for multiple ends, there will be some overlapping papers in the three RQs. It should be noted that we will focus on introducing the optimization techniques proposed by each paper on the corresponding end in different RQs. Figure 5(a) presents the number of papers we discuss per year of publication, illustrating the increasing relevance of the topic. Figure 5(b) shows that 47.1% of them were published in conference proceedings, accounting for the largest proportion, followed by Journal (39.7%) and ArXiv (13.2%).

4 ANSWERING RQ1: WHAT ARE THE QUERY-END OPTIMIZATION METHODS IN CODE SEARCH STUDIES?

In the realm of code search, the quality of a query is paramount as it directly influences the quality of the search results [93, 94]. A low-quality query, characterized by the use of uncommon abbreviations or redundant and noisy phrases, may result in the retrieval of suboptimal code snippets. As a countermeasure, researchers have progressively proposed a variety of optimization techniques aimed at optimizing and understanding user queries. These endeavors collectively serve to enhance the performance of code search. In a nutshell, query-end optimization endeavors to elevate the quality of user queries, consequently heightening the quality of code search.

As shown in Figure 2, the optimization techniques employed at the query end can be mainly categorized into two facets: feature mining from the user query and subsequent representation of these extracted query features. For query feature mining, existing efforts predominantly fall within three categories: query reduction, query expansion, and query transformation. Query reduction improves the quality of the user query by eliminating

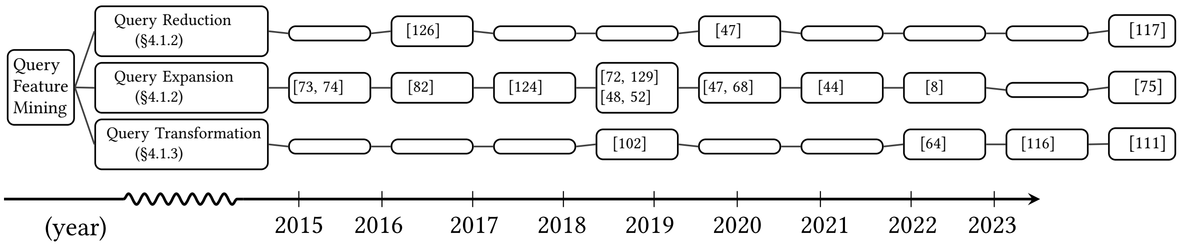


Fig. 6. Evolution of feature mining techniques in query end

redundant content [68]. Query expansion is to enrich the user query by incorporating available information, such as software-specific expansion words sourced from Stack Overflow [73, 82]. Query transformation first transforms the user query into an alternative form, such as API [90], and then retrieves related code snippets using the transformed form. Figure 6 lists some representative works of the three types of query feature mining techniques at various temporal junctures. More details of these techniques are discussed in Section 4.1. As for query feature representation, existing endeavors are generally classified into two classes: IR-based methods and DL-based methods. The main difference between the two lies in the adoption of different techniques to embed the natural language query and generate the corresponding vector representations. Figure 7 presents some notable works of the two categories of query feature representation techniques across different points in time. More details of query feature representation are discussed in Section 4.2. In the following subsections, we will intricately introduce the three types of query feature mining techniques and the two categories of query feature representation techniques aforementioned in detail. Additionally, we will succinctly encapsulate their trajectories of development and emerging trends.

4.1 Query Feature Mining

4.1.1 Query Reduction. As mentioned earlier, query reduction strives to eliminate superfluous components from the query, which contributes to the code search technique to grasp the core intent of the query accurately, thereby enhancing performance. Existing studies [40, 91] find that the keywords that occur in more than 25% of the documents in a corpus are less discriminating. Thereby, the key of query reduction is to remove the redundant, noisy, ambiguous, or less discriminating keywords [92], making the results of code search more reasonable and suitable for users to reuse. The first line of Figure 6 shows the code search papers that employ query reduction to reformulate user queries. It can be seen intuitively that query reduction is less common for code search. In the following, we will detail how these two code search works perform query reduction.

Yu et al. [126] propose an information retrieval-based technique called APIBook to help users search the code of API methods. Given a user query also called "API description" in their paper, APIBook extracts semantic information and type information from the query. The extracted information will be used for matching later. The semantic information of an API description refers to the meaning of words in the API description. They use nouns, verbs, and adjectives as the semantic information of the API description. The type information of an API description refers to the information that concerns types, such as "String" and "StringBuilder". In simple terms, they perform query reduction by removing content other than semantic and type information, such as stop words, prepositions, and adverbs.

Huang et al. [47] propose a deep learning-based method called QESC for effective query expansion. Although only query expansion is emphasized in the title of the paper, QESC also performs query reduction. Therefore, it can be said that QESC employs a combination of query expansion and query reduction to optimize query feature mining. In this section, we focus on how QESC performs query reduction, and the content of query expansion is introduced in Section 4.1.2. Specifically, they decompose the code search into two steps: first-pass retrieval and second-pass retrieval. In the first-pass retrieval, on receiving a query, the search engine produces the initial query results. They train an inference model based on the Deep Belief Network [42], which can infer the fine-grained changed code terms that will most likely occur in the initial results. They represent a code term by a triplet of $(\langle label\rangle, \langle role\rangle, \langle operation\rangle)$. $\langle label\rangle$ represents the textual information of Abstract Syntax Tree (AST) nodes. $\langle role\rangle$ has two options that decide if a term is changed or dependent. $\langle operation\rangle$ has three operations that decide if a term is unchanged, new, or deleted. In the second-pass retrieval, they reformulate the query with the selected changed terms. If the $\langle operation\rangle$ of the term is "deleted", they see it as an irrelevant term and remove its $\langle label\rangle$ from the query. In short, QESC performs query reduction by deleting irrelevant terms identified by the inference model.

Wang et al. [117] find the existing code search tools usually return a ranked list of candidate code snippets without any explanations, making the developers often find it hard to choose the desired results and build confidence on them. To address this issue, they propose XCoS, an explainable code search approach based on query scoping and knowledge graph. Query scoping essentially performs the process of query reduction, aiming to extract different parts from a query, including functionalities, functional constraints, and nonfunctional constraints. Specifically, given a query, XCoS first removes the starting words for a question, such as "how to" and "how can I". Then XCoS uses an NLP tool spaCy to analyze the part of speech and dependence tree of the remaining query. After that XCoS extracts different parts from the query using linguistic rules. The linguistic rules for the code search query are borrowed from the NLP field where linguistic rules are commonly used to extract functionality and constraints [49, 85, 113]. To build linguistic rules, they randomly sample 50 question titles as a validation dataset to iteratively refine and validate the rules by observing the extraction results. In the rules, VERB, DOBJ, PREP, POBJ, and MOD denote verb, direct object, preposition, preposition object, and modifier, respectively. Based on these rules, they summarize that *Functionality: VERB or VERB DOBJ; Functional Constraint: PREP POBJ, to VERB DOBJ, or using DOBJ; Nonfunctional Constraint: adverbs, adverb clauses, relative clauses, or phrases such as in MOD way that are used to modify or qualify the functionalities or functional constraints.* XCoS composes the functionality part and functional constraint part together to retrieve code snippets. The knowledge graph is used to guide the generation of explanations for code snippets and its construction is an offline task. In practice, given a query, XCoS first identifies different parts (i.e., functionalities, functional constraints, nonfunctional constraints) from it and uses the expressions of functionalities and functional constraints to search the codebase. It then links both the query and the candidate code snippets to the concepts in the knowledge graph and generates explanations based on the association paths between these two parts of concepts together with relevant descriptions.

In summary, the advantage of query reduction lies in eliminating noise, redundant parts, and some ambiguous keywords in the query. Query reduction ensures that the remaining keywords make the search results more accurate and reasonable. However, if query reduction is not precise enough, it may result in the loss of some of the query's intent. Therefore, ineffective query reduction might have a negative impact on the results.

4.1.2 *Query Expansion.*

Query expansion is to expand the user's original query through the inclusion of other available information, such as synonymous words. This approach serves as a strategy for reformulating queries, particularly in cases where the original query yields a poor retrieval result. Originally, in the field of information retrieval, query expansion was used for cross-language information retrieval [10], the words/terms suggested by query expansion techniques can rich retrieval code results. Given the effective promotion of search performance, the researchers in the software engineering community introduce them to code search tasks. The second line Figure 6 presents the code search papers that leverage query expansion to enrich user queries. It is apparent that, compared with query reduction, query expansion has garnered extensive attention and application within code search. Subsequently, we will discuss how these code search endeavors conduct query expansion.

Query expansion from API. Utilizing API to expand queries can enhance the semantic understanding of queries, providing more accurate search results. Furthermore, it also increases the possibility of finding reusable APIs.

Lv et al. [74] find that one major limitation of existing code search tools is the lack of query understanding. These tools often adopt conventional text similarity matching techniques to retrieve relevant code snippets. They do not consider query understanding, which could lead to inaccurate return results. Therefore, they propose CodeHow, a code search approach that considers both API understanding and text similarity matching. CodeHow understands a query by identifying the APIs that the query may refer to. CodeHow expands the user query with the identified APIs and applies the Extended Boolean model to retrieve the code snippets that match the expanded query. Specifically, CodeHow decomposes code search into two phases, i.e., the API understanding phase and the code retrieval phase. In the API understanding phase, CodeHow first collects the description of each API in the API library from its online documentation. It then calculates two similarity values, one between the API description and the query, and one between the API name and the query. Finally, it returns the potentially relevant APIs that match the query according to two similarity values. In the code retrieval phase, CodeHow constructs a Boolean query expression for retrieving code snippets that match the query in terms of text similarity. It retrieves code snippets that contain the potentially relevant API as well as other query terms in the method body and method name. CodeHow also constructs Boolean query expressions for each API recommended in the API understanding phase, which is intended to search for code snippets that contain the potentially relevant API. The query expressions above are combined to obtain an expanded query expression for retrieving code snippets. Finally, the expanded query expression is passed to an Extended Boolean Model as the input, which will return relevant code snippets according to their similarity to the expanded query expression.

Zhang et al. [129] find that the proportion of identifiers (e.g., class and method names) plays a key role in retrieving relevant code examples from code search engines. Therefore, they propose to perform query expansion by recommending semantically related identifiers (particularly Application Program Interface (API) class-names) to expand natural-language queries. Specifically, they first use the continuous bag-of-words model (CBOW) [79] to extract vector representations that are used to map natural-language queries with identifiers. Then based on similarities between vector representations of the query and API class-names, they find relevant API class-names (identifiers) from the corpus to expand the given query. Finally, they retrieve code snippets from the corpus by executing the expanded query.

Query expansion from Question & Answer (Q&A). Q&A not only includes highly precise answers but also questions posed by developers using natural language. This format is highly suitable to improve the performance of query expansion.

To overcome the term mismatch problem inherent in text retrieval-based techniques, Nie et al. [82] propose Query Expansion based on Crowd Knowledge (QECK) to improve the performance of code search. Specifically, given a query, QECK ranks all Question & Answer (Q&A) pairs collected from Stack Overflow [7] using the information retrieval model Lucene [8]. The top-m Q&A pairs are identified as the Pseudo Relevance Feedback (PRF) documents, which will be treated as relevant to the query. Then, QECK identifies useful expansion words from PRF Q&A pairs. Each word in PRF Q&A pairs is assigned an expansion weight. According to the weights of words, top-n words are selected as useful expansion words and added to the original query to generate the expanded queries. Finally, QECK ranks all code snippets in the corpus for the expanded query, and the top k code snippets are recommended to developers as the search results.

Q&A from programming forums that contain abundant exchanging knowledge about programming issues are crucial resources for code retrieval and annotation. However, Hu et al. [44] find that mining software repositories

[7]https://stackoverflow.com/
[8]http://lucene.apache.org

in such open and unrestricted forums is challenging, since the posts can be arbitrary and noisy. To overcome this challenge, they propose Code-Description Mining Framework (CodeMF), an unsupervised framework to eliminate noisy posts and extract high-quality software repositories from programming forums. Specifically, CodeMF is the combination of two proposed frameworks kernel principal component analysis (KPCA) and wavelet time-frequency transform feature fusion (WTFF) which are applied in extracting high-quality software repositories. KPCA is the principal component analysis tool used to reduce the dimension of features collected from the software repositories, and then further extract high-quality mappings between query-code pairs. Regarding the novel framework WTFF, it transforms the multiple dimension features into a time-frequency domain through wavelet transformation to reduce the computational complexity. This helps to extract the principal components of the software repository features more easily. Finally, CodeMF leverages the QECK [82] to retrieve the code snippets. As we introduced above, QECK uses the Q&A context-text pairs to expand the NL queries. CodeMF enhances the Q&A Pairs Search Engine which is a crucial component of QECK. Thus, it offers high-quality expanded keywords, leading to improved code search performance.

Query expansion from Source Code. Utilizing source code to expand queries is one of the most common expansion strategies. The source code contains a wealth of code context, including variable names, function names, comments, and other information. Using this information for expansion can significantly enhance the performance of code search.

Code tokenization is a key preprocessing step in code search techniques, which aims to convert query or code snippets into lexicons. Karnalim [52] find that most code search techniques rely on programming-language-dependent features to extract source code lexicons. However, these techniques would require manual updates to accommodate new programming languages, a process that can consume a significant amount of time. To handle this issue, Karnalim proposes a language-agnostic code retrieval approach. It does not rely on programming-language-dependent features. Instead, it relies on the Keyword & Identifier lexical pattern which are typically similar across various programming languages. The recognized lexicons are classified by lexicon categorization based on Keyword & Identifier lexical pattern. This lexical pattern is selected as the main concern, because its rules are similar in most programming languages. This pattern is also adapted to query expansion. Specifically, the keyphrases found in the most descriptive paragraph are regarded as the query expansion candidates. They are selected from top-K retrieved documents and limited by lexical pattern. A lexicon is only viewed as a candidate if its category is similar to the category of the query term, either a keyword-like or identifier-like lexicon. After that, the candidates are sorted by the importance score which is weighted by term frequency and one-to-many association. Finally, the query expansion candidates are used to improve the code search performance, which expands the query based on terms found in the top k retrieved source codes.

Lu et al. [72] find that formulating (e.g., exchanging, adding, and deleting) the related words identified based only on the positional proximity is not enough to optimize the query. The presence of word relations prevalent within the source code, such as compound words and synonyms, is useful for query reformulation. Therefore, they propose a novel method called INQRES to leverage these word relations to expand the query interactively. Experiments show that the performance of reconstructing queries through expansion by INQRES is very effective, far surpassing the expansion methods proposed by Lu et al. [73] Specifically, given a query, INQRES first extracts meaningful keywords from it. Then INQRES expands the keywords based on identifiers from the source code, meaningful words from the comments, and synonyms from the WordNet thesaurus. To find related expansion words from the three sources, INQRES excavates five word relations in the source code, including the Inheritance Relation (InR), Implementation Relation (ImR), Synonym Relation (SynR), Same-word Relation (SamR) and Compound Relation (ComR). All extracted word relations are saved in a word set called word-relation library (WRLib). Then, WRLib is used to extend the related words in the search query. If some words in the query also occur in WRLib, the related words are recommended, which are annotated by the relation level (SynR-1, SynR-SamR-2, etc.), which consists of the type of word relations and the grade of the relation between the recommended words and the query words. All extended related words are sorted based on the grade of relation level and their frequency used in the source code. After ranking all the related words, they further demonstrate them using the "AND" or "OR" relation for developers to understand and select. Inspired by code search engines where the developers often use the "advanced search" to optimize the query results, that is, query words are combined with the "AND" and "OR" relations, INQRES builds "AND" and "OR" relations in an interactive way for the developer to select suitable words for query expansion. SynR and SamR indicate similar relations between words and thus the words in these two sets are defined as the "OR" relation. The meanings of ComR, InR, and ImR are complementary, and thus the words in these three sets are defined as the "AND" relation. In INQRES, all related words are shown in an interface. Users are empowered to evaluate the relevance of these words to the original query. Those deemed relevant are recognized as valid and subsequently incorporated into the original query. INQRES can iteratively use the expanded query to identify other effective related words in a similar way, until the search results are satisfied.

Yang et al. [124] discover a significant issue where code search results are often modified manually. This phenomenon is caused by the inability to predict intent accurately with code search tools. To address this problem, they propose an intent-enforced code search approach called IECS. IECS can predict potential intents for a query before performing code retrieval. It utilizes intent to enhance the search to meet user needs. Specifically, IECS extracts intents from the given query by the intent extraction algorithm. This algorithm first uses the AST to identify modifications from the past method records. It then records the mapper between the identifiers and the concrete instances according to the modifications. After that, IECS can extract the intents from the mapper. Finally, it expands the query with intents and applies the Extended Boolean Model to retrieve the relevant code without any subsequent modification. Experiment results show that IECS performs well when performed to code search tools, with a 28.5% increase in the precision of the first returned results compared to CodeHow [74].

Huang et al. [47] find a potential issue where expanded queries may inadvertently include irrelevant terms. This phenomenon, known as the "overexpansion problem", can lead to confusion within the search engine and subsequently result in worse outcomes. To avoid the overexpansion problem, they propose a novel query expansion algorithm based on the semantics of change sequences, named QESC. Change sequences are generated from the commits of each method. They contain the changed terms (the new or the deleted code terms) as well as dependent terms (the unchanged code terms). As we mentioned earlier, DBN is trained on the change sequences. Then, QESC uses the DBN to generate changed terms which are composed of operation, label, and role. If the operation of the term is new, it means the term is relevant to the original query, promoting QESC to expand the query with its label. According to the obtained changed terms, QESC guarantees the expanded terms are relevant to the original query, and second-pass retrieval with an expanded query will perform better. Furthermore, compared with two other recent code expansion methods, CodeHow [74] and QECK [82], QESC also generally have better performances.

Query expansion from Model Inference. Utilizing DL models to assist in query expansion is a very promising approach. Among these methods, some directly use the models to infer the content of the expansion, which has been proven to enhance the performance of code search.

NCS [96] is a useful code search tool that can correctly search repositories of existing source code for code snippets. However, Liu et al. [68] find that the performance of NCS regresses with shorter queries. To address this issue, they explore an additional way of using neural networks in code search. They develop NQE, a neural model that takes in a set of keywords and predicts a set of keywords to expand the query to NCS. NQE with NCS can perform better than using NCS alone. Specifically, NQE is an encoder-decoder model, given a query as input, which outputs the most likely sets of expanded keywords. NQE learns to predict keywords that co-occur with the query keywords in the underlying corpus, which helps productively expand the query. Besides, beam search is also utilized to obtain the top-k most likely sequences of method names. This enhances the performance of NQE in finding the most relevant keywords for expanding the query.

QECC (Query Expansion with Code Changes) [48] applies a similar approach to QESC, improving its ability to infer expansion words. Similar to QESC, QECC also extracts (changes, contexts) pairs from the abstract syntax trees (ASTs) of changed methods, which are used to detect changes and extract contexts. However, the difference between the two lies in the output of the inference model, the association-based inference model trained by QECC can directly infer the suggested words to expand the query. Precisely, upon receiving a query, QECC retrieves initial results from the code corpus. Subsequently, the model extracts the contexts of these initial results and deduces potential expansion words by assessing the basis of initial results. Ultimately, the model constructs an expanded query by incorporating these expansion words into the original query. This expanded query is then utilized to execute a final search within the code corpus.

Query reformulation is a widely utilized technology that can be regarded as similar to query expansion for enriching user requirements and enhancing the outcomes of code search. However, Mao et al. [75] find that training a query reformulation model requires a large parallel corpus of query pairs (i.e., the original query and a reformulated query) that are confidential and not publicly available. This restricts the practicality of query reformulation in software development processes. Therefore, they propose SSQR, a self-supervised query reformulation method that does not rely on any parallel query corpus. Specifically, SSQR treats query reformulation as a masked language modeling task conducted on an extensive unannotated corpus of queries. SSQR extends T5 [89] (a sequence-to-sequence model-based on Transformer) with a new pre-training objective named corrupted query completion (CQC), which randomly masks words within a complete query and trains T5 to predict the masked content. Subsequently, for a given query to be reformulated, SSQR identifies potential locations for expansion and leverages the pre-trained T5 model to generate the appropriate content to fill these gaps. In this way, SSQR enhances the performance of code search from the unsupervised query reformulation. Compared to baseline expansion method proposed by Lu et al. [73], SSQR brings a giant leap of over 50% in search accuracy.

Query expansion from Other Sources. There are also other sources for query expansion that can improve the accuracy of code search.

Lu et al. [73] propose an approach for expanding queries using WordNet to generate synonyms. This approach ensures that the query can accurately search for code snippets with similar semantics (synonyms). Specifically, after preprocessing the query text, this approach first uses a Part-of-Speech Tagger (POS Tagger) to determine the part of speech of each word in the query. Then, it uses WordNet [61] to determine synonyms for each word to expand the original query. Since WordNet may return some inappropriate synonyms, this approach only returns synonyms from WordNet that have the same part of speech as the words in the query as the expanded word set. Finally, the expanded query is matched with identifiers extracted from the code snippets, and the results are sorted based on the similarity of the matches.

Cai et al. [8] find the retrieved source code from the existing code search techniques is not compatible with local programming language since the evolution and production of multiple versions of libraries. To solve this issue, they propose DCSE, a deep code search model based on evolving information. Specifically, DCSE first deeply excavates evolved code tokens and evolution descriptions in the code evolution process. It then treats evolved code tokens and evolution descriptions as one feature of source code and code descriptions, respectively. Therefore, DCSE can retrieve the source code that is compatible with the local programming language. DCSE embeds source code and its code descriptions into a high-dimensional shared vector space. It retrieves the initial result using the given query from the repositories. If the initial search result is incompatible with the local programming language, users could add the error report of IDE to query for a second-time search. This is due to the error report being semantically related to the evolution description while the evolution description is one feature of code description, so the distance between the expanded query vector and the compatible source code vector will be closer. Finally, DCSE can leverage the expanded query to retrieve the compatible source code. As demonstrated by the experimental results, DCSE outperforms QESC [47] by 9–12% in precision and 12% in mean reciprocal rank.

In summary, the advantage of query expansion methods lies in their ability to reconstruct queries by adding content, which has a good chance of covering the query's intent and serving as an effective supplement to the query. However, one limitation of query expansion lies in the availability/accessibility of expansion content. Additionally, increasing the expansion content introduces additional query overhead, making it challenging to limit the number of expansions and control the quality of expansion content.

4.1.3 Query Transformation. Considering the gap in syntax and structure between the query in natural language and the code snippet in programming language, there are naturally some obstacles if directly matched between them. To address this issue, some researchers believe that finding an intermediate pattern to bridge this gap holds promise as a viable solution. They successively propose some query transformation techniques, which can convert natural language queries into other forms or augment the queries with those forms, such as Q&A posts and other customized forms [102, 116]. In short, query transformation turns the direct matching problem between query and code into an indirect one. From the third line of Figure 6, it is observed that the research enthusiasm for both query transformation and query reduction is comparable but notably lower than that observed for query expansion. The subsequent sections will delve into a detailed discussion of how these works conduct the process of query transformation.

Sirres et al. [102] find a significant issue where source code terms such as method names and variable types are often different from conceptual words mentioned in a query. This is called a mismatch problem which occurs from the poorly documented or non-explicit names of source code. To reduce this mismatch problem, they present COCABU, a novel approach that leverages common developer questions and the associated expert answers to transform the augmented queries from the original user queries according to the relevant Q&A sites. Specifically, in the first step of COCABU, the search proxy takes an original query as input and returns a set of relevant posts collected from developer Q&A sites as an output. The obtained Q&A posts are used to find out how natural language concepts can be translated into program elements, that is, to collect potential translation rules. These translation rules can alleviate the vocabulary mismatch problem between user queries and source code. To transform the original query into the augmented query, the code query generator module extracts structural code entities from code snippets in Q&A sites. Besides, the code query generator only considers the accepted answers in Q&A sites. Thereafter, based on the Lucene search engine, COCABU preserves the terms from search results and combines them with the types of structural code entities collected from Q&A sites (e.g., unqualified/partially qualified method invocations or classes) to form the augmented query. Finally, the code search engine takes an augmented query produced by the code query generator and provides a list of search results to the developer.

Ling et al. [64] find most code search techniques ignore the deep structured features when processing both queries and code snippets. To address this problem, they propose an end-to-end deep graph matching and searching (DGMS) model based on graph neural networks for the task of semantic code retrieval. Considering the rich, important semantic structure information within the queries and code snippets, DGMS builds the graph-structured data to represent that structure information. Specifically, for the natural language query text, DGMS builds the text graph based on the constituency parse tree [23] and word ordering features. These features provide both constituent and ordering information of sentences to establish the graph-structured data. In a nutshell, DGMS performs query transformation by converting the query text into a text graph. For code snippets, DGMS also generates corresponding code graphs. After transforming both queries and code snippets into unified graph-structured data, DGMS uses the proposed graph matching and searching model to retrieve the best matching code snippet.

Wang et al. [116] find that the user query is relatively shorter than the code description (also known as code comments) and limited in context. It implies a knowledge gap between the query and the code description.

The code description contains more semantic keywords like code snippets rather than the query. However, existing research ignores this gap, resulting in low code search accuracy for code search models trained based on code descriptions rather than real queries. To reduce the impact of the knowledge gap, Wang et al. propose a query-enriched code search model called QueCos. QueCos performs query transformation by generating the corresponding code descriptions from the given query. Those descriptions are utilized for improving the code search performance. Specifically, QueCos collects the code-description pairs dataset from GitHub. The designed crawler saves the code snippets referred in the Stack Overflow posts and the corresponding code descriptions. Then, a query semantic enriching model is designed to generate the corresponding descriptions for queries based on the collected dataset, during which reinforcement learning is adopted to enable the code snippets retrieved by the generated descriptions to be ranked higher. The generated descriptions are treated as semantically enriched queries and not necessarily to be exactly close to the ground-truth descriptions. Finally, both the semantically enriched queries and original queries are employed for the ultimate code search.

Existing code search techniques [25, 32, 34] have primarily relied on intricate matching and attention-based mechanisms. However, Tang et al. [111] find that those techniques often lead to computational and memory inefficiencies, posing a significant challenge to their real-world applicability. To tackle this challenge, they propose a novel technique, the Hyperbolic Code QA Matching (HyCoQA). HyCoQA leverages the unique properties of Hyperbolic space to express connections between code snippets and their corresponding queries. Specifically, HyCoQA transforms the code search task into a Q&A pair matching paradigm. It constructs a dataset with triple matches characterized as '⟨negative code, description, positive code⟩'. In this case, the primary objective is to maximize the margin between the scores of the correct Q&A pair and the negative Q&A pair, ensuring that the retrieve system can robustly differentiate between accurate and inaccurate solutions based on the given description. Thus, these triple matches are subsequently processed via a static BERT embedding layer, yielding initial embeddings. A novel mathematical concept, hyperbolic geometry is used by HyCoQA. Unlike traditional Euclidean spaces, hyperbolic spaces excel at depicting hierarchical structures, which often underlie the relationship between code and its corresponding natural language description. Therefore, HyCoQA utilizes a hyperbolic embedder to transform the initial embeddings into hyperbolic space, calculating distances between the codes and descriptions. The process concludes by implementing a scoring layer based on these distances and leveraging hinge loss for model training.

In summary, query transformation bridges the gap between natural language and programming language by transforming queries into intermediate forms or enhancing queries with these forms. The reconstructed query can enhance the performance of code search. However, restructuring queries through query transformation consumes a significant amount of computational resources, which may result in lower overall efficiency.

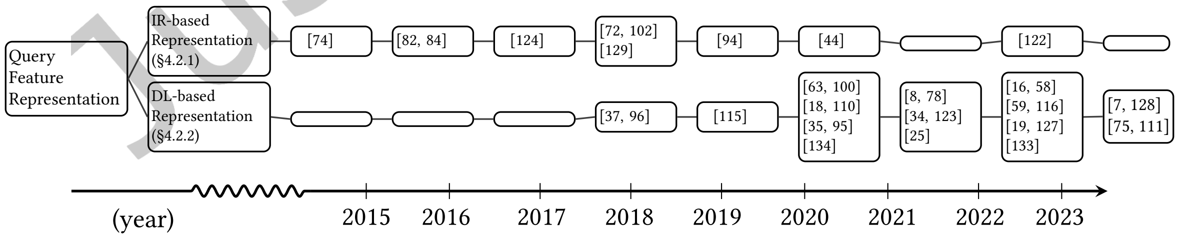


Fig. 7. Evolution of feature representation techniques in query end

4.2 Query Feature Representation

4.2.1 IR-based Feature Representation. Information retrieval (IR) is the technique of finding relevant information from large amounts of information according to the needs of users [101]. In the field of code retrieval (i.e., code search), the IR technique has naturally been introduced by researchers. There are many mature IR-based methods to handle natural language queries. Those methods represent queries as vectors or plain texts, and researchers utilize these representations to optimize the query end, thereby enhancing the final results of code search. Generally speaking, traditional IR methods represent queries as some form of index (such as keyword terms and vectors), which can reflect the semantic information contained within queries. Thereby, the code search model can utilize these representations to optimize the query end and improve the final performance. The first line of Figure 7 showcases the code search works that adopt IR-based techniques to represent query features. It is observed that most early code search efforts employ IR techniques to represent queries. This demonstrates that IR technology is simple and effective in feature representation, and we will describe these works in detail later.

Plain text is a common feature representation method in the field of information retrieval, which refers to preprocessed text from the query. The plain text representation can be used to optimize the query end. As mentioned in Section 4.1, CodeHow [74] expands the query with the APIs and performs code retrieval. To find the related APIs, CodeHow needs to represent the descriptions of APIs and the given query before calculating their similarity. Therefore, Codehow utilizes plain text as the representation of features used in traditional IR methods, while preprocessing of the text is required before generating the plain text representation. In the preprocessing stage, CodeHow finds some words that appear very often and do not have a definite meaning, called stop words (e.g., on, the, are, etc.). To reduce the impact of those meaningless words in the queries and descriptions, they remove those stop words after adopting the text normalization. Thereafter, they also perform stemming. The goal of those preprocessing steps is to represent the text into keyword terms that can be used to compute the similarity by Vector Space Model (VSM) [98]. In the code retrieve stage, the method of representing the queries expanded with APIs is also similar to the preprocessing stage. The expanded queries, represented in the form of plain text, are used for Boolean Model (BM) [97] to match with code snippets.

Nie et al. [82] also employ plain text to represent query features and query expansion word features. As mentioned in Section 4.1.2, they propose a query expansion tool called QECK. In QECK, there are two times of feature representation. The first one involves representing the original query for first-pass retrieval, while the second one involves representing the expanded candidate words to select the most suitable ones. These two instances of feature representation both utilize plain text. In this way, QECK directly processes the text into the search without converting it into vectors. Specifically, the original text is split by Camel-case and separators (e.g., '_'). Then, it filters these words by removing the stop words. Besides, the remaining words are handled by stemming. After the preprocessing, the obtained plain text can be used to retrieve the code snippets. Lu et al. [72] and Hu et al. [44] also utilize a similar feature representation rule to extract the plain text representation for user queries. They regard the processed plain text representation as a keyword set that can retrieve the code more easily.

In order to extract the plain text representation of queries more accurately, Rahman et al. [94] propose a Part-of-Speech (POS) tagging on the query before normal preprocessing. POS can extract meaningful words such as nouns and verbs from the query. Once a query is submitted, they first perform POS and then apply standard natural language preprocessing (i.e., stop word removal, splitting, and stemming) on the query to extract the stemmed words. This more precise query feature representation can further enhance the final performance of code retrieval.

Using traditional IR methods to represent queries, whether expanded or reduced, as vectors is also a common approach. For example, the use of VSM as a method for representing queries as vectors has been applied in many techniques. For example, Sirres et al. [102] and Niu et al. [84] use the VSM to determine the relevancy of the user query. It represents the query as a vector where the term is computed by the TF-IDF weighting.

Another widely used IR method is the bag-of-words model (BOW) [79]. BOW represents a text document as a collection of vocabulary, disregarding the sequence and context of words, only focusing on the frequency of word occurrences within the document. For example, Wu et al. [122] employ the BOW to represent each term in a normalized query and generate the term vector to represent the semantic feature. In BOW, text is regarded as an unordered collection of vocabulary, disregarding its grammar and context. Feature information of query is obtained using the TF-IDF, which calculates the term frequency (TF) and inverse document frequency (IDF). Thereafter, the query vectors are calculated by summing up the vectors of each term. Yang et al. [124] also adopt BOW to represent the query feature. After the normal preprocessing, they convert the query and the intents into a BOW model. The output from the BOW model is regarded as the representation of the query. To map queries with identifiers that are used to expand the queries, Zhang et al. [129] utilize the continues bag-of-words model (CBOW) to convert the original query into a vector representation. CBOW is an improved version of the traditional BOW model. Unlike the BOW model, which focuses solely on the frequency of words within a document, the CBOW model pays attention to the vocabulary information in the context. Therefore, the semantic features represented by the query vectors generated through CBOW are more accurate.

4.2.2 DL-based Feature Representation. DL-based feature representation applies deep neural networks to encode the given feature data to produce semantic-preserving numerical vector representations (also known as embeddings). It has also been widely used to optimize query feature representation in code search. When code search techniques apply DL techniques/models to transform queries into embeddings, models will extract fine-grained semantic and structural information in queries. The second line of Figure 7 presents the code search techniques that apply DL techniques to represent queries. These techniques transfer different neural networks/models from the natural language processing (NLP) or computer vision (CV) fields to encode queries. We will introduce these techniques in subsequent paragraphs.

Ling et al. [63] and Kong et al. [58] utilize FastText [5] to build word vector representations. FastText is an open-source model released by Facebook. Researchers can directly download pre-trained FastText models from open-source repositories to represent the queries they need to process. Besides, FastText employs CBOW and Skip-gram models to learn word vector representations. It focuses on the context of words, generating dense, low-dimensional vectors for each word. Therefore, word representations learned by FastText are more reliable and effective.

Sachdev et al. [96] are among the first researchers to adopt DL techniques to represent query features. They propose a neural code search tool named NCS. In NCS, query feature representation is decomposed into two steps: 1) building word embeddings and 2) building document embeddings. NCS treats a query as a document. In step 1), NCS uses a variant of the Word2vec model, called FastText [5] to build word vector representations (i.e., embeddings). It employs the continuous skip-gram model with a window size of 5, i.e., all pairs of words within a distance of 5 are considered nearby words. In step 2), NCS expresses the intent of the query in the same high–dimensional vector space as the word embeddings, by aggregating the representations of all the words extracted from the query. The authors found that building document embeddings by simply averaging word embeddings does not work well for their purposes. Therefore, they further tried three variants of the combination method: i. Average over all the words; ii. Average over the unique words in each document; iii. Weighted average of all unique words in a document according to the following equations.

$$v_d = u\left(\sum_{w \in d} u(v_w) \cdot \mathrm{tfidf}(w, d, C)\right) \tag{1}$$

$$\mathrm{tfidf}(w, d, C) = \frac{1 + \log \mathrm{tf}(w, d)}{\log |C| / \mathrm{df}(w, C)} \tag{2}$$

where d is a multiset of words representing a document; C is the corpus containing all documents; u is a normalizing function where $u(v) = \frac{v}{|v|}$; tfidf, short for term frequency–inverse document frequency, is a function that assigns a weight for a given word in a given document [81]. A word has a higher weight if it appears frequently in the document but is also penalized if it appears in too many documents in the corpus. Their experiments show that the weighted average method works significantly better than the others.

Recurrent Neural Network (RNN) is also widely used to encode queries. For example, Gu et al. [37] utilize an RNN to represent query features. It is known that RNN has a recurrent structure within the network where hidden layers are recurrently used for computation. Therefore, unlike traditional feed-forward neural networks, RNN can encode sequential queries using its internal memory. The hidden state represents the feature of the query as the final output of RNN. The query representation through RNN can be computed as:

$$h_i = \tanh\left(W\left[h_{i-1}, w_i\right]\right) \tag{3}$$

where each hidden state h_i is generated from the previous hidden state h_{i-1} and w_i is the one-hot representation of each word in query; while W is the matrix of trainable parameters in the RNN, while $\tanh(\cdot)$ is a non-linearity activation function of the RNN.

Wan et al. [115] propose a comprehensive multi-model representation method for source code called MMAN. To match with the source code representation, MMAN exploits the standard LSTM to learn the representation of the given query. In MMAN, building a query feature representation consists of two steps: 1) embedding the query into a vector and 2) generating the hidden state from LSTM. In step 1), MMAN builds a word embedding layer that uses the one-hot embedding function to compute the given query into embedding. In step 2), MMAN applies an LSTM to represent the query. As an improved version of RNN, LSTM can better capture long-term dependencies. Each LSTM unit contains an input gate, a memory cell, and an output gate. The unit receives the embedding from step 1) and generates the hidden state given to the next unit. The last hidden state can be used as the query feature representation denoted h_i^{query}. It can be computed as:

$$h_i^{query} = \mathbf{LSTM}\left(h_{i-1}^{query}, w(q_i)\right) \tag{4}$$

where $w(\cdot)$ is the word embedding layer to embed each word into a vector; q is the input of the word embedding layer which donates the query; the last hidden state h_d^{query} is the final output of the LSTM. Due to its outstanding performance and efficiency, LSTM is also widely employed by other code search techniques for query feature representation. For instance, CSSAM [7] and DCSE [8] all follow almost the same process as MMAN.

Unlike NCS in building document embeddings step, CARLCS-CNN [100], CoNCRA [18], and CSRS [16] introduce the use of the convolutional neural network (CNN) for query feature representation extraction after obtaining the word embedding. As for the query, it is usually short, but it contains informative keywords that reflect the intention of the user. Therefore, CNN is suitable for extracting query feature representation from word embedding. The feature representation D of a query can be computed as:

$$v_i = f\left(W_i * E_{i:i+h-1} + b\right) \tag{5}$$

$$\mathrm{D_h} = [v_1, v_2, \ldots, v_d] \tag{6}$$

$$\mathrm{D} = \mathrm{D_1} \oplus \ldots \oplus \mathrm{D_n} \tag{7}$$

where E is the embedding of the query; W is the convolution kernels for convolution operation; and f is a non-linear function such as the hyperbolic tangent. After generating the feature scores v, the embedding matrix D_h can also be obtained. Thereafter, the final feature representation D is accomplished by merging all the embedding matrices.

Differing slightly from the aforementioned methods, EAGCS [133] and CRaDLe [35] incorporate a maxpooling layer after LSTM. They take the hidden states generated by each LSTM unit and feed them into the maxpooling layer to obtain the final query representation. The purpose of adding a maxpooling layer is to perform further feature extraction, obtaining the significant features. Therefore, the final representation of the query, denoted $\boldsymbol{v}^{query}$, can be formulated as:

$$\boldsymbol{v}^{query} = \mathbf{maxpooling}\left([h_1, \ldots, h_i]\right) \tag{8}$$

where $\mathbf{maxpooling}(\cdot)$ refers to maxpooling layer; the h is the hidden state from each LSTM unit and calculated by Equation (4).

As a variant of the traditional LSTM, bi-directional LSTM (Bi-LSTM) is also employed by Ren et al. [95], Wang et al. [116], and Meng [78] to generate query feature representations. Unlike the traditional LSTM, Bi-LSTM simultaneously considers the past and future information at each time step in the sequence to better capture contextual relationships. Bi-LSTM consists of two LSTM layers, one processing the input sequence from left to right in terms of time steps (i.e., forward LSTM), and the other processing the input sequence from right to left (i.e., backward LSTM). The outputs of these two LSTM layers are concatenated to form a representation that incorporates bidirectional contextual information. The final feature representation generated by Bi-LSTM for the query can be formalized as:

$$\boldsymbol{v}^{query} = h_q = \left[\overrightarrow{h}_q, \overleftarrow{h}_q\right] \tag{9}$$

where $\overrightarrow{h}_q$ and $\overleftarrow{h}_q$ are the forward and backward hidden states produced by the final layer of LSTM.

Considering the self-attention mechanism is suitable for capturing the structural and semantic features within query sequences, Fang et al. [25], Zhu et al. [134], Gu et al. [34], Kong et al. [59], and Sun et al. [110] utilize the self-attention mechanism to extract features and generate representations. The framework they designed for constructing the query feature representation by the self-attention mechanism can be divided into two steps: 1) generating word embedding of query and 2) obtaining feature representation from the self-attention mechanism. In step 1), the framework applies a word embedding tool to build the embedding layer. Sun et al. and Kong et al. select the one-hot code to represent the word embedding of the query. The rest of the techniques exploit the well-known word embedding tool, called word2vec [9]. Word2vec uses the CBOW to embed words. Compared with one-hot representation, distributed representation produced by word2vec can build semantic relations between different words. In step 2), the word embedding generated by the embedding layer is further embedded by the self-attention mechanism. The self-attention mechanism can effectively extract features from the query and generate the feature representations. The following equation simply summarizes the above process of generating a query feature representation through the embedding layer and attention mechanism.

$$\boldsymbol{v}^{query} = \mathbf{Att}\left(E\left(s^{query}\right)\right) \tag{10}$$

where s^{query} is the given query treated as a list of words; while $E(\cdot)$ represents a word embedding layer; $\boldsymbol{v}^{query}$ is the final output of attention mechanism.

To fully understand the relationships between query and code, Deng et al. [19] propose a code search model named FcarCS. The approach of representing query features in FcarCS is similar to the aforementioned self-attention framework, but it replaces the self-attention mechanism with the co-attention mechanism as the

[9]https://github.com/danielfrg/word2vec/

feature-extracted tool. FcarCS constructs a new fine-grained co-attention mechanism to learn interdependent representations for each code snippet and query, respectively. The workflow of this co-attention mechanism can be divided into computing semantic association, extracting semantic information, and calculating semantic vector parts. According to those steps, the co-attention mechanism establishes dynamic attention relationships between queries and code snippets. Therefore, it can explore more fine-grained semantic correlations between each code snippet and query, and enrich the query feature representations.

There also exists an architecture that utilizes both attention mechanisms and a co-attention mechanism to generate query feature representations. To bridge the semantic gap between code snippets and queries effectively and efficiently, Xu et al. [123] propose a two-stage attention-based model for code search, called TabCS. The first stage leverages attention mechanisms to extract semantics from queries considering their textual features. The second stage leverages a co-attention mechanism to capture the semantic correlation between queries and code snippets. Therefore, the co-attention mechanism contributes to better query representation.

Yu et al. [127] propose a novel deep neural network named Method-Description-Joint Embedding Neural Network (MD-JEnn), which uses a joint embedding technique to model the semantic relation between code snippets and descriptions. The description embedding module (DE-Module) is a component of MD-JEnn that embeds natural language descriptions (queries) into vectors. The word embedding model and Bi-LSTM are used in MD-JEnn to embed queries into query vectors first. Since some words in a description are important, it is necessary to assign higher weights to these important words. Thus, MD-JEnn introduces an attention mechanism to aggregate the query vectors of the description into a feature-represented vector by calculating a scalar weight for each vector of the description word. The individual vectors are aggregated to a feature-represented vector $\boldsymbol{v}$ via attention:

$$a_i = \frac{\exp(\widetilde{h_i}^\intercal \cdot \alpha)}{\sum_{j=1}^{n} \exp(\widetilde{h_j}^\intercal \cdot \alpha)} \tag{11}$$

$$\boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot \widetilde{h_i} \tag{12}$$

where h is the the hidden states of the previous Bi-LSTM layer, which represents the query vector and α_i is a learnable matrix initialized in random, α_i also represents the attention weight of each $\widetilde{h_i}$.

Zeng et al. [128] also propose an embedding module similar to MD-JEnn, but they replace Bi-LSTM with LSTM. They utilize the structure of an LSTM layer along with an attention layer to accurately extract feature representations containing keyword information from user queries. Besides, Hu et al. [45] design a similar structure to obtain feature representation. However, instead of using one-hot representation or pre-trained word2vec embeddings to generate the word embedding, they use a structure embeddings matrix to incorporate word-level structure information and get the structure embeddings of queries.

Tang et al. [111] leverage the unique properties of Hyperbolic space to express the feature of user queries. Unlike traditional Euclidean spaces, hyperbolic spaces excel at depicting hierarchical structures. Therefore, they first exploit the BERT [20] (a pre-trained NLP model) as the embedding layer to embed the original query. Thereafter, they use an advanced hyperbolic embedder to encode the query embedding into hyperbolic spaces as the final representation. This representation can express the feature of the query deeply and easily to match with the code snippets.

SSQR [75] utilizes masked language modeling task (MLM) to conduct the extensive unannotated corpus of queries to reformulate queries. SSQR takes the T5 [89] (a pre-trained NLP model) as the backbone model for the MLM task since it has a sequence-to-sequence architecture. The queries are converted into the embeddings by the MLM and masked out a span of 15% consecutive words from a randomly selected position. The purpose of the MLM task is to encourage the model to learn contextual relationships and better understand the semantics of language by inferring missing words in a given context. Following MLM training, the T5 model can accurately capture the semantic meanings of each word in the queries and generate high-quality query embeddings. Therefore, these query embeddings are used for the final code retrieval.

In summary, using IR-based methods or DL-based methods for query feature representation has its advantages and disadvantages. IR-based methods are relatively simpler and more efficient but lack a deep semantic understanding of queries. DL-based methods using neural network models require significant computational resources for training but can provide more accurate feature representations. Choosing the appropriate method from these two for feature representation is crucial depending on the situation.

Summary for RQ1: Existing code search techniques mainly optimize the query end from two aspects: query feature mining and query feature representation. For query feature mining methods, query reduction, query expansion, and query transformation continue to receive attention. Among them, query expansion overall is the most, followed by query transformation, and query reduction. Compared with query reduction, research on query transformation has become more popular in recent years and continues to be a hot trend. For query feature representation methods, most early code search techniques apply information retrieval techniques (e.g., plain text, bag of words, and TF-IDF) to represent query features. In recent years, with the emergence of deep learning technology, most researchers have turned to using neural networks (e.g., FastText, CNN, RNN, LSTM, Bi-LSTM, BERT, and T5) and attention mechanisms (e.g., self-attention and co-attention mechanisms) to represent query features, and this phenomenon will obviously continue.

5 ANSWERING RQ2: WHAT ARE THE CODE-END OPTIMIZATION METHODS IN CODE SEARCH STUDIES?

Understanding code semantics also plays an important role in code search. Only by correctly understanding what a code snippet is doing can we better match it to the query. As a human-readable text written in a specific programming language, code snippet contains semantic information not only in the text but also in its structure, such as AST. To learn about the semantics of the code snippet more comprehensively, it is often necessary to leverage such semantic information and convert it into other representations that can be used to match queries at a later stage. As a result, many code-end optimization techniques have been proposed to capture and understand the semantics of the code snippet to improve the effectiveness and efficiency of code search.

As illustrated in Figure 2, similar to the query end, the optimization techniques devised for the code end can likewise be divided into two primary parts: code feature mining and code feature representation. For code feature mining, existing works mainly focus on mining textual and structural features of the code snippet. The textual features include method names, API sequences, and tokens. The structural features, also referred to as intermediate representations, encompass abstract syntax tree (AST), data flow graph (DFG), control flow graph (CFG), program dependence graph (PDG), and variable-based flow graph (VFG). Details of the above textual and structural features are discussed in Section 5.1. For code feature representation, similar to the query end, the techniques used to represent code features can also be divided into two major classes, i.e., IR-based feature representation and DL-based feature representation. Details of these code feature representation methods are discussed in Section 5.2. In the following subsections, we will discuss these code features and their representation ways in detail, extract their commonalities, find their differences, and summarize the development trends.

5.1 Code Feature Mining

5.1.1 Code Textual Feature Mining.

The code snippets we discuss in this survey are at the method (function) level. A method-level code snippet

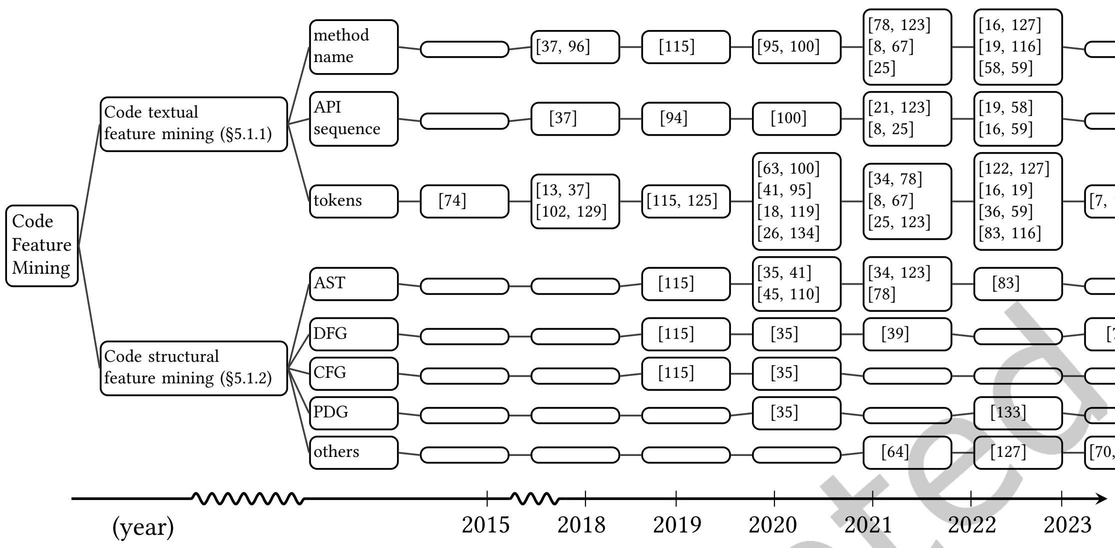


Fig. 8. Evolution of feature mining techniques in code end

usually consists of a method name, parameters (optional), and a method body. In code search, commonly used code textual features include method name, API sequence, and tokens, all of which contain semantic information [59].

Method name. The method name often outlines the functionality (semantics) of a code snippet (at the method level) [100]. Therefore, it plays an important role in understanding the code snippet. As shown in the first line of Figure 8, many works [8, 16, 19, 25, 37, 58, 59, 67, 78, 95, 96, 100, 115, 116, 123, 127] mine and utilize this feature. The extraction of method names is relatively simple and intuitive, and some works describe this process in detail. For example, CodeMatcher [67] develops a tool called JAnalyzer [10], which transforms a method into an AST with the Javaparser [11] library and then extracts method name by traversing the AST. CodeHunter [58] first uses JDT (i.e., Eclipse Java Development Tools) to construct the AST of the source code, and then invokes the `getName()` method in the MethodDeclaration class to obtain the method name. Considering that a method name typically consists of multiple words, it is a common practice to split the camel-case (camelCase) or snake-case (snake_case) concatenated method names into multiple separate tokens for later embedding.

API sequence. API (Application Programming Interface) is of great significance in code search. It refers to a predefined and encapsulated function. Rather than implementing a method from scratch, developers often invocate APIs in their code to facilitate their development activities [38]. What's more, the naming of an API usually briefly describes its functionality (semantics), so the API sequence extracted from a code snippet can help code search models understand how this code snippet is implemented. As illustrated in the second line of Figure 8, many works [8, 16, 19, 21, 25, 37, 58, 59, 94, 100, 123] make use of this feature. Next, we will introduce two ways to extract API sequences. Rahman et al. [94] extract API sequences from Stack Overflow posts with island parsing techniques. They first isolate code snippets from the HTML source of each answer from Stack Overflow using <*code*> tags, and then use a regular expression for Java class to extract the API class tokens having camel case notation. DeepCS [37], DCSE [8] and CodeHunter [58], follow the method described in DeepAPI [38] to extract

[10]https://github.com/liuchaoss/janalyzer
[11]https://github.com/javaparser/javaparser

an API sequence from each Java method. They all employ the Eclipse JDT compiler to parse and traverse the AST. After obtaining the AST, the API sequence is generated by the following rules [37, 59]:

- For a constructor invocation `new C()`, they produce `C.new`.
- For a method call `o.m()` where `o` is an instance of class `C`, they create `C.m`.
- For a method call passed as a parameter, they append the method before the calling method.
- For a statement sequence $s_1; s_2; \ldots; s_N$, they extract the API sequence a_i from each statement s_i and concatenate them to form the API sequence a_1-a_2-$\ldots$-a_N.
- For conditional statements such as $if(s_1)\{s_2;\}else\{s_3;\}$, they produce a sequence from all possible branches, i.e.,a_1-a_2-a_3, where a_i is the API sequence extracted from the statement s_i.
- For loop statements such as $while(s_1)\{s_2;\}$, they generate a sequence a_1-a_2, where a_1 and a_2 are API sequences extracted from statement s_1 and s_2, respectively.

Tokens. Tokens are bags of words that are parsed from the method body of a code snippet. They include useful information such as constant names, variable names, and comments written by the developer. Usually, data preprocessing for tokens removes duplicate words, stop words, and keywords in the programming language, which improves the quality of tokens. As a result, as presented in the third line of Figure 8, almost all code search techniques [7, 8, 13, 16, 18, 19, 25, 26, 34, 36, 37, 41, 59, 63, 67, 74, 78, 83, 95, 100, 102, 111, 115, 116, 119, 122, 123, 125, 127, 129, 134] take advantage of this feature. Most of them tokenize code snippets through camel case splitting or snake case splitting. Zhang et al. [129] focus on query-end optimization and aim to extend natural-language queries with API class-names, so they keep them intact.

5.1.2 Code Structural Feature Mining.

In code search, commonly used code structural features include AST, DFG, CFG, PDG, and other structural features.

AST. Abstract syntax tree (AST) abstractly represents the syntactic structure of a code snippet in the form of a tree [115]. Each node in the tree denotes a structure in the code snippet, such as loop structure, conditional judgment structure, method call, and variable declaration [123]. There is no doubt that AST is very helpful in understanding the code as these structures denote the logic of the code. Consequently, AST is the most commonly used structural feature in code search techniques [34, 35, 41, 45, 78, 83, 110, 115, 123]. They use various different AST parsing tools to generate AST. For instance, MMAN [115] parses C code into AST via an open-source tool named Clang [12]. PSCS [110] extracts AST paths using PathMiner [60], an open-source Java library for mining path-based representations of code. At-CodeSM [78] employs javalang to generate an AST from the code snippet. SPT-Code [83] first uses an AST parser [13] to get the AST. Then, it utilizes a simplified version of structure-based traversal (SBT) [46] called XML-like SBT (X-SBT) to traverse the AST and parse it into a sequence. By employing SBT, the resulting sequence can be reduced by more than half in length.

DFG. Data flow graph (DFG) is a type of intermediate representation. DFG refers to the data flow of a program, which describes how the data in a piece of code flows and how it is processed. Many code search works [7, 35, 39, 115] use DFG as a feature to capture the data dependencies between code elements. For instance, GraphCodeBERT [39] is a pre-trained model for the programming language, which utilizes data flow in the pre-training stage. Such a semantic-level structure is less complex and does not bring an unnecessarily deep hierarchy of AST, the property of which makes the model more efficient. It first parses a code C into an AST, whose terminals (leaves) are used to identify the variable sequence, denoted as $V = \{v_1, v_2, \ldots, v_k\}$. Then, it takes each variable as a node of the graph, and a direct edge $\epsilon = \langle v_i, v_j \rangle$ from v_i to v_j means that the value of j-th variable comes from i-th variable. Taking $x = expr$ as an example, $expr$ is an expression (e.g., $(a+b)*c$), and edges

[12]http://clang.llvm.org/

[13]https://tree-sitter.github.io/tree-sitter/

from all variables in $expr$ to x are added into the graph. The set of directed edges is denoted as $E = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_l\}$ and the graph $\mathcal{G}(C) = (V, E)$ is data flow used to represent dependency relation between variables of the code. Hu et al. [7] establish a code semantic representation graph (CSRG) based on AST and DFG, which is a graph structure more compact than AST. Their performance tests show that CSRG significantly reduces the complexity of input data, improves the training speed compared with only using AST, and improves the accuracy of search results compared with just using DFG. Since AST has more node information than DFG, the training time of the AST model is longer than the DFG model. Therefore, the training time of the CSRG model is increased compared with the DFG model, but less than the AST model. At the same time, its Mean Reciprocal Rank (MRR) score is higher than that of AST and DFG models.

CFG. Control flow graph (CFG) is another type of intermediate representation of the code feature. CFG means the computation and control flow of a program, which has the function of representing all possible execution paths for the program. CFG is also a code feature commonly used by code search techniques [35, 115] to capture control dependencies between code elements. For example, Wan et al. [115] directly make use of the CFG of a code snippet in their code search model. They first parse C functions into CFGs via an open-source tool named SVF [106] [14]. Then, for nodes with the same statement, they keep the nodes that appear first in the output of SVF, remove their child nodes, and connect the children of their child nodes to them. For nodes without statements, they delete them and link their child nodes to their parent nodes.

PDG. Program dependence graph (PDG) can represent the data dependencies and control dependencies of each operation in a program [27]. It is built on AST, but not as deep as AST in structure, and reserves only the execution paths that will affect the execution results [35]. Gu et al. [35] utilize PDG to extract code structures. They propose CRaDLe, a code retrieval model based on semantic dependency learning, to learn the matching relationship between the code and description pairs which helps to retrieve the related code snippets. Zhao et al. [133] establish a statement-level advanced program dependence graph (APDG), which introduces the statement execution information and control logic missed in PDG. In addition, their statistics on the maximal/average/minimal number of nodes and edges of ASTs and APDGs indicate that APDG effectively reduces the complexity of code graphs and is conducive to model training. Based on APDG, they propose EAGCS, a novel code search approach that largely enhances the expression of structural and semantic information in source code. APDG helps EAGCS to learn a deeper understanding of code vectors which improves the performance of retrieving code snippets for a given query.

Others. Some code search techniques define new structural features based on those above. For instance, DGMS [64] and GraphSearchNet [70] both construct a program graph to represent the source code, which helps them precisely learn unified semantic relation representation of the source code and queries. Given a code snippet c, a program graph is a multi-edged directed graph $g(V, \boldsymbol{A}) \in \mathcal{G}$ extracted from c. V is a set of nodes built on the AST and $\boldsymbol{A} \in \{0, 1\}^{k \times m \times m}$ is the adjacency matrix representing the relationships between the nodes (i.e., the edges), where k and m are the total number of edge types and nodes in g, respectively. Particularly, the leaf nodes of AST correspond to the identifier in the code snippet, and the non-leaf nodes represent different compilation units such as "Assign", "BinOp", "Expr". In addition, GraphSearchNet also builds the syntactic edges (i.e., "NextToken", "SubToken") and data-flow edges (i.e., "ComputedFrom", "LastUse", "LastWrite") based on AST nodes, which can represent the code snippet.

Noticing that the API sequence generated by traversing the AST tree ignores semantics contained in the structure of the code snippet, SQ-DeepCS [127] introduces a novel method called *program slice* to preserve structural information. To generate a program slice, first, the AST of a code snippet is parsed, and then different processing methods are applied to different statements extracted from the code snippet. Taking the loop statement as an example, the program slice reserves its judgment conditions and loop body and adds the `for` keyword.

[14]https://github.com/SVF-tools/SVF

For instance, `for(c1; c2; c3){s4;}` is converted to `for(p2){p4;}`, where `p2` and `p4` are the program slices of condition `c2` and statement `s4`.

A variable-based flow graph (VFG) is proposed by Zeng et al. in their work DeGraphCS [128]. They think that tokens and structural features cannot accurately express the in-depth semantics of source code. To overcome this limitation, they propose VFG, which integrates tokens, data flow, and control flow. VFG is constructed on LLVM IR instructions. Specifically, to construct VFG, they first extract the identifiers in each LLVM IR instruction as nodes. Then, they build data dependencies and control dependencies between nodes according to different types of instructions. The data dependencies are built based on the address operation instructions (e.g., "load") and the computation-related constructions (e.g., "add"). The control dependencies are constructed based on the jump instructions (e.g., "br") and address operation instructions. Finally, they apply an optimization mechanism to remove the redundant nodes without changing the semantic information.

In summary, code features are mainly divided into two aspects: textual and structural. Textual features include method names, API sequences, and tokens that reflect the semantic information and functionality of the code snippet. Structural features focus on the structural information, including AST, DFG, CFG, and so on. Different features reflect different aspects of the code snippet, providing information from various levels and perspectives. They give more comprehensive support for code understanding and presentation.

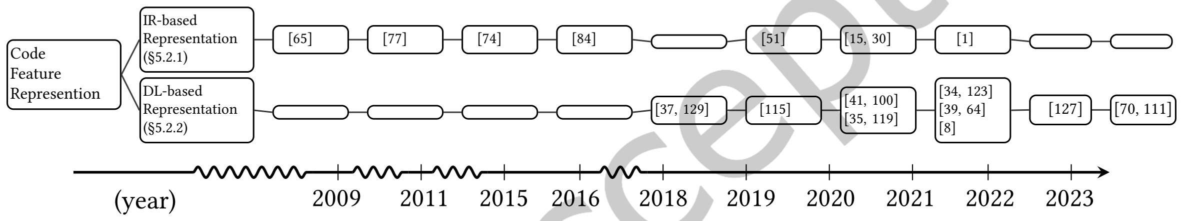


Fig. 9. Evolution of feature representation techniques in code end

5.2 Code Feature Representation

5.2.1 *IR-based Feature Representation.*

As mentioned earlier, code Search is a successful application of IR where the information retrieved is code snippets. The core idea of IR-based code search methods is to treat source code as text, so the code and the query can be matched by their textual similarity. The first line of Figure 9 presents the code search works that adopt IR-based techniques to represent code features. We will describe these studies in detail later.

While an approach based entirely on IR was effective in the early days, as time goes by, researchers have come to realize that treating code as just plain text does not provide a comprehensive understanding of its semantics. As a result, later techniques combine it with some additional information, such as the relational representations of code [65], API [1, 74], comment [30], and keywords [15], etc. Another research [77] aims at making it easier for developers to make use of the code snippet retrieved by showing other possibly related functions and the call graph between them. In this section, we will introduce several typical IR-based code feature representation techniques.

In most applications of mining and searching software corpora, the code's structural aspects, as well as the relevant metadata surrounding it, are always ignored. To utilize this information, Linstead et al. [65] propose Sourcer, which combines standard text IR techniques with source-specific heuristics and a relational representation of code. It employs a relational database consisting of two tables to store the data: (1) program entities: uniquely identifiable elements from the source code; and (2) their relations: any dependency between two entities. In addition, they also store compact representations of attributes for fast retrieval of search results, including keywords from FQNs and comments, and fingerprints used to support structural searches of source code. All entity keywords and metadata are indexed using Lucene to support fast search. In this way, Sourcerer provides a comprehensive, multi-modal platform for searching and finding reusable software components.

Niu et al. [84] propose a code search approach that utilizes a machine learning technique to automatically train a ranking schema. They represent each code snippet as a 12-dimension vector, each dimension denotes the value of a feature extracted from the code snippet. The 12 features are classified into four categories: similarity, popularity, code metrics, and context. The similarity is the textual similarity between a query and candidate code snippets, computed by the Vector Space Model (VSM) [98]. The popularity category has two features: frequency and probability. Frequency is the number of times that the frequent method call sequence of a candidate code snippet occurs in the corpus, while probability is the likelihood of following the method call sequence in a candidate code example. The code metrics category contains eight features that only reflect characteristics of a code snippet regardless of the query, including the line length, the average number of identifiers per line, and so on. The context similarity refers to the similarity between the context of the query and the candidate code snippets.

Similar to Niu et al. [84], Jiang et al. [51] also leverages textual and structural features like the textual similarity between a query and a candidate code snippet and line length of a code snippet to represent the code. Besides, they also consider the topic similarity between a query and a code snippet since a code snippet can be viewed as a textual document describing one or more technical topics. They first generate a term-by-document matrix M, given a collection of code snippets and queries. Then, they employ Latent Dirichlet Allocation (LDA) [4] to identify the latent variables (topics) hidden in the data and generate as output a topic-by-document matrix. A generic entry θ_{ij} of this matrix denotes the probability of the j^{th} document to belong to the i^{th} topic. Finally, the topic similarity between a query and a code snippet can be calculated based on this matrix.

In summary, IR-based code feature representation is relatively simple, is of high efficiency, and requires fewer computing and time resources. However, it usually only represents and matches based on the lexical level and lacks an in-depth understanding of code semantics, making it difficult to deal with polysemy problems.

5.2.2 *DL-based Feature Representation.*

DL-based feature representation on the code-end optimizes the capture and representation of code features (semantics) by introducing advanced deep learning techniques, thereby improving the performance of code search. In this technique, both queries and code snippets are transformed into feature vector representations (also called embeddings). Embedding learns to represent entities (e.g., words, sentences, and graphs) as vectors with the aim of making vector representations of similar entities close to each other [79, 80]. These embeddings are randomly initialized and then fixed via an end-to-end supervised training paradigm. The purpose of embedding training in code search is to bridge the lexical gap between code snippets in programming languages and queries in natural language, so as to better understand the semantics of the code snippets. In the task of code search, both code and query are embedded into a unified vector space through Joint Embedding, which is a technique to jointly embed heterogeneous data. As a result, similar concepts with different modalities are close to each other in this space and the code relevant to a query can be measured by the distance between their vector representations, such as their cosine similarity [37].

Gu et al. [37] first introduces deep learning to the field of code search. They propose a novel deep neural network called CODEnn to learn a unified vector representation of both code snippets and natural language queries. Based on CODEnn, they develop a prototype named DeepCS to support code retrieval. After DeepCS, many DL-based feature representation techniques have sprung up. In this section, we will first discuss how they leverage advanced deep learning techniques to represent a single feature of a code snippet, and how they utilize these features to build high-performance code search models.

To begin with, we will introduce the representation methods for the textual features of a code snippet as mentioned in Section 5.1.1.

For the method name, DeepCS [37] and SQ-DeepCS [127] both use a Recurrent Neural Network (RNN) to encode the sequence of the tokens obtained by splitting the method name, as RNN can capture the semantics of sequence information well. It is computed using the same Equation (3) mentioned in Section 4.2.2. DCSE [8] employs Bi-LSTM to learn the semantic information since the method name has word order. Empirically finding that the average length of each method name sequence is only 2 or 3 in its training data, CARLCS-CNN [100] applies a Convolutional Neural Networks (CNN) instead of RNN, which is supposed to be good at extracting robust and abstract features. Therefore, in CARLCS-CNN, the representation of the method name is calculated as:

$$\boldsymbol{m}_{1:n} = \boldsymbol{m}_1 \oplus \boldsymbol{m}_2 \oplus \cdots \oplus \boldsymbol{m}_n \tag{13}$$

$$\boldsymbol{c}_i = f\left(W_M * \boldsymbol{m}_{i:i+h-1} + \boldsymbol{b}\right) \tag{14}$$

$$M_h = [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{n-h+1}] \tag{15}$$

where $\boldsymbol{m}_i \in \mathbb{R}^k$ is the k-dimensional word vector; $\oplus$ is the concatenation operator; $W_M \in \mathbb{R}^{k\times h}$ is a *filter* involved in the convolution operation, which is applied to a window of h words to produce a feature; $\boldsymbol{b} \in \mathbb{R}$ is a bias term; $*$ is the convolution operator and f is a non-linear function such as the hyperbolic tangent; M_h is a *feature map* produced after applying the filter to each possible window of words in the method name. What's more, CARLCS-CNN uses three types of filters with varying window sizes h from 2 to 4, with the number of each type of filter set to d. Then, it completes the convolution operation through these filters to extract three distinctive feature maps, i.e., $M_{h_1}, M_{h_2}, M_{h_3} \in \mathbb{R}^{d\times(n-h+1)}$,, and finally concatenates them into a feature matrix M:

$$M = M_{h_1} \oplus M_{h_2} \oplus M_{h_3} \tag{16}$$

TabCS [123] utilizes attention mechanism-based search models to improve the efficiency of training and testing. Let $\boldsymbol{m}_i \in \mathbb{R}^k$ be a k-dimensional word initial vector corresponding to the i-th word in a method name. Given a sequence of length n $\{\boldsymbol{m}_1, \ldots, \boldsymbol{m}_n\}$, the attention weight for each $\boldsymbol{m}_i$ is computed as follows:

$$\alpha_{m_i} = \frac{exp(\boldsymbol{a}_{m_i} \cdot \boldsymbol{m}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{m_i} \cdot \boldsymbol{m}_i^T)} \tag{17}$$

where the attention vector $\boldsymbol{a}_{m_i}$ is optimized during model training. The attention weight for each initial vector is computed by applying the softmax function to the product of the initial vectors and attention vectors. Then, it multiplies each initial vector with its corresponding attention weight, and concatenates the resulting weighted vectors:

$$M = \alpha_{m_1}\boldsymbol{a}_1 \oplus \alpha_{m_2}\boldsymbol{a}_2 \oplus \cdots \oplus \alpha_{m_n}\boldsymbol{a}_n \tag{18}$$

For the tokens, considering that tokens are the informative keywords of code, CARLCS-CNN [100] uses CNN to encode them. COSEA [119] also leverages CNN, since CNN has the natural ability to capture locality information which can be used for capturing code blocks' information. Moreover, COSEA utilizes layer-wise attention to learn the code semantic representation. After transforming a code snippet into embedding vectors, the model can get longer and longer code block representation through convolutional modules. Finally, attentive pooling is performed on these block representations and the ultimate semantic embedding of the code snippet is obtained.

DeepCS [37] and SQ-DeepCS [127] simply encode tokens via a multilayer perceptron (MLP), i.e., the conventional fully connected layer, as tokens are unordered in the code snippet. Let $\boldsymbol{\tau}_i \in \mathbb{R}^d$ represent the embedded representation of the token τ_i, $\boldsymbol{W}^\Gamma$ represent the matrix of trainable parameters in the MLP, the embedding vector

$\boldsymbol{h}_i$ of the i-th token is computed as:

$$\boldsymbol{h}_i = \tanh(\boldsymbol{W}^{\Gamma}\boldsymbol{\tau}_i), \forall i = 1, 2, \ldots, N_{\Gamma} \tag{19}$$

Next, the individual vectors are summarized to a single vector t through maxpooling:

$$\boldsymbol{t} = \text{maxpooling}([\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{N_{\Gamma}}]) \tag{20}$$

Zhang et al. [129] use Word2vec [15] to extract vector representations of tokens, which is an efficient implementation of the continuous bag-of-words model (CBOW) [79]. Considering that not all tokens contribute equally to the final semantic representation of the code snippet, MMAN [115] introduces the attention mechanism on tokens to extract the ones that are more important to the representation of a sequence of code tokens after encoding them via LSTM. The specific calculation process of LSTM and attention mechanism refer to Equation (4) and Equation (17), respectively. HyCoQA [111] transforms tokenized code snippets into numerical representations through a BERT embedding layer [20], because the word vectors generated by BERT can be dynamically adapted to the context provided by neighboring words.

For the API sequence, similar to the handling of the method name, DeepCS [37] encodes it into a vector representation using an RNN with maxpooling. In view of the dynamic sequential features of the API sequence, CARLCS-CNN [100] implements Bi-LSTM to do the embedding, using the Equation (9) mentioned in Section 4.2.2. Then, the API sequence is encoded by concatenating all the output hidden states to a feature matrix. Finally, TabCS [123] performs an attention mechanism on the randomly initialized feature matrix of the API sequence, the same as its process to the method name and tokens.

Next, we will introduce the representation methods for the structural features of a code snippet as mentioned in Section 5.1.2.

For the AST, MMAN [115] utilizes Tree-LSTM whose unit contains multiple forget gates and adopts the hidden state of the root node as the AST modality representation. Considering a node N with the value x_i in its one-hot encoding representation, which has a left child N_L and a right child N_R, the Tree-LSTM recursively computes the embedding for N from the bottom up. Assume that the left child and the right child maintain the LSTM state $(\mathbf{h}_L, \mathbf{c}_L)$ and $(\mathbf{h}_R, \mathbf{c}_R)$, respectively, then the LSTM state (h, c) of N is computed as:

$$(\mathbf{h}_i^{ast}, \mathbf{c}_i^{ast}) = \text{LSTM}\left(\left(\left[\mathbf{h}_{iL}^{ast}; \mathbf{h}_{iR}^{ast}\right], \left[\mathbf{c}_{iL}^{ast}; \mathbf{c}_{iR}^{ast}\right]\right), w(x_i)\right) \tag{21}$$

where $i = 1, \ldots, |x|$ and $[\cdot;\cdot]$ means the concatenation of two vectors.

MT-CAT [41] converts the AST representation of a code snippet by the deterministic parser to a string using SBT [46], and then applies FastText [5] as the word embedding module to map a string to an embedding. TabCS [123] converts tree nodes into initial vector embeddings by building vocabularies. Then, since only part of the nodes can reflect the method's function, TabCS performs an attention mechanism and concatenates the weighted vectors into a feature matrix, which extracts the important nodes. The final concatenation is the feature matrix of the AST. The computation process can be referred to Equation (17) and Equation (18), where the word sequence $\{m_1, \ldots, m_n\}$ is replaced by a node sequence $\{\boldsymbol{ast}_1, \ldots, \boldsymbol{ast}_n\}$. Multimodal [34] transforms the AST into a novel tree structure named Simplified Semantic Tree (SST) to make the tree structure semantically better for code search, and then serializes SST to a linear token sequence by sampling tree-paths [3, 54] or traversing tree-structures [14, 46]. Finally, multimodal adopts a SelfAtt model to encode the tree sequence. The SelfAtt model is a transformer-based model that leverages a self-attention mechanism and BERT's positional embedding to learn from contextual information [20, 114].

For the CFG, as it is a directed graph, MMAN [115] applies a gated graph neural network (GGNN) to represent it, which is a neural network architecture developed for graphs. Define a graph as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V}$ is a set of vertices (v, ℓ_v) and $\mathcal{E}$ is a set of edges (v_i, v_j, ℓ_e). ℓ_v and ℓ_e are labels of vertex and edge, respectively. In the code

[15] https://code.google.com/archive/p/word2vec/

search scenario, each vertex is the node of CFG, and each edge denotes the control flow of code. GGNN learns the graph representation through the following procedures: First, the hidden state for each vertex $v \in \mathcal{V}$ is initialized as $\mathbf{h}^{cfg}_{v,0} = w(\ell_v)$, where w is the one-hot embedding function. Then, for each round t, each vertex receives the vector $\mathbf{m}_{v,t+1}$, which is the "message" aggregated from its neighbours. It can be formulated as follows:

$$m_{v,t+1} = \sum_{v' \in \mathcal{N}(v)} \mathbf{W}_{\ell_e} \mathbf{h}_{v',t} \tag{22}$$

where $\mathcal{N}(v)$ are the neighbours of vertex v. Message from each neighbour is mapped into a shared space via $\mathbf{W}_{\ell_e}$ in round t. GGNN updates each vertex v's hidden state using the gated recurrent unit (GRU) [17], which can be formulated as follows:

$$\mathbf{h}^{cfg}_{v,t+1} = \text{GRU}(\mathbf{h}^{cfg}_{v,t}, \mathbf{m}_{v,t+1}) \tag{23}$$

Finally, after T rounds of iteration, the embedding representation of the CFG is obtained by aggregating the hidden state of all vertices.

For the DFG, given a source code $C = \{c_1, c_2, \ldots, c_n\}$ with its comment $W = \{w_1, w_2, ..., w_m\}$, GraphCodeBERT [39] first obtains the data flow graph. $\mathcal{G}(C) = (V, E)$ as discussed in Section 5.1.2. Next, it concatenates the comment, source code, and the set of variables V in the DFG as the sequence input, and converts the sequence into an input vector by summing the corresponding token and position embeddings. Then, the model applies N transformer layers over the input vector to produce contextual representations. Specifically, it defines a graph-guided masked attention function to incorporate the graph structure into a transformer. At last, after being pre-trained on three tasks, namely masked language modeling, edge prediction, and node alignment, GraphCodeBERT can be applied to some downstream tasks, such as code search.

For the PDG, CRaDLe [35] first constructs a dependency matrix $\Upsilon \in \{0,1\}^{(l)\times(l)}$ according to the extracted PDG, where l is the number of statements in the code snippet. The element $v_{ij} = 1$ if the i-th statement has a data/control dependency on the j-th statement; otherwise $v_{ij} = 0$. Then, it uses a one-layer MLP to encode the matrix Υ according to Equation (19). Note that $\boldsymbol{\tau}_i$ in the Equation (19) is replaced by v_i, and $\boldsymbol{h}_i$ is the embedding of the dependency information for each statement here. After that, CRaDLe concatenates the dependency embedding with statement-level token embedding to get the representation of the statement. Next, it adopts Bi-LSTM to encode the sequence of the statement embeddings and uses the same encoder to get the description embeddings. Finally, after calculating the cosine distance, the model will rank the code snippets and return the higher-ranked ones to the developer.

For the program graph, DGMS [64] adopts one variant of GNNs—Relational Graph Convolutional Networks (RGCNs) to learn its node embedding. Particularly, given the program graph of a code snippet $G_e = (\mathcal{V}_e, \mathcal{E}_e, \mathcal{R}_e)$ with nodes $e_i \in \mathcal{V}_e$ and edges $(e_i, r, e_j) \in \mathcal{E}_e$, where $r \in \mathcal{R}_e$ represents edge type, RGCN calculates the updated embedding vector $\boldsymbol{e}_i$ of each node $e_i \in \mathcal{V}_e$ as follows:

$$\boldsymbol{e}_i^{(l+1)} = \text{ReLU}\left(W_\Theta^{(l)} \boldsymbol{e}_i^{(l)} + \sum_{r \in \mathcal{R}_e} \sum_{j \in \mathcal{N}_i^r} \frac{1}{|\mathcal{N}_i^r|} W_r^{(l)} \boldsymbol{e}_j^{(l)}\right) \tag{24}$$

where $\boldsymbol{e}_i^{(l+1)}$ is the updated embedding vector of node e_i in the $(l+1)$th layer of RGCN, $\mathcal{N}_i^e$ is the set of the neighbors of node e_i under the edge type $r \in \mathcal{R}_e$, $W_\Theta^{(l)}$ and $W_r^{(l)}$ are parameters of the RGCN model to be learned. Thus, the node embeddings $\boldsymbol{X}_e = \{\boldsymbol{e}_j\}_{j=1}^N \in \mathbb{R}^{(N,d)}$ for the program graph G_e is obtained, where d represents the embedding dimensions of each node.

GraphSearchNet [70] introduces a Bidirectional Gated Graph Neural Network (BiGGNN) to encode the program graph, which learns node embeddings from both incoming and outgoing directions for the program graph $g(V, \boldsymbol{A})$ extracted from the code snippet. During each hop n, for node v, it applies an aggregation function to take a set of incoming (or outgoing) neighboring node vectors as input and outputs a backward (or forward) aggregation vector. The summation function is selected as the aggregation function, where $N_{(v)}$ denotes the neighbors of node v and ⊣ / ⊢ is the backward or forward direction.

$$\boldsymbol{h}^{n}_{\mathcal{N}_{\dashv(v)}} = \text{SUM}(\{\boldsymbol{h}^{n-1}_{u}, \forall u \in \mathcal{N}_{\dashv(v)}\}) \tag{25}$$

$$\boldsymbol{h}^{n}_{\mathcal{N}_{\vdash(v)}} = \text{SUM}(\{\boldsymbol{h}^{n-1}_{u}, \forall u \in \mathcal{N}_{\vdash(v)}\}) \tag{26}$$

Then, the node embeddings for both directions are fused as follows:

$$\boldsymbol{h}^{n}_{\mathcal{N}_{(v)}} = \text{Fuse}(\boldsymbol{h}^{n}_{\mathcal{N}_{\dashv(v)}}, \boldsymbol{h}^{n}_{\mathcal{N}_{\vdash(v)}}) \tag{27}$$

The fusion function is formulated as a gated sum of two inputs:

$$\text{Fuse}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{z} \odot \boldsymbol{a} + (1 - \boldsymbol{z}) \odot \boldsymbol{b} \tag{28}$$

$$z = \sigma(\boldsymbol{W}_z[\boldsymbol{a}; \boldsymbol{b}; \boldsymbol{a} \odot \boldsymbol{b}; \boldsymbol{a} - \boldsymbol{b}] + \boldsymbol{b}_z) \tag{29}$$

where $\odot$ is the component-wise multiplication, σ is a sigmoid function and z is gating vector. Then, GRU is used to update node representations. At last, after k hops of computation, the final node representation $\boldsymbol{h}^{k}_{v}$ is obtained and max-pooling is applied over all nodes $\{\boldsymbol{h}^{k}_{v}, \forall v \in V\}$ to get a d-dim graph representation $\boldsymbol{h}^{g}$:

$$\boldsymbol{h}^{g} = \text{maxpool}(\text{FC}(\{\boldsymbol{h}^{k}_{v}, \forall v \in V\})) \tag{30}$$

where FC is the fully-connected layer.

In summary, DL-based code feature representation possesses powerful learning capabilities, which enable it to learn to process complex semantics from large-scale data and better understand code snippets. However, it requires the design and training of complex neural network models, consuming more computing and time resources. The training of models also relies on large-scale and high-quality data.

Summary for RQ2: Similar to the query end, existing code search techniques mainly optimize the code end from two aspects: code feature mining and code feature representation. Regarding code feature mining, The usage frequency of the three code textual features (i.e., method name, API sequence, tokens) is similar. Moreover, there are many works that simultaneously utilize multiple types of code textual features. Among code structural features, AST is used most frequently, followed by DFG, CFG, and PDG. Some recent works have explored some new code structural features, e.g., VFG, program paragraph, and program slice. Also, some works simultaneously utilize multiple code structural features. It is worth noting that there are many code search techniques that consider both code textual features and code structural features. Regarding code feature representation, existing code search techniques have designed distinct code representation methods for various code features. Works in recent years have widely adopted DL-based methods to generate numerical vector representations of code textual or structural features.

6 ANSWERING RQ3: WHAT ARE THE MATCH-END OPTIMIZATION METHODS IN CODE SEARCH STUDIES?

In general, developers anticipate that a code search system will prioritize the most relevant and possible code snippets as search results to facilitate their development. However, identifying the code snippet that best matches a query from thousands of snippets can be highly complex. Even with a decent understanding of the semantics of the query and the code snippet, it may still cost a large amount of time. In order to find the code snippet that matches the query efficiently and correctly, numerous methods and algorithms have been devised to optimize the match-end of the code search. In short, a better match-end optimization method can improve the performance of code search. It finds the most effective and efficient method to match the code snippet and query after making comprehensive use of their syntax and semantics.

According to the feature representation form of the query and code, existing match-end optimization techniques can be divided into three distinct groups: text-based matching, vector-based matching, and classification-based matching, as shown in Figure 2. The text-based matching techniques aim to utilize the keywords from query and code snippet to make the matching, detailed in Section 6.1. The vector-based matching techniques are widely used in recent work. They can be further divided into two categories: vector distance-based, and embedding distance-based, detailed in Section 6.2. The classification-based matching techniques aim to regard the matching tasks into classification tasks, detailed in Section 6.3. Figure 10 showcases the evolution of the above three types of match-end optimization techniques. In the following subsections, we will discuss them in detail and summarize their development trends.

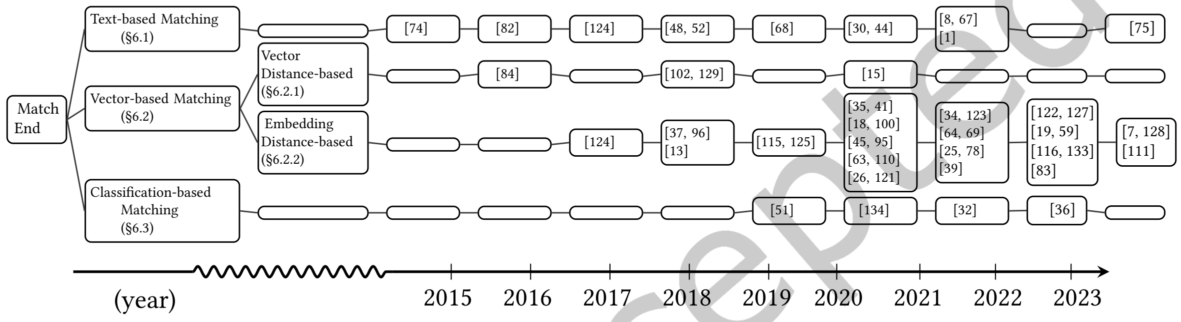


Fig. 10. Evolution of techniques in match end

6.1 Text-based Matching

As a traditional matching method, text-based matching utilizes textual features such as word frequency or keywords to match code snippets with queries. These matching methods are simple and efficient and are usually adopted by the early code search methods. As shown in Figure 10, we have summarized a total of 12 articles that have used text-based matching methods and provided detailed summaries and overviews of them.

A basic method of text-based matching is calculating the number of common keywords shared in the text of code snippets and queries. CodeMatcher [67] is a good example of directly utilizing the keywords for matching. CodeMatcher is an IR-based code search model that inherits the advantages of DeepCS [37] (i.e., the capability of understanding the sequential semantics in important query words). CodeMatcher performs well and clearly outperforms existing solutions, such as CodeHow [74]. CodeMatcher first collects metadata for query words to identify irrelevant/noisy ones. According to this preprocessed step, the collected keywords are used to launch an iterative fuzzy match on indexed method names. Then, iteratively performs the fuzzy search with important query keywords on the codebase to obtain the top k candidate code snippets. To refine the fuzzy search results, CodeMatcher designs a reranking step to measure the matching degree between query and candidate code snippets. During the reranking, the method name and body are regarded as two different components for the search. This is because the method name is often defined in natural language whose semantic representation is close to the query, but the method body implements the goal of the method name in programming languages. Therefore, when matching keywords for a candidate code snippet, CodeMatcher calculates the characters matched keywords in the method name of code snippets as S_{name}. It indicates a level of the ranked method with more overlapped tokens between the query and code snippet. CodeMatcher reorders code snippets based on S_{name} in descending order to retrieve the final search results. In the case of tied scores, a similar approach is employed to calculate the keyword coverage scores on code methods as S_{body}. A higher S_{body} implies better keyword matching between query and code snippets in the method body. Thus, the candidate code snippet with a higher S_{body} will rank ahead.

Lu et al. [73] also calculate common keywords as a matching strategy for the expanded query and code snippets. Specifically, they first preprocess the code snippets and extract the identifiers. Then, they calculate the number of common keywords between each code snippet's identifier set and the expanded query, and use the proportion of this number to the maximum word count in either the identifier set or the expanded query as the similar score. Finally, they use this score to determine the matching results.

Extended Boolean Model (EBM) is an information retrieval model that incorporates features of both the traditional Boolean model and the VSM. It combines the advantages of precisely matching from the Boolean model with the semantic similarity from VSM. CodeHow [74] applies the EBM to consider the impact of both text similarity and APIs on code search. According to the CodeHow introduction in Section 4.1.2, it can calculate similar scores to obtain the potentially relevant APIs that match the query. Thus, CodeHow considers both the text and APIs in EBM to match the query with code snippets. EBM conducts keyword retrieval on method name and method body of code snippets through Boolean operators and obtains similar scores. As mentioned in CodeMatcher, the method name and body are regarded differently in matching the query and code snippets. The method name is more important than the method body. Therefore, the similarity scores for method name and method body are weighted differently in the calculation of the total score, with weights of 1.5 and 1.0, respectively. Besides, the final score also includes the API score obtained during the API understanding phase, with a weighting coefficient of 1.5. IECS [124] also utilizes EBM to conduct the similarity between the original query with code and the expanded query with code, respectively. The sum results are used to match the code snippets with the query as the final output.

BM25 is a well-known information retrieval technique that is designed to improve the traditional TF-IDF model. BM25 is employed to assess the relevance between textual and queries, so it is commonly applied to optimize the matching end. Both query expansion code search models, NQE [68] and QECK [82] leverage BM25 to match code snippets with queries. Similar to the computation for TF-IDF, it uses the following formula to give the score for the code snippets document d and a given query word x:

$$\mathrm{BM\,25}(d,x) = \mathrm{IDF}(x) \cdot \frac{\mathrm{TF}(x,d)\cdot(k+1)}{\mathrm{TF}(x,d) + k\cdot\left(1-b+b\cdot\frac{|d|}{\mathrm{avgdl}}\right)}, \tag{31}$$

where $TF(\cdot)$ is a function that calculates the term frequency; $IDF(\cdot)$ is a function that computes the inverse document frequency; and *avgdl* is the average document length of all code snippets. k and b are tunable parameters that adjust the impact of term frequency on the final score, thereby enhancing the flexibility of the BM25 model. Based on the scores calculated using BM25, NQE, and QECK output the code snippets with the highest score as the matching result for the given query. Furthermore, many code search methods that do not involve deep neural networks also utilize BM25 as a retrieval model to match queries with code snippets [44, 48, 52]. This indicates that BM25 is not only effective but also easy to use.

Lucene is a conventional text search engine behind many existing code search tools. It is a versatile tool that can be embedded in various code search methods, providing efficient code retrieval capabilities. It incorporates text similarity, FQN (full quality name) of entities, and code popularity to rank the code snippets. DCSE [8], FACER [1], and SSQR [75] use Lucene as a retrieval tool to match the expanded queries with code snippets, demonstrating the effectiveness of their query expansion technique.

Fu et al. [30] regard each method as recommendation objects, denoted as documents, and store the objects in the indexes in the Lucene framework. Each document will be divided into multiple fields according to different code characteristics. When matched with the given query, they get a series of keyword tokens from the query and implement the keyword-based search in Lucene to find the retrieved results.

In summary, text-based matching primarily draws on traditional IR models or algorithms, and it is easy to implement these methods to code search for matching the query with code snippets. However, these methods typically capture the textual information and do not consider the deep semantic connections between queries and code snippets. This is precisely a limitation of such text-based matching methods.

6.2 Vector-based Matching

According to Section 4.2 and Section 5.2, it can be found that vector is the most common form of feature representations of queries and code snippets. Therefore, there are naturally more code search techniques that use vector-based matching methods to find relevant code snippets. Vector-based matching methods calculate distances using vectors that extract fine-grained semantic and syntactic features from code snippets and queries. Therefore, compared to text-based matching, these methods can retrieve code snippets that have similar features to the query, and improve overall code search performance. The second line of Figure 10 shows that most current code search papers adopt vector-based matching methods. Generally, there are two categories of methods for generating feature vector representations for code snippets and queries: using traditional IR methods to generate feature vectors and using DL neural networks to generate embeddings (vectors generated by DL-based methods referred to collectively as embeddings). Thereby, in this section, we divide vector-based matching into vector distance-based methods and embedding distance-based methods. Then, we will discuss and analyze vector distance matching and embedding distance matching as two separate parts.

6.2.1 Vector Distance-based Matching.

As mentioned earlier, vector distance-based matching methods retrieve the relevant code snippets by calculating the distance between the IR-based feature representation of both query and code snippet. In intuition, the closer the code snippet and the query are in these spatial distances, the more likely they are compatible in the semantics. Therefore, finding the code snippet in the vector space that is closest to the query is the core of distance-based matching.

Before calculating the distance between code snippets and queries, representing them in the vectors is the first step of vector distance-based matching methods. Traditional IR methods are always used to establish the vectors. These vectors represent the semantics information of queries and code snippets and are used to match the query with code snippets in the distance and retrieve the ranking results. The most general way to measure the distance is cosine distance or Euclidean distance [97]. In the following, we will introduce 4 representative code search works that execute the vector distance-based matching.

As a traditional IR technique, researchers find that the Vector Space Model (VSM) is suitable for basic code search matching tasks [84, 102]. VSM utilizes the common representation method TF-IDF to convert queries and code into vectors and then calculates their similarity using the cosine similarity formula:

$$\cos(v_c, v_q) = \frac{{v_c}^T v_q}{\|v_c\| \|v_q\|} \tag{32}$$

where v_c and v_q are the vectors of the code snippet and query, respectively. The higher the similarity, the more related the code is to the query. Thus, we categorize the technique that employed the VSM as a part of vector distance-based matching.

Zhang et al. [129] also utilize VSM to calculate vector cosine distances, but in addition, they consider other features to measure the matching results between queries and code snippets. Therefore, their proposed weighted-sum ranking schema incorporates a total of five feature components in the calculation, including fv, which is the similarity score generated by Lucene; fs, which is the cosine distance similarity; fn, which represents keyword term frequency; fp, which is the number of parameters in the code example; and fa, which denotes the score for recommended API class names in query expansion. The scores of these components are weighted and summed to produce the final matching result.

Chen et al. [15] propose a semantics-based search for Java methods named Quebio. Compared with most methods in the matching end, Quebio combines a customized keyword-based search with a distance-based search to check the relevant code snippets with a given query quickly. Quebio designs the matching method in two steps. In the first step, a keyword-based search is employed for quick filtering, which calculates the query keyword frequencies in the code text. Only code snippets with results exceeding the threshold enter the second step. In the second step, the selected code snippets are preprocessed and sorted using the TF-IDF method, with the top 5 most frequently appearing words selected as the code snippet's summary. Then, VSM is used to construct vectors for these summaries, and the cosine distance between the summary vector of the code snippet and the query vector is calculated as the final retrieval result.

6.2.2 *Embedding Distance-based Matching.*

As its name implies, embedding distance-based matching methods utilize the embeddings of the code snippet and query to calculate the distance, and optimize the ranking results. The embeddings are obtained by the deep neural network model, which can capture the semantics and structure in fine-grained, thus the ranking performance can be significantly improved.

Similar to the vector distance-based matching method mentioned above, it is necessary to measure the degree of match between queries and code snippets by computing the distance between their embeddings. Cosine distance is the most widely used in the field of feature similarity comparison [120]. Cosine distance evaluates the similarity of embeddings by calculating the cosine of their angle. The computation is similar to Equation 32 but replaces v_c and v_q with $\boldsymbol{v_c}$ and $\boldsymbol{v_q}$, representing the embeddings of code and query. Specifically, most works [7, 13, 18, 19, 25, 34, 35, 37, 45, 59, 63, 64, 69, 78, 83, 95, 96, 100, 110, 115, 122, 123, 127, 128, 133] match the code snippets with the query on their cosine distance in the embedding space.

Actually, cosine similarity can be widely used not only in Euclidean spaces but also in non-Euclidean spaces (i.e., hyperbolic spaces). As we mentioned in Section 4, HyCoQA [111] introduces the Hyperbolic space to express connections between code snippets and their corresponding queries. Unlike traditional Euclidean spaces, hyperbolic spaces excel in representing hierarchical structures, which frequently underlie the connection between code and its corresponding natural language description. Thus, after utilizing the BERT embedding layer to represent code snippets and queries into embeddings, HyCoQA utilizes a hyperbolic embedder to transform the initial embeddings into hyperbolic space. Finally, using Equation (32) to calculate the cosine similarity between code snippet embeddings and query embeddings in hyperbolic spaces, the obtained rankings are used for the retrieved result.

Both CoaCor [125] and $TranS^3$ [121] generate comments for code snippets to assist in code search. Therefore, they combine two cosine similarities as the final score:

$$\text{score}(Q, C) = \beta * \cos\left(e_q, e_c\right) + (1 - \beta) * \cos\left(e_q, e_s\right) \tag{33}$$

where e_q, e_c, and e_s are the embeddings of query, comments, and code snippets, respectively. β is a weight parameter that ranges from 0 to 1, and $cos(\cdot)$ is the cosine similarity function. The first cosine similarity is calculated based on the query and code snippet, and the second one is calculated based on the query and generated comments. While it applies a weighting parameter to each of the two cosine distances and then combines the results to obtain the final similarity score for ranking code snippets for a given query.

Similar to Equation (33), QueCos [116] also employs a hybrid ranking approach that combines the weighted sum of two cosine similarities as the final similarity score. The difference lies in that the first cosine similarity is calculated between enriched queries and code snippets, and the second cosine similarity is calculated between the original query and code snippets. QueCos can enrich the original query semantically through reinforcement learning, generating enriched queries. Therefore, the designed hybrid ranking approach considers both the original query and enriched queries to return the ultimate search results.

The inner product distance is also a widely used measure of the similarity between embeddings. Both CodeBERT [26] and GraphCodeBERT [39] leverage the inner product distance to match the code snippets with the query. CodeBERT and GraphCodeBERT are the pre-trained models for the programming language, which can be used in a series of downstream tasks, including code search. In their experiments, six programming language datasets are used to fine-tune the downstream code search tasks. All the tasks calculate the inner product of code and query embeddings as relevant scores to rank candidate code snippets.

Euclidean distance (i.e., L2 distance) calculates the straight-line distance between two points in a multidimensional space. It is also a commonly used measure of the similarity between embeddings. MP-CAT [41] computes the L2 distance in the similarity module:

$$\operatorname{sim}\left(e_c, e_q\right) = 1 - d\left(\frac{e_c}{\|e_c\|_2}, \frac{e_q}{\|e_q\|_2}\right) \tag{34}$$

where $d(\cdot)$ denotes the L2 distance calculation for the dimensional code embeddings e_c and query embeddings e_q. The similarity module selects the code snippet with a close similarity to the given query. Finally, the output of the similarity module is regarded as the final output of the retrieved result.

In summary, vector-based matching uses feature vectors or embeddings generated from IR models or DL models to calculate the distance in the feature space. Therefore, it can match queries and code snippets that have similar syntactic and structural features. However, there are potential problems, such as overlap problems (e.g., "message" and "msg"), which may inevitably impact the accuracy of matching queries with code snippets.

6.3 Classification-based Matching

The classification-based matching methods are different from the distance-based method. Instead of calculating the distance of vectors or embeddings from the code snippets and queries or using the textual information to match the code snippets with the query, classification-based matching methods transform the matching task into a classification task. Such methods utilize the classifier to predict the probability of semantics matching and use the predicted probabilities to rank the candidate code snippets. In the following, we will discuss them in detail.

To implement an effective and efficient code search system, Gotmare et al. [32] propose a hybrid ranking framework called CASCODE, which includes the fast encoder and slow classifier to improve the performance of retrieving the search results. They prove that leveraging classification tasks involving NL-PL sequence pairs for code retrieval can achieve an optimal result. However, adopting this approach would be impractical due to the large number of candidates to be considered for each query. Therefore, they divide the retrieval process into two stages. In the first stage, the transformer encoders jointly transform the natural language query and code snippet into the embeddings and calculate the cosine distance to provide the top k candidate code snippets. In general, this fast encoder stage ensures that the framework can quickly retrieve the top few candidate code snippets that are relatively close to the query from the code repository. In the slow classifier stage, they utilize a transformer encoder-based classifier to predict the top k candidate code snippets from the first stage. The researchers' experiments demonstrate that the well-trained classifier retrieves code snippets by returning the match probabilities between queries and code snippets in semantics, resulting in excellent performance. However, using a classifier is costly and slower. As for the limited number of candidate code snippets, a transformer classifier jointly processes the query sequence with each of the candidates to predict the probability of their semantics matching has become feasible. The predicted score of each candidate is regarded as the metric to rank the final retrieved results.

To address the problem of overlaps (e.g., “message” and “msg”), which has a negative impact on retrieved results, Zhuet et al. [134] propose a novel neural architecture called OCoR. In the matching end, similar to the slow classifier stage of CASCODE [32], OCoR also utilizes a transformer classifier to predict the probability of two classes. The first class denotes that the input natural language query and the input code are related, whereas the second class denotes that the input natural language query and the input code are unrelated. The predicted classification probability of the first class is the relevance score between the input natural language query and the code snippet. The relevance scores are used for ranking the relevant code snippets.

To provide a better code retrieved result, Jiang et al. [51] propose a novel method combining both information retrieval and code classification, called ROSF. ROSF decomposes code search into two stages. Specifically, in the first stage, ROSF utilizes the IR-based method (BM25) to retrieve a candidate set that contains N code snippets. This is a coarse-grained search for identifying a few relevant candidate code snippets. In the next fine-grained re-ranking stage, ROSF views the problem of ranking the candidate code snippets as a multi-class classification task. For each instance of the candidate set for a new query, ROSF employs the learned linear predictor function to predict the probabilities of four possible relevance scores. In other words, each instance has four probability values corresponding to four relevance scores, denoted as $score1$, $score2$, $score3$, $score4$. Then, the relevance score with the maximum probability value is selected as the predicted relevance score for the instance. Among the candidate sets, ROSF first sorts the subset containing the code snippets with predicted $score4$ according to the predicted probability values in descending order. Then, ROSF selects the top k code snippets as the final results. If the size of this subset is less than k, ROSF considers the subset with $score3$ until it collects k code snippets.

To accelerate the retrieval efficiency of deep learning-based code search approaches, Gu et al. [36] propose a novel approach called CoSHC, which adopts the recall and re-rank mechanism with the integration of code classification and deep hashing to improve code search performance. CoSHC first generates the code and description embeddings from deep learning networks. Then, a deep hashing module is utilized to generate the corresponding binary hash codes for the embeddings in binary hashing space. Since the capacity of binary hashing space is very limited compared to Euclidean space, they cluster the source code whose representation vectors are close to each other into the same category. After obtaining the code representation categories in hashing space, the classifier in the category prediction module will calculate the probability distribution of categories for the given query. The number of code candidates R_i for each category i will be recalled according to this probability distribution, which can be computed as:

$$R_i = \min\left(\lfloor p_i \cdot (N-k) \rfloor, 1\right), i \in 1, \ldots, k, \tag{35}$$

where p_i is the predicted probability for category i; N is the total recall number of source code and k is the number of categories. In the final re-ranking stage, the original representation vectors of these recalled code candidates will be retrieved and utilized for the cosine similarity calculation.

In summary, classification-based matching transforms the matching task into a classification task and utilizes DL models to predict the matching relationship between queries with candidate code snippets. Classification-based matching leads to high-precision matching results. The obvious downside is the high computational resources required for the classifier. In the process of conducting code searches on large datasets, classification-based matching consumes more time than text-based or vector-based matching methods. Therefore, typical classification-based matching methods generally rely on filtering and optimization strategies to reduce the number of candidate sets of code snippets for matching.

Summary for RQ3: There are three optimization methods utilized in the match end by existing code search techniques, i.e., text matching-based methods, vector matching-based methods, and classification-based methods. All three types of methods have received widespread attention. Among them, with the popularity of DL technology, vector matching-based methods (especially embedding distance-based methods) are the most common. Although classification-based methods are also based on embeddings, they require training additional classification layers, which means they have higher training costs. It is foreseeable that embedding distance-based methods will remain mainstream for some time in the future.

7 CHALLENGES AND OPPORTUNITIES FOR FUTURE WORK

After analyzing the existing code search techniques in both advantages and disadvantages, we can suggest that further research will also have practical and research significance. From the practical significance of implementing code search, these excellent techniques will certainly help us solve the difficulties in the development of software. On the other hand, code search still has many challenges to be solved. We pose some of the challenges and opportunities that have emerged from the papers below.

7.1 Challenges

7.1.1 Accurate query understanding. Similar to information retrieval, accurately understanding the query intent is a prerequisite for accurate code search. Accurately understanding query semantics contributes to bridging the semantic gap between queries and code snippets. As mentioned in Section 4, code search techniques typically enhance the understanding of the query through feature mining and feature representation on the query end. Clearly, feature mining and representation for a query is a highly challenging task. The answer to RQ1 demonstrates that existing code search techniques have devised a variety of solutions for query feature mining and representation. Therefore, the challenge of accurate query understanding can further be attributed to the difficulties encountered in accurate query feature mining and representation.

Accurate query feature mining. In Section 4.1, we meticulously detailed three approaches for mining query features, including query reduction, query expansion, and query transformation. However, numerous challenges remain in mining query features to achieve better query understanding. For instance, query reduction can be used to remove noisy/redundant terms of the query and preserve important query features. However, the identification of noisy/redundant terms is not trivial. Incorrect removal of certain terms will result in a loss of semantic context, making it challenging to capture the user's intent accurately. In addition, users might have different expectations regarding the reduction process, and striking a balance between code search precision and recall is challenging. Query expansion aims to add new terms to the original query to enrich query features. However, the relevance and quality of the newly added terms play a significant role. Using irrelevant or inaccurate terms can hinder the search process. How to control the quality of extensions is challenging. Query transformation aims to transform the original query feature into an alternative form feature, e.g., code API. However, transforming a query into an appropriate code-related feature while preserving its semantic meaning is challenging. Ensuring accurate transformation is crucial for retrieving relevant results. Additionally, some users search for code in domain-specific programming languages that might not have direct mappings from natural language queries. Adapting query transformation rules to handle domain-specific code retrieval is a challenge.

Accurate query feature representation. Similarly, in Section 4.2, we showcased the variety of query feature representation techniques adopted in existing code search research. We salute the researchers for their dedicated explorations into feature representation techniques. Obviously, accurately representing the features of a query still poses many challenges. For example, IR-based query feature representation methods represent queries as some form of index (such as keyword terms and vectors). However, queries and code snippets might use different vocabularies or terminologies, leading to a mismatch in the representation. How to align the vocabulary effectively is crucial and challenging for accurate retrieval. DL-based query feature representation methods apply deep neural networks to encode the given query feature to produce embeddings. However, queries often contain ambiguous terms or phrases that can have multiple meanings in the context of code. It is challenging for DL models to discern the intended meaning of ambiguous queries accurately. In addition, queries can vary significantly in length, making it challenging to process them uniformly. The applied models need to handle variable-length inputs effectively.

7.1.2 Accurate code understanding. Accurately understanding the functionality (semantic) of the code snippet is also a prerequisite for accurate code search. It contributes to filling the semantic gap between queries and code snippets. The answer to RQ2 demonstrates that existing code search techniques have devised a variety of methods for code feature mining and representation. Therefore, the challenge of accurate code understanding can further be attributed to the difficulties in accurate code feature mining and representation.

Accurate code feature mining. In Section 5.1, we meticulously sorted out the various code features that existing research has extracted and utilized to enhance code understanding accuracy. However, there are still many challenges in adequately extracting features to achieve optimal code search performance. In addition, there is information overlap between different code features. For example, Token and the labels of AST leaf nodes overlap. Therefore, how to deal with this redundant information is also a challenge when considering multiple features simultaneously. Finally, whether structural features without Token information (e.g., syntactic structure in AST, control flow in CFG, and data flow in DFG) actually help facilitate code representation and code search tasks still lacks systematic research. The diversity of feature parsing tools and preprocessing methods makes such systematic research challenging.

Accurate code feature representation. Similarly, based on our summary of research on code feature representation in Section 5.2, we find that this field presents challenges. Using DL techniques to represent code features has become mainstream nowadays. Compared to traditional IR-based code feature representation methods, the code representations generated by DL-based cod feature representation methods can better capture the semantics of code. However, DL-based methods require large, high-quality labeled datasets for effective training. Obtaining such datasets with accurately labeled code snippets can be challenging, especially for specific programming languages or domains. In addition, code snippets also vary significantly in length, from short expressions to lengthy functions. Designing representations that handle variable-length inputs effectively is a challenge. Moreover, as mentioned earlier, code contains multiple textual and structural features. Integrating these diverse features into a cohesive representation without losing essential information is challenging.

7.1.3 Efficiently Query-Code Matching. As mentioned in RQ3, optimizing the methods for matching code snippets with queries is also one of the keys to improving the effectiveness of code search. Since different developers have different wording habits, this will cause the text-based matching method to fail to retrieve the expected results. Therefore, mapping relationships between query vocabulary and code vocabulary is challenging for the text-based matching method. For vector distance-based matching methods, it is necessary for the query vectors and code vectors to be in a unified vector space, and the distance between vectors should accurately reflect semantic relevance. However, whether utilizing IR techniques or DL techniques, mapping queries and code snippets to a unified vector space while preserving their respective semantics is a challenging task. Instead of directly calculating distance using query and code vectors, classification-based matching methods involve concatenating the two vectors and utilizing a classifier for binary classification. However, correctly concatenating query and code vectors is challenging because vectors are abstract, and DL is low in interpretability. Last but not least, an aspect that requires more attention is the efficiency challenge faced by all query-code matching methods when dealing with extensive code search corpus. Therefore, designing efficient matching methods is also a challenging task.

7.1.4 Practical Applications of Code Search techniques. Existing research on code search primarily focuses on designing advanced code search models to enhance search accuracy. There is less attention to the practical implementation and application of code search techniques/models in production environments. Creating usable and user-friendly code search tools or plugins is a highly challenging task. Advanced code search tools have a considerable distance to cover before they can be practiced in production environments. The main challenges in the practical application of code search tools lie in the updates of the backbone models and in achieving user-friendly interactions.

Updating of the backbone models. The actual software development is very complex, and the code snippets that developers search for may exceed the training data used in the current code search models. Therefore, if code search tools want to solve real-world problems, it is important to update the backbone model in a targeted manner. However, how to update the backbone model of tools in production environments based on user feedback has not yet received sufficient attention and research.

Implementation of user-friendly interaction. Despite the numerous code search models that have been proposed, there are still many potential barriers before these models can be practically implemented in practical applications. Developing excellent programming assistance tools, including code search tools, is a complex and systematic engineering task. This requires not only a full stack of programming skills from developers but also consideration of the actual user experience. How to achieve efficient search result recommendations and how to make it easier for developers to use have not yet received sufficient attention and research.

7.2 Opportunities

7.2.1 Addressing the Challenges Mentioned in Section 7.1. Typically, challenges in research can also be seen as opportunities from another perspective. Focusing on the challenges outlined in Section 7.1 and designing corresponding solutions offer many excellent research opportunities and directions. The achievements in these directions will significantly enhance the performance and practical usability of code search.

7.2.2 Applying LLM in code search. Recently, with the success of large language models (LLMs) in natural language processing (NLP) [22, 88], an increasing number of software engineering (SE) researchers have started integrating them into the resolution process of various SE tasks [24, 43, 131], such as code generation [12, 118], program repair [11, 130, 132], and code summarization [108, 109]. The recent work by Li et al. [62] has also demonstrated the potential of using LLMs to improve code search performance. They first utilize LLMs to retrieve exemplar code snippets, and then synthesize the original query and these exemplar codes to formulate an augmented query. Essentially, they utilized LLMs for query expansion, thereby improving code search.

In addition, we envision two research opportunities for leveraging LLMs to enhance code search as follows. (1) Using LLMs to mine and represent query and code features. It is known that LLMs are trained on vast corpora (including code corpora), which endow them with powerful natural and programming language understanding capabilities. Therefore, leveraging LLMs for mining and representing query and code features holds promise for achieving more accurate semantic matching between queries and code snippets. (2) Using LLMs to bridge programming languages and natural languages. Several recent studies have shown that LLMs have code summarization capabilities [108, 109], i.e., generating natural language descriptions (also known as summaries) of code snippets. These summaries can be further used to substitute the original user queries for code retrieval.

8 THREATS TO VALIDITY

In this chapter, we explore the validity threats to our study. Validity concerns the relationship between the research results and the actual situation, as well as how the conclusions might be incorrect. We have categorized the threats to the validity of our study into two types, including internal validity and external validity. Our discussions on these identified threats are as follows:

8.1 Threats to Internal Validity

The threats to internal validity refer to experiment errors and human biases [112]. In our study, we employ a cautious research strategy to mitigate threats to internal validity. Firstly, we establish three specific research questions covering the different essential dimensions of the code search technique (detailed in Section 3.1). Subsequently, we utilize the PIO approach to generate keywords from these research questions and collect papers from six widely used electronic databases (including DBLP, Google Scholar, IEEE Explore, etc.). These collected papers under multiple rounds of manual filtering and screening based on our defined research scope and criteria (detailed in Section 3.2, 3.3). From the remaining 1427 papers, we meticulously review 68 of them, analyzing their technical aspects according to the three research perspectives. Additionally, we explore and discuss potential future developments based on existing work within each perspective. As a result, the systematic exploration has instilled confidence in our work and established a certain level of effectiveness in the research outcomes.

8.2 Threats to External Validity

External validity concerns the extent to which the results can be generalized beyond the scope of the study, even when specific cause-and-effect relationships have been established in the study. Threats to external validity involve the generalizability of reported research findings [112]. For our study, we meticulously select 68 papers from six commonly used electronic databases. Hence, these papers likely cover the core studies in the code search field. Therefore, the conclusions and findings drawn in our study can potentially apply to the techniques inadvertently missed in this paper or those emerging as new techniques in the near future. Furthermore, within each research question section, we have conducted theoretical qualitative analyses. Hence, we believe that external validity threats are not particularly challenging for this study.

9 CONCLUSION

This article provides a 3-dimensional survey of the technological developments in code search over the past thirty years. This survey focuses on the three core components of code search technology, i.e., query understanding component, code understanding component, and query-code matching component. We classify and discuss the optimization techniques proposed for each component. Specifically, we divide the techniques for optimizing the query and code ends into two parts, namely feature mining and feature representation, respectively. For the match end, we categorize existing optimization techniques into three major classes, i.e., text-based matching, vector distance-based matching, and classification-based matching. For each end (each optimization category), we provide a detailed introduction to some representative optimization technologies and summarize the technology development trends. Based on the comprehensive observation of optimization techniques proposed in existing code search papers, we summarize some challenges that still need to be addressed and suggest research opportunities.

10 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by the National Natural Science Foundation of China (61932012, 62372228, 62272220), the Science, Technology, and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003), the Program B for Outstanding PhD Candidate of Nanjing University (202201B054), and the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] Shamsa Abid, Shafay Shamail, Hamid Abdul Basit, and Sarah Nadi. 2021. FACER: An API usage-based code-example recommender for opportunistic reuse. *Empirical Software Engineering* 26, 5 (2021), 110.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–17.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th Conference on Programming Language Design and Implementation*. ACM, Philadelphia, PA, USA, 404–419.

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, January (2003), 993–1022.

[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*. ACM, Boston, MA, USA, 1589–1598.

[7] Bo Cai, Yaoxiang Yu, and Yi Hu. 2023. CSSAM: Code Search via Attention Matching of Code Semantics and Structures. In *Proceedings of the 30th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Taipa, Macao, 402–413.

[8] Fuqi Cai, Changjing Wang, Qing Huang, Zhengkang Zuo, and Yunyan Liao. 2021. Search for Compatible Source Code. *International Journal of Software Engineering and Knowledge Engineering* 31, 3 (2021), 477–502.

[9] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 13th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Tallinn, Estonia, 964–974.

[10] Guihong Cao, Jianfeng Gao, Jian-Yun Nie, and Jing Bai. 2007. Extending query translation to cross-language query expansion with markov chain models. In *Proceedings of the 16th Conference on information and knowledge management*. ACM, Lisbon, Portugal, 351–360.

[11] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. 2023. A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *CoRR* abs/2304.08191, 1 (2023), 1–12.

[12] Hailin Chen, Amrita Saha, Steven Chu-Hong Hoi, and Shafiq Joty. 2023. Personalized Distillation: Empowering Open-Sourced LLMs with Adaptive Learning for Code Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 6737–6749.

[13] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, Montpellier, France, 826–831.

[14] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 1–11.

[15] Zhengzhao Chen, Renhe Jiang, Zejun Zhang, Yu Pei, Minxue Pan, Tian Zhang, and Xuandong Li. 2020. Enhancing example-based code search with functional semantics. *Journal of Systems and Software* 165, 1 (2020), 110568.

[16] Yi Cheng and Li Kuang. 2022. CSRS: code search with relevance matching and semantic matching. In *Proceedings of the 30th International Conference on Program Comprehension*. ACM, Virtual Event, 533–542.

[17] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555, 1 (2014), 1–9.

[18] Marcelo de Rezende Martins and Marco Aurélio Gerosa. 2020. CoNCRA: A Convolutional Neural Networks Code Retrieval Approach. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. ACM, Natal, Brazil, 526–531.

[19] Zhongyang Deng, Ling Xu, Chao Liu, Meng Yan, Zhou Xu, and Yan Lei. 2022. Fine-grained Co-Attentive Representation Learning for Semantic Code Search. In *Proceedings of the 29th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 396–407.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 23rd Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Minneapolis, MN, USA, 4171–4186.

[21] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning Approach for Semantic Code Search. In *Proceedings of the 30th International Conference on Information & Knowledge Management*. ACM, Queensland, Australia, 2994–2998.

[22] Mengnan Du, Fengxiang He, Na Zou, Dacheng Tao, and Xia Hu. 2022. Shortcut Learning of Large Language Models in Natural Language Understanding: A Survey. *CoRR* abs/2208.11857, 1 (2022), 1–10.

[23] Jacob Eisenstein. 2019. *Introduction to natural language processing*. MIT press, USA.

[24] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *CoRR* abs/2310.03533, 1 (2023), 1–23.

[25] Sen Fang, Youshuai Tan, Tao Zhang, and Yepang Liu. 2021. Self-Attention Networks for Code Search. *Information & Software Technology* 134, 1 (2021), 106542.

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, Online Event, 1536–1547.

[27] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

[28] Edwin T Floyd and J Tom Kinser. 1990. Service Code Search High Performance Medical Records Indexing on a Budget. In *Proceedings of the Annual Symposium on Computer Application in Medical Care*. American Medical Informatics Association, USA, 408.

[29] William B. Frakes and Brian A. Nejmeh. 1987. Software Reuse Through Information Retrieval. *SIGIR Forum* 21, 1-2 (1987), 30–36.

[30] Shanqing Fu, Bing Li, Yi Cai, Zhuang Liu, and Junxia Guo. 2020. Recommendation Based on Java Code Analysis and Search. In *Proceedings of the 6th Fuzzy Systems and Data Mining*, Vol. 331. IOS Press, Virtual Event, 514–521.

[31] Inc. GitHub. 2008. GitHub. site: https://github.com. Accessed: 2022.

[32] Akhilesh Deepak Gotmare, Junnan Li, Shafiq Joty, and Steven C.H. Hoi. 2021. Cascaded Fast and Slow Models for Efficient Semantic Code Search. *Computing Research Repository* abs/2110.07811, 1 (2021), 1–12.

[33] Luca Di Grazia and Michael Pradel. 2022. Code Search: A Survey of Techniques for Finding Code. *Computing Research Repository* abs/2204.02765, 1 (2022), 1–30.

[34] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution*. IEEE, Luxembourg, 483–494.

[35] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2020. CRaDLe: Deep Code Retrieval Based on Semantic Dependency Learning. *Neural Networks* 141, 1 (2020), 385–394.

[36] Wenchao Gu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Michael R. Lyu. 2022. Accelerating Code Search with Deep Hashing and Code Classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Dublin, Ireland, 2534–2544.

[37] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 933–944.

[38] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering*. ACM, Seattle, WA, USA, 631–642.

[39] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations*. OpenReview.net, Virtual Event, Austria, 1–18.

[40] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE Computer Society, San Francisco, CA, USA, 842–851.

[41] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A Multi-Perspective Architecture for Semantic Code Search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics,ACL*. Association for Computational Linguistics, online, 8563–8568.

[42] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.* 18, 7 (2006), 1527–1554.

[43] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR* abs/2308.10620, 1 (2023), 1–62.

[44] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, Wang Gao, and Mengting Yuan. 2020. Unsupervised software repositories mining and its application to code search. *Software - Practice and Experience* 50, 3 (2020), 299–322.

[45] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, and Mengting Yuan. 2020. Neural joint attention code search over structure embeddings for software Q&A sites. *Journal of Systems and Software* 170, 1 (2020), 110773.

[46] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, Gothenburg, Sweden, 200–210.

[47] Qing Huang, Yang Yang, and Ming Cheng. 2019. Deep learning the semantics of change sequences for query expansion. *Software - Practice and Experience* 49, 11 (2019), 1600–1617.

[48] Qing Huang, Yangrui Yang, Xue Zhan, Hongyan Wan, and Guoqing Wu. 2018. Query expansion based on statistical learning from code changes. *Software: Practice and Experience* 48, 7 (2018), 1333–1351.

[49] Ishrar Hussain, Leila Kosseim, and Olga Ormandjieva. 2008. Using Linguistic Knowledge to Classify Non-functional Requirements in SRS documents. In *Proceedings of the 13th International Conference on Applications of Natural Language to Information Systems*. Springer, London, UK, 287–298.

[50] Stack Exchange Inc;. 2008. Stack Overflow. site: https://stackoverflow.com/. Accessed: 2022.

[51] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. 2019. ROSF: Leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets. *IEEE Transactions on Services Computing* 12, 1 (2019), 34–46.

[52] Oscar Karnalim. 2018. Language-agnostic source code retrieval using keyword & identifier lexical pattern. *International Journal of Software Engineering and Computer Systems* 4, 1 (2018), 29–47.

[53] Kisub Kim, Sankalp Ghatpande, Dongsun Kim, Xin Zhou, Kui Liu, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2024. Big Code Search: A Bibliography. *ACM Comput. Surv.* 56, 1 (2024), 25:1–25:49.

[54] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by Feeding Trees to Transformers. In *Proceedings of the 43rd International Conference on Software Engineering*. IEEE, Madrid, Spain, 150–162.

[55] Barbara A. Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *Information & Software Technology* 55, 12 (2013), 2049–2075.

[56] Barbara A. Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *Information & Software Technology* 55, 12 (2013), 2049–2075.

[57] Yutaka Kobayashi and Yasuhisa Niimi. 1989. An Efficient VQ Code Search Algorithm Using Signal Continuity. In *Processings of the 1st European Conference on Speech Communication and Technology*. ISCA, Paris, France, 1446–1449.

[58] Xianglong Kong, Hongyu Chen, Ming Yu, and Lixiang Zhang. 2022. Boosting Code Search with Structural Code Annotation. *Electronics* 11, 19 (2022), 3053.

[59] Xianglong Kong, Supeng Kong, Ming Yu, and Chengjie Du. 2022. Joint Embedding of Semantic and Statistical Features for Effective Code Search. *Applied Sciences* 12, 19 (2022), 10002.

[60] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE / ACM, Montreal, Canada, 13–17.

[61] Chodorow Leacock and M Chodorow. 1998. Wordnet: An electronic lexical database-combining local context and wordnet similarity for word sense identification, in wordnet: An electronic lexical database.

[62] Haochen Li, Xin Zhou, and Zhiqi Shen. 2024. Rewriting the Code: A Simple Method for Large Language Model Augmented Code Search. *CoRR* abs/2401.04514, 1 (2024), 1–11.

[63] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive Deep Code Search. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, Seoul, Republic of Korea, 48–59.

[64] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep Graph Matching and Searching for Semantic Code Retrieval. *ACM Transactions on Knowledge Discovery from Data* 15, 5 (2021), 88:1–88:21.

[65] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

[66] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. 2022. Opportunities and Challenges in Code Search Tools. *Comput. Surveys* 54, 9 (2022), 196:1–196:40.

[67] Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E Hassan, and Shanping Li. 2021. CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–37.

[68] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural query expansion for code search. In *Proceedings of the 3rd International Workshop on Machine Learning and Programming Languages*. ACM, Phoenix, AZ, USA, 29–37.

[69] Shangqing Liu, Xiaofei Xie, Lei Ma, Jing Kai Siow, and Yang Liu. 2021. GraphSearchNet: Enhancing GNNs via Capturing Global Dependency for Semantic Code Search. *Computing Research Repository* abs/2111.02671, 1 (2021), 1–14.

[70] Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2839–2855.

[71] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API recommendation without historical software repositories. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM/IEEE, Montpellier, France, 282–292.

[72] Jinting Lu, Ying Wei, Xiaobing Sun, Bin Li, Wanzhi Wen, and Cheng Zhou. 2018. Interactive Query Reformulation for Source-Code Search With Word Relations. *IEEE Access* 6, 1 (2018), 75660–75668.

[73] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Montreal, QC, Canada, 545–549.

[74] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *Proceedings of the 30th International Conference on Automated Software Engineering*. IEEE/ACM, Lincoln, NE, USA, 260–270.

[75] Yuetian Mao, Chengcheng Wan, Yuze Jiang, and Xiaodong Gu. 2023. Self-Supervised Query Reformulation for Code Search.

[76] Lee Wei Mar, Ye-Chi Wu, and Hewijin Christine Jiau. 2011. Recommending Proper API Code Examples for Documentation Purpose. In *Proceedings of the 18th Asia Pacific Software Engineering Conference*. IEEE Computer Society, Ho Chi Minh, Vietnam, 331–338.

[77] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, Waikiki, Honolulu, HI, USA, 111–120.

[78] Yao Meng. 2021. An Intelligent Code Search Approach Using Hybrid Encoders. *Wirel. Commun. Mob. Comput.* 2021, 1 (2021), 9990988:1–9990988:16.

[79] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations*. OpenReview.net, Scottsdale, Arizona, USA, 1–12.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems*. MIT Press, Lake Tahoe, Nevada, United States, 3111–3119.

[81] Bhaskar Mitra and Nick Craswell. 2017. Neural Models for Information Retrieval. *Computing Research Repository* abs/1705.01509 (2017), 1–52.

[82] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query Expansion Based on Crowd Knowledge for Code Search. *IEEE Transactions on Services Computing* 9, 5 (2016), 771–783.

[83] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1–13.

[84] Haoran Niu, Iman Keivanloo, and Ying Zou. 2016. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22, 1 (2016), 259–291.

[85] Rahul Pandita, Kunal Taneja, Laurie A. Williams, and Teresa Tung. 2016. ICON: Inferring Temporal Constraints from Natural Language API Descriptions. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Raleigh, NC, USA, 378–388.

[86] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: an update. *Information & Software Technology* 64, 1 (2015), 1–18.

[87] Andreas Polydoros and Charlesl Weber. 1984. A unified approach to serial search spread-spectrum code acquisition-Part I: General theory. *IEEE Transactions on communications* 32, 5 (1984), 542–549.

[88] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023. Is ChatGPT a General-Purpose Natural Language Processing Task Solver?. In *Proceedings of the 28th Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 1339–1384.

[89] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[90] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. Association for computing machinery, Austin, TX, USA, 357–367.

[91] Mohammad Masudur Rahman and Chanchal K. Roy. 2016. QUICKAR: automatic query reformulation for concept location using crowdsourced knowledge. In *Proceedings of the 31st International Conference on Automated Software Engineering*. ACM, Singapore, 220–225.

[92] Mohammad Masudur Rahman and Chanchal K. Roy. 2021. A Systematic Literature Review of Automated Query Reformulations in Source Code Search. *Computing Research Repository* abs/2108.09646, 1 (2021), 1–68.

[93] Mohammad Masudur Rahman, Chanchal Kumar Roy, and David Lo. 2016. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Computer Society, Suita, Osaka, Japan, 349–359.

[94] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. 2019. Automatic Query Reformulation For code search using crowdsourced knowledge. *Empirical Software Engineering* 24, 4 (2019), 1869–1924.

[95] Leiming Ren, Shinmin Shan, Kai Wang, and Kun Xue. 2020. CSDA: A novel attention-based LSTM approach for code search. *Journal of Physics: Conference Series* 1544, 1 (2020), 012056.

[96] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd International Workshop on Machine Learning and Programming Languages*. ACM, Philadelphia, PA, USA, 31–41.

[97] Gerard Salton, Edward A. Fox, and Harry Wu. 1983. Extended Boolean Information Retrieval. *Commun. ACM* 26, 11 (1983), 1022–1036.

[98] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[99] Abdus Satter and Kazi Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. In *Proceedings of the 19th International Conference on Computer and Information Technology*. IEEE, Dhaka, Bangladesh, 586–591.

[100] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. In *Proceedings of the 28th International Conference on Program Comprehension*. Association for Computing Machinery, Seoul, Republic of Korea, 196–207.

[101] Amit Singhal. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[102] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering* 23, 5 (2018), 2622–2654.

[103] Keele Staffs et al. 2007. *Guidelines for performing systematic literature reviews in software engineering.* Technical Report. Technical report, ver. 2.3 ebse technical report. ebse.

[104] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* 10, 5 (1984), 494–497.

[105] Kathryn T. Stolee, Sebastian G. Elbaum, and Matthew B. Dwyer. 2016. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software* 116, 1 (2016), 35–48.

[106] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, Barcelona, Spain, 265–266.

[107] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 388–400.

[108] Weisong Sun, Chunrong Fang, Yudu You, Yuchen Chen, Yi Liu, Chong Wang, Jian Zhang, Quanjun Zhang, Hanwei Qian, Wei Zhao, Yang Liu, and Zhenyu Chen. 2023. A Prompt Learning Framework for Source Code Summarization. *CoRR* abs/2312.16066, 1 (2023), 1–23.

[109] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *CoRR* abs/2305.12865 (2023), 1–13.

[110] Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. 2020. PSCS: A Path-based Neural Model for Semantic Code Search. *Computing Research Repository* abs/2008.03042, 1 (2020), 1–7.

[111] Xunzhu Tang, Zhenghan Chen, Saad Ezzini, Haoye Tian, Yewei Song, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Hyperbolic Code Retrieval: A Novel Approach for Efficient Code Search Using Hyperbolic Space Embeddings. *Computing Research Repository* abs/2308.15234, 1 (2023), 1–10.

[112] Yuan Tian, David Lo, and Julia L. Lawall. 2014. Automated construction of a software-specific word similarity database. In *Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE Computer Society, Antwerp, Belgium, 44–53.

[113] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. 2015. Extracting Development Tasks to Navigate Software Documentation. *IEEE Transactions on Software Engineering* 41, 6 (2015), 565–581.

[114] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems*. MIT Press, December 4-9, 5998–6008.

[115] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 13–25.

[116] Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. 2022. Enriching query semantics for code search with reinforcement learning. *Neural Networks* 145, 1 (2022), 22–32.

[117] Chong Wang, Xin Peng, Zhenchang Xing, Yue Zhang, Mingwei Liu, Rong Luo, and Xiujie Meng. 2023. XCoS: Explainable Code Search based on Query Scoping and Knowledge Graph. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 140:1–140:28.

[118] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *CoRR* abs/2401.06391, 1 (2024), 1–13.

[119] Hao Wang, Jia Zhang, Yingce Xia, Jiang Bian, Chao Zhang, and Tie-Yan Liu. 2020. COSEA: Convolutional Code Search with Layer-wise Attention. *Computing Research Repository* abs/2010.09520 (2020), 1–10.

[120] Shuohang Wang and Jing Jiang. 2017. A Compare-Aggregate Model for Matching Text Sequences. In *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net, Toulon, France, 1–11.

[121] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranSˆ3: A Transformer-based Framework for Unifying Code Summarization and Code Search. *Computing Research Repository* abs/2003.03238, 1 (2020), 1–12.

[122] Chen Wu and Ming Yan. 2022. Learning Deep Semantic Model for Code Search using CodeSearchNet Corpus. *Computing Research Repository* abs/2201.11313, 1 (2022), 1–6.

[123] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Honolulu, HI, USA, 342–353.

[124] Yangrui Yang and Qing Huang. 2017. IECS: Intent-enforced code search via extended Boolean model. *Journal of Intelligent & Fuzzy Systems* 33, 4 (2017), 2565–2576.

[125] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *Proceedings of the 28th The World Wide Web Conference*. ACM, San Francisco, CA, USA, 2203–2214.

[126] Haibo Yu, Wenhao Song, and Tsunenori Mine. 2016. APIBook: an effective approach for finding APIs. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*. ACM, Beijing, China, 45–53.

[127] Hao Yu, Yin Zhang, Yuli Zhao, and Bin Zhang. 2022. Incorporating Code Structure and Quality in Deep Code Search. *Applied Sciences* 12, 4 (2022), 2051.

[128] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2023. deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 34:1–34:27.

[129] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2018. Expanding Queries for Code Search Using Semantically Related API Class-names. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1070–1082.

[130] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 55:1—55:69.

[131] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *CoRR* abs/2312.15223, 1 (2023), 1–57.

[132] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *CoRR* abs/2310.08879, 1 (2023), 1–12.

[133] Wei Zhao and Yan Liu. 2022. Utilizing Edge Attention in Graph-Based Code Search. In *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc., Virtual Conference Center, USA, 60–66.

[134] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An Overlapping-Aware Code Retriever. In *Proceedings of the 35th International Conference on Automated Software Engineering*. Institute of Electrical and Electronics Engineers, Melbourne, Australia, 883–894.