UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2021-050
The Faculty of Science, Technology and Medicine

Dissertation

Presented on the 07/07/2021 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Kisub KIM
Born on 7^{th} SEPTEMBER 1988 in Cheongju, Korea

Steps Towards Semantic Code Search

Dissertation Defense Committee

Dr. Tegawendé Bissyandé, Dissertation Supervisor
Associate Professor, Université du Luxembourg, Luxembourg

Dr. Yves Le Traon, Chairman
Professor, University of Luxembourg, Luxembourg

Dr. Dongsun Kim, Vice Chairman
Assistant Professor, Kyungpook National University, Korea

Dr. Lingxiao Jiang
Associate Professor, Singapore Management University, Singapore

Dr. Xin Xia
Director of the Software Engineering Application Technology Lab at Huawei, China

Dr. Jacques Klein, Expert
Associate Professor, Université du Luxembourg, Luxembourg

Abstract

Code search can be a core activity in software development for enhancing productivity. Developers commonly reuse existing source code fragments by searching for codebases available in local or global repositories. Code search helps developers ease the implementation by supplying code snippets to reuse or understand specific concepts deeper during software development by providing various code snippets for the same tasks. In addition, reading real-world examples (the results of code search) is helpful for developers to make programs more reliable, faster, or secure as the examples have been tested and reused by many other developers. However, it is getting more challenging as the codebases are becoming larger since the large codebase can derive too many code candidates. Thus, the research community has invested substantial efforts in developing new techniques, combining methods, and applying more extensive data to improve the performance and efficiency of code search.

Despite the significant efforts made by researchers in the field, code search still has many open problems that the community needs to address, such as lack of benchmarks, vocabulary mismatch (between natural language and source code), and low extensibility on programming languages. Our work focuses on the open issues and the momentum of the domain on semantic code search, which considers the meaning of the user query rather than concerning the syntactic similarity that most other studies have approached. The thesis begins with exploring general issues on code search by conducting a systematic literature review. The survey organizes and classifies the code search approaches with various directions such as learning-based, feedback-driven, dynamic techniques. It reveals insights and new research directions. Given the research directions by the survey, we concentrate on alleviating the vocabulary mismatch problem between free-form text query and source code to improve the overall performance of code search first. To understand the free-form text query, we leverage crowd knowledge. The survey also discovered that there are only a few code-to-code approaches and investigation on crowd-knowledge indicated there exists demand, especially on finding semantically similar source code, i.e., source code that is syntactically different but performs the same functionality. Therefore, we go further, reformulating the user code query with real-world code snippets. This allows catching the semantics from the source code. Given the semantic information, a user can search for desired source code by using their code fragments.

In this context, the present dissertation aims to explore semantic code search by contributing to the following three building blocks:

- *Review of state-of-the-art*: Despite the growing interest in code search, a comprehensive survey or systematic literature review on the field of code search remains limited. We conducted a large-scale systematic literature review on the internet-scale code search. Our objective in this study was to devise a grounded approach to understand the procedure for the code search approach. We built an operational taxonomy on top of each procedure to categorize the approaches and provide insights on the selection of various approaches. Our investigation on the open issues from the literature guide researchers and practitioners to future research directions.
- *CoCaBu*: Source code terms such as method names and variable types are often different from conceptual words mentioned in a search query. This vocabulary mismatch problem can make code search inefficient. We presented COde voCABUlary (CoCaBu), an approach to resolving the vocabulary mismatch problem when dealing with free-form code search queries. Our approach leverages common developer questions and the associated expert answers to

augment user queries with the relevant but missing structural code entities to improve matching relevant code examples within large code repositories. To instantiate this approach, we built GitSearch, a code search engine, on top of GitHub and Stack Overflow Q&A data. Experimental results, collected via several comparisons against the state-of-the-art code search and existing online search engines such as Google, show that CoCaBu provides qualitatively better results. Furthermore, our live study on the developer community indicates that it can retrieve acceptable or attractive answers for their questions.

- *FaCoY*: Most existing approaches focus on serving user queries provided as natural language free-form input. However, there exists a wide range of use-case scenarios where a code-to-code approach would be most beneficial. For example, research directions in code transplantation, code diversity, patch recommendation can leverage a code-to-code search engine to find essential ingredients for their techniques. Given the wide range of use-case for code-to-code search, we propose FaCoY, a novel approach for statically finding code snippets that may be semantically similar to user input code. FaCoY implements a query alternation strategy: instead of directly matching code query tokens with code in the search space, FaCoY first attempts to identify other tokens, which may also be relevant in implementing the functional behavior of the input code. The experimental results show that FaCoY is more effective than all the existing online code-to-code search engines, and it can also be used to find semantic code clones (i.e., Type-4). Moreover, the results proved that FaCoY could be helpful in code/patch recommendation.

"Everybody is a Genius. But If You Judge a Fish by Its Ability to Climb a Tree, It Will Live Its Whole Life Believing that It is Stupid" Albert Einstein

Acknowledgements

This dissertation would not have been possible without the support of many people who, in one way or another, have contributed and extended their precious knowledge and experience in my PhD studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my supervisor, Assoc. Prof. Tegawendé Bissyandé, who has given me this great opportunity to come across continents to pursue my doctoral degree. He was the one who showed me passion, opportunity, intuition, and the future.

Second, I am equally grateful to my daily advisor Asst. Prof. Dongsun Kim, and co-advisers, Prof. Yves Le Traon and Assoc. Prof. Jacques Klein who have introduced me to the world of Code Search. Since then, working in this field is just joyful for me. They have taught me how to perform research, write technical papers, and conduct fascinating presentations. Their dedicated guidance has made my PhD journey a fruitful and fulfilling experience. I am pleased about the friendship we have built up during the years.

Third, I would like to extend my thanks to all my co-authors, including Prof. Woosung Jung, Prof. David Lo, Asst. Prof. Eunjong Choi, Asst. Prof. Li Li, Asst. Prof. Kui Liu, Asst. Prof. Anil Koyuncu, Mr. Sankalp Ghatpande, and Mr. Jaekwon Lee for their valuable discussions and collaborations. It was always passionate, insightful, and pleasant conversations with them.

I want to thank all my PhD defense committee members, including Assoc. Prof. Lingxiao Jiang, Asst. Prof. Xin Xia, my supervisor Assoc. Prof. Tegawendé Bissyandé, and my daily advisers Prof. Yves Le Traon, Assoc. Prof. Jacques Klein, and Asst. Prof. Dongsun Kim. It is my great honor to have them on my defense committee, and I appreciate very much their efforts to examine my dissertation and evaluate my PhD work.

I would like to also express my great thanks to all the friends I have made in the Grand Duchy of Luxembourg for our memorable moments. I will not forget my friends forever. Moreover, I would like to thank all the team members of TruX and SerVal at SnT for the peaceful coffee breaks with exciting discussions.

Finally, I would like to thank my father and mother for bringing the most considerable insights of the world and happiness to my life. The reason that I am the happiest person in the world is because they are my parents, and I know we love each other infinitely.

Kisub Kim
University of Luxembourg
July 2021

Contents

List of figures

List of tables

1 Introduction

In this chapter, we first introduce the motivation for steps towards semantic code search. Then, we summarize the challenges for both researchers and developers in semantic code search. Finally, we present the contributions and roadmap of this dissertation.

Contents

1.1 Motivation

Developing and maintaining software is still a time-consuming and complex task [73, 275]. Many practitioners rely on existing third-party libraries or frameworks to save time and alleviate the complexity. To do so, developers frequently explore the documentation and source code examples written by other developers for reuse purposes. On the other hand, it is not easy to locate specific code examples from the documentation due to their deficiency or poor quality. Moreover, code examples are overwhelmingly scattered in practice, disturbing the proper code selection to apply and plundering developers' time.

To aid these issues, researchers proposed the code search concept. Code search is an activity that developers can look for source code examples. In this way, the developers no longer need to look through all the details of the documentation or they do not have to open source code files to locate a source code snippet that performs a specific functionality. Furthermore, it helps developers to build the software faster, easier, and even reliable since they are often developed and tested by many other developers (e.g., development communities). Many code search approaches have been proposed [11, 14, 37, 82, 161, 163, 201, 353, 358, 478, 566] being based on textual or structural similarities. Furthermore, several internet-scale commercial code search engines [144] such as Krugle [271] and Searchcode [469] have been proposed. These engines also leverage either textual- or structural-based approaches to match the keywords from user free-form query and source code. Unfortunately, these commercial code search engines have an accuracy issue due to their textual matching, i.e., they treat source code as natural language (NL) documents. Source code, however, is written in a programming language, while most query terms are expressed in natural human language. As a result, searching source code using query keywords in NL often leads to irrelevant search results unless there is an exact keyword match with the keywords from programs. According to Hoffmann et al. [193], approximately 64% of developer queries for code search are merely descriptive but do not contain exact keywords of libraries, packages, classes, methods, etc.

As in any search engine, the terms in a code search query must be mapped with an index built from the code. Unfortunately, the construction of such an index, as well as the mapping process, is challenging since "no single word can be chosen to describe a programming concept in the best way" [138]. This is known in the literature as the vocabulary mismatch problem (VMP): user search queries frequently mismatch a majority of the relevant documents [138, 174, 601, 603]. This problem occurs in various software engineering research work, such as retrieving regulatory codes in product requirement specifications [94], identifying bug files based on bug reports [386], and searching code examples [174, 175, 188]. The vocabulary mismatch problem is further exacerbated in code search engines, where the source code may be poorly documented or may use non-explicit names for variables and method names [262]. Further research efforts are needed to address this semantic gap between the query terms and the terms from the relevant source code.

There exists a different perspective to resolve this semantic gap by using user code queries. Logically, code-to-code matching should be more feasible, and it finds alternative implementations of some functionalities. Recent automated software engineering research directions for software transplantation or repair constitute further illustrations of how a code-to-code search engine can be leveraged [319].

Although the existing code clone search approaches [25, 35, 227, 239, 270, 304, 312] identify similar code fragments, the contemporary static approaches still miss finding such fragments which have similar behavior even if their code is dissimilar [233]. Researchers have relied upon dynamic code similarity detection to find similarly behaving code fragments, which consists of identifying programs that yield similar outputs for the same inputs. State-of-the-art dynamic approaches generate random inputs [228], rely on symbolic [297] or concolic execution [273], and check abstract memory states [256] to compute function similarity based on execution outputs. The most recent state-of-the-art dynamic code clone search focuses on the computations performed by the different programs and compares instruction-level execution traces to identify equivalent behavior [501]. Although these approaches

can effectively find semantic code clones, the dynamic execution of code is not scalable. It also implies several limitations for practical usage (e.g., the need for exhaustive test cases to ensure confidence in behavioral equivalence).

Except for the low accuracy caused by semantic gaps, individual developers' different code element (e.g., library) selection, and dynamic approaches' scalability issue, current code search approaches still have different limitations to overcome. Our investigation reveals that code search has extensibility issues related to various programming languages that imply "there is no silver bullet" in code search. It lacks consensus that indicates current code search engines lack to consider various computing resources (e.g., one needs to minimize the memory consumption while the other should prioritize the secure source code). Furthermore, replicability of code search approaches is also a problem as Liu et al. [313] emphasized that only 18% of the approaches have an accessible link for the replication packages in their survey.

Code search holds the promise of reducing developers' efforts, especially on the implementation, by providing source code written by other developers. This is why, in production, code search is the most frequent activity of developers. In the literature, indeed, code search has shown to be effectively applied to software development. Nevertheless, there still exist open issues that need to be addressed.

In this dissertation, we explore code search approaches from two major perspectives: 1) general procedure of code search that researchers and practitioners should bear in mind to select the appropriate one to research or apply, i.e., the correct decision reveals appropriate code snippets; and 2) semantic code search which still has open issues.

1.2 Challenges of Code Search

In this section, we present the technical challenges we encounter while stepping towards semantic code search. Specifically, we discuss the challenges related to a code search approach (i.e., general view) and extracting desired concepts of users for semantic code search. Specifically, general challenges consist of a lack of existing benchmarks, extensibility, consensus, usability, and replicability. We present the vocabulary mismatch for NL query to source code and code query to source code from the search base for understanding semantics.

1.2.1 General Challenges in Code Search

- **Complication on selection of code search.** Given a wide variety of existing approaches, developers who search for source code can be confused because they cannot be sure which one is the best for their specific tasks. For instance, developers who are new to their field need a code search engine that can adequately reformulate their queries because they cannot formulate the query based on the task's context. For example, some developers need users' feedbacks to meet the complex requirements and considering points. They can leverage the interaction that the engine provides to address the task step by step.
- **Lack of Benchmarks.** Although researchers and practitioners all know that a more general and trustworthy conclusion for the code search will be drawn with a high-quality benchmark [14], there is still a lack of a benchmark for code search approaches. We observed that comparisons of existing approaches are common in evaluating code search approaches during the review process. However, many of them collected their datasets, and they re-implemented the target approaches and applied their datasets to compare.
- **Extensibility.** Software development often uses multiple programming languages to build one full product. Consequently, code search approaches do not always prove to be a good solution because of their limitation towards specific programming languages. The extensibility of code

search approaches towards all viable programming languages is an essential issue in the domain. Being able to apply a particular search approach towards different programming languages would provide more usefulness and convenience.

- **Consensus.** The result from a code search engine is not always what a user expected. Different approaches rarely consider different contextual requirements of the users. For example, a user looking for a code snippet that is memory efficient might not consider just a regular code snippet as an answer as it fails to satisfy the requirement of being memory-optimized.
- **Usability.** Specific code search approaches such as the binary code search are limited in its usability. All the executables of software or component are made of binary code. The analysis of such executables and their underlying binary code is crucial in information security to detect vulnerabilities that would adversely affect its working. Furthermore, even the use of binary code as dataset requires a strong assumption that they are not obfuscated. Obfuscated binaries make it extremely difficult to perform any form of code matching or abstraction. Additionally, certain programming languages such as C or C++ are capable of hardware specific optimization that further complicates the effects for any form of code matching.
- **Replicability.** Most code search approaches do not have publicly available replication packages. This poses an obstacle for developers who need to apply such an approach. A shared replication package helps developers in two significant ways: 1) first being a time-efficient way to deploy and test the approach; 2) a reference implementation to check for errors and issues. Re-implementation of an approach is a time-consuming task, even if the approach is well-explained. It can also derive potential errors where the performance is different from the reported one.

1.2.2 Challenges on Understanding Semantics

The ideal code search approach should understand the meaning of the user query and retrieve the best appropriate code snippet. However, for the NL-to-code case, every user has their own query style, and various scenarios can be drawn as follows: 1) one does not know what to search for; 2) another one does not know how to describe the question; and 3) the other knows everything except the specific name of the desired library. In this scenario, the critical obstacle is different terms between NL description and source code. Fundamentally, source code is structured following a specific rule and keywords that indicate it is difficult to retrieve relevant results with a simple matching process. Although the researchers proposed to address this by leveraging enormous synonyms from the dictionary, API documentation, etc., the issue remains and retrieves unsatisfiable results since understanding the semantic still lacks. For the other scenario called code-to-code case, i.e., a developer queries using a code fragment. This can be work better since both sides use the same concept of writing (source code). However, each developer has different library preferences and even different word choices, making the code search impenetrable.

1.3 Contributions

We now summarize the contributions of this dissertation as below:

- *Systematic Literature Review on code search*: We aim to provide a clear overview of the state-of-the-art in the field of code search by analyzing the trend, pinpointing the key techniques applied, and reveal the challenges encountered by the code search approaches and the research directions for further improvement on code search. With this direction, we conduct a Systematic Literature Review (SLR) where we identified 136 code search approaches published in principal conferences and journals until 2020. We exhaustively analyze and review each paper to find insights. We successfully reveal the challenges that code search researchers should contemplate deeply to boost the field with such insights. Finally, we propose several future research directions based

on the challenges previously revealed, including broad issues such as continuous increases of software repositories and specific like vocabulary mismatch, extensibility, and consensus issues. *This work is a technical report that will be submitted to a journal in 2021.*

- *Semantic query augmentation for user free-form query*: We discovered that many code search approaches focused on alleviating the vocabulary mismatch problems but we found out that such problems remain unresolved from the systematic literature review. Our investigation discloses that the source code terms such as method names and variable types are often different from conceptual words mentioned in a search query. This vocabulary mismatch problem can make code search inefficient. We presented COde voCABUlary (CoCaBu), an approach to resolving the vocabulary mismatch problem when dealing with free-form code search queries. Our approach leverages common developer questions and the associated expert answers to augment user queries with the relevant but missing structural code entities to improve matching relevant code examples within large code repositories. To instantiate this approach, we built a research prototype, GitSearch, a code search engine, on top of GitHub and Stack Overflow Q&A data. Experimental results, collected via several comparisons against the state-of-the-art code search and existing online search engines such as Google, show that CoCaBu provides qualitatively better results. Furthermore, our live study on the developer community indicates that it can retrieve acceptable or attractive answers for their questions.
  This work has led to a research paper published on the Empirical Software Engineering (EMSE) in 2018.
- *Semantic query reformulation for user code query*: Our investigations from the literature review on existing approaches revealed that most of them focus on serving user queries provided as NL free-form input. However, there exists a wide range of use-case scenarios where a code-to-code approach would be most beneficial as well as we catched the demand from the software development community by finding questions like "How to correct this code?". Research directions in code transplantation, code diversity, patch recommendation can leverage a code-to-code search engine to find essential ingredients for their techniques. Given the wide range of use-case for code-to-code search, we propose FaCoY, a novel approach for statically finding code snippets that may be semantically similar to user input code. FaCoY implements a query alternation strategy: instead of directly matching code query tokens with code in the search space, FaCoY first attempts to identify other tokens, which may also be relevant in implementing the functional behavior of the input code. The experimental results show that FaCoY is more effective than all the existing online code-to-code search engines, and it can also be used to find semantic code clones (i.e., Type-4). Moreover, the results proved that FaCoY could be helpful in code/patch recommendation.
  This work has led to a research paper published on the 40^{th} *International Conference on Software Engineering in 2018 (ICSE'18).*

1.4 Roadmap

Figure 1.1 illustrates the roadmap of this dissertation. Chapter 2 gives a brief introduction on the necessary background information including developers reuse activity by using code search, the procedure of code search, and semantic code search. In Chapter 3 reports a systematic literature review on code search to deepen the knowledge and provide insights on the overall background as well as the state-of-the-art. In Chapter 4, we present CoCaBu, an approach to resolving the vocabulary mismatch problem when dealing with free-form code search queries. In Chapter 5, we propose FaCoY, a novel approach for statically finding code snippets that may be semantically similar to user input code after a semantical reformulation. Finally, in Chapter 6, we conclude this dissertation and discuss some potential future works.

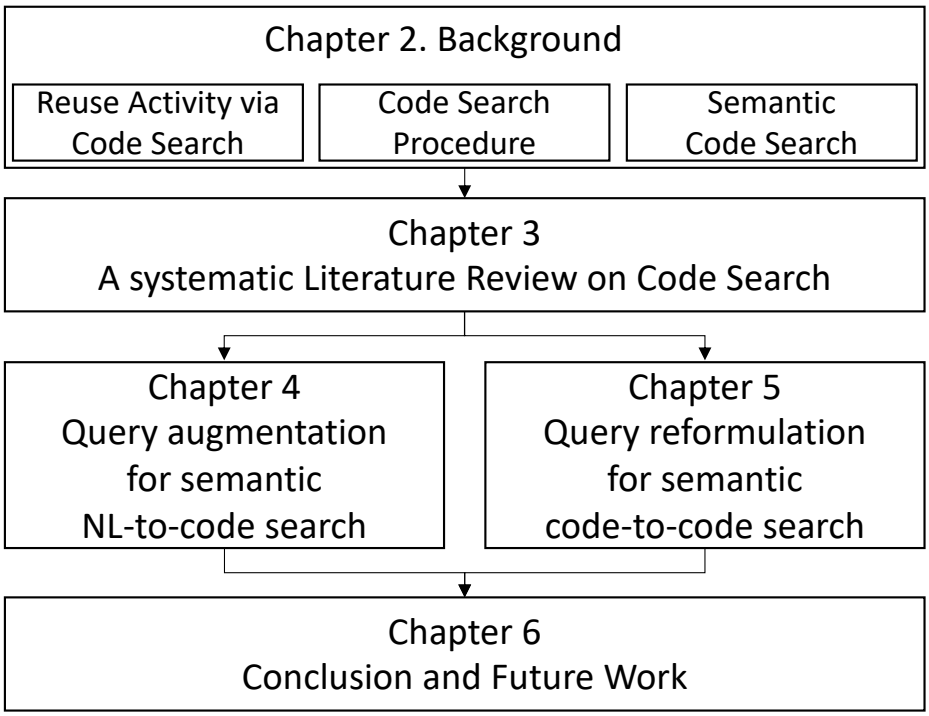


Figure 1.1: Roadmap of This Dissertation.

2 Background

This chapter provides the necessary background to understand the purpose, key concerns, and technical details of the three research studies we conducted in this dissertation. Specifically, we revisit some details and basics of code search, general procedure, semantic code search, and the datasets involved in this dissertation. Code search holds the promise of reducing developers' efforts, especially on the implementation, by providing source code written by other developers. This is why, in production, code search is the most frequent activity of developers. In the literature, indeed, code search has shown to be effectively applied to software development.

Contents

2.1 Code Search

Code Search is an activity that takes queries from users and retrieves the most relevant, appropriate, and ranked source code snippets. Many programs consist of developing routines, data structures, and designs that are also implemented in other programs [142]. This indicates that developers apply many code snippets written by other developers to their development tasks. Therefore, code search became a hot research topic, and there exist various approaches in the literature. We describe the essential components of code search and state-of-the-art code search in this section.

2.1.1 Essential Components

Figure 2.1 shows the basic concepts and main components of the code search. We now detail the major parts in the below list.

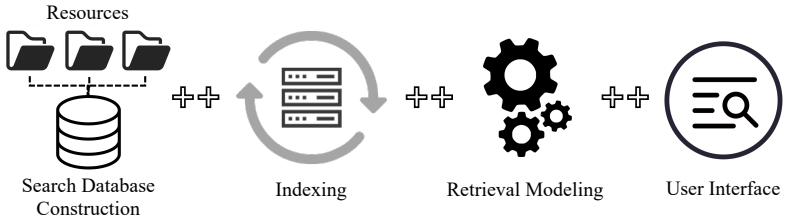


Figure 2.1: The Basic Code Search Components.

- **Search Base Creator** A search base is a source code repository or a dataset where a code search approach searches for and retrieves code snippets from. A general search base creator consists of web crawlers for collecting source code files or other resources such as API documentation, dictionary data, or question & answer pairs. Researchers and practitioners decide the type of search base (e.g., relational database, file database, or text files) and design the search base in this step.
- **Indexer** Indexing is a process that is used to optimize certain accesses towards data managed within its search base. Indexers for code search also leverage several techniques to speed up source code retrieval. For example, the most popular and classical indexing technique is inverted indexing, an index data structure containing a mapping from content such as words to its locations in a document or a set of documents. There exists other indexing techniques like B+ Tree indexing, Graph indexing, ID-based indexing, and Positional indexing.
- **Retrieval Model** A retrieval model is the crucial component of a code search approach. It predicts and explains what a user will find relevant given the input (i.e., it matches the query and the source code from the search base). Moreover, different retrieval models usually categorize code search approaches as they take the principle position behind them. Various retrieval models can be classified into three categories: matching text, matrix computation, or vector similarity measure depending on the specific techniques.
- **User Interface** The user interface is the last essential component that interacts with the users. It presents the results in an appropriate platform. As each user has different requirements and feasibility, code search approaches should be provided in different platforms such as independent code search engines or integrated into a specific development environment (IDE). Independent code search engines can either be local or online to interact with the users, while some others are implemented as a plugin of different IDEs allowing them to leverage the search within the development process.

2.1.2 State-of-the-art Code Search

Since the milestone work of Bajracharya et al. 15 years ago, code search has progressively become a hot research topic. According to our investigation, this research area has the most extensive publications in the recent two years among the entire period [259]. Various approaches have been proposed in the literature, which can be compressed into three categories: code search based on static information, code search based on dynamic information, and learning-based code search.

2.1.2.1 code search based on static information

Code search approaches based on static information broadly can be classified into three categories; 1) accounting code (i.e., keywords) as text to match, 2) leveraging structural information of source code, and 3) investigating semantic features that can be extracted from source code [259].

Code search can leverage static information such as text in source code, which is similar to general-purpose search engines [156, 364, 579]. For example, a code search tool can make an index of class names, variable names, parameter types as well as comments. The approaches with these information are called Keyword-based Code Search (KBCS). The KBCS approach [135, 378, 380, 518] was based on retrieving code snippets without considering any code-specific information (e.g., structure, sequence, etc.). KBCS approaches leverage additional features such as Abstract Syntax Trees (ASTs) [478] to extract the keywords from the source code to match against the user query. In addition to tokens in source code, it is able to use tokens in other sources such as API documentation, which are linked to source code. Code search inevitably suffers from the vocabulary mismatch problem [482] as the languages used for describing and implementing functionalities are often different from each other. To deal with this intrinsic problem, many code search engines [82, 161, 163, 353, 358] leverage API documents. Others leverage some other techniques such as weighting [594] or summarization [220] in KBCS.

Structural information of source code can be beneficial for code search as source code is different (i.e., each programming language has its own pre-defined structures and dependencies, which KBCS approaches ignore) from NL. For instance, an early approach named Prospector [339] employs a graph search to synthesize chains of objects and methods calls. Following this feasibility, majority of structural based code search engines [14, 78, 97, 300, 339, 356, 359, 415, 523, 546, 549, 551] leverage graph-based techniques with various types of graphs such as Api Signature Graph (ASG), Directed Acyclic Graphs (DAG), Function Call Graph (FCG), System Dependence Graph (SDG), Program Dependence Graph (PDG), Method Call Relationship (MCR) graph, and Program Expression Graph (PEG). Other techniques [5, 78, 97, 323, 356, 359, 452, 453, 549, 551] for structure-based code search are object instantiation, tree similarity, hierarchical information (e.g., parent and children types), Spreading Activation Network (SAN) [98, 99, 104], Dependence Query Language (DQL) [551], PageRank [403], etc.

Keyword-based approaches and code search based on structural information are limited since they ignore the semantic concepts on either user query and source code. Researchers [5, 11, 186, 261, 300, 314, 354, 379, 480, 482, 483, 607] investigate and extract semantic features to improve the performance and accuracy of code search approaches. In particular, to conduct semantic code search, several Natural Language Processing (NLP) techniques such as Latent Dirichlet Allocation (LDA), phrasal concepts (PC), Locality Sensitive Hashing (LSH) [110], degree-of-interest (DOI) [251], Softpedia [487] (a feature description service), fuzzy matching, as well as software-related models like Software Word Usage Model (SWUM [184]), Library-Sensitive Language Model (LSLM) [607] are leveraged. Moreover, to retrieve semantically similar code snippets, recently researchers [261, 480, 482, 483] leverages the online posts from developer Q&A forums (e.g., StackOverflow) as a semantic translator.

2.1.2.2 code search based on dynamic information

Unlike static approaches such as KBCS, code search approaches with dynamic information [143, 194, 196, 198, 213, 224, 229, 253, 278–280, 286, 289, 427, 428, 430, 502, 552] takes inputs (e.g., test cases or interfaces) and executes the program inside the search engine to value the candidates for qualitative retrieval. By using dynamic information, code search engines can assist developers who lack the expertise of the desired code [143] and non-native speakers of the language in which the repository is based [286], assure faster retrieval of code snippets [278], provide more guarantee the correctness of the behavior of retrieved code snippets [278].

2.1.2.3 Learning-based code search approaches

Learning-based approaches are classified into two categories based on typical machine learning or deep learning algorithms. These approaches mainly based on mapping between either NL and code, or code and code embeddings into a common space (e.g., vector space or matrix).

Typical machine learning approaches [4, 168, 203, 225, 241, 389, 393, 508, 564, 609] leverage techniques like collaborative filtering [503], X-Means algorithm [407], multinomial logistic regression [55], Support Vector Machine (SVM) [102], graph embedding techniques [160], graph kernel (code as object usage graph [394]), word2vec [366], and Markov Random Fields (MRF) [362].

Many recent studies have taken steps towards enabling more advanced code search by leveraging techniques based on neural networks [71]. From the early stage, code search approaches [167, 308, 586] started to leverage such techniques to alleviate the semantic gap between high-level intent from the user query and the low-level implementation details in the source code. These approaches have learning models that discover the correlations from the dataset and usually embed code snippets and the user query into a common hyperspace. The final phase for retrieving is composed of computing similarity between the vectors using similarity functions such as cosine similarity [459]. Various techniques for neural networks such as fastText [71, 308, 449], Recurrent Neural Network (RNN) [167, 308, 316], Convolutional Neural Network (CNN) [472], Long Short-Term Memory (LSTM) [167, 214, 308, 432, 466, 472, 541, 586, 588, 606], Auto-Encoder (AE) [88], Transformer [550, 590], Feed-Forward Neural Network (FFNN) [137], and Gated Graph Neural Network (GGNN) [541] are leveraged by the researchers.

In this dissertation, our studies are primarily limited in analyzing global(internet)-scale code search. Although knowledge of other approaches such as intra-scale code search which is also known as feature/concept location [121] are also important. For example, feature/concept location might be more important for cases like finding a specific bugs in one software. In this case, having the data rather than such specific software (i.e., global-scale) might hinder to locate the correct code snippets. In particular, we explore the overall code search approaches during conducting a systematic review and then focus on semantic code search by adopting query reformulation techniques. These let code search engines adapt to human natural language and then retrieve the best appropriate code snippets.

2.2 General Procedure of Code Search

This section describes the general procedure of code search for a better understanding of code search.

The general procedure of code search guides how a code search approach should be built and worked. The procedure consists of several steps: (1) selecting output type, (2) creating search base, (3) indexing data, (4) formulating input (query), (5) building retrieval model and retrieve, and (6) presenting the results.

Providing a general procedure can derive a baseline (i.e., how to design a code search approach) for researchers while developers can understand the characteristics of each step and figure out the best

approach to apply or use to conduct their tasks. In other words, this should serve as a practical guide to researchers who are new to this field and developers who are confused about picking a code search approach.

Depending on the design of each step in the procedure, a code search approach can have a significant difference in terms of various performances. For instance, Creating a search base with good quality code snippets can improve the baseline performance, which implies poor quality code lead to poor results even though the other steps are well-designed. Formulating the query with a specific query language may help avoid a well-known vocabulary mismatch problem to enhance the code snippets' relevancy. Designing the retrieval model with learning-based approaches shows a significant improvement in accuracy, while classical models are still much faster to retrieve code snippets.

In this dissertation, we investigate, analyze, and provide the details (e.g., different techniques for each procedure) with the overall taxonomy of code search.

2.3 Dataset

This section presents three categories of datasets related to code search that are fundamentally important to the research community and this dissertation.

2.3.1 Open Repositories

In the last decades, there has been widespread use and adaptation of open source software. This has created thriving open-source software development communities that leverage collaborative coding systems and are made centrally available for everyone to use. Researchers within the code search have adopted the data sources that form a large set of different open source projects when testing their approaches (e.g., [14, 523]).

The leveraged data source, known as the dataset, is in a file archive, where most of the source code files are extracted from different software projects. These repositories are often accessible online on code hosting services such as `Github` [557], `Sourceforge` [488], `Bitbucket` [53], etc. Software repositories are a rich source of code snippets created and curated by developers around the globe. Furthermore, the curated source code snippets in the form of datasets provide opportunities to investigate and research new ways for code search techniques.

Representatively, many research prototypes [261, 334, 346, 349, 368, 376, 419–422, 482, 497, 552, 585] have utilized source code files collected from the code repository platform `Github` as their search base. `Github` is one of the largest super-repositories built on top of the `Git` VCS [27]. It is the most prominent hosting service [426] with more than 100 million repositories hosted as of January 2020.

2.3.2 Other Data Sources

There exist other resources that have been leveraged for different approaches towards the code search problems. Resources such as the developer Q&A forums, metadata, and API documents are applied for many code search approaches [11, 221, 248, 261, 284, 301, 335, 345, 368, 395, 398, 421, 482, 494, 497, 498, 536, 537, 594]. Such additional resources have demonstrated an increase in the code search techniques' efficiency and accuracy.

Question and Answer (Q&A) forums are community-driven platforms that allow users to share knowledge with other users who participate in them. Q&A forums offer social mechanisms to evaluate and improve the quality of both the question and answer that implicitly leads to brevity in questions and qualitative answers, potentially with source code snippets [261]. Many of the Q&A forums

determine a user's reputation by scores awarded by others, based on the clarity and correctness of the answers [1]. These forums have recently gained immense popularity having over 10 million users as of January 2020. The posts within such forms tend to include code snippets within the question and its answer. It makes them ideal for forming the part of the dataset for some code search approaches. A majority of the code search approaches with Q&A dataset have utilized the `Stackoverflow`. `Stackoverflow` is the largest Q&A forum that mainly contains questions and answers on programming-related topics [159]. Answers from `Stackoverflow` often are an alternative explanation for corresponding official product documentation wherein the documentation is either insufficient, does not exist, or lacks in-depth information [32]. As of January 2020, the public data dump [491] lists include a total of 176 communities (categories such as LaTeX, Linux, etc.), 58 million posts, and over 10 million registered users. A significant portion of answers includes code snippets that demonstrate the solution for the corresponding programming problem. These solutions may even contain information about the usage of a particular function from either a library or a framework [261, 528, 581]. Recently, large and systematically mined dataset using Stackoverflow are proposed and learning-based code search approaches [114, 177, 204, 542, 600] leveraged these in their evaluation. For instance, CoNaLa dataset [591] by Yin et al. consists of two parts, a manually curated parallel corpus of 2,379 training and 500 test examples, and systematically mined 600k pair examples. The pairs include Python code snippets and their corresponding annotations in natural language. Unlike CoNaLa, StaQC [587] is specifically intended for code search (i.e., not for code generation or summarization) and it consists of 148K systematically mined Python and SQL question-code pair dataset. It is mainly composed of straightforward "How-to-do-it" questions and one-to-one alignment to raise usability.

Software project metadata includes information such as the project name, owner, description, homepage, rating of the project, creation date, code commit dates, license of files, the number of developers, etc. [44, 51, 100, 126, 172, 450]. Code search empirical studies [145, 171] have leveraged project metadata by claiming that metadata can play an important role in the design and implementation of code search approaches. It can be used to restrict the result sets [60, 245], supplement searching accuracy along with user query [356], change the search range in the sense of iterative search [346], and give its users a better understanding with more information of projects [261, 482].

Some code search studies [82, 258, 345, 352] leveraged well-known Application Programming Interface (API) documentation. Generally, developers refer to the API documentation for examples [301] of a concerned project where such documents are the official source of information, such as the `JavaDoc` [400] or `MSDN` Library [363]. These documentations are known and followed by all vendors for consistencies usually written by multiple people, undergoing multiple reviews based on feedback from other developers [116]. The API documentation also serves as the natural bridge between the natural language query and API methods invoked in code snippets [301].

2.3.3 Existing Benchmarks

A benchmark is a package that serves as a basis for evaluation or comparison. Many studies on code search collected their datasets from various sources and incorporated several evaluation metrics. Approaching this direction leads to inconsistencies for fair comparison because their search base has different snippets, and different metrics may indicate different results. Furthermore, different queries for the same task make the retrieved results vary. These problems motivate researchers to build benchmarks for code search using common search bases and metrics or to have the same queries and answer code snippets. Therefore, the benchmarks provide the dataset, metrics, and sample results that lead the users to a fair comparison in the code search evaluation phase.

There are several code search benchmarks [292, 580] proposed recently. For instance, Li et al. [292] introduced a benchmark in the field of code search by providing the results of their neural code search approaches [71, 449]. Such benchmark consists of the dataset as a natural language query (from

Stackoverflow) and code snippet pairs about Android software development (from Github). Another benchmark CosBench [580] combines a search base (from GitHub), query and answer pairs (from Stackoverflow), and a set of metrics. A deep learning approach [298] showed the first usage in the evaluation for code search.

3 Big Code Search: a Bibliography

In this chapter, we aim at providing a clear overview of the state-of-the-art works around the topic of code search, in an attempt to highlight the main trends, pinpoint the main methodologies applied and enumerate the challenges faced by the code search approaches as well as the directions where the community effort is still needed. To this end, we conduct a large-scale Systematic Literature Review (SLR) during which we eventually identified 136 relevant research papers published in leading conferences and journals until 2020. This study further helps researchers and practitioners in selecting code search approaches that are the most appropriate for a specific task. To be more specific, this study devises a grounded approach in order to understand the procedure for code search, and build an operational taxonomy capturing the key facets of the code search procedure.

This chapter is based on the technical report that is ready for submission.

Contents

3.1 Introduction

Many programs consist of developing routines, data structures, and designs that are also implemented by other programs [142]. Developers are indeed recurrently writing code to address similar tasks or cloning (e.g., via copy/paste) other code. Towards easing software development, searching for source code on the internet became a typical activity for developers [32, 476, 497]. In particular, developers often search for source code to reuse or to consider as reference examples [145,476], to help them identify the programming concepts that are required for solving coding tasks [23, 63, 195, 347, 474] or to fact-checking (i.e., in contrast to exploratory usages [343]) on the availability of different implementations for a given algorithm. Furthermore, the core concept and principles of code search research is also applied to other software engineering tasks such as *concept/feature/concern location* [121], *code clone detection* [247], *code completion* [66, 436], *match and transform* [402], *vulnerability detection* [79, 578], *bug localization* [4], and *automatic program repair* [319].

Given the importance of code search in software development, the research community has invested substantial effort in the field, developing new techniques, applying new methods, and collecting new data to improve the efficiency and effectiveness of code search. In broad terms, code search is a procedure that is composed of a series of activities aiming to retrieve relevant code snippets according to the user specification. These activities include i) selection of the output type, ii) creation of a search base, iii) indexing the search base, iv) formulation of user specifications into search queries v) obtainment of code snippets relevant to the user query vi) demonstration of relevant results to users. The ability to characterize the properties of each activity remains crucial towards understanding the core concepts and principles of code search. To that end, we summarize the properties and characteristics of code search approaches and use them to establish a procedure for code search. Eventually, we propose a grounded approach to establish a standard procedure for code search and build an operational taxonomy on this procedure.

The task of understanding the process and studying the right approach is a challenging task. Lack of resources that may serve as a systematization of this knowledge that would provide a newcomer with adequate information is a problem we hope to address in this work. Yet, to the best of our knowledge, there has not been any attempt to undertake work that would address this issue and provide literature covering the code search domain in its entirety. Recently, Liu et al. [313] submitted a survey that focuses on the input and outputs of code search engines. Their work focuses primarily on understanding the publication trend within the domain that provides a segmentation between the types of publication undertaken, venues, and the trends. Furthermore, the survey focuses on synthesizing different metrics and modeling techniques and the different evaluation methodologies used in the domain. We believe that further in-depth analysis and systematization of the knowledge is a necessary and vital step. We undertake this work on code search, where we propose the first operational and classifiable taxonomy applicable throughout the domain. Concretely, our work provides many essential contributions such as:

- A novel systematic literature review on 136 code search approaches published until the end of 2020.
- A profound literature search strategy with snowballing to bring to light unrevealed approaches of code search.
- An overview of the field of code search and its historical evolution trend to highlight what has been done so far, and what needs to be done.
- Devising a general procedure of code search that can help researchers and developers understand fundamental concepts and principles of code search.
- An operational taxonomy of code search that can guide researchers and practitioners to locate code search approaches that are most suited to their tasks.
- Analysis of existing infrastructure and the dataset of code search.
- Discussion of the open issues and possible solutions.

In this paper, we consider a total of 136 internet-scale code search approaches and they are all based on multiple projects (i.e., at least two projects as a search base). This survey aims to provide the reader with a complete overview of the field, starting from the first known publication in the code search domain.

The rest of the paper is structured as follows: Section 2 investigates publication trend for the entire field of code search; Section 3 introduces the general and typical procedure of building code search engines; the overall taxonomy of the existing code search approaches are presented in Section 4; Section 5 and 6 include code search infrastructures and datasets, respectively; Section 7 deals with evaluation methods and metrics; We discuss the past, present, and future of code search in Section 8; Section 9 summarizes and concludes the survey.

3.2 Paper Collection and Review Schema

This section introduces the survey scope, the paper collection methodology, and the brief analysis of the collected papers.

3.2.1 Survey Scope

The scope of our paper targets comprehensive internet-scale code search engines that take input from users and then retrieve/recommend/suggest code snippets/examples or related information.

We apply the following criteria for the inclusion of papers in this survey.

- The paper that proposes or discusses a general idea of code search.
- The paper implements a code search/retrieval/recommendation engine/framework/infrastructure.
- The paper proposes an approach/study that targets specific code search techniques.
- The paper presents a dataset or benchmark especially designed for code search.

Some papers adopt code search engines to improve the performance for other research fields, such as code clone detection, which implies that such papers do not improve the field of code search. These papers are considered beyond the scope of our work. Furthermore, our survey excludes studies that introduce code search techniques but never retrieve code snippets or related information (i.e., we only cover end-to-end approaches). For instance, some approaches [207] improve the user query by reformulating them but, they do not retrieve code snippets. Such approaches are not being considered in the scope of the survey. Moreover, we excluded the papers that explicitly mention a single search base scope (i.e., approaches limited to a single specific software repository) as they are local approaches (e.g., feature location) rather than internet-scale/multi-project based code search.

Another criterion we have applied is discarding approaches that focus on detecting code and locating specific code elements. Even if the same approach is applicable towards searching, detecting, or locating a code element, its use-cases differ. There are code clone detection and concept/feature location techniques that very closely resemble the code search. Also, some code search engines leverage them to evaluate the performance. However, we do not consider code clone detection as code search. Their purpose is generally different from code search, and feature location is finding the concerned spot in a single project.

Sometimes, the literature of code search also consists of posters that present either work in progress or ideas by different authors at various venues. As most of the posters are later extended in the form of a conference paper or a journal, we do not consider posters in our study as separate ideas and/or approaches. However, we cover the papers that have elaborated an idea without performing a proper/complete evaluation as short papers.

Table 3.1: Overview of different publishing avenues for code search domain.

Venue Category	Short Name	Full Name	Count
Conference	ICSE	International Conference on Software Engineering	20
	ASE	Automated Software Engineering	13
	MSR	International Conference on Mining Software Repositories	8
	SUITE	Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation	5
	PLDI	ACM SIGPLAN Conference on Programming Language Design and Implementation	4
	OOPSLA	Object-Oriented Programming, Systems, Languages & Applications	4
	RSSE	International Workshop on Recommendation Systems for Software Engineering	4
	SANER	IEEE International Conference on Software Analysis, Evolution and Reengineering	3
	COMPSAC	Annual Computer Software and Applications Conference	3
	CSMR-WCRE	IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering	3
	ESEC/FSE	European Software Engineering Conference and Symposium on the Foundations of Software Engineering	2
	SAC	ACM Symposium on Applied Computing	2
	WWW	The World Wide Web Conference	2
	SBES	Brazilian Symposium on Software Engineering	2
	SCAM	International Working Conference on Source Code Analysis & Manipulation	2
	VL/HCC	Visual Languages and Human-Centric Computing	2
	ISSTA	International Symposium on Software Testing and Analysis	1
	MAPL	ACM SIGPLAN International Workshop on Machine Learning and Programming Languages	1
	FASE	International Conference on Fundamental Approaches to Software Engineering	1
	RecSys	ACM Conference on Recommender Systems	1
	ACIIDS	Intelligent Information and Database Systems	1
	UIST	ACM Symposium on User Interface Software and Technology	1
	WEH	International Workshop on Exception Handling	1
	SBCARS	Brazilian Symposium on Software Components, Architectures, and Reuse	1
	ACL	Annual Meeting of the Association for Computational Linguistics	1
	ICoICT	International Conference on Information and Communication Technology	1
	WSDM	ACM International Conference on Web Search and Data Mining	1
	ICCIT	International Conference on Computer and Information Technology	1
	CCS	ACM SIGSAC Conference on Computer and Communications Security	1
	RCoSE	International Workshop on Rapid Continuous Software Engineering	1
	Programming	International Conference on the Art, Science and Engineering of Programming	1
	Internetware	Asia-Pacific Symposium on Internetware	1
	MOBILESoft	International Conference on Mobile Software Engineering and Systems	1
	IWSC	International Workshop on Software Clones	1
	SERVICES	IEEE World Congress on Services	1
	ASC	ACM Southeast Conference	1
	ICSEW	International Conference on Software Engineering Workshops	1
	IJCNN	International Joint Conference on Neural Networks	1
	CIRCLE	CEUR Workshop	1
		Total Conference Venues	102
Journal	TSE	Transactions on Software Engineering	3
	EMSE	Empirical Software Engineering	3
	JSS	Journal of Systems and Software	3
	IEEE Access	IEEE Access	3
	SPE	Practice and Experience	3
	TOSEM	Transactions on Software Engineering and Methodology	2
	IST	Information and Software Technology	2
	TSC	IEEE Transactions on Services Computing	2
	SCIS	Science China Information Sciences	2
	ASE_Journal	Automated Software Engineering Journal	1
	JIFS	Applications in Engineering and Technology	1
	ISF	Information Systems Frontiers	1
	JPCS	Conference Series	1
	PACMPL	ACM on Programming Languages	1
	IEEE Software	IEEE Software	1
	KBS	Knowledge-Based Systems	1
	KIES	International Journal of Knowledge-based and Intelligent Engineering Systems	1
	WUJNS	Wuhan University Journal of Natural Sciences	1
	JIT	Journal of Internet Technology	1
	SEKE	International Journal of Software Engineering and Knowledge Engineering	1
		Total Journal Venues	34
		Total Number of Studies	136

3.2.2 Paper Collection Methodology

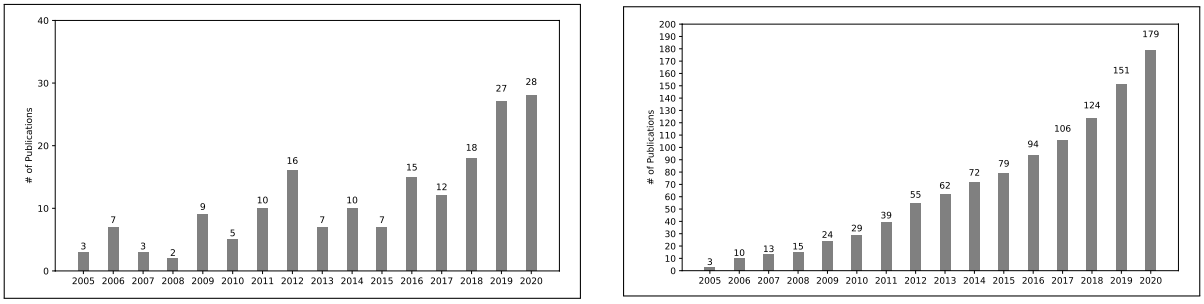
To collect the papers across different research avenues that would cover as many papers as feasible, we initiated the keyword search first on popular scientific databases. The databases are listed up as follows; ACM Digital Library[1], IEEE XPlore[2], DBLP[3], Springer Link[4], Wiley Online Library[5], Elsevier Online Library[6], and arXiv[7].

Researchers have used diverse keywords for the fundamentally identical concept (e.g., 'code snippet'-'code example' and 'retrieval'-'recommendation'). Therefore, we employed keyword combinations for code-related keywords (i.e., source code, code snippet, code fragment, code example) and search-related (i.e., retrieve, recommend, and suggest) for the text searching across the repositories and published until 2020.

To ensure that we cover all the internet-scale code search [145] papers and avoid confusion from missing papers, we conducted snowballing on each paper found by text searching following a well-known guideline [567]. Snowballing is a process that traces citations of papers continuously until there are no missing papers in a domain. The main idea of such a process is to avoid missing studies related but are not discovered by keyword search. This process allowed us to add studies that satisfy our inclusion criteria in Section 3.2.1. For instance, CodeLikeThis [347], Strathcona [194, 196, 198] are missing from the keyword search but found by such process.

3.2.3 Collection Results

Figure 3.1a shows the number of approaches that were published each year, and Figure 3.1b illustrates the cumulative trend. As these figures reports, the research within the code search domain gradually increased from 2006, reaching its peak in the recent years of 2019 and 2020. Finally, we ended up with 179 approaches code search approaches. We carefully exclude all the duplicate papers (i.e., journal first, short versions) for our taxonomy which reveals 136 approaches. These trends emphasize the importance of code search, and it is still in the growth.



(a) Number of papers published

(b) Cumulative number of papers published

Figure 3.1: Publication trend.

Figure 3.2 shows the distribution of papers published in different research conferences (either in the form of short or full papers), journals, or arXiv. Overall, a full paper in the conference covers by 41.9% of the total distribution, followed by short papers at 31.4%. The journals as the venue for publication occupy a 19.2%. Finally, the publications on arXiv are non-peer-reviewed but need to be considered as the venue used by major industrial R&D as a publication venue.

[1]https://dl.acm.org/
[2]https://ieeexplore.ieee.org/
[3]https://dblp.org/
[4]https://link.springer.com
[5]https://onlinelibrary.wiley.com/
[6]https://elsevier.com/
[7]https://arxiv.org/

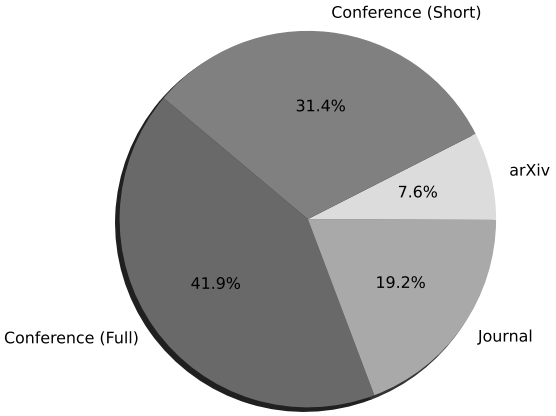


Figure 3.2: Publication venue distribution.

Table 3.1 lists the different conferences and journals that cater towards the code search domain. The International Conference on Software Engineering (ICSE) has been the target of 20 out of 102 overall approaches that target the conference as a publishing venue. Similarly, the journals Transactions on Software Engineering (TSE), Empirical Software Engineering (EMSE), Journal of Systems and Software (JSS), and IEEE Access are the most target journals in the domain. The public repository that contains all the publications is available at our Github page[8].

3.3 Code Search Engines - General Procedure

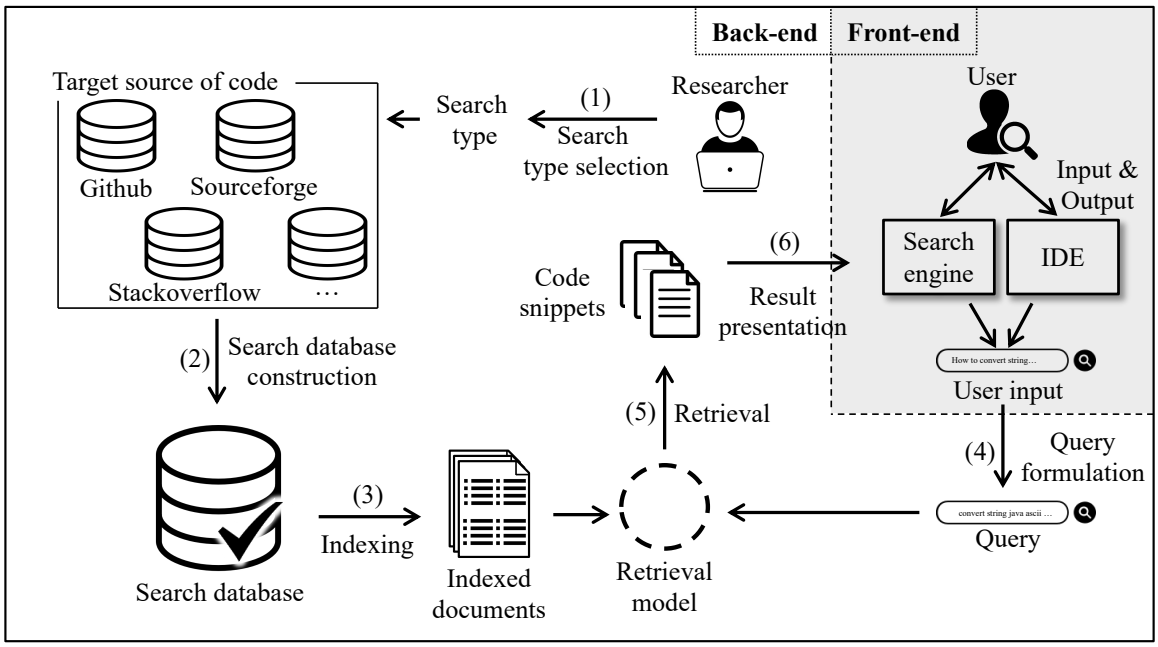


Figure 3.3: General Code Search Process.

The general procedure of code search guides how a code search approach should be built and worked. Figure 3.3 shows the process of general code search and a procedure of code search approach consists

[8] https://github.com/FalconLK/CodeSearch_Survey.git

of several steps: (1) selecting search type, (2) search base creation, (3) indexing data, (4) formulating input (query), (5) building retrieval model and retrieve, and (6) presenting the results.

Providing a general procedure can derive a baseline (i.e., how to design a code search approach) for researchers while developers can understand the characteristics of each step and figure out the best approach to apply or use to conduct their tasks. In other words, this should serve as a practical guide to researchers who are new to this field and developers who are confused about picking a code search approach.

Depending on the design of each step in the procedure, a code search approach can have a significant difference in terms of various performances. For instance, creating a search base with good quality code snippets can improve the baseline performance, which implies poor quality code lead to poor results even though the other steps are well-designed. Formulating the query with a specific query language may help avoid a well-known vocabulary mismatch problem to enhance the code snippets' relevancy. Designing the retrieval model with learning-based approaches shows a significant improvement in accuracy, while classical models are still much faster to retrieve code snippets.

Following the importance of the procedure in the field of code search, we focus on each step and deliver detailed information. These steps are also the base of our operational taxonomy in Section 3.4.

3.3.1 Search Type Selection

The first step to designing a code search approach is to decide the search type. Developers want to search for various types of code, such as some need code snippets related to a specific API while the others require interface-related codes. However, most developers search for available code snippets to address their problems or implement their tasks. These types vary by the desire of target users.

General Code Snippet: Generally, code search approaches retrieve the most relevant general code snippets rather than other source code types (e.g., API usages, GUI, or binary).

API Usage Example: Many software developments leverage the use of APIs that provide common functionalities towards smooth developments. The developers understand and learn about the API by referring towards the API documentation and tutorials (e.g., JDK), looking towards Q&A forums such as StackOverflow, or searching for code examples on super repositories such as Github. These resources are usually provided with search engines, but they are generally not optimized to search for examples of a particular API. Such usage to search for code examples is limited to identifying the targeted API's name without considering other elements such as API-type. This requires developing different search engines that cater to developers searching for target API and seeking to have usage examples for them. Many researchers (e.g., [12, 342, 368, 392]) proposed approaches that specifically aim to provide API usage examples for the given type of input. The API usage-based code search engines can be seen as a use-case for code search engines targeting a specific subset of code snippets.

GUI code: A code is used not just for writing software that provides specific functionalities; instead, it has also been used for user interface development that serves as the interface for user's interaction with underlying software. The development of the user interface (also known as Graphical User Interface [GUI]) consists of multiple widgets, dynamic user information resulting in a functional user interface in the form of a sketch that can be a picture, an SVG file, or an XML file. Due to the existence of different attributes in the GUI source codes, the traditional retrieval techniques cannot be effectively applied, and hence some researchers [39, 431, 574] proposed approaches that specifically target the GUI code search.

3.3.2 Search Base Creation

Search base is a repository or dataset where a code search approach can search for and take the code snippets relevant to user's specification. This is important because the overall performance of a search engine is influenced by the quantity and the quality of data [405]. Considering the importance of the data, we focus on how developers and researchers create their search base in this section and leave the description of the data itself used for code search in Section 3.6. Many approaches within the literature [261, 328, 423, 482] leverage the super repository such as the GitHub [557] for their search base because of such repositories being a rich source of community maintained source code.

Many studies [86, 134, 387, 411] implied that narrowing down the search base allows the search engines to hit more relevant results to the queries. To narrow the search base and get higher precision on the relevancy of code snippets, project metadata such as the programming language, creation date, popularity, etc., are further considered in the code search domain. For some approaches, the researchers [261] consider specific repositories that a certain number of users has starred to avoid toy projects that would contribute as noise within the data. Furthermore, [210] leverages information such as commit logs (e.g., commit messages) to improve their engines' performance by narrowing the search base or mining them for specific topics. A dataset from Q&A forums where questions are mapped with answers containing code snippets is used in many studies (e.g., [261, 299, 316, 422–425, 482]). Using only appropriate questions and correct answers (i.e., up-voted by the developer community) from such data can evaluate the quality of the search engine. It can be a supervised technique (i.e., as the correct answer exists per a question).

Most of them designed the structure of the repositories, crawled the data from the web, and organize them to use as the search base of a code search engine.

3.3.3 Index

Indexing is a process that is used to optimize certain accesses towards data managed within its search base. Code search approaches also leverage several indexing techniques to speed up source code retrieval.

Inverted Indexing: An inverted index is an index data structure containing a mapping from content such as words to its locations in a document or a set of documents. Inverted indexing is the most popular indexing technique that many search engines, like Google, employ in the real world. Many code search engines also take this technique to index their source code snippets in their search base. To apply the inverted indexing, a number of researchers [14, 261, 454, 482] utilized Lucene [558] that is a representative open source search engine library. This library is commonly used to demonstrate many code search techniques (e.g., where they do not need specific indexing techniques). As for another example, the inverted indexing technique has been applied to traditional SQL to address a problem that users cannot use INDEX with LIKE search. It implies that a code search that leverages the traditional SQL utilizes the inverted indexing behind.

B+ Tree Indexing: B+-Tree indexing is an alternative mechanism to index a sequential set of data elements. Databases such as MySQL leverages B+ Tree as their indexing method, which creates a specific structure from the data in the search base. A B+ Tree is primarily utilized for implementing dynamic indexing on multiple levels. It stores the data pointers only at the tree's leaf nodes, making the retrieval, addition, and deletion process more accurate and faster in code search. Some researchers [39, 143, 286] in code search have relied on this indexing for boosting speed performance.

Graph Indexing: Graphs have been used extensively to model the complicated structures and relationships between different entities within a programming code. Conceptually, graph structure can be used to accurately describe the relationship between two entities of a code, such as a class and

its method. Such conceptualization uses graph structures for searching code similarities, an ideal tool for code search approaches. For every query provided as input towards these approaches, it is transformed in graph structure used to retrieve similar related graphs from the repositories.

The storage of graphs requires the use of graph indexing for effective retrieval. Researchers [241, 502, 574] have leveraged the graph structure and its graph indexing techniques for code search engines. These approaches leverage the graph reachability index algorithm to derive a labeled directed graph where each node has a unique label assigned from a known label set. The algorithm indexes all the reachable relationships between node labels. Users can query and obtain all the instances of a reachability pattern with the constructed indices.

ID-based Indexing: ID-based (or File-based) indexing mechanism is used to optimize access to data managed in the form of a single access file or point. This type of mechanism typically leverages a specifically created index file that stores only the search key's value and a pointer towards the file's storage pointer. Within code search engines, the use of ID-based indexing means that for each of the potential searchable source code files, an index is stored that can point towards its storage index. Within code search engines, ID-based indexing creates an index for each potential searchable source code file. This index is the reference towards a files' storage index. Certain code search engines [72, 229, 523] leverage the ID-based index by assigning a unique searchable ID towards each file, class, or method name that can be matched for optimized retrieval.

Positional Indexing: Positional indexing is the mechanism that leverages the existence of tokens within a document. Each positional information (e.g., line number, the position of a token within a sentence) of every term is indexed and leveraged with the retrieval techniques.

3.3.4 Input (Query)

Formulating input to a query is one of the procedures related to the user side (also known as the Front-end). In particular, the client-side of code search starts with taking the input from the user. The form of input varies because each user has their own specific tasks. Such tasks include new development or maintenance (i.e., software refactoring and performance optimization in terms of the higher category. For instance, for developers who want to find a similar code to their code, a code search approach takes a code as the input is needed. If a developer has test cases to pass, the perfect-fit code search engine may take the test case as the input. According to these situations, it is crucial to design the input variety in code search. We report the inputs and their properties used to design code search approaches.

NL query: Developers often find it easier to formulate the query in natural language for the desired search describing the expected code snippet. A natural language (NL) query consists only of regular terms used in a user's language, without any specific syntax or format. An NL query allows a developer to enter terms in any form, either a statement, a question, a list of keywords associated with programming elements (class or method name). Many developers do not always have the knowledge or know-how about the technical expression required to express their search formats. Because of this, most code search approaches are designed for NL queries. Community-driven forums for developers such as Stackoverflow [492] and super repositories like Github [557] utilize NL-based search as input for the query of the search.

Code Fragment: We define "code fragment" as the source code that the search engine takes and "code snippet" as the source code that the search engine retrieves. A scenario that a developer inputs code fragments and takes code snippets is named code-to-code search in general. The code-to-code approach is beneficial for research directions in code transplantation, code diversity, patch recommendation to find essential ingredients for their techniques. One of the popular commercial engines, Krugle [271] has its snippet-based search in the advanced option, while SearchCode [469] is capable of taking input as both NL and code fragment in its main interface. Krugleś such

option appears to identify only identical code examples, and SearchCode yields similar code snippets according to an empirical investigation by Kim et al. [261].

Query Language: A query language is specifically designed to model the structure and properties (e.g., loops, method calls, and exception handling). Modeling such properties can express more complicated search patterns such as 'find all code examples that call the *ABC* method in a *loop* and handle an *exception* of type *abcException*'. In the case of typical text-based approaches, this cannot be accurately expressed [460, 486].

Binary Code: In some cases, developers may need to search for code similar to functions existing within a binary file. Developers mostly use the use-case of binary code search to search for critical vulnerabilities or find code snippets that implement the given binary input file's functions. Researchers have proposed approaches (i.e., [79, 111, 578]) that take a binary file as input and perform code search to retrieve code snippets that implement semantically similar functions.

Test Case: A test case is a programming entity that consists of different input types for a target code that tests the code's defined logic and the expected output. A test case is executed and reaches a state of pass if the output from the code and the provided output matches are considered to have failed. Test cases are used heavily in software development and form the basis of Test-Driven Development (TDD) [38]. In many cases, Test-Driven code search approaches take either test cases or a set of keywords that can define a required test case. Test cases are beneficial as they provide instant feedback about the suitability of a particular code result. The requirement for designing test cases guides the user in searching for self-contained and manageable software pieces.

Class/Interface Type: Similar to test cases, Interface-Driven code search (IDCS) approaches leverage the interface types existing within the input snippet. These interface or class types are similar to existing interface within a function such as '*string f(string)*' in Java. In the example, the types (as input of a code search) are 'string'. Some researchers [194, 196, 198, 284] have leveraged the use of interface or class type for matching reduces the potential search space and provide effective code search engines.

Note that IDCS is performed as a stage of TDCS: the interface of the desired method is extracted from the tests to find the initial set of candidates.

Software Specification: Software specification is what the user specifies as their requirement and condition towards the search. These specifications are a set of conditions imposed by the user on the retrieved result candidates and are expected to satisfy. The use of software specifications provides further criteria that a search engine applies over potential candidates to provide more relevant results. For example, a user may wish to retrieve code that only has a limited set of parameters or asynchronous only API code, etc. Software specifications have been leveraged by researchers [342, 498] to specify the security constraints (i.e., method pre- and post-conditions) for retrieving results that are aligned with user's needs.

3.3.5 Retrieval model

A retrieval model predicts and explains what a user will find relevant given the input (i.e., it matches the query and the source code). The correctness of the model's predictions can be validated in various experiments. Moreover, different retrieval models usually categorize code search approaches as they take the position of the principle behind them. The retrieval phase of code search generally consists of matching text, computation of matrix, or vector similarity measure depending on the specific techniques.

Textual Similarity: In code search, the textual similarity consists of matching tokens, keywords, or even a sequence of tokens with keywords between two code snippets. Several different models have been defined for use for textual similarity.

As for the standard models, the Boolean model and vector space model (VSM) are famous classical information retrieval models adopted many times by code search engines. The BM is based on Boolean logic and classical set theory. Query and the target code files are conceived as sets of terms in such a model, and the retrieval is based on whether or not the code files include the terms from the query. It has an explicit limitation that at least one of the query terms must be present in the document side when an OR connective is used. On the other hand, the vector space model (VSM) represents the document (source code files) as the vector of identifiers such as method name, class name, or tokens. Generally, the comparison between such two vectors is derived using the cosine similarity. It does not support structural queries which consist of complex AND/OR relations. Moreover, it may lead to imprecise results in the context of code search because of the differences in term frequencies. Therefore, the revised VSM integrated both BM and VSM models to address the limitations by weighting both query and document terms and by computing the similarity between query and documents.

Another method used to compute the textual similarity is the term frequency-inverse document frequency TF-IDF that assigns weights to each token that appears within a file (source code file) of the code repository. Researchers also use other metrics such as Jaccard Distance [223] and Euclidean Distance [108] for code search engines.

The distance-based metric for textual similarity, known as edit distance, calculates the number of operations necessary for an algorithm to transform the first input into the second one. In code search, for given two code snippets, the edit distance would be the metric that shows the number of transformations requires to transform one of the snippets to other semantically. Different algorithms are used to compute the edit distance between two input code snippets, including Levenshtein Distance [593], Hamming weight [563], etc.

Graph Similarity: A source code is a collection of specific keywords that follow a strict semantic representation governed by syntactic rules. Treating these various entities in source code as text fails to capture the code semantics and needs a different form of representation. Graph representation is an optimal way to model the different source code elements and leverage them for the code search. The code search approaches that leverage graph-based retrieval does not explicitly accept input in a 'graph' form; instead, it accepts a typical query in the form of NL or code fragments. The query is transformed in a suitable graphical representation such as AST before being passed as input for the underlying search technique. This transformation leverages the code-specific structure and relationship between the different entities to refine the retrieving of relevant results. The type of graphical representation selected for a particular search engine depends on the context of the targeted input. For instance, a binary code search engine is likely to transform the binary input in the form of a Control Flow Graph (CFG); a code search engine targeting object-oriented programming language would use Data Flow Graph (DFG), etc. Each graphical transformation is compared against a similar graphical representation used as search space by the approach. In the form of similarity computation, this comparison consists of leveraging graph traversing techniques for optimized results.

Matrix Computation: Matrix computation is an elaborate method used as a retrieval technique in many code search approaches. The result of the computation provides a characterized correspondence between the elements of the matrix. Many applications have adopted these matrix computations because of their scalability that retains the results' predictive accuracy. Similarly, code search employs the matrix computation for measuring the co-occurrences of specific features (such as terms) within the source code. The approaches that use this technique compute a score that correlates between a given query and each source code. This score is then used to retrieve the relevant results.

Embedding Vector Similarity: Code search approaches using neural networks have been a trend recently. Common across such approaches is the idea of embedding user queries and code into vectors (i.e., an embedding refers to a real-valued vector representation). The computation of the distance between the embedding vectors is the correlation that forms the basis of retrieving the suitable candidates. As code search approaches based on neural networks show that it can adequately learn

the features of both the query and code from a large dataset, it helps address the issues from the previous techniques.

Execution Trace: A source code in the form of binary format is hard to read and understand. An efficient way towards understanding what a binary file is performing is to trace its execution flow, known as the execution tract. This execution flow provides information on the behavior of a program, such as the logical sequence of execution, changes in the data variables, etc. The similarity between the execution trace of two syntactically different programs provides information on how similar (or dissimilar) the two binary programs are.

Clone Detection: Code search is another way of identifying similar or identical snippets of source codes. Generally, code-to-code approaches sometimes leverage code clone detection techniques in the middle of their retrieval process to narrow down the search space by clustering similar source codes [5, 248, 322, 373]. Clone detection techniques are also directly used to determine the lexical similarity between the query code and candidate snippets [421].

Type Link: Type Link is a methodology that resolves a given function name by referencing the function (such as class extending another class in Java) and attempting to link it with its canonical form for the same type. For example, a class extending another class may have complex inheritance and nested types from its parent class. Such a type of class or type of code context requires a different approach when searching within them.

Solver: Solver or SMT solvers (short for satisfiability modulo theories) are instances represented as a formula in first-order logic. These formulas represent functions and symbols that follow specifications in the form of input and output of a program. For any given query, represented in the input and output of a program, the solver finds programs for which these specifications and constraints are satisfied. The satisfied candidates thus form part of the results presented to the user.

3.3.6 Result Presentation

The final step of the code search procedure delivers and presents the results in an appropriate platform. As each user has different requirements and feasibility, code search approaches should be provided on different platforms such as independent code search engines, integrated into a development environment (IDE), or presented as an idea in a paper. Independent code search engines can be either local or online to interact with developers with the underlying techniques, while some others are implemented as a plugin of different IDEs allowing them to leverage the search within the development process. Finally, others are presented just as an idea form proposed as potential work in code search.

Search Engine: A search engine catered explicitly towards the code search is the popular means of presenting the search approach results. These search engines are either working online (accessible over the internet) or offline (where the engine is deployed locally over the dataset chosen by the user).

IDE Extension: Rather than providing the code search in a search engine, some researchers have attempted to provide code search facilities integrated within the programmers' user development environment. These approaches integrate within the IDE of the user (such as Eclipse or IntelliJ) plugins. The majority of the code search implementation in the form of IDE extensions take as input either a pure NL query or a code snippet to output a relevant code snippet. The IDE plugin's use allows the code search to leverage the code given as input for the approach to find code examples.

Idea: Among all the different code search approaches, not all of them propose an evaluation or implementation; instead, they are ideas that can be incorporated within a potential code search implementation. Such approaches are classified as 'Idea' that proposes an approach without any evaluation or implementation details described in our work. Such ideas are valuable contributions within the field, some of which are later implemented as part of a larger code search engine or an IDE plugin.

3.4 Taxonomy of Code Search Techniques

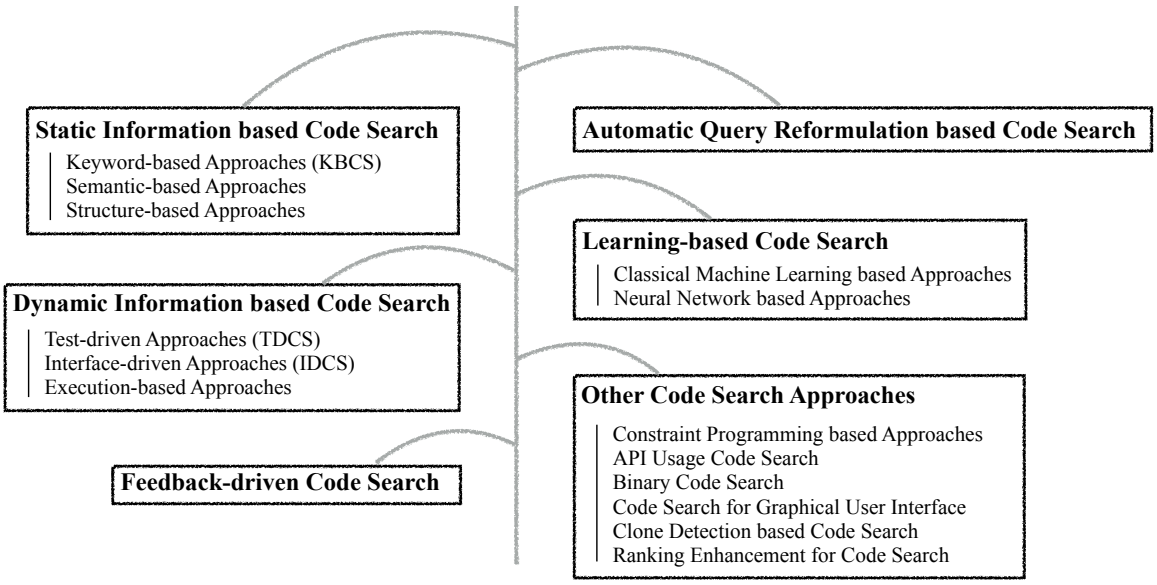


Figure 3.4: Taxonomy overview.

This section presents an overview of the code search literature based on the characteristics of each approach. We organize published papers on code search by focusing on the fundamental ideas (e.g., query reformulation for code search) as well as specific techniques (e.g., machine learning-based code search). When a paper has two individual techniques, we categorize the paper based on the main idea rather specific techniques utilized in the idea. Figure 3.4 shows the overview of our taxonomy which classifies the approaches into six classes.

3.4.1 Static Information based Code Search

Code search can leverage static information such as text in source code, which is similar to general purpose search engines [156,364,579]. For example, a code search tool can make an index of class, name, variable names as well as parameter types. Recent code search techniques with static information consider more information such as code comments; they generate code summaries to match against the query.

3.4.1.1 Keyword-based Code Search (KBCS)

Many of early code search engines rely on general text retrieval approaches; they treat source code as either plain or structured text. To that end, those approaches build an index of the textual information, i.e., keywords. Given a user query, the approaches try to compute a token-level similarity between the query and index. This type of code search is categorized into Keyword-based Code Search (KBCS) in this study.

The earliest known KBCS approach [135] was based on retrieving code snippets without considering any code-specific information (e.g., structure, sequence, etc.). It merely matched tokens from the query with the tokens from the source code. However, the resulting precision and recall were low, attributed to the tokens' mismatch, which created the need to consider additional information such as code structure and API documentation.

Table 3.2: Code search approaches leveraging static information

Static based Code Search		Output		Dataset					Indexing				Input				Retreival			Presentation			
Category	Approach	*General Code*	*API Usage*	*Specific Open Source Projects*	*Super Repositories*	*Developer Q&A*	*Language/API Documentation*	*Metadata*	*Inverted*	*Database (B+ Tree)*	*Graph Index*	*File Prefix*	*Natural Language*	*Code Fragment*	*Query Language*	*Class/Interface Type*	*Textual Similarity*	*Graph Similarity*	*Embedding Vector Similarity*	*Search Engine*	*Idea*	*IDE Extension*	Programming Language
Keyword-based approaches	Frakes et al. [135]	✓		✓									✓	✓				✓		✓			
	Jsearch [478]	✓									✓			✓			✓					✓	Java
	Sourcerer [14]	✓			✓				✓				✓				✓			✓			Java
	SNIFF [82]		✓	✓						✓			✓				✓					✓	Java
	McMillan et al. [358]		✓	✓					✓				✓				✓			✓			Java
	Exemplar [161, 163, 353]	✓			✓				✓				✓				✓			✓			Java
	Hill et al. [186]	✓		✓									✓				✓			✓			Java
	JECO [11]	✓				✓							✓				✓			✓			Java
	SnipMatch [566]	✓											✓				✓					✓	Java
	Hsu et al. [201]	✓			✓				✓				✓				✓			✓			Java
	APPROX [37]	✓		✓										✓			✓			✓			Java
	ANNE [538]	✓				✓							✓				✓			✓		✓	Java
	Example Overflow [594]	✓				✓		✓	✓				✓				✓			✓			Java
	Selene [378, 518]	✓		✓					✓					✓			✓			✓		✓	Java
	SoCeR [220]	✓		✓					✓				✓						✓	✓			Python
	Mentor [336]	✓		✓					✓				✓				✓				✓		
Semantic/Structure based Approaches	Prospector [339]	✓					✓									✓		✓				✓	Java
	XSnippet [453]	✓					✓			✓						✓		✓				✓	Java
	PARSEWeb [523]	✓				✓	✓					✓				✓		✓				✓	Java
	Coogle [452]	✓		✓										✓				✓				✓	Java
	Mendel [323]	✓		✓										✓				✓		✓			
	Portfolio [356, 359]	✓		✓				✓	✓				✓				✓	✓		✓			C, C++, Java
	Akhin et al. [5]	✓		✓							✓			✓				✓			✓		
	Wang et al. [551]	✓		✓							✓				✓			✓		✓			C
	Wang et al. [546]	✓		✓							✓				✓			✓		✓			C
	AutoQuery [549]	✓		✓						✓				✓				✓		✓			C, C++
	Chan et al. [78]		✓	✓							✓		✓				✓	✓		✓			Java
	Murakami et al. [379]	✓		✓						✓				✓			✓				✓		Java
	McMillan et al. [354]	✓			✓					✓			✓				✓			✓			Java, C#
	Vinayakarao [536]	✓				✓				✓				✓			✓				✓		Java
	Lancer [607]	✓			✓					✓				✓			✓	✓				✓	Java
	RACS [300]		✓	✓		✓	✓			✓			✓				✓	✓		✓			Javascript
	Aroma [328]	✓			✓					✓				✓				✓	✓	✓			Hack, Java, Javascript, Python
	YOGO [415]	✓				✓				✓				✓				✓		✓			Python, Java
	Barbosa et al. [29, 30]	✓			✓					✓				✓			✓						Java
	Test Recommender [409]	✓		✓						✓				✓			✓					✓	Java
	CodeMatcher [314]	✓			✓				✓				✓				✓			✓			

Later KBCS techniques leverage additional information from the structure of source code after parsing its Abstract Syntax Trees (ASTs). Then, the tokens extracted from these ASTs are used as plain-text keywords. Transforming a source code into an AST allows the code to be expressed as code entities like a package, interface, class, method, constructor, field, initializer, etc. In these approaches, each of the token keywords within the search space (i.e., source code) is mapped to a list of these entities associated with additional information (e.g., location, version of the source code, etc.). Once the search engines receive a user query, they match keywords with their corresponding entities retrieved as results. JSearch [478] is one of the approaches that use the information after parsing ASTs.

By leveraging the same concept, Bajracharya et al. [14] further considered the complex relations of code (e.g., inside, extends, implements, calls, etc.) with particular heuristics such as text-based, structured-based, and graph-based on the ranking methods. Reporting and arguing that the existing code search engines (until 2006) do not leverage complex relations present in the source code, they introduced a search engine called Sourcerer. This approach is partially based on using KBCS. A user query (i.e., list of keywords) is matched against the set of keywords maintained by the Lucene [558]. Each keyword is mapped to a list of entities such as the package, interface, class, method, constructor, field, and initializer. Each entity has its location, version, and rank associated with it. Finally, a list of entities under the matched keywords are retrieved from the information attached. The main heuristics suggested in their approach are text-based, structured-based, and graph-based ranking methods. The goal of Sourcerer aims three search types; (1) search for implementations (i.e., certain functionality), (2) search for reuse (i.e., existing piece of code), and (3) search for characteristic structural properties (i.e., certain properties or patterns).

In addition to tokens in source code, it is able to use tokens in other sources such as API documentation, which are linked to source code. Code search inevitably suffers from the vocabulary mismatch problem [482] as the languages used for describing and implementing functionalities are often different

each other. To deal with this intrinsic problem, many code search engines leverage API documents. For example, SNIFF [82] combines tokens in JavaDoc into those of source code to build an index. Another approach by Mcmillan et al. [358] even uses only JavaDoc as its search space; this approach retrieves code snippets that use classes and methods whose corresponding API documents have tokens of a user query. Exemplar [161, 163, 353] also resorts to a similar idea but this tool focuses on searching for applications rather than code snippets.

Hill et al. [186] presented a search technique based on phrasal concepts (PC) by claiming that bag-of-words approaches can suffer from reduced accuracy by ignoring the relationships between terms. The PC-based approach uses term position and their semantic role in method signature to calculate relevance between terms. The approach also introduces a relevance scoring mechanism that leverages four semantic roles: action (verb), theme (i.e., direct object), secondary arguments (indirect objects), and any auxiliary arguments captured by Software Word Usage Model (SWUM [184]).

Arwan et al. [11] proposed a tool named JECO by building a topic model that enhances the code search's effects by applying the Latent Dirichlet Allocation (LDA) model. The LDA creates a topic model for source code and each of the tokens within the input query. Later a cosine similarity [459] is computed with the topic scores to retrieve a ranked list of the relevant code snippet.

SnipMatch [566] incorporates crowd-sourced source code and introduces a simple markup language used for indexing and matching relevant code snippets. The markup permits snippet authors to specify search patterns and integration instructions. The tool leverages the markup and current code context (e.g., variable types or imported libraries) to improve the ranking and filtering process.

Hsu et al. [201] addressed the keyword-based code search by proposing a new information retrieval technique. This approach is based on leveraging the block-structure scheme that breaks the source code file into a hierarchy of blocks. It consists of file, class, method, and code blocks categorized in code-data and meta-data blocks for which different stemming processes are applied. The different stemming process for code-data block that is applied specifically for code (i.e. removing the frequent code words, etc.). The hierarchical structure of the source file allows a flexible approach towards the indexing and ranking process. Similarly, they also propose a query technique for users seeking specific types of blocks from the source code files.

Vinayakarao et al. [538] found that in code search, queries that include natural language with programming concept or syntactic forms of code tend to perform poorly. Such poor performance is due to the lack of mapping between different programming concepts and their associated syntactic form. They propose a technique that discovers mapping between syntactic code forms and their natural language terms to address this. The technique first infers the programming concepts, such as named entities, from the NL terms using Part of Speech (PoS) [92] tagging and pattern matching of its sequences. The syntactic forms of such terms together form the bases of the entity knowledge base. Finally, each line of a given source code is annotated with its associated NL terms from the knowledge base that permits keyword-based search on programming concept terms. A proof of concept named ANNE (ANNotation Engine) was developed to demonstrate the usefulness of entity knowledge-based for C and Java languages.

Example Overflow [594] uses jQuery-specific posts from StackOverflow. They employ lucene [558] for keyword matching against the posts. They also put weights on titles, labels, questions, answers, and source code of the posts to prioritize the matching process. They measure their results using the comparison of average returned result and context switching (number of clicks).

Based on the associated search engine GETA [380], Selene [378, 518] is a proposed approach towards a source code recommendation tool. The approach relies on pure code-to-code matching where the full source code is taking as a set of keywords to be searched against the code-base, and the top of the retrieved code is shown for the user. Primarily working as an Eclipse-based plugin over Java score base, Selene demonstrates the concept set of keyword-based code search.

APPROX [37] is an approach that takes source code fragment as input query and returns syntactically similar snippets. It achieves this by leveraging the algorithm proposed by Chang et al. [80]. The algorithm was initially designed for use in computational biology, and the authors adopted it for code search. It works in three phases: partitioning (creating partitions of text that needs to be searched for the occurrence of the query tokens), search (identify and eliminate irrelevant partitions using suffix trees) and check (locate the corresponding snippet using hybrid dynamic programming).

SoCeR [220] is an approach based on code summarization technique. It extracts and analyzes the content of source code (e.g., classes, methods, variables, docstrings, and comments) to generate code summary for each method. Then, it leverages the summaries to match against the user NL queries. First, SoCeR's pre-processor parses source code into AST to extract the contents. Later, the description generator combines the extracted methods and their contents to generate code summaries. Once a user query comes in, SoCeR leverages TF-IDF technique with its vectorization module named TfidfVectorizer[9] and computes the cosine similarity between the query and each code summary to retrieve code snippets.

Mentor [336] is a proposed tool that manages change requests and makes recommendations of source code related to a change request. It proposes the code by leveraging the Prediction by partial matching algorithm that calculates the similarity between change requests of a repository. The PPM works in two steps, first by creating a model from each change request within the repository, including the newly submitted one. This model is based on using the description, summary and comments present in the change request and assigning probabilities on its occurrences. In second step, it competes entropy of each model with the input change request that provides similarity probability score. Finally, the output with highest score is presented to the developer as candidate result.

Barbosa et al. [29, 30] proposed a code recommendation system that aims to provide code snippets for handling exceptions in the code. The approach first creates a pair of three 'terms' representing the three types of heuristic and their values from the code corpus. It applies the first heuristic based on the type of exception within the code, such as IOException. The second heuristic applies on the method calls such as open, read, etc. The third heuristic is based on the type of variable type existing within the code. The same type of heuristic is applied to the query that consists of a code snippet that the developer has focused on. Later the engine composes all the terms as a single disjunctive Boolean expression used to query the code corpus for similar examples. Finally, the candidates found with the heuristics are passed through their ranking function. This ranking function ranks the candidates based on the number of 'terms' that match with query snippet and the code in the repository. Higher the grade assigned based on that, the higher position within the list of recommended the candidates.

3.4.1.2 Semantic or Structure based Approaches

Unlike natural languages, programming languages (source code) have their specific structures and dependencies which KBCS approaches ignore. This has led to the consideration that code tokens as keywords where applying keyword matching may decrease the performance of the code search. The code search based on the structural or semantic similarities is based on leveraging the graph-based techniques, and where API usage patterns may be used to increase the performance.

For instance, Prospector [339] employs a graph search to synthesize chains of objects and methods calls. It structures a query language that comprises a pair of source and target object types. To traverse the search space, Prospector leverages the pre-computed graph where the nodes are types and edges connect two types of an API declaration. The retrieval is accomplished by traversing through the shortest path algorithm. The target is a set of paths (API method call sequences) from source and target object types. Finally, it generates a chain of declaration calls that can be directly fit into the context of user code as either code examples or Java Standard Development Kit (SDK).

[9] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

XSnippet [453] extended Prospector by providing an additional range of queries, ranking heuristics, and mining algorithms. The tool suggests relevant code snippets for the object instantiation task at hand. A range of instantiation queries is invoked from the java editor, including generic query that returns every code snippets to instantiate its type, specialized type-based and parent-based queries that return either type-relevant or parent-relevant results. XSnippet invokes the BFSMINE algorithm as the key component. This breath first mining that traverses a code model instance and retrieves the output (i.e., a set of paths representing code snippets that meet the user‘s desire).

PARSEWeb [523] is another synthesis tool that utilizes statistics from open source repositories. It takes a user query of the form “source object type → destination object type” to suggest relevant method-invocation sequences. The tool’s code analyzer employs AST and Directed Acyclic Graphs (DAG) for analyzing partial code (from an external code search engine) to handle control-flow information and method inlining. PARSEWeb traverses the DAG and leverages the shortest path algorithm to extract and retrieve Method-Invocation Sequences (MISs) with code snippets to the user.

Coogle [452] extends the concept of similarity measures (often used to find similar documents for a given query) to source code repositories. It detects similar Java classes in software projects using tree similarity algorithms after parsing the source code to extract ASTs. Once a parser generates ASTs, these are converted into an intermediary model called FAMIX [117, 527], a programming language-independent model for representing object-oriented source code. FAMIX tree representations of two classes are used to compute the tree similarity using the tree similarity algorithm proposed by Valiente [534].

Mendel [323] aids developers based on the entity the developer currently browses. The tool focuses on traits (e.g., which class to methods or referenced types are likely missing from a method) that may be missing. It leverages object-oriented programming properties to decide the set of family members of a class considering direct super- and sub-classes. This class family concept allows them to use genetic metaphor in the algorithm that analyzes source code entities related to the current working context. Once an entity comes in, Mendel computes a set of family members and retrieves the traits such as the entity’s functions.

Portfolio [97, 356, 359] employs keyword matching for application metadata and the user query to retrieve relevant functions. The search space is constructed with a function call graph (FCG) generated by their source code analyzer. To enhance the ranking performance, it leverages PageRank [403] on the FCG and computes the rank vector for each function. The retrieved functions by keyword matching along with FCG serve as an input to the algorithm Spreading Activation Network (SAN) [98, 99, 104]. Then, SAN computes the spreading activation vector of scores for the functions associated with the retrieved functions. Ranking vectors from both PageRank and SAN are combined into a resulting vector to retrieve code results to the user.

Akhin et al. [5] proposed a code search approach by engaging a tree-based code clone detection technique. The clone detection technique used inside utilizes characteristic vectors (w.r.t. all similar AST subtrees) to represent source code features and Locality Sensitive Hashing (LSH) for clustering. The key idea to retrieve code snippets is keeping breadth-first search (BFS) indexes for every node. Every query node is matched independently, and they build a sequence of matching nodes by using BFS indices. All the matched sequences are reported as search results to the user.

Some approaches toward code search focused on defining a precise query language to capture the users need. Working on similar approach, Wang et al. [551] proposed a code search engine with query language based on common dependence-related properties of source code. Their Dependence Query Language (DQL) allows users to formulate the queries by describing dependence properties on top of textual properties. DQL has four parts: (1) node declaration (ndecl) declares node variables and their types, (2) node description (ndesc) specifies constraints on declared node variables, (3) relationship description (rdesc) specifies constraints on the relations among declared node variables,

and (4) targets (target) specifies the variables specified in ndecl that are desired search targets. Once the system gets a DQL query from a user, it transforms the query into a series of graph reachability patterns and matches them in the System Dependence Graph (SDG) [479] of the source code using graph pattern matching [91]. SDG describes the dependence relationships between program points in the source code, and they used CodeSurfer [521] to extract SDG. Overall, when a DQL query is processed on an SDG, nodes in the SDG that match the node variables specified in the target and satisfy the constraints specified in the ndecl, ndesc and rdesc would be returned as result code snippets of the search.

Wang et al. [546] also proposed integrating the LDA method and a graph matching approach. They built an enriched SDG by assigning topics extracted by LDA to the nodes of SDG. To help users formulate queries, they apply the topics to DQL initially designed by Wang et al. [551]. The formulated query is transformed into a graph representation (i.e., query graph). Finally, the search engine retrieves code snippets by matching query graphs against enriched SDGs.

Based on the previous works [546, 551] that need users to construct dependence queries manually, Wang et al. [549] proposed an automatic query generation technique called AutoQuery to alleviate the users' burden. It consists of two major processes, namely Program Dependence Graphs (PDGs) generation and query generation. PDGs generation process returns the PDGs of the code fragments given by a user, and it is fed to the query generation engine. To generate PDGs, the approach uses Codesurfer [521] to convert the compilable code to their corresponding PDGs. These PDGs (expressed as multi-labeled graphs with textual and not type labels) are analyzed, and their commonalities are highlighted to be mined by a new graph mining solution. The sub-graph is the results of the mining step and is converted to a dependency query. For the searching step, they leverage a previously built search engine [551] to retrieve the code snippets.

Chan et al. [78] employs shortest path indexing scheme to recommend API code snippets. The approach first model classes and methods in an API library along with their invocation information as an API graph. Then, it retrieves the initial API candidates (methods or classes) based on textual similarity. They try to find an optimum sub-graph with the highest score from the API graph among all possible connected sub-graphs. Here a possible connected sub-graph is a graph where phrase of the query can be similar to at least one of the nodes in the graph. When the total number of nodes in the graph is smaller and the accumulated textual similarity is higher, the code snippets of such sub-graph is likely to be retrieved to the user. A greedy sub-graph search algorithm named RarestFirst (RF) [277] is employed and modified to build their own RarestGainFirst (RGF) algorithm to fit the problem of API code search.

The approach by Murakami et al. [379] proposed the use of degree-of-interest (DOI) model proposed by Kersten and Murphy [251] as the context information for code recommendation. A DOI model is capable of capturing the developers actives within the IDE, and assigns each of the programming element (i.e., method, field, function and class) a relevance score to the task the developer is working on. For example, for a developer working on program involving reading file as input, then method such as the input file would be considered a high DOI scored element. They implement a DOI-based recommendation system by extending Selene [518], that uses Eclipse as IDE, GETA [517] as search engine and UCI Source Code Dataset as code repository. The monitoring and DOI calculation are extended from the Maylyn's implementation.

McMillan et al. [354] proposed code recommendation by mining the feature description and code from the repositories. The approach works by first extracting feature description for application from Softpedia [487] that serves as repository for the feature descriptions for an application. By using incremental diffusive clustering (IDC) algorithm and feature identification approach [123] they extract the features for application. The clusters formed using the IDC represents a single feature, and the feature is named by identifying the most representative descriptor for that cluster.

Vinayakarao [536] claimed that use of repository of topics along with the structural information of source code extracted from various discussion forum could be benefitial for code search. The approach takes proximity search [131] for interleaved code problem and user defined terms (UDT) for overlapping structure problem. For a given snippet, it extracts the structural representation and to provide that to proximity search as a parameter. After getting the results, it checks the count of the topics in the results to avoid disambiguation by using UDT.

Pointing out the out-of-vocabulary (OOV) problem, Lancer [607] has been proposed with token-granularity method modeling. It is a context-aware code-to-code search tool that presents a new language model called Library-Sensitive Language Model (LSLM) and new ranking algorithms. LSLM is based on an extended version of the open-vocabulary cache N-gram model [181] to capture the code patterns for each library. Furthermore, Lacer leverages Topic-Sensitive PageRank [180], a graph-heuristic association mining algorithm to compute the relevance scores between libraries. Once a code query comes (i.e., an incomplete method), LSLM retrieves the most relevant libraries and predicts several subsequent tokens by combining all language models' predicted probability scores.

Javascript is a complex programming language where its harder to infer the relationship between method calls because of the use of keywords that naturally occur in a NL such as 'when' and 'after'. Furthermore, many of the Javascript frameworks such as jQuery use Document Object Model (DOM) elements to manipulate the code's flow (where the elements are defined in HTML code while functions in Javascript). Li et al. [300] proposed a Relationship-Aware Code Search (RACS) approach that specifically finds the code snippet using Javascript frameworks for a given NL query. RACS leverages the semantic information between the API method calls in the code snippet and the relationship between actions in the query. The RACS works on three operations: mining API usage patterns to generate Method Call Relationship (MCR) graph from the search space, abstracting the NL query to generate the Action Relation (AR) graph from the user query, and retrieving the snippets by matching the MCR with AR graphs.

Aroma [328] is a three-phase approach that leverages the structural features (e.g., method calls, variable usage, etc.). The light-weight phase of Aroma parses and creates a simplified parse tree for each of the method body in the code corpus. It takes a query code fragment and retrieves a list of the best overlapping methods in the parse trees of the query and each method based on their custom features. The Prune and Rerank phase re-ranks the obtained list of method bodies by leveraging their pruning algorithm on the first phase is generated parse tree. In the last phase, named Cluster and Intersect, Aroma clusters the re-ranked code snippets based on the method's similarity. A custom clustering algorithm that takes the constraints into account is developed to ensure the recommendation's quality. The final intersection process is considered to each cluster so that Aroma can retrieve the recommended code snippets.

YOGO [415] is presented as an equational reasoning based code-to-code approach that is built upon dataflow equivalences. It takes a library of pre-written rewrite rules and a high-level concept expressed as a dataflow pattern. In order to recognize if a code fragment is equivalent to one of implementations for the same concept, YOGO takes a fusion approach of equality saturation [520] which is based on Program Expression Graphs (E-PEGs) and the Programmer's Apprentice [435]. Equality saturation allows the tool to discover equivalent fragments by applying low-level equations and rewrite rules for representing all fragments as a structure called an e-graph [118, 382, 383]. Programmer's Apprentice ensures that the high-level concepts can be identified as dataflow patterns so that YOGO can recognize if a program reaches the same goal with different algorithms. Yogo translates source code from the search space into Program Expression Graphs (PEGs), it matches the rule for De Morgan's law, the equality saturation engine extends the PEGs with the newly generated nodes. Later, it adds dashed equivalence edge between the rules. Finally, it matches the rules for retrieving matched code snippets.

Barbosa et al. [29, 30] proposed an approach that specifically provides code snippets for handling exceptions in the code. The approach first creates a pair of three 'terms' representing the three types

Table 3.3: Dynamic Information based Code Search Approaches

Dynamic Code Search		Output		Dataset		Index			Input						Retrieval				Presentation			
Category	Approach	*General Code*	*Test Case/Test Code*	*Specific Open Source Projects*	*Super Repositories*	*Database (B+ Tree)*	*Graph Index*	*File Prefix*	*Natural Language*	*Code Fragment*	*Query Language*	*Test Case*	*Class/Interface Type*	*Software Specification*	*Textual Similarity*	*Graph Similarity*	*Test Case/Tested Input*	*Linear Programming*	*Search Engine*	*Idea*	*IDE Extension*	Programming Language
TDCS	CodeGenie [278–280, 289]	✓				✓				✓						✓					✓	Java
	Lemos et al. [286]	✓			✓	✓				✓					✓						✓	Java
	Code Conjurer [213, 224]		✓			✓				✓					✓						✓	Java
	EQMINER [229]		✓	✓				✓		✓						✓	✓		✓			C
	S6 [427, 428, 430]					✓			✓					✓		✓			✓			Java
	Marcus and Atkinson [253]	✓			✓	✓				✓							✓			✓		
	HUNTER [552]	✓			✓	✓					✓	✓						✓	✓			Java
IDCS	Strathcona [194, 196, 198]					✓				✓			✓								✓	
	Lemos et al. [284]	✓			✓	✓			✓				✓		✓				✓			Java
Execution-based	CodeHint [143]	✓		✓		✓								✓	✓						✓	Java
	DyCLINK [502]	✓		✓			✓			✓						✓					✓	MYSQL

of heuristic and their values from the code corpus. It applies the first heuristic based on the type of exception within the code, such as IOException. The second heuristic applies on the method calls such as open, read, etc. The third heuristic is based on the type of variable type existing within the code. The same type of heuristic is applied to the query that consists of a code snippet that the developer has focused on. Later the engine composes all the terms as a single disjunctive Boolean expression used to query the code corpus for similar examples. Finally, the candidates found with the heuristics are passed through their ranking function. This ranking function ranks the candidates based on the number of 'terms' that match with query snippet and the code in the repository. Higher the grade assigned based on that, the higher position within the list of recommended the candidates.

Another approach, Test Recommender [409] attempts to match the changes in production code with the useable tests code from existing test sets. Test Recommender first identifies the changes proposed towards the production code using a versioning system. It parses the proposed changes, line-by-line classifying them as Project classes, non-project classes (core libraries), primitive data types (consisting of types like int, string, etc.), and stored as an element of the changeset. In parallel, all the past test cases code snippets are stored as testset. Finally, Test Recommander compares the changeset with each testset element and returns the tests ranked based on their match with the changeset elements.

CodeMatcher [314] is proposed to simplify code search approaches by arguing the fact that deep learning based models such as CODEnn [167], are time consuming as well as they can be inaccurate comparing to simple approaches. It combines tokens from the user query with original order, performs a fuzzy search on name and body strings of methods, and returned best-matched methods with a longer sequence of used tokens. The approach is compared with CODEnn and it differs as it represents code as a <method name, token sequence> instead of <method name, API sequence, token> and the training phase is omitted as it is not a learning based approach. CodeMatcher generates token metadata such as property (e.g., verb), frequency (i.e., occurrence number of the camel split tokens), and importance (i.e., the naming worth of token property). These metadata is used to remove query tokens that are not commonly used for method naming. Based on the filtered query tokens, it generates regular match strings in the form of *token$_1$.*...*token$_n$.* and launched iterative fuzzy match to filter out the repeated method.

3.4.2 Dynamic Information based Code Search

Unlike static approaches such as KBCS, code search with dynamic information takes inputs and executes the program inside the search engine to value the candidates for qualitative retrieval. By using dynamic information, code search engines can assist developers who lack the expertise of the desired code [143] and non-native speakers of the language in which the repository is based [286], assure faster retrieval of code snippets [278], provide more guarantee the correctness of the behavior of retrieved code snippets [278].

3.4.2.1 Test-driven Approaches - TDCS

TCDS is one of the dynamic approaches in code search that takes a test case (e.g., a test case is a set of inputs, execution conditions, and expected output for a specific function of a program) and identify the appropriate code snippets that pass the given test case. The use of test cases serves two purposes: (1) they define the behavior of the desired functionality to be searched; and (2) they test the matching results for suitability in the local context [278]. A typical TDCS procedure starts with designing test cases, providing a code query (e.g., important keywords such as class or method names from the test cases) to the code search service. Finally, each program has to be executed for validation and retrieved for given test cases.

The first example of dynamic approach is proposed by Lemos et al. [278–280,289] named as CodeGenie. It takes test cases designed by developers for a desired feature, automatically searches for the features, weaves the results into the developer's project. Finally, CodeGenie performs tests using the original test case. It relies on the well-known code search engine Sourcerer [14].

Later they introduced the issue of vocabulary mismatch problem (VMP), where the results are poor due to faulty choice of keywords selected by the user. To circumvent the VMP, they extended the CodeGenie with a specific concept called *thesaurus-based tag clouds* [286]. A thesaurus for the language similar to the code in a given repository is used to explore similar terms for a given initial token, taken from the method name. These terms are searched in the codebase, and the tag 'cloud' is configured according to their frequency. The closer the terms are to the method name, the larger their weight.

To enhance the performance of the TDCS, Janjic et al. [213, 224] added the concept of proactive to TDCS and implemented Code Conjurer (integrated in merobase [361] platform). They argued that CodeGenie is not proactive that requires developers to test all reuse candidates locally in their IDE manually. Such proactive mode can issue new search requests each time the developer deletes, inserts, or changes an interface-defining part of the software to improve effectiveness.

Reiss [427, 428, 430] introduced S6 that lets the user define keywords, test cases, and other developer-friendly specifications. The approach uses keywords to find initial source code candidates. It utilizes transformations to extract, convert, and manipulate the initial candidates into a form that matches user's specification. Furthermore, it includes a dynamic approach (i.e., test-driven) by using Ant and JUnit, and employs semantics (i.e., specifications) to restrict the results. Later, Reiss [429] attempts to integrate S6 with Code Bubbles [61, 62] which is a redesign of the user interface to programming for providing intuitive arrangement of working sets (e.g., classes, other modules).

EQMINER [229] is a another TDCS that generates random inputs and it relies on symbolic [297] or concolic execution [274]. It checks the abstract memory states [257] to compute function similarity based on execution outputs.

Marcus and Atkinson [253] proposed augmenting the continuous software engineering with the rapid and continuous reuse of software code units by integrating a TDCS approach. Specifically, test-driven searches can be triggered based on the analysis and extraction of test cases from the code changes. Moreover, suppose a test-driven search retrieves the results as a list of software components. In that case, a reuse report is then pushed through the continuous integration platform to the developers or managers, hence making reuse a reactive activity.

HUNTER [552] is a proposed tool that searches for code relying on finding its relevant methods and synthesizing with necessary wrapper code for the results. The tool consists of three components: code search (responsible for the ranking list of candidates), interface alignment (that infers matches between the parameters and desired methods from the query), and synthesis (applies the type of similarities between the inferred mapping). For every candidate generated by the approach of HUNTER, it runs those methods with user-provided test-cases. Essentially, for any test case that fails on the retrieved candidates, HUNTER gathers another set of candidates.

3.4.2.2 Interface-driven Approaches - IDCS

IDCS is similar to TDCS but it is based on interface that a software component offers rather than a test case of object oriented programming. By specifying an interface for the desired function, IDCS aims to reduce the search space (e.g., by restricting the parameter and return type of a function to *int*) and generally improves the performance of code search. *Signature Matching* [596] from 1995 originally inspired the concept of IDCS [284] but the search for the *Signature Matching* is limited towards software libraries.

Strathcona [194, 196, 198] is an IDCS approach proposed in very early stage. It determines the structural context from the source code that the user is working on and automatically formulates a query to retrieve code snippets with similar context from the codebase. It employs six heuristics (one Inheritance Heuristic, three Calls Heuristics, and two Uses Heuristics) to match both sides' structural context descriptions. These heuristics are based on the types of classes and methods. The final results are retrieved once all the heuristics are applied and decided based on the initial result's frequency.

As for another example, Lemos et al. [284] extended Sourcerer infrastructure [19] to implement an IDCS for their empirical study. This implementation came with a feature that allowed it to accept Solr-formatted [468] queries so that it can catch the return and parameter types in addition to the keyword-based query.

3.4.2.3 Execution-based Approaches

Some code search engines leverage a programming language's execution-related features, such as the Java runtime elements, to retrieve its execution trace.

For instance, CodeHint [143] attempts a different path towards finding meaningful information from a given code snippet. It exploits the Java runtime capabilities by executing the debugger at runtime to collect data such as the code's call method. By executing the candidates in a concrete program state and using the partial specifications, it can synthesize code that uses I/O, reflection, and native calls. Thus, CodeHint allows users to input the refined query using classical keyword matching approaches and compare them with the runtime data.

DyCLINK [502] is another dynamic approach that computes the similarity of execution traces to detect whether two code fragments are relatives (i.e., that they behave (functionally) similarly) or not. It traces each program's execution creating a dynamic dependency graph that captures behavior at the instruction level. It records instruction-level traces from sample executions, organizes the traces into instruction-level dynamic dependence graphs, and employs a specialized subgraph matching algorithm to efficiently compare the executions of candidate code relatives.

3.4.3 Feedback-driven Code Search

Feedback-driven approaches toward code-search provide a user-friendly concepts that works by performing interactions with the user. Many times developers are unsure what they are looking for and do not search for the desired code using a single query. Therefore, they issue multiple queries [24, 63, 195, 475, 493] by adding, removing, or editing their initial queries. This process is iterated until the developer is satisfied with the results. Furthermore, developers often do not have a clear idea of what they are working with [112, 477]. They tend to learn or consider initial ideas at the beginning of the search, which can later be expanded or changed based on the newfound understanding of the problem. This leads to the change within the idea that is considered [112,127,329,373]. Multiple researchers have investigated an approach towards building feedback-driven (i.e., iterative) approaches based on such observation.

Table 3.4: Feedback-driven approaches

Feedback-driven Code Search		Output		Dataset					Index	Input			Retrieval		Presentation			Programming Language
Approach	Key Technique	*General Code*	*API Usage*	*Specific Open Source Projects*	*Super Repositories*	*Developer Q&A or Tutorial*	*Language/API Documentation*	*Metadata*	*Inverted*	*Natural Language*	*Code Fragment*	*Query Language*	*Textual Similarity*	*Vector Similarity*	*Search Engine*	*Idea*	*IDE Extension*	
Mica [500]	Web-based (Google API) query expansion		✓	✓			✓			✓			✓		✓			Java
SAS [18]	Using feature like Tag-cloud		✓		✓				✓	✓			✓		✓			Java
Conquer [440]	Finds and leverages co-occurring set of tokens from initial result	✓		✓						✓			✓				✓	Java
Extended Conquer [189]	concept-based searching [187] that combines the vocabulary and contextual search	✓		✓						✓			✓				✓	Java
Wang et al. [547]	Uses 4-point Likert scale [6]	✓		✓						✓			✓		✓			C,C++
CodeExchange [346]	Specific characteristics used to rank/arrange initial results	✓			✓			✓	✓	✓			✓		✓			Java
CodeLikeThis [347]	Hybrid diversity ranking algorithm [349]	✓			✓				✓	✓			✓		✓			Java
INQRES [325]	Word relations - Inheritance Relation (InR), Implementation Relation (ImR), Synonym Relation (SynR), Same-word Relation (SamR), and Compound Relation (ComR)	✓		✓						✓			✓		✓			Java
Cosoch [299]	Markov decision process (MDP) and query logs with user interaction behaviors from a search session to reassign weights for reinforcement learning algorithm	✓				✓				✓			✓		✓			Java
Contextual Search [185]	Leverage hierarchy and context of the words in the given query	✓			✓					✓			✓				✓	Java
Refoqus [176]	RSV [437], Dice [120], and Rocchio [438, 439] expansion and one elimination of non-discriminating	✓		✓					✓	✓			✓				✓	Java, C++
ALICE [486]	*Inductive logic programming* (ILP) [375, 418], a relational learning form that supports structured data encoded as logical facts	✓		✓						✓		✓		✓			✓	java
SNIPR [460]	Twist, a command-line language for search result interactions and code re-targeting									✓		✓	✓			✓		
DeepAPIRec [84]	Tree-LSTM organizes the code information in tree-based structure and invertible back to the source code to support incremental expansion.		✓		✓						✓			✓	✓			Java

The very first approach named CodeFinder [132] proposed by Henninger in 1993, presented the possibility for recommending further keywords (i.e., keyword refinement) to the user from the retrieved results. The approach is based on the Spreading Activation Network (SAN) [98, 99, 104].

Mica [500] augments standard web search results described for the desired functionality. This approach allows users to select keywords to augment the query based on their requirements/needs. It explicitly offers Java SDK refinement recommendations by extracting keywords from the search results retrieved by Google's web API [153]. Keyword matching is performed against the occurrences within the Java SDK Libraries.

Later, Bajracharya et al. [18] proposed an idea called Sourcerer API Search (SAS). It is a search interface that incorporates specific features such as Tag-cloud (i.e., catching the popular words from the initial results), Hits with Snippets, and Top APIs. The provided features allow the users to either modify the query based on the popular words or filter the results to those that use specific top API elements.

Based on the insight that the same queries can be used for a different context, Conquer [440] uses NL techniques to find a co-occurring set of tokens among the initial results to help a developer refine the subsequent query. The tool mainly provides four features: (1) quick feedback on the relative frequency of query tokens in the result set, (2) display alternative query tokens to help reformulation, (3) organize the results by action and theme to show a variety of search results, (4) report the results in a list sorted by the overall score to make the results easier to be skimmed without requiring the cognitive overhead.

The Conquer was later extended [189] with a technique that leverages the concept-based searching [187] that combines the vocabulary with contextual search approaches. This approach is not restricted, i.e., it does not need to have ordered words within the query for effective retrieval.

Wang et al. [547] required users to decide on the relevancy of the result. They used the 4-point Likert scale [6], where the scale from 1-4 indicates the relevancy of the results from entirely relevant towards irrelevant. By going through their refinement engine, users can retrieve the re-ranked list. The feedback from the previous queries and its corresponding refined results are cached to identify and refine similar future cases.

Interacting with the user to refine the results was further explored by Martie et al. [346, 347]. They introduced CodeExchange (CE) and CodeLikeThis (CLT) to lead the user towards leveraging the initial results for formulating the future query. CE provides options to help the developer decide specific characteristics (e.g., the code's complexity) to arrange the initial results. Rather than using specific characteristics, CLT can help the developer find code analogous to the full result by leveraging the Hybrid diversity ranking algorithm [349].

Further work in the same direction was undertaken by Lu et al. [325] who implemented INQRES that interacts with users to reformulate the query. It extends related words from the source code and leverages the word relations such as Inheritance Relation (InR), Implementation Relation (ImR), Synonym Relation (SynR), Same-word Relation (SamR), and Compound Relation (ComR). The extracted words are demonstrated in the interactive user interface with the "AND" and "OR" relations for users to decide.

Cosoch [299] casts a code search session into a *Markov decision process* (MDP), in which rewards measuring the relevance between the queries and the resulting code documents (i.e., code snippets with textual explanations). It utilizes query logs and user behaviors within a search to dynamically adjust the feature weights by the reinforcement learning algorithm, providing an optimized ranking list of documents.

Contextual Search [185] provides another iterative refinement process for code search. The approach automatically captures the context of the words in the given query by extracting and generating natural language phrases from the initial code snippet results. The extracted phrases from the source code form a hierarchy of related phrases from general to specific ones. The leaf nodes of the hierarchy are the specific code elements that match the phrases. Therefore, this hierarchy aids developers identify relevant code elements by reducing the number of relevance judgments and reformulating their initial query.

Feedback-driven approaches overlook the fact that a query depends on many factors, and the different properties of a query may require different reformulation techniques. In their prototype called Refoqus [176], it proposed the use of reformulated query to developers after investigating the quality of the original query. It consists of four techniques such as RSV [437], Dice [120], and Rocchio [438, 439] expansion and one elimination of non-discriminating (i.e., tokens that are occurred more than 25% in the corpus). Overall, the user can use the information to decide if she/he has to investigate the list of the results or seeks a reformulation of the query to get better results.

SNIPR [460] introduced a new concept called 'code retargeting' to complement code search. This concept intends to narrow the search for a suitable code. It allows developers to understand the results easier and explore code modification ideas without leaving the search interface. Their search space stores all the possible snippets-mappings (i.e. mapping similar code snippets) generated by humans for better accuracy. Furthermore, they proposed Twist, a command-line language for search result interactions and code retargeting. It allows user queries and retargeting operations (commands) to be intermixed or to be used separately. For example, with Twist, the queries like 'fibonacci series +Int=>Array[Int] : reformat > read-only fields' can be specified, and it implies that the user is capable of specifying the requirements in a fine-grained manner.

Sivaraman et al. [486] designed a query language that models a program's structural and semantic properties as logic facts and expresses a search pattern as a logic query. In particular, the tool takes *Inductive Logic Programming* (ILP) [375,418], a relational learning form that supports structured data encoded as logical facts. The implementation, ALICE based on ILP uses the logical predicates (e.g., if, loop, parent, next, methodcall, type, exception, methoddec) to represent each code snippet factbase (a database of the logical predicates). Therefore, the query language can represent search patterns such as 'find examples that call the readNextLine method in a loop and includes FileNotFoundException', which cannot be represented in text-based techniques.

Leveraging one of the recent deep nueral network technique named tree-based LSTM [516] (Tree-LSTM), DeepAPIRec [84] has been proposed for a feedback-driven approach. Tree-LSTM organizes the rich code information in a tree-based structure and invertible back to the source code and it can support incremental expansion. Furthermore, combining tree-based structure with LSTM's superior ability to preserve long-distance dependencies, it allows the approach to model and reason about variable-length, preceding, and succeeding code contexts. DeepAPIRec also uses statistical parameter models for continually generating a complete line of code. The overall architecture contains three components. First, *Statement Model Training* is to predict the next statement by applying Tree-LSTM to the abstract representations of statements which neglect concrete variable names. Second, *Parameter Model Construction* is to complete the prediction by generating concrete parameters. This step analyzes the source code from the corpus to extract data dependency and then constructs a statistical parameter model for parameter concretization. Finally, based on the previous models, *Code Recommendation* step takes a program with a hole as input and generates a list of ranked code statement with concrete parameters. Once a user chooses a recommended statement, the code recommendation engine further recommends other statements based on the updated program. Note that the *Statement Model Training* predicts abstract statements and the *Parameter Model Construction* generates concrete parameters for the abstract statements by identifying and evaluating compatible variables or objects from the code context.

3.4.4 Automatic Query Reformulation based Code Search

A query plays one of the most critical role for the code search engines [189, 410] and many researchers have worked towards improving the quality of search by reformulating the queries automatically (i.e., without user's intervention). Query reformulation is conducted because the vocabulary mismatch problem [182] (multiple words for the same topic), polysemy (one word with multiple meanings), and the general words in the query complicate the code search engines. A query can be reduced, expanded (augmented), or even entirely alternated by their properties [174]. This section only covers code search with automatic reformulation techniques (i.e., without user intervention here.

Query Expansion has two major classes: global and local approaches [75, 341, 575]. The former approach expands the query using thesauruses such as WordNet [561] to leverage the synonyms. In contrast, the *local approaches* leverage user's feedback or automatically extracts expanded tokens from the top-ranked documents.

Nie et al. [395] is one of the local approach that employs a concept called Pseudo Relevance Feedback (PRF) [341] and Stackoverflow as one variation of PRF to improve the performance of code search. In their implementation of a prototype named QECK leverages Stackoverflow Q&A pairs to expand the queries. Moreover, their method built upon Rocchio's model [438, 439] that assumes a fixed number of top-ranked documents are relevant to the initial query and extracts tokens from such documents to expand the query.

Table 3.5: Automatic Query Reformulation based Code Search

Automatic Query Reformulation based Code Search		Output		Dataset							Index			Input		Retrieval			Presentation			
Approach	Key Technique	*General Code*	*API Usage*	*Specific Open Source Projects*	*Super Repositories*	*Developer Q&A or Tutorial*	*Language/API Documentation*	*Code Clone*	*Challenge/Competition*	*Metadata*	*Inverted*	*ID-based*	*Positional*	*Natural Language*	*Code Fragment*	*Textual Similarity*	*Matrix Computation*	*Embedding Vector Similarity*	*Search Engine*	*Idea*	*IDE Extension*	Programming Language
QECK [395]	Pseudo Relevance Feedback (PRF)	✓			✓	✓					✓			✓		✓			✓			Android Java
Yang et al. [582]	Semantical pairs on context in code and comment	✓		✓										✓		✓			✓			Java
Durão et al. [124]	Ontology model based expansion	✓		✓								✓		✓		✓					✓	Java
NLP2CODE [72]	TaskNav algorithm	✓	✓			✓								✓		✓					✓	Java
SCP [484]	model for term-term positional proximity	✓		✓									✓	✓		✓			✓			Java, C, C++
QExpandator [462]	Term-term matrix based on click logs (Koders logs)	✓		✓										✓			✓		✓			Java
FWSMF [464]	WordNet [561]	✓		✓							✓			✓		✓			✓			Java
CodeX [463]	Synonyms based expasion	✓		✓							✓			✓		✓				✓		N/A
SnippetGen [206, 568, 585]	Intent-relevant context	✓			✓						✓			✓		✓			✓			C#
CodeGenie 2.0 [287]	Type-based thesaurus	✓			✓						✓			✓		✓					✓	Java
CoCaBu [482]	Query augmentation with Stack Overflow posts	✓			✓	✓				✓	✓			✓		✓			✓			Java
FaCoY [261]	Query Stack Overflow posts	✓			✓	✓		✓	✓	✓	✓				✓	✓			✓			Java
CodeHow [334]	Enriched API documents	✓			✓						✓			✓		✓					✓	C#
RACK [422, 423]	Keyword-API co-occurrence from Stack Overflow	✓			✓	✓	✓							✓		✓					✓	Java
NLP2API [424]	Pseudo-relevance feedback and term weighting algorithms for query expansion	✓				✓					✓			✓				✓	✓			Java
NQE [316]	Conditional distribution for predicting method names	✓			✓	✓					✓			✓				✓	✓			Android Java
SENSORY [3]	BWT transformation algorithm [68]	✓			✓							✓		✓		✓			✓			Java
QESR [231]	Inferring evolving contexts with probabilistic model	✓			✓		✓				✓			✓		✓			✓			Android Java, Java
QECC [210]	Model to learn <changes, contexts> pairs	✓			✓						✓			✓		✓			✓		✓	Java
GKSR [208]	Request and its corresponding conversation (R&C) pairs from "pull-requests"	✓			✓						✓			✓		✓			✓			C#, Java, Android Java
QESC [209, 608]	Change information along with the reason of the change by (DBN) [191]	✓			✓		✓				✓			✓		✓			✓			Java

In contrast to the local approach, the global one addresses the ambiguity of the query keywords. In such ambiguity, the query is matched with irrelevant information leading to less accurate results. Li et al. [305] introduced a global approach that employs Stackoverflow to either alter or expand the query. The method executes a synonymy substitution on each term in the initial query (e.g., "db" → "database"). It then expands the query using the related words based lexical database built on Stackoverflow. Another global approach from Lu et al. [326] introduced a query reformulation based on WordNet [561] and as part-of-speech of each word in the query.

Yang et al. [582] proposes a technique to extend the queries in keyword-based code search with semantically related keywords. They achieve this by automatically identifying the semantically related words by levering the context in which they appear, i.e., if the words appeared as pairs in comments, code, or a combination of both. Furthermore, they consider phrases from the comments and code for accurately leveraging the context when creating the semantical pairs. Later, this approach was extended [583] to consider cross-project semantical checks.

The semantics problem was further extended by Durão et al. [124] who proposed reformulating the queries with a semantic layer. This semantic layer utilizes the domain ontology to enhance the query construction by adding terms related to the query. Besides, the semantic layer performs analysis by using Naive Bayes on the codebase, enabling a domain-based classification. The classification allows the user's reconstructed query to be matched within the same domain (e.g., a query with keyword 'connection' within the domain 'network' will be reconstructed with additional keywords from the same domain), thereby increasing the precision.

Other approaches in query reformulation attempted to conceptualize the natural language expression for a given query. For instance, a statement "Best strategy to add lines of text to a text file" is conceptualized to "add lines to text file". Such conceptualization allows for a meaningful matching between the main keywords from the given query. NLP2CODE [72] is one such approach that mines titles from Stackoverflow questions via TaskNav [530, 531], an algorithm that conceptualizes software tasks as verbs. It is associated with a direct object and suggests automatic expansion in auto-completing for the user's needs of API usage examples.

Sisman et al. [484] proposed Spatial Code Proximity (SCP), a model for measuring term-term positional proximity. The model is especially appropriate for source code because the keywords that consistently appear close to the query keywords in the software are likely to be related to the query. It assumes that tokens with the same concepts in source code are usually proximal to one another in the same files. The model retrieves the additional tokens from the source code and injects them into the final query to reformulate it.

QExpandator [462] expands the user query with a search topic and content-specific keywords. Specifically, it extracts pairs of a user query, a clicked code fragment from the dataset [13] and a representation of each query in a vector. Each user query in the query logs is transformed as a document vector as well. A term-term matrix is constructed where semantically and contextually similar terms are stored in the same matrix row. The similarity score is calculated by applying the Jaccard similarity formula on term co-occurrences within the same query. The top-k relevant terms are obtained from the matrix to expand each term.

Feature-Wise Similar Method Finder (FWSMF) [464] is a technique that focuses on semantic similarity and query expansion. This technique uses call and data dependency graphs to identify the invoked methods and find the fields declared outside the method body but used by each method within the search space. For further essential factors such as the signature (e.g., input and output types), a self-executable method is regenerated by constructing a Data Dependency Graph for the method. For a given set of self-executable methods, the similarity is checked by running each method and checking the output to cluster them. For each cluster, terms that appear in most of the methods of that cluster are chosen to be indexed. The query is finally expanded by leveraging WordNet [561].

According to Satter et al. [463], a code search engine should retrieve self-executable code snippets that are comprehensive and relevant. They aim to achieve this using the proposed CodeX that employs parsing of all the methods and uses the SourcererCC [454] for removing duplicate methods from the search space. To make the results self-executable, it uses a slicing approach introduced by [532] to resolve the dependencies. Once a user query is received as a boolean query, the query terms are expanded by adding synonyms using the boolean OR operation. CodeX also supports the test cases and test reports for the retrieved results to show the level of correctness.

SnippetGen [206, 568, 585] is another query expansion approach based on intent-relevant semantic and structural matching. It has three components, namely the intent-enriched, intent-first, and intent-expanded. The intent-enriched component contains the code modifications recorded from its VCS. Such component exploits the intents based on intent-relevant context with control-, data-, and containment- dependency analysis. The intent-first component takes user queries and retrieves results by computing the combination of two similarity scores (semantic and structural) between them. Retrieval is based on the combined score of both Bag-of-Words and Bag-of-Function-Calls by using VSM. If the results cannot be retrieved, the intent-expanded component is triggered to expand the user query by considering both the intent and the text similarity. Specifically, once SnippetGen gets k potentially relevant methods from the intent-first component, it tokenizes the intent to get a keyword list for each method. Then it constructs boolean expression (to leverage Extended Boolean Model [457]) and combine them to generate an expanded query.

Another direction of query expansion targets IDCS, which generally gets high precision but is often compensated by the recall's corresponding loss. Query expansion can be applied to every part (i.e., method name, return types, and parameter types) of the query in IDCS. Their prototype, CodeGenie 2.0 [287] utilizes the WordNet [561] to generate English synonyms and a code-related thesaurus with software-related word relations used to expand the method name present in the query. For the other part of the query consisting of types, CodeGenie 2.0 uses a type-based thesaurus (e.g., both Integer and int are considered to be the same) to expand the return types and parameter types.

One of the main motivations of query reformulation is Vocabulary Mismatch Problem (VMP) [139,315]. The use of NL queries often leads to irrelevant results due to VMP. It is considerably worse in code search (i.e., only 12% accurate [24]) because of poor documentation or lack of explicitly informative names for classes, methods, and variables [263]. Moreover, the likelihood of two people selecting the same keyword among a word family is only between 10 to 15% [139].

To overcome the VMP, CoCaBu [480, 482, 483] augments and structures the free-form query taken by the search proxy into a code-friendly one by leveraging developer Q&A forum Stackoverflow. Specifically, CoCaBu uses the user query to find the most relevant posts' code snippets and parse them to structure a code query. The search engine matches the structured query (e.g., method_declaration: getValue, used_class: Writer) against the structured code tokens data in the search space.

As a progression of CoCaBu, FaCoY [261] leverages Stackoverflow to reformulate the user code query (i.e., code fragment). In contrast to CoCaBu, FaCoY matches the most similar answer snippets from the Stackoverflow answer posts and finds the corresponding question posts. Then, the search engine computes the textual similarity to find the most similar question posts. Given the question posts, the answers (only the posts with code snippets) are considered semantically similar code snippets. Following this process, the user's query is translated to another code query, and the search engine retrieves the best matching code snippets from the search space.

Code search engine such as CodeHow [334] is based on API understanding i.e.,text similarity between a search query and enriched API documents. It expands the query with the APIs and performs code retrieval using the Extended Boolean model [457], which considers the impact of both textual similarity and potential APIs. The APIs are enriched by obtaining API descriptions from their online documents (MSDN as it targets C#), including the full qualified API name, summary, and the remarks (full descriptions).

Researchers [422–424, 598] investigated and discovered the fact that the applications with similar functionalities often use the same set of API classes.

RACK [422, 423] is one of the state-of-art that leverages this finding. It suggests a list of relevant API classes for a natural language query by exploiting the API-keyword association extracted from a developer Q&A forum (i.e., crowd-sourced knowledge). The approach has two major steps: (1) Construction of mapping (i.e., Keyword-API) database by taking questions-answer code snippets from Stackoverflow, and (2) API recommendation by employing the three heuristics (i.e., Keyword-API Co-occurrence, Keyword Pair-API Co-occurrence, and Keyword-Keyword Coherence) for covering APIs that are also functionally consistent with the query keywords.

NLP2API [424, 425] is another approach that identifies relevant API classes from Stackoverflow for a natural language query and reformulates the query using such API classes. API classes are collected using PRF [341] and two-term weighting algorithms. The technique identifies and ranks the candidates using Borda count and semantic proximity between the query terms and collected API classes. Determining the terms' semantics is based on their positions within a high dimensional semantic space developed by fastText [56]. In contrast to the other query expansion approaches, it analyzes both local (e.g., PageRank [403]) and global (e.g., semantic proximity) contexts of the terms in the query to enhance the performance of code retrieval.

The use of neural network language model for automated query reformation by Zhang et al. [597, 598] extended a study [582]. Their approach discovers the API class-names that will be expanded based on the NL query using a neural network language model. It captures API-class's semantic representation from the source code using the continuous bag-of-words model (CBOW) [365]. Also, the training phase that utilizes Word2Vec [366] to map NL queries with identifiers (i.e., API class names described in Java API documentation). The semantic distance between the query and the API class-name selects the class-names to expand the user's initial query.

Liu et al. [316] introduced a neural model named NQE (a.k.a. Neural Query Expansion) for code search. The model is based on encoder-decoder. The encoder embeds the query and feeds the vectors to a Recurrent Neural Network (RNN) while the decoder uses Gated Recurrent Units (GRU). The conditional distribution of NQE learns to predict method names that contain the query tokens and co-occur with each other in the dataset. Therefore, it helped productively expand the query and demonstrated improvement of the NCS [449] by combining the NQE model. This model was the first query reformulation based on neural networks for the field of code search.

Investigating a different perspective of the existing query reformulation techniques (e.g., [334, 395]) shows that it may negatively impact the code search because of the addition of irrelevant tokens to the query. QREC, proposed by Huang and Wu [207], is a query reformulation technique with evolving contexts that refer to added/deleted tokens during the maintenance. This reformulates a user query by considering added tokens as the relevant and deleted tokens as the irrelevant [416]. QREC is designed as a two-pass retrieval (1) First-pass: the search engine produces the initial query results, and code tokens are extracted as dependent tokens from the initial results. Their positive/negative inference model computes the probability of added/deleted tokens triggered by the dependent tokens. (2) Second-pass: the search engine reformulates the query by applying added/deleted tokens to retrieve the final method-level results. While it is not a code search engine, the technique defined is applicable to existing search engines. In the literature, it is applied to code search engines named Portfolio [356, 359] and CodeGenie [289].

SENSORY [3] proposes incorporating the programming context references to achieve finer-grained results (i.e., at a statement-level). It parses source code from the collected dataset to extract method-level code snippets information such as method name, return value, parameters, and variable names. For each of this extracted information, SENSORY replaces all the variable declaration with its type and transforms it into the Last Column (LC)-Sequence with the BWT transformation algorithm [68]. The BWT algorithm builds an array whose rows are cyclic shifts of the tokens parsed from the code

snippets in the dictionary order. The last column (LC) consists of the identical tokens, retaining the ordering. The input query consists of the programming context that includes a code snippet near the cursor position in an IDE and structural information about the method. A similar process of extracting and replacing the variable declarations to get the BWT transformation is applied to the query. The similarity of each pair <query, a code-snippet> (i.e., compressed token strings) from the BWT transformation is measured using the Jaccard similarity and assigned weight based on its score. Finally, using a modified variation of the BM25 text retrieval algorithm ranks the resulting candidates based on their match weight.

Arguing the state-of-the-art may fail to find relevant code snippets for different versions of software code snippets, Jin et al. proposed a query expansion based code search framework named QESR [231]. It is based on evolving contexts (EC), which imply added/deleted terms and dependent terms (i.e., the code terms that are either control- or data-dependent on the added/deleted terms) in commits. The framework has three components: (1) EC inference, (2) Search, (3) Integration. An EC is defined as a tuple of (<dependent terms>, <distance>, <changed terms>, <operation>) where a dependent term is defined as a tuple of (<changed term>, <dependence distance>, <AST node type>, <label>). Based on obtained EC, a probabilistic model is trained, and the model computes probability scores. By appending the added terms and excluding the deleted terms in a query, the user query is expanded and used for the retrieval process based on Extended Boolean Model [457] and BM25 [341]. Finally, the Support Vector Machine (SVM) [102] ranking is leveraged to rank the candidate snippets to provide them to the user.

QECC [210] is a query expansion approach based on investigating code changes that may accurately capture the user's needs. The implementation for QECC named InstaRec builds <changes, contexts> pairs from the commits from Github, and it learns the pairs statistically to infer code changes and expand the given query. Specifically, the context of the initial search result is leveraged. It infers the changes as expansion keywords by computing the probability of changes triggered by the contexts using the association-based inference model. The user query is expanded based on the probabilistic scores of the candidate keywords.

GKSR [208] is a query expansion approach for code search based on Github knowledge (GK) along with existing features such as API information and crowd knowledge. GK is based on the "pull-requests" of Github repositories, and it consists of each request and its corresponding conversation (R&C) pair. For more details, GK includes descriptions of the request, commits, and comments from participants (i.e., crowd knowledge (CK)) and API information from the changed files. GKSR first retrieves the initial results by calculating BM25 [341] similarity scores between a query and R&C pairs. Then, it selects candidate keywords with TF-IDF to expand the query. The second retrieval phase computes similarity scores between the expanded query and code snippets in the search space for the final retrieval. Multiple expansion methods are integrated for considering more features in parallel. GKSR retrieves results for each expansion method based on GK, API, CK, and individual components. Finally, the ranking process takes computations of the weighted sum of the components with the SVM [102] ranking.

QESC [209, 608] is a query expansion algorithm with the semantics of change sequences to capture change information itself along with the reason of the change based on Deep Belief Network (DBN) [191]. The system first generates sequences of change and code respectively by detect changes and parsing code. To generate change sequences, QESC leverages ChangeDistiller [133] for changed terms and Crystal [10] for dependent terms and put them together in the sequence in the format of (<label>,<role>,<operation>). <Label> represents the textual information of AST nodes, <role> has two options that decide if a term is changed or dependent, and <operation> has three options that decide if a term is unchanged, new, or deleted. For code sequences, it takes input as a method and outputs a code sequence by parsing code and put them in the sequence like generating change sequences. After obtaining the sequences, QESC converts them into vectors of terms to apply them

[10] https://code.google.com/archive/p/crystalsaf/

to DBN for training the semantics of sequences. This step generates an inference model for inferring the fine-grained changes. In order to bridge the semantic gap between NL query and changed terms, the system takes two-pass retrieval. Initial search results are leveraged to infer changed terms and QESC formulate the query with the selected changed terms (i.e., new terms to add, deleted terms to remove). On receiving a formulated query, the search engine calculates BM25 textual similarity scores between a query and the index words to retrieve.

3.4.5 Learning based Code Search

3.4.5.1 Typical Machine Learning based Approaches

Weimer et al. [564] extend collaborative filtering technology named Maximum Margin Matrix Factorization (MMMF) and apply it to code search domain. The approach contains MMMF with two new

Table 3.6: Learning based Code Search

Learning based Code Search		Output		Dataset								Index		Input		Reterival				Presentation			
Category	Approach	*General Code*	*API Usage*	*Specific Open Source Projects*	*Super Repositories*	*Developer Q&A*	*Language/API Documentation*	*Code Clone*	*Challenge/Competition*	*Existing Benchmark*		*Inverted*	*Graph Index*	*Natural Language*	*Code Fragment*	*Textual Similarity*	*Graph Similarity*	*Matrix Computation*	*Embedding Vector Similarity*	*Search Engine*	*Idea*	*IDE Extension*	Programming Language
Machine Learning based	MMMF [564]	✓		✓											✓			✓		✓			Java
	Surisetty [508]	✓		✓								✓		✓					✓		✓		
	ROSF [225]	✓		✓								✓		✓		✓			✓	✓			Java
	Source Forager [241]	✓			✓								✓		✓				✓			✓	C/C++
	Zou et al. [609]	✓				✓	✓						✓	✓					✓	✓			Java
	CodeKernel [168]		✓	✓			✓						✓		✓				✓	✓			Java
	SCOR [4]	✓			✓							✓		✓					✓	✓			Java
	CODEC [377]	✓			✓										✓	✓			✓	✓			Java
	ExAssist [389, 393]	✓		✓											✓					✓		✓	Java
	CodeMF [203]	✓				✓								✓		✓				✓			C#, SQL
Neural Network based	Nguyen et al. [390]		✓	✓			✓							✓		✓			✓	✓			Java
	CODEnn [167]	✓			✓									✓					✓	✓			Java
	MP-CAT [177]	✓				✓								✓					✓	✓			Python
	CSDA [432]	✓			✓									✓					✓	✓			Java
	CARLCS-CNN [472]	✓			✓									✓					✓	✓			Java
	BVAE [88]	✓				✓								✓					✓	✓			C#, SQL
	SLAMPA [606]	✓						✓							✓				✓	✓			Java
	SCS [214]	✓		✓										✓					✓	✓			Python
	NCS [449]	✓			✓	✓						✓		✓					✓	✓			Android Java
	UNIF [71]	✓			✓	✓						✓		✓					✓	✓			Android Java
	Fujiwara et al. [137]	✓						✓							✓				✓	✓			Java
	MMAN [541]	✓			✓									✓					✓	✓			C
	AdaCS [308]	✓			✓									✓					✓	✓			Java
	CoaCor [586]	✓				✓								✓					✓	✓			Python, C#
	CODE-NN [222]	✓				✓								✓		✓				✓			C#, SQL
	Ye et al. [588]	✓				✓								✓					✓	✓			Python, SQL
	TranS^3 [550]	✓			✓									✓					✓	✓			Python
	Yin et al. [590]	✓				✓								✓					✓	✓			Python, SQL
	CDRL [205]	✓			✓									✓					✓	✓			Java
	Schumacher et al. [466]	✓			✓									✓					✓	✓			Python
	HECS [296]	✓				✓								✓					✓	✓			Python, C#
	MSR [122]	✓		✓										✓		✓			✓	✓			Java
	PSCS [507]	✓							✓					✓					✓	✓			Python, Javascript, Ruby, Go, Java, and PHP
	Heyman and Cutsem [183]	✓				✓								✓					✓	✓			Python
	COIL [298]	✓			✓					✓				✓					✓	✓			Java
	CoNCRA [114]	✓				✓								✓					✓	✓			Python, SQL
	COSEA [542]	✓				✓								✓					✓	✓			Python, SQL
	DGMS [309]	✓			✓	✓			✓	✓				✓					✓	✓			Java, Python
	APIRec-CST [85]		✓		✓		✓							✓						✓			Java
	Zhao et al. [600]	✓				✓								✓					✓	✓			Python, SQL
	CodeCMR [592] (vague to add)														✓				✓	✓			
	CRaDLe [166]	✓							✓					✓					✓	✓			Python
	NJACS [204]	✓				✓								✓					✓	✓			C#, SQL, Java, Python

loss functions (i.e., weighted soft margin and weighted logistic regression [283]). These loss functions allow the natural trade-off between precision and recall of the search results. Overall, combining the two new loss functions with collaborative filtering outperforms the rule-based approachesCO in terms of the F1 score.

Surisetty [508] proposed a Behavior-based approach that leverages clusters of code snippets. The hypothesis behind this is that the code snippets in a single cluster tend to have similar behavior. The clustering engine computes parent clusters by clustering scripts with similar vectors by using X-Means algorithm [407]. Later, the sub-clustering engine computes the cosine similarity [459] between vectors in the parent cluster and groups all the scripts with a cosine similarity value >= 0.9 into a single child cluster. Finally, the enumeration engine computes the average tf-idf of each parent and child cluster. The search engine uses the stored values to retrieve the cluster so that the search engine can work based on the snippets' behavior.

ROSF [225] is a combined approach between information retrieval and supervised learning based on multi-aspect features. The approach consists of two stages: coarse-grained searching and fine-grained re-ranking. Coarse-grained searching is based on an information retrieval method BM25 [341], and the retrieved code snippets are represented as instances with its Vector builder and used as a test set of the training phase. The re-ranking stage trains a probabilistic model to re-rank corresponding code snippets by applying different relevance scores for each instance in the test set. The multinomial logistic regression [55] is utilized for training and generating the models for re-ranking.

Source Forager [241] pursues various aspects (i.e., feature-classes) of a code by extracting all possible code features. It has multiple code feature-classes that are extensible to enhance the coverage of the search. A supervised-learning technique of binary classification Support Vector Machine (SVM) [102] is employed to compute different feature-classes' relative importance. SVM training process generates relative and fine-tuned weights for feature-class similarity scores. Such similarity scores every feature-class between two functions are integrated into a similarity vector. The model is trained on the data of similarity vectors for both similar and dissimilar functions. Finally, the model transforms the user query to its feature-vector and computes the similarity to retrieve the most similar code snippets.

An approach with graph-embedding technique to capture local and global structural information has been proposed by Zou et al. [609]. The approach automatically constructs code graphs from the search space and represents each code element using graph embedding. Graph embedding is an offline process in the approach, and LINE (Large-scale Information Network Embedding [519]) has been leveraged. Given an NL query, the approach takes BoW to represent and retrieves candidate nodes by their specific text matching rules and weights. The subgraph's generation takes a beam search-based algorithm and then extends the selected nodes to a connected subgraph.

CodeKernel [168] is a first graph kernel-based approach for API code snippet retrieval. It represents source code as object usage graphs [394] which can be seen as an abstraction of source code instead of feature vectors [258]. Then, it clusters graphs by an embedding (which conserves full aspects of the original graphs [46]) them into a continuous space using a graph kernel [58, 59]. CodeKernel employs GrouMiner [394] at the function level to represent source code as graphs and adopts Borgwardt's code in Matlab [540]. To cluster the graphs, it leverages Spectral Clusterer for Weka [330] which can perform a typical clustering algorithm named spectral clustering [70]. CodeKernel retrieves code snippets by selecting a representative graph from each cluster using empirically designed ranking metrics (i.e., Centrality and Specificity that are designed based on sigmoid and IDF).

ExAssist [389, 393] recommends exception handling code for given android bytecode with API-related code. It starts by building API usage models represented as graph-based object usage model using GROUM [394]. Then it extracts the API exception usages from these models. Later it uses their proposed XRank, a fuzzy-based model to rank and suggest unchecked exception types for the query code snippet. For every exception with high confidence output, it is considered likely a candidate to

be suggested. Furthermore, a machine learning model called XHand is utilized to recommend an exception that best suits the API method call.

To consider semantics and order of tokens, SCOR [4] leverages the combination of word2vec [366] and Markov Random Fields (MRF) [362] which is based on undirected graphs. In MRF framework, one of the nodes represents a source code file that is being evaluated for its relevance to a given query and other nodes represent the individual terms in the query. The edges between the nodes represent probabilistic dependencies between the nodes. Using the MRF allows the system to select different kinds of probabilistic dependencies to encode. The approach embeds the source code files and the query individually by using word2vec skip-gram model to capture the semantic relationships between the terms. It employs two match layers to compute cosine similarity similarity values. The retained values are summed and normalized to obtain the relevance score based on their Per-Word Semantic Model (PWSM). In order to consider the order of the terms, one of the match layers contains cells that represent similarity between two consecutive query terms and file terms. Finally the system computes the composite score of single term based and consecutive based relevance scores to retrieve the code snippets.

CODEC [377] is a contextualized and synthesis-based approach that takes the query as the surrounding context in an IDE to recommend suitable code snippets for the missing spot. The system learns to encode the context and decode it into a program instead of learning to encode contexts and source code into a latent space. It extracts contexts such as class name, the Javadoc text for the method with the missing body, the desired return type, and the formal parameter's name. The contexts are built as <context, code fragment> pairs. Then, CODEC decompiles each code fragment into a SKETCH language, which captures the essence of the Java fragment (e.g., API calls) but ignores lower-level details (e.g., variables). The model is trained using a subset of the decompiled code utilized to power a maximum likelihood estimation, where the goal is to select the parameter set. Once the parameter set has been learned, each SKETCH code to be indexed is transformed into an intermediate representation. The set of each pair is distributed across the set of servers used to power the search. When the user query comes in, the query context is extracted from an incomplete program. It is processed to compute the top K pairs that maximize the posterior distribution is retrieved to the user.

CodeMF [203], a feature-fusion approach, mines Stackoverflow posts to retrieve high-quality software repositories for code search. Instead of focusing on only structural features of code snippets, CodeMF is a prototype to prove that social features shared on programming datasets are useful. Given the stripped query-code matching pair data from Stackoverflow, CodeMF retrieves top k pairs by their scores as high-quality software repositories. It consists of five steps: (1) Q&A data structuring and normalization to strip versions of query-code pairs and construct a K-dimension features matrix, (2) Kernel principal component analysis (KPCA) to normalize each feature vector and reduce the dimension of the matrix to D-dimension features matrix, (3) Wavelet time-frequency transformation to train the dimension-reduced matrix and transform them to frequency-domain coefficients, (4) Expectation Maximization (EM)–based fusion and inverse transformation to merge all corresponding leaf-node coefficient sets into the high-frequency and low-frequency coefficient vectors and reconstruct them to a new signal (NT), and (5) Score ranking and high-quality pairs extraction to rank the value of the NT in descending order and extracts the top k scored query-code pairs. The framework is applied to a query expansion based code search approach namely, QECK [395] for the validation.

3.4.5.2 Neural Network based Approaches

Many recent studies have taken steps towards enabling more advanced code search by leveraging techniques based on neural networks [71]. From the early stage, code search approaches [167, 308, 586] started to leverage such techniques to alleviate the semantic gap between high-level intent from the user query and the low-level implementation details in the source code. These approaches have learning

models that discover the correlations from the dataset and usually embed code snippets and the user query into a common hyperspace. The final phase for retrieving is composed of computing similarity between the vectors using similarity functions such as cosine similarity [459]. Various techniques for neural networks such as fastText [71,308,449], Recurrent Neural Network (RNN) [167,308,316], Convolutional Neural Network (CNN) [472], Long Short-Term Memory (LSTM) [167,214,308,432,466, 472,541,586,588,606], Auto-Encoder (AE) [88], Transformer [550,590], Feed-Forward Neural Network (FFNN) [137], and Gated Graph Neural Network (GGNN) [541] are leveraged by the researchers.

Nguyen et al. [390] combines Revised Vector Space Model (rVSM) [605] with a shallow neural network for embedding named Word2vec model [366] to achieve better code retrieval accuracy. They used a new score function that is a linear combination of the two scores from Word2vec and rVSM. They normalize the scores for each side and apply a trade-off parameter that controls the importance of the two similarity scores. The trade-off parameter is adjusted based on Jaccard similarity between the query and the code snippet. Finally, the method that will be taken is decided by a specific threshold value (e.g., Word2vec is used when trade-off parameter is less than 0.65, rVSM otherwise).

One of the representative approaches with neural networks is CODEnn [167] that jointly embeds code snippets and natural language descriptions into a high-dimensional vector space. In this way, the model contains code snippets and their corresponding descriptions as similar vectors. DeepCS is their research prototype based on the CODEnn model. It embeds code snippets in the dataset into vectors, and when a user query comes in, the code snippets that have the nearest vectors to the query are returned. It consists of three components: (1) a code embedding network (CoNN) to embed source code into vectors, (2) a description embedding network (DeNN) to embed natural language descriptions into vectors, and (3) a component for similarity measure that computes the similarity between code and descriptions vectors. A CoNN embeds the sequence of camel split tokens, API sequence using a Recurrent Neural Network (RNN) with maxpooling [264]. As general tokens have no strict order in the source code, they are embedded by using a multi-layer perceptron (MLP) (i.e., the conventional fully connected layer [372]). Finally, three code aspects' vectors are concatenated and fused into a vector through a fully connected layer. For DeNN, it embeds the NL descriptions using a RNN with maxpooling. Overall, the cosine similarity score is computed between the vectors from CoNN and DeNN to retrieve the most relevant code snippets against the user query.

Based on CODEnn [167] Haldar et al. [177] proposed a multi-perspective cross-lingual neural framework for code-text matching to capture both global and local similarities. The proposed model is named MP-CAT, which combines a multi-perspective (MP) and AST-based (CAT). CAT takes the code, AST, and natural language sequence to retrieve two vectors (i.e., the first vector represents code and AST, and the other represents the NL description). The other module, MP takes the same process, but it contains local information where CAT ignores as a global embedding. CAT has three BiLSTM [509] layers that are followed by three maxpool layers for the context representation module. At the same time, MP consists of three sets of BiLSTM layers, which compute each token's contextual representation in the corresponding input sequence. Bilateral Multi-Perspective Matching (BiMPM) [554] approach is leveraged to compare and aggregate the sequence embeddings into two fixed-length multi-perspective hidden representations by passing them through two different BiLSTM layers. The approach concatenates the vectors from CAT and MP modules to get the final vectors and takes L2-distance to retrieve code snippets.

CSDA [432] is an attention-based LSTM neural network that uses jointly embedding technique which includes source code features such as method name, API invocation sequence, and method token orders that CODEnn [167] is missing. It is possible to provide improved semantic similarity measurements between the user's NL query and code snippet. CSDA captures API invocation sequence and token orders by assigning attention weights to different word locations of semantic features. The overall architecture contains four embedding layers for method names, API sequences, method token orders, and query descriptions. All the embeddings are obtained using BiLSTM [509] layers, and an attention layer is leveraged to generate weight vectors for method token orders and the

query description. These vectors are used to compute the weighted sum and for the final semantic representation of the tokens. The similarity module measures the semantic similarity between the semantic code vector, and description semantic vector and query description semantic vector by leveraging cosine similarity [459].

Similarly, Shuai et al. [472] pointed out that CODEnn [167] ignored the internal semantic correlations of code and query. The research prototype is named CARLCS-CNN, which learns interdependent representations for the code and query with a co-attention mechanism. Such a mechanism allows to learn correlation matrix and co-attends their semantic relationship via row/column-wise max-pooling. CARLCS-CNN model takes a commented code as inputs and performs individual embedding on code features (method name, API sequence, and tokens) and corresponding comments. Then, it learns co-attentive representations. Specifically, CARLCS-CNN takes CNN to embed method name, method tokens and paired description, while it takes BiLSTM [509] to embed API sequence. Two feature matrices, for code and description sides, are generated and the correlation matrix is computed by introducing a parameter matrix, which is to be learned by neural networks. Finally, description attention vectors are obtained via Column-wise max-pooling and softmax while it takes Row-wise max-polling and softmax for code attention vectors. Cosine similarity is used to compute the similarity measure between the generated representations.

Another approach by Chen et al. [88] presented a neural framework that bidirectionally maps source code and natural language. The framework named BVAE has two Variational AutoEncoders (VAEs), C-VAE for source code and L-VAE for the natural language. These two VAEs are jointly trained to reconstruct their input (a standard way of AutoEncoders) with regularization that captures the similarity between the latent variables of code and description. Overall, BVAE learns semantic vector representations for code and description to generate new descriptions for random code snippets. Given a natural language query, its mean for the posterior distribution is computed and similarity-search algorithms such as Best-Bin-First [40], VA-File [54, 562], and FFVA [545] are applied to retrieve the most relevant code snippets.

SLAMPA [606] is an approach for retrieving code snippets based on statistical language model and clone detection techniques. It is a code-to-code approach that infers first given code query's intention using a neural language model and it retrieves code snippets from the search space with the support of their clone detection technique named Hybrid-CD. SLAMPA employs RNN [595] (specifically LSTM [192] to analyze the intention of the code query. This LSTM network analyzes the names and tokens from the written incomplete code and infers an ordered sequence of tokens which are likely to be used. Later, the inferred tokens are concatenated to the original token sequence to match against each code snippets in the search space by embedding them and compute the vector similarities. It takes two concepts of similarity measurements; high-level and low-level. On one hand, *High-level Similarity Measurer* embeds features like class and method name as well as the token sequence into a fixed-length vector by using fastText [56]. To embed tokens, SLAMPA employs BiLSTM [509]. Finally the class/method embeddings and the token embedding are concatenated into a larger embedding called snippet embedding. By taking two snippet embeddings as input, it calculates the similarity. On the other hand, *Low-level Similarity Measurer* considers the frequencies of tokens contain some information to retrieve code snippets. To record frequency of each token and convert the token sequence into a token-frequency dictionary, they followed Cclearner [294]. Similarities from two concepts are combined for the final results.

SCS [214], a supervised neural code search system using multiple sequence-to-sequence networks, has been implemented by the GitHub engineering team. The approach trains a Sequence to Sequence (Seq2Seq) model to act as an encoder for the source code (i.e., predict a docstring from code) in the dataset. It also trains a separate language model to use as a sentence encoder for the descriptions. The generated code embedding was then mapped into a vector space with the sentence embedding using dense layers. Finally, a vector similarity computation was performed to match and retrieve the results against the user query.

NCS [449] combines word embedding [56], TF-IDF [369] weighting, and higher-dimensional vector similarity search [232] to build a neural code search. It contains a continuous vector embedding of each code fragment at method-level granularity. When an NL query comes in, NCS maps the query to the same high-dimensional vector space and then uses vector distance to simulate the relevance of code fragments to the query. To embed the tokens from the code, fastText [56] has been leveraged, and the generated vectors are used to represent the intent of source code as "document" vectors. As for the query's embedding, NCS averages the vector representations of each word to create a document embedding for the query. Finally, NCS takes FAISS (Facebook AI Similarity Search), implementing various efficient similarity search algorithms for the final results.

Later, UNIF [71] has come out based on an attempt to understand adding the supervision to neural code search and investigations on existing neural code search systems. It is considered an extension to NCS [449] that minimally adds supervision and performs better than NCS. The principal behind UNIF is using a bag-of-words based network that has not been used in more sophisticated approaches such as CODEnn [167] and SCS [214] which are based on sequence-of-words. Furthermore, UNIF computes the code token weights with a neural network while NCS uses TF-IDF weighting.

A deep neural network (DNN) model with a code mutation technique has been used for code-to-code search. Fujiwara et al. [137] presented an approach for code block search using a Feed-Forward Neural Network (FFNN). In the learning phase, they generated mutated source code fragments and clustered them. Later the feature vectors are generated by the extracted code blocks from the labeled clusters to train the FFNN. For a given query code, the feature vectors are generated for that query, and the trained model computes and retrieves a label. Finally, the original code block of the corresponding label is the result for the user.

Deep learning based approaches tend to lack explainability, making it hard to interpret the results and almost impossible to understand which features are important for the retrieval. The prototype MMAN [541] addresses the issue by leveraging Multi-Modal Attention Network for semantic code search. The tool consists of an LSTM for the sequential tokens of code, a Tree-LSTM for the AST of code, and a GGNN (Gated Graph Neural Network) for the CFG of the code. Additionally, a multi-modal attention fusion layer is applied to capture the multi-modal features simultaneously. MMAN assigns different weights to different parts of each source code modality to overcome the existing DL-based code search's explainability issue. Then the tool integrates them into a single hybrid representation. For a given query, MMAN feeds it to the LSTM module to generate the corresponding representations and compare it with each code snippet from the hyperspace, trained by a multi-modal representation module.

AdaCS [308] is an unsupervised domain adaption based approach that can be trained once and transferred to new codebases. To allow the models to be adaptive, it decomposes the training process into embedding domain-specific words. An unsupervised word embedding is leveraged to build a matching matrix to represent the lexical similarity. Then an RNN is utilized to capture latent syntactic patterns from these matrices in a supervised way. A supervised task learns general syntactic patterns that across domains that allow AdaCS to be transferable to new codebases. AdaCS takes fastText [56] to learn domain word embeddings, represents interaction structures of each <text, code> pair as a lexical matching matrix, and leverages a LSTM [192] to encode the pairs, and prediction takes a RNN [447].

Yao et al. [586] claimed that annotating source code can improve code search, and their research prototype CoaCor validated it. The fundamental motivation behind this is that code annotation represents a code snippet's semantic meaning and can be used to alleviate semantic mismatch [538]. Annotation generation is formulated as a Markov Decision Process [41] and the CA model is trained to maximize the received reward by using an advanced reinforcement learning [510] from A2C framework [370]. Once code annotation (CA) model is trained, CoaCor can build two kinds of pairs (i.e., <query, code> (QC) and <query, annotation> (QN)). CoaCor contains two code retrieval (CR)

models called QC-based CR and QN-based CR based on BiLSTM [509] network and leverages the combination of the scores from both to rank code snippets.

A neural model for code summarization and search named CODE-NN [222] has been proposed by leveraging LSTM with an attention mechanism to produce sentences. It jointly performs content selection and surface realization based on the code snippets and NL descriptions from Stackoverflow. An LSTM model guided by attention on the code snippet to generate a summary one word at a time. Here, a global attention model [332] that computes a weighted sum of the embeddings of code snippets based on the current LSTM state. By using a specific scoring function, CODE-NN produces an NL sentence and retrieves code snippets that maximize the score.

Inspired by CoaCor [586], an approach that proves the similarity between generated annotations could improve the performance of code search, Ye et al. [588] assumed that generated code could also be beneficial. Building upon this assumption, they proposed a framework named CO3 by exploiting the probabilistic correlation between code summarization and code generation with dual learning [570]. The model utilizes the two encoders from dual learning to train the code retrieval task through multi-task learning. To embed NL descriptions and code snippets, CO3 takes separate embedding layers to convert input sequences into high-dimensional vectors. For code summarization and generation, LSTM and Bi-LSTM cells are used to extract all the trainable parameters. Later, the dual learning [570] generates the marginal distribution for both summarization and generation. The code retrieval module takes an NL input and computes the similarity against all the code snippets to retrieve the code sequences with the highest scores.

CDRL [205] is a code-description based representation learning model for code search based on attention to reduce the generally shared space into a specific one. For the dataset, the frequently considered aspects (i.e., method name, API sequence, tokens of code) [169, 300, 334] are used for the code side. The model first embeds and generates vectors for source code and the corresponding descriptions. Each aspect is embedded individually (e.g., convolution operation based on nonlinear activation for the method name, CNN for API sequence, and MLP [372] for tokens of code) From these vectors, CDRL extracts import features and represents their semantic information in the code-based and description-based feature extraction layers. The attention layer allows CDRL to narrow the code tokens to take only important code tokens for descriptions and vice versa. Then, it employs a refining layer to refine feature vectors from a specific shared space. Finally, it inputs the refined feature vectors into a Factorization Machine (FM) [433] to model the higher order feature interactions for rating prediction. The rating prediction computes the relevance scores between the user query and each source code to retrieve.

Schumacher et al. [466] proposed a combined network that contains neural and classical machine learning approaches to address a well known problem named Out-of-Vocabulary (OOV). On one hand, they employ Pinter-Mixture Network [293] that alleviates OOV problem by using Pointer Networks [539] with multiple LSTM layers are leveraged for the neural network side. On the other hand, a probabilistic language model named PHOG [45] which uses production rules from a context sensitive grammar and has little restrictions in regard to vocabulary size and long-range dependencies is taken for classical machine learning side. To combine two different networks, they added an output-dimension to the output layer of the neural network for each probabilistic language model which they want to utilize in the combined network. Each of these added output-dimensions holds an estimated probability of how the corresponding language model produces the results. The key idea of combined network is to avoid the Out-of-Vocabulary (OoV) problem by denying the prediction of unknown words when leveraging the learning models in code search field.

HECS [296] is a deep neural network that based on hierarchical embedding by considering sequential information of source code and developer Q&A posts. It consists of two hierarchies for code and query that allows HECS to capture potential information by two modules (i.e., Intra-language encoding module (ILEM) and Cross-language encoding module (CLEM)). HECS first uses SGNS-based [470] word2vec model to learn code structure embedding from the Q&A dataset. ILEM is built upon

ON-LSTM to capture the token order structure of source code (i.e., a code snippet can be expressed as a hierarchy), hence HECS can model the natural representation of this hierarchy during training. CLEM is based on attention mechanism that calculates the interactive between queries and code snippets. The attention to the corresponding code snippets on queries and the corresponding queries on code snippets is calculated, respectively. This allows HECS to incorporate the importance weights of each other's presentation of learning and HECS's input layer more interactive at the token level of queries and code snippets. Finally, the vectors for both query and code side are generated and the similarities are computed to retrieve the code snippets.

MSR [122] is a model from the perspective of text representation to maximum semantic matching. The source code snippets and query text are first re-encoded by the text cleaning step and they are processed by MSR model for code element extraction and text normalization. Furthermore, it retrieves text features, implicit semantic features, deep semantic features, and other related features. The semantic representation features classified into three levels: text feature (edit distance, co-occurrence index, and vector distance), latent semantic feature (latent semantic, topic feature), and deep semantic feature (word2vec, weighted word2vec). Learning-to-rank [67] model takes the output of the multi-level representation features and gives code snippet sorting by their semantic relevance.

PSCS [507] is a path-based neural model that encodes both the semantics and structures of code represented by AST paths. It adopts tree-based embedding that extracts paths from the AST of code [7] to get a comprehensive view of code. By taking the tree-based approach PSCS can explore both the semantics and structures of code by walking from one leaf node to another. PSCS embeds the queries and AST paths into a vector space by using a query encoder and a code encoder, respectively. To embed source code, it takes BiLSTM [509] and an attentive fusion layer which can be used to compute the contribution of each path and integrate the hybrid path representations into a single representation. The model updates the parameters and embedding matrices by learning the correlation between the query and the code. Finally the trained model generates embedding vectors so that PSCS can compute the similarities to rank the code snippets and retrieve.

Heyman and Cutsem [183] proposed formulating the code snippet retrieval problem as a k-nearest neighbor problem. The queries and snippets from the projects are represented into a shared vector space such that they can be matched with any given query with the matching snippets. Due to the lack of sufficient training dataset, they independently train two embedding models that separately capture the similarity between (1) queries and descriptions; and (2) queries and code fragments. The two query-description similarity models are generated by using Neural bag-of-words (baseline), and the universal sentence encoder (USE) [77]. For the query-code similarity model, they leveraged NCS [449] and UNIF [71] models from Facebook. Finally, an ensemble model computes query-code similarity as a linear combination of the fine-tuned USE, and NCS model is generated to retrieve the results.

COIL [298] employs a hybrid representation to model query-code correlations, which aid to address the OOV problem by representing low-frequent tokens. It also learns the query-code interaction so that it may avoid independent similarity matching and the small training dataset problems. The hybrid representation indicates that the local representations are used for lexical matching and the distributed for semantic matching. The query-code interaction is conducted using a 2D interactive matching matrix and a CNN model to compute the similarity between code and queries. It allows COIL to understand each token better. The principle behind the COIL is to treat code search as a problem of image recognition. Lexical and semantic matching matrices could be viewed as images to measure the similarities.

CoNCRA [114] is a proposed approach based on deep neural network CNN to train sentence embedding, which consists of Stackoverflow question and answer posts. To obtain the sentence embedding, the approach combines the word embeddings generated by Word2vec's skip-gram. CNN is leveraged for the combination to prioritize local interactions (i.e., terms nearby). CoNCRA adopted an objective

function that prioritizes the relative preference of the code snippets. This helps the model to separate the correct answers from incorrect ones during the training phase for joint embedding.

Another CNN based approach named COSEA [542] is proposed with layer-wise attention for capturing the valuable code's intrinsic structural logic. The approach contains a variant of contrastive loss for the training phase to distinguish the ground-truth code from the most similar negative samples. The intuition behind is code blocks are crucial to understanding the comprehensive meaning of a code snippet, and previous studies ignored this. Therefore, to capture the semantic information more accurately, a well-designed network is proposed and, captures code blocks' features, and combines them. Then, CNN is leveraged to learn the block representation. Each convolutional layer can aggregate local information based on shorter block representation output by previous layers. After the aggregation, layer-wise attention learns the respective weight on each output representation vector and re-scales this vector with the weight before being fed into the next layer. The role of such weights is to let the successive convolutional layers pay attention to more important representation vectors when composing extended block representations for code.

DGMS [309] is built on graph neural networks for semantic code search. The approach is based on the author's investigation that implies higher abstraction requires capturing more semantic information. Intuitively, graph-structured data represents a much higher abstraction than plain sequences or tokens. To utilize such data, a specific graph generation method that represents both query terms and source code into a unified graph-structured data is introduced. Furthermore, DGMS leveraged the power of graph neural networks (GNN) [465] to learn all node embeddings for graphs to capture semantic information for individual NL-term graph or code graph. Finally, DGMS reports a semantic matching operation using the cross-attention mechanism to get more fine-grained semantic relations between the NL-term graph and the corresponding code graph for updating the embedding of each node from both graphs. Code search is performed by aggregating all node embeddings from both sides to obtain two graph-level embeddings, and DGMS computes ultimate similarity score to retrieve code snippets.

APIRec-CST [85] combines the API usage with the text information in the source code based on an API context graph network and a code token network. As such, the model can simultaneously learn structural and textual features for recommending correct APIs. An API context graph contains all semantics in subgraphs and integrates these semantics as a whole. The API context graph network consists of an embedding layer and Gated Graph Neural Networks (GG-NNs) [303] which learns to extract informative structural features. The token network contains a bag of code tokens and infers the developer's intent jointly with API context graph network. To obtain a joint vector, the joint layer combines vectors from two different networks by concatenating them and then applying a fully connected layer designed to further learn the joint semantics from both sides in a holistic way. APIRec-CST leverages softmax considering each API as a class and the API recommendation as a classification task to retrieve correct APIs with its code snippets.

Zhao et al. [600] applies adversarial learning [367] that plays the discriminator role to distinguish ground-truth code snippet from <natural language question, code snippet> pairs. The goal of the approach is to learn a question-code (QC) model to retrieve the highest score code snippets for a given question. Adversarial learning alleviates the bi-modal learning challenges, representing learning of two heterogeneous but complementary modalities, which has been known to be difficult. To do so, an adversarial QC generator selects unpaired code snippets, which are difficult for the QC model to discriminate for its ability to distinguish top-ranked positive and negative samples. Furthermore, a question-description (QD) relevance model is employed to provide a secondary view of the generated adversarial samples. QD model predicts a relevance score for a pair of a given query and natural language descriptions of code snippets. A pairwise ranking loss is computed between the ground-truth code and the adversarial code to decide the final code snippets to recommend.

CRaDLe [166] is a statement-level semantic dependency learning approach that purifies code representations via fusing both the dependency and semantic information. It learns a unified vector

representation for each code and description pair for modeling the matching relationship. CRaDLe leverages <code, description> pairs to generate vectors by the code encoder and query encoder, respectively. The encoding code phase is separated into two (i.e., Statement dependency and Statement-level tokens). The dependency information is extracted by leveraging PDGs of the code snippets and leverages multi-layer perceptron to embed statements. To embed statement-level tokens, an embedding layer embeds tokens in each sequence into vectors individually and an attention layer is utilized to compute a weighted average. Later, bi-LSTM network is adopted to encode the sequence of the statement embeddings and use the last hidden state as the vector representation of the code. Encoding description phase also takes bi-LSTM model with maxpooling to embed. Finally, cosine distance calculation is used to decide the candidate code snippets for a given description.

NJACS [204] is a two-way attention-based (i.e., global attention and attentive pooling) neural network based on the data from Stackoverflow. NJACS first embeds the queries and codes using a bi-LSTM with pre-trained structure embeddings separately. Next layer uses the sequence-based embedding RNNs to learn the underlying semantic representation by summarizing the sequential structure vectors of tokens and words from both directions. Then, it learns an aligned joint attention matrix and generate the different directions of attentive pooling based weights for each other's representation. Deriving the pooling-based projection vectors in different directions, it can guide the attention-based representations. Finally, NJACS has a similarity matching layer to computes the similar values of the bi-model attention-based vector representations to score the matching degree between NL queries and code snippets to retrieve.

By raising the suffering points of CNN-based and RNN-based approaches, Wang et al. proposed a framework named TranS^3 [550] for code summarization and search based on Transformer [535]. CNN-based approaches are suffering from long-distance dependency problem [544], and RNN-based approaches are suffering from excessive load imposed by sequential computation [212]. The code summarization component initializes by preparing annotated <code, comment> from the dataset to train a tree-transformer model that encodes the source code into hidden vectors. These pair vectors are also used for a deep reinforcement learning model (i.e., actor-critic framework). The actor network is a formal encoder-decoder model to generate comments given the input code snippets. In contrast, the critic network measures the accuracy of the generated comments and provide feedback to the actor-network. To search for code snippets, TranS^3 encodes NL query, previously generated comments, and the code snippets into the vectors by using transformer and tree-transformer. Two similarity scores of query/code and query/comment are measured to derive their weighted scores and retrieve the results.

Yin et al. [590] proposed a search tool that combines neural network and question-driven approach. Specifically, they constructed the search space with question ("how-to-do-it" posts) and the corresponding answer code snippets from Stackoverflow (StaQC [587] dataset). The description (question) and the source code are converted to word embedding and fed into a representation module based on stacked attention layers of Transformer [535]. Such a module builds multi-grained attention representation matrices. The representation matrices are integrated into semantic matching matrices using matrix multiplication, and such matrices are integrated into a single matching image again. The matching score is computed using the matching image with two convolution layers of max-pooling. Finally, the matching score is computed according to the features extracted by such layers using a single-layer perceptron [442].

3.4.6 Other Approaches

Researchers have attempted to model the problem of code-search as a solvable problem from other domain such as constraint programming 3.4.6.1, approaches that leverage the API usage 3.4.6.2, attempting code search over binary executable 3.4.6.3, searches that focus on design and user

interfaces 3.4.6.4, utilizing clone detection techniques 3.4.6.5 and providing refinement on results from other code search engines 3.4.6.6.

3.4.6.1 Constraint Programming based Approaches

Some code search approaches attempt to support users searching for code using example-based query models to conceptualize the users' needs during the search. These approaches rely on an input/output query model where users provide an example of what they want to do but may be difficult to describe without one. The approach transforms the source code repositories and encoded them as constraints. For every query, the I/O example is transformed into constraints and sent as input to SMT Solver along with the encodings of the source code repositories. The SMT Solver identifies the code from the repository that meets the I/O example, which forms the results.

Satsy [495–497] is one search approach that works in two phases: offline encoding and online search. The encoding module encodes each snippet's semantics from the dataset as a logical formula concerning the input and output variables used in the snippet. Satsy replaces the values from the given input/output examples with its variables in the logical formula in the other online search phase. It then checks the constraint's satisfiability using a solver where if the constraint as satisfiable, then the snippet is considered a potential match.

As for an extension of the previous work [497], Stolee et al. [498] extended Satsy by supporting the search for programs with multiple paths. To do so, the constraint solver must be invoked on each path in a program, rather than each program as a whole [497], to determine if the program matches a specification. Encoding multi-path programs as constraints require symbolic execution. Satsy also uses a new ranking algorithm based on the scored match between an input/output query and the symbolic execution-driven program paths. They further generalize the encoding to include multi-path, non-looping programs by integrating symbolic execution into the indexing phase. Satsy joins the encoded specification with each path with a type signature match and invokes the Z3 SMT solver [113] for final matches.

Quebio [230] is an SMT solver based approach towards code search. It consists of two phases: the symbolic encoding phase and the query phase. Quebio begins by performing a symbolic encoding phase on the given Java methods by extracting the traversing path from the method's CFG. These paths are then changed to Static Single Assignment (SSA) [2] form and encoded in its symbolic constraints via syntax-directed translations [2]. For all the code identifiers such as API class name, Quebio extracts its alternative specifications from the documentation. In the query phase, for a given set of Input/Output examples, Quebio finds the candidate methods with the same types of variables (i.e., integers, strings, etc.) and invokes them as constraints in every path of the methods. For every path where the IO examples satisfy the full path, the method is considered a candidate for the search.

Later in 2020, Quebio [90] was updated to use symbolic execution engine [93] for using with collected Java methods and specifications of the APIs. The engine generates and explores the execution paths, computing the values of variables as the expression on the inputs' symbolic value. For each path, every expression's values and conditions are combined as a logical formula representing the method's semantics on that path. Finally, for each method, keywords found from the API documentation are extracted, and the top 5 frequent appearing keywords, calculated using TF-IDF, are associated with the method. In the search phase, Quebio takes as input a group of input/output examples that specify the expected IO behavior of the requested method and an optional set of keywords that capture the method's semantics. When searching, if there are optional keywords provided, Quebio calculates the textual similarity between given keywords and the summary of each method within the codebase. Only the methods that had positive similarity from the keyword matching are used as potential candidates for example-based matching. Each method is checked against the IO examples

by calling in the symbolic execution engine for each method's set of parameters. If the engine's output is a satisfiability result for the method, it is considered a likely candidate.

3.4.6.2 API usage Code Search

Some researchers have focused on API usage patterns to retrieve users' desired API code snippets. Using some APIs with complex not well-documented is known as such a challenge [571]. Despite the existence of such documentation, they are often outdated [290]. This constitutes the main reason why developers often look for source code examples for specific APIs on the web.

For example, MAPO [571] mines frequent usage patterns of APIs. It aims to retrieve the common usage patterns of APIs, including the order they are used. The framework consists of five components: a code search engine, a source code analyzer, a sequence pre-processor, a frequent miner, and a frequent sequence post-processor. The key idea is to use BIDE [543] algorithm that enumerates closed sequential patterns (i.e., without any proper super-pattern while having the same frequency) in a depth-first search. MAPO leverages this algorithm to mine closed sequential patterns and retrieve API usages for the given query.

PRIME [368] is a proposed tool that focuses on how a particular API is used. Using software code repositories, they proposed using code snippets for a particular API from various resources, including open-source scripts, tutorials, and forums. These snippets then undergo typestate analysis on the API's specification, allowing the capturing of sequences for a particular API method revocation. These specifications are consolidated in the form of a sequence of invocation for the API. A tool called `PRIME` was implemented in this approach to demonstrate its effectiveness.

SUSHI [392] extracts API usages from the bytecode of mobile apps to train an improved version of a Hidden Markov Model [417] (HAPI [391]). Sushi first uses an API method sequence extractor to extract API call sequences from the dataset using GROUM [394], a graph-based model of object usages. Second, a HAPI learner component is leveraged to train HAPIs from the sequences that describe the method calls involving one or more API objects. Each of the sequences consists of several states that represent the internal states invoked in method calling. Third, the Indexer that extracts and tokenizes the API names, technical terms, and index code examples. Fourth, Word Vector Learner is used to learn word vector representation for the technical terms using the Glove [408]. Finally, once an NL query comes in, the search process is activated after embedding the query. The similarities are computed and rank code snippets them to retrieve the potential candidates.

PropER-Doc [342] is an API type code search approach that recommends results that meet the API usage. It takes the name of the API type as an input query and attempts to construct links between the API elements. These links are constructed structurally (i.e., type hierarchy and contentment relationship), and conceptually (i.e., descriptions in the API document) considered the API call type. Later, PropER-Doc collects potential candidate code snippets from a web-based code search engine such as Koders[11]. PorpER-Doc parses them and checks for the invoked API calls within them. These parsed API calls are annotated based on their types and compared with the query types. For each relevant API call, a level is used as an indicator to demonstrate its importance with the query API calls. Finally, every candidate is divided into a different group based on its type usage. A diagrammatic representation is generated that provides an overview of object interaction of the API usage. These groups are then presented as ranked results based on the significance of the types of API calls.

LibFinder [401] is one such approach that leverages the usage history and semantic similarity between a library and the given software. The underlying technique to find the correct library candidates is to treat it as a multi-objective optimization problem. The main objectives to be solved by the optimization problem for the given library and current code are: (1) maximize the co-usage in them, (2)

[11]Koders is a no longer maintained project

find semantically similar libraries, and (3) mining the overall recommended libraries. These objectives are searched using the algorithm NSGA-II [115], a non-dominating genetic sorting algorithm.

There exists a need for code generation as not all API documents have code examples, and web pages retrieved by general search engines such as Google [156] or Bing [364] tend only to have API names without any examples. SWIM [419] retrieves code based on capturing API names and discovering the API usage examples. It has a component called API name mapper, which leverages clickthrough data, a log of (query, URL) pairs from a general search engine Bing [364]. These clickthrough data helps SWIM to expand the user query with API names. As aforementioned, many clicked web pages tend to mention only the API name without any code snippets, SWIM only takes API names and synthesizes code snippets using such APIs. SWIM analyzes (1) how the object is to be constructed, (2) the sequence of methods to be invoked, (3) the control flow between the object actions, and (4) the appropriate variable names. Additionally, structured call sequences are leveraged to generate code snippets with control- and data-flows. Overall, on receiving the user query, their probabilistic natural language model based on expectation maximization (EM) algorithm recommends a ranked list of APIs. It also finds relevant structured call sequences from the index using those APIs as search keys. Such structured call sequences are synthesized into solution snippets.

APIREC [385] is another code search tool that computes the most likely API call to be inserted by developer based on context of the location. The tool works on three key ideas. First it developed an association-based model to capture change patterns I.e., capture the frequent, co-occuring, fine-grained code patterns. Second it adds the 'change context' towards the captured change pattern that includes the next method call added or revoked. Furthermore, APIREC adds code context from the location of the changed code lines in the form of code tokens. Finally, APIREC cleans the context and changes that are not useful in recommendation by relying on observation that for a larger number of changes across project, the project-specific changes are less likely to appear frequently to other projects. In the final step, APIREC determines the likelihood of user inserting an API method call towards a location by considering the context changes from their model, code context of the recommendation point and the trained inference model. If the output determines that an API method insertion is likely, then it returns that in the list of resulting candidates.

Lee et al. [282] focused on leveraging the comments within the code snippets and proposed a comment-driven API usage code search. They defined several <comment, code> patterns based on their observation and extracted such pairs from regularized code snippets. This process explores more code snippets related to the corresponding comments, increasing the chances of finding more API usage patterns. Once finding an API usage pattern (i.e., API usages that recurrently appear in multiple projects), several terms will be identified to identify similar patterns from the comments. When the user provides an NL query to search API usage code snippets, the system performs matching by Vector Space Model [459].

AUSearch [12] recommends code snippets related to API usages. It ensures that the retrieved code snippets are invocations of the APIs specified in the query by performing type resolutions. The query is defined as it takes one or more API signatures to the code snippets, including all the specified API signatures. By leveraging GitHub API, AUSearch obtains Java files with the user query, and its package analyzer takes the files to get required Jar files, which contain corresponding imported packages. A type resolver takes a query, Java, and Jar files to resolve the types of variables inside the files and return resolved Java files. Finally, AUSearch matches between resolved files and the user query to retrieve the results by checking the class's resolved type the method belongs to matches the type in the user query.

3.4.6.3 Binary Code Search

Traditional code search approaches rely on analyzing source code's presentations, such as structural information leveraged for accurate results. However, in some cases, such approaches are not feasible

due to a lack of access to full source code, i.e., only compiled and binary format of the code is available. Some approaches [79, 111, 254, 578] address this challenge by using the binary and executable binaries as search space.

David and Yahav [111] proposed a new approach towards code search over executable binaries. The approach is based on computing the similarity between functions in its binary form. They propose decomposing the functions by considering their CFG as tracelets that are continuous, short, and partial execution traces of the functions. To find similar tracelets, they consider it as a constraint-solving problem. The similarity distance between different tracelets is based on a similar number of constraints satisfied by each of the tracelets. These constraints are transformations applied to each tracelet that transforms the memory layout and the register allocation for each function. This approach was implemented as a tool called TRACY that leveraged the IDA Pro disassembly toolkit to extract the functions and tracelets.

Rendezvous [254] is a search engine approach that leverages the binary code to build the search space. Its approach is based on performing three independent analyses on binary code's disassembled functions: mnemonics, control-flow graph, and the constants. The disassembled functions are tokenized and used to extend the search query terms further. The terms themselves are used as n-perm model [240], a set-based as opposed to the classical n-gram model, which is sequential based.

BINGO [79] is a proposed binary search engine that performs semantic matching by combining a set of different techniques to find relevant code. First, it uses selective inlining, where the libraries and user-defined functions are inlined into callers to capture functions' semantics. The approach selects the callee functions explicitly based on invocation dependency patterns that narrow the list of potential candidates. Second, it uses an architecture and OS neutral filtering technique that generates functional models of partial traces' variant lengths. Finally, the semantic features from the function models are extracted where the semantic features capture the machine state transition in the form of input-output pairs. This allows scoring to be assigned for functions based on similarity, permitting a comparison between the binary functions and the input code.

BINGO-E [578] is a binary code search that considers high-level semantic and structural features, based on the previous approach named BINGO [79]. The high-level semantic features are used to find the library calls, and the structural features are used to find information about the system calls. BINGO-E leverages a clone detection technique [87] that uses CFG as the centroid of 3D coordinate to reduce the problem of graph isomorphism (graph equivalence) as a simple distance calculation between centroids. As opposed to capturing execution traces (i.e., BINGO), BINGO-E uses the framework UNICORN [533] for emulating and extracting the partial execution traces from the functions of the given binary. In principle, it makes BINGO-E compare the traces (i.e., graphs) of the functions from the binary code in the search space with the traces input query code using Jaccard Distance to compute the similarity.

Gemini [577] is a neural network-based approach for detecting binary code similarity for code search. Gemini utilizes a new graph embedding network to convert the graphs into embeddings by assuming that a binary function can be accurately represented as a CFG with attributes attached to each node. Attributed Control Flow Graph (ACFG), which treats each vertex as a basic block labeled with a set of attributes in the graph, has been used instead of normal CFG. To compute graph embedding, it combines a graph embedding network named Structure2vec [107] into a Siamese network [65] which captures that the graph embeddings of two similar functions should be close to each other in the vector space. Finally, the computation of vectors is performed to retrieve the binary code.

Li et al. [302] investigated the novel problem of deep collaborative hashing codes on user-item ratings to build a deep learning framework for *Deep Collaborative Hashing* (DCH). DCH employs neural networks to learn user and item representations and make these close to binary codes. They also targets to minimize the quantization loss which is a well-known problem with hash collaborative filtering. To conduct collaborative filtering first, DCH adopts matrix factorization which is proven

to be effective [268]. It decompose the rating matrix into two low-rank matrices which are good at reconstructing the rating. The model is composed of two neural networks for learning user preference and item latent features, respectively and to reconstruct ratings. For binary representation learning, DCH needs to exploit the obtained model to binary code. As aforementioned, the normal hash collaborative filtering approach introduce large quantization loss. Therefore, DCH takes soft function sign (softsgn) which converges polynomially and its activation distribution is around the knee and has smoother asymptotes which makes it not easy to be saturated. Also, softsgn is differentiable which makes the training networks using gradient descent possible. Finally, DCH takes the vectors to compute the similarity and it can retrieve the binary code as the final results.

3.4.6.4 Code Search for Graphical User Interface

The graphical user interface (GUI) for software development is a complicated process. In many cases, despite being a designing task, it needs to be done using code to achieve a good user experience. The designs that are created are difficult to maintain and understand, leading to issue with code-reuse. However, the user interface is a critical component for the success of any software.

Researchers have proposed different approaches that aim to simplify and automate the process of exploring and building GUI's by lettering users reuse the existing similar GUI's within data repositories. The retrieved result is provided in the form of an interface for users to interact with the code to replicate them. Some approaches [39] take input sketches from the user, identified its features, and apply various code search techniques to retrieve similar designs from the search space. While another approach [431] empowers the search by seeking user to 'draw' the required interface design within the search engine, potentially with additional context such as keywords relevant towards the search.

GUIFetch [39] is a search technique that provides GUIs and transitions similar to the input sketches and keywords provided by the user. In the first phase of analysis, GUIFetch finds relevant apps from code repositories using the keywords from the query in two ways: (1) looking for Java source files that contain the keywords along with the term 'Android'; (2) search for Android manifest, XML files for the keywords and terms of 'Android', 'manifest', 'application', and 'activity'. For each of the repositories retrieved earlier, it uses existing plagiarism detection technique [414] to remove duplicates. In parallel, a sketch parser is used to extract the GUI hierarchies from the given sketch. The sketch parser extracts the attributes such as the type, height, width, dimension, and texts from the sketch and the hierarchical elements. Using a combination of elements and attributes, GUIFetch builds a graph where nodes are the elements (screens) and edges are the attributes that provide the transition between the screens. Similarly, the same process is applied to the repositories retrieved earlier. The second phase of GUIFetch takes as input the source code of the non-duplicate apps, the GUI hierarchies, and the graphs, computed in the analysis phase. This phase's output is an overall similarity score between the app and the sketch by computing the similarity between each screen in the app and that in the sketch. This similarity is based on the type matching where elements and attributes within the sketch are mapped from the source code files. Similarly, a transition score between each graph's nodes and its edges between the screen is computed. The combined scores of the computed scores are used to rank the candidates.

SUSIE [431] takes as input a set of keywords and a Scalable Vector Graphics (SVG) file that can either be provided by the user or drawn by the user using the interactive interface of the engine. The SVG file is transformed in describing its hierarchical components, i.e., its position within the sketch and the relationship with other components. For every component description, it also includes a set of Java Swing/AWT or Android widget (component) type that can implement a similar component. The keywords from the user query are forwarded to S6 code search tool [430] that searches for Java classes or methods based on the keywords provided by the user and the semantics of the target code. The S6 searches for user interface snippets that can be used to implement the Swing/AWT component or a

non-private method that results in such component. For each of the resulting candidate, S6 ensures that the code compiles and runs successfully. Finally, each of the interfaces retrieved is checked for the description of its hierarchical components, and every candidate that matches is presented to the user as a result.

Xie et al. [574] also proposed an approach for UI code search with a tree matching algorithm that considers both the tree structure and the visual representation information. The approach converts code snippets into visual-representation trees that encode visual information for subsequent comparisons. The model takes manually labeled sketches (e.g., a picture, an SVG, or an XUL file) by the authors and their corresponding UI code to be learned. To convert a sketch into a visual-representation tree, they adopt pix2code [43] that can extract code structure. For the neural network, they employ CNN (i.e., VGGNet in the paper) that can learn the types and positions of UI elements from a given sketch. The CNN model outputs leaf nodes of the visual-representation tree and the system train again to recover the internal nodes by using seq2seq model. Previous step returns an input sketch or candidate code as an ordered list of root-to-leaf paths with visual information. It compares the similarity of two such lists (pair-wise) next to allow the system to support inexact matches. Later, they also consider comparing two ordered lists of sequences by taking a concept, weighted sum of sub-region similarities which assumes that if two user interfaces have similar sub-regions, the similarity tends to be large.

3.4.6.5 Clone Detection based Code Search

Some code search approaches proposed using techniques from other software engineering domains such as code clone detection [446].

The approach by Keivanloo et al. [248] uses a combination of Baker's p-strings [26] for Type-2 (i.e., identical code fragments except for variations in comments, identifiers, literals, types, and whitespace) similarity with Carter et al. [76]'s vector space model based approach for Type-3 (i.e., clones with added, removed, and modified statements) similarity search. Also, they leveraged frequent itemset mining [57] to detect popular abstract solution. Overall, the approach (1) generates a query with the most relevant encoded code patterns, (2) extracts the complete popular abstract solution, and (3) retrieves the most similar code snippets to the output of the second step.

Liu et al. [322] proposed refining the search results to understand the candidate methods. The research prototype called CodeNuance provides similarities and differences between the resulting code candidates in a graph structured format. The tool takes input a ranked list of returned methods by their keyword-based search engine called KeywordRec, as a query. CodeNaunce then groups these methods and uses CCFInderX [239] to find the Type-1 and Type-2 clones [446]. Later it applies code difference techniques [306, 307] to detect the code differences between the method groups. Leveraging the output, the tool models the relationship between method groups with their differences as an interactive exploration graph. The visualization of the exploration graph allows users to evaluate the difference between methods with relative ease.

Rahman and Roy [421] proposed a context-aware recommendation for exception handling code examples. The approach leverages code snippets' structural and lexical features along with the quality of the exception handlers. It identifies all the API objects and their static relationships, including the data dependencies in the code, and generates an API usage graph. Furthermore, it applies a lexical feature-based code cloning technique called NiCad [443] to determine the lexical similarity between context code typed in the IDE and the candidate snippets.

Another approach towards leveraging the code clone detection technique for code search is MUSE [373] which focuses on method usage examples. Once a user provides a project's source code with Javadoc (if available) and a list of client projects using the given project as a library, MUSE retrieves method examples. It parses client projects to collect method usages and computes a static backward slice

(i.e., JDeodorant [532]) for each usage statement to detect the sequence of relevant steps of the method invocation and prune out irrelevant examples. Since different projects tend to use the same methods, MUSE identifies similar examples through Simian clone detector [179] (clustering similar code examples).

3.4.6.6 Ranking Enhancement for Code Search

Niu et al. [398] trained a ranking schema to rank code snippets for new queries at run-time. For a given query, the approach retrieves all the code snippets with the class or method names written in the query from the search space and computes the cosine similarities [459] between the query and the corresponding code snippets. The retrieved candidates are ranked using the trained ranking schema. To create the training data, they identified 12 specific features and collected the relevances between the queries and the corresponding code snippets. A pairwise learning-to-rank algorithm called RankBoost [136] among all the types [291] (i.e., Pointwise, Pairwise, and Listwise) is used to train/learn how the 12 features should be combined to build a ranking schema. The algorithm is primarily intended to train the weights automatically.

QualBoa [119] is designed based on a code search infrastructure named Boa [125, 126] to locate relevant code snippets that are reusable by computing quality metrics. It accepts the user query in the form of a signature, which is similar to the Java interface that consists of all desired class methods. Boa engine [125, 126] retrieves the files with values of quality metrics for each. Later, the files are parsed to generate ASTs. Then, Functional Scorer decides whether each file fulfills the functionality posed by the query. Reusability Scorer takes quality metrics for each file to value their reusabilities. Their functional scores are used to rank the final results.

Kessel and Atkinson [252] consider the candidate ranking as an optimization problem. Their approach's main component expects a list of code candidates and user-selected quality criteria, such as the indicator metric measures, objectives, and priorities for desired non-functional properties. This ranking component is implemented to compute a partial ordering of non-dominated sets, containing all the solutions with the optimal trade-off with selected criteria. It determines the next sub-criteria for the iterative ranking of the list based on the priority. Once it exhausts all the sub-criteria in the list, a non-dominating set is left that is returned as ranked code snippets consisting of this non-dominated set. In the next step, a joined list with all the resultants of the sub-ranking operation with the current non-dominating set creating a super non-dominated set (i.e., consisting of the list of candidates that were iteratively ranked) These computations' results is a list of final ranked candidates where each stores information about its computed rank. This ranking approach is integrated into a code search platform called Merobase [361].

Table 3.7: Other approaches in Code Search

Other Code Search Approaches			Output			Dataset			Index				Input								Retrieval								Presentation			
Category	Approach	Key Technique	*General Code*	*Sketches/GUI*	*API usage*	*Specific Open Source Projects*	*Super Repositories*	*Developer Q&A*	*Inverted*	*Database (B+ Tree)*	*Graph Index*	*File Prefix*	*Sketch File*	*Natural Language*	*Binary*	*Code Fragment*	*Query Language*	*Other*	*Input/Output*	*Software Specification*	*Textual Similarity*	*Graph Similarity*	*Clone detection*	*Execution Trace*	*Usage Model*	*Type Links*	*Vector Similarity*	*Solver*	*Search Engine*	*Idea*	*IDE Extension*	Programming Language
Solver based	Satsy [495–497]	Composes the code fragment as elements of solveable equation	✓			✓				✓							✓			✓								✓	✓	✓		Java, C
	Extended Satsy [498]	Extends the logical solver by treating the equation as syntax tree	✓			✓				✓		✓							✓	✓		✓						✓	✓			Java
	Quebio [230]	Uses CFG to extract elements for Z4 SMT solver	✓				✓									✓			✓			✓						✓	✓			Java
	Extended Quebio [90]	Uses Input/Output with NL query	✓				✓	✓						✓					✓		✓								✓			Java
API based	MAPO [571]	Provides code snippets of API usage based on query leveraging other source engines	✓		✓									✓							✓								✓			Java
	PRIME [368]	Leverages the partial temporal specifications of input to find similar typestates to find sample	✓		✓	✓									✓						✓								✓			Java
	SUSHI [392]	Extracts API usage from bytecode of mobile app and uses them as vectors to compute similarity	✓													✓	✓				✓					✓				✓		Java (Android)
	PropER-Doc [342]	Captures the structural and conceptual links between API implementation and documentation	✓		✓	✓														✓					✓						✓	Java
	LibFinder [401]	Uses NSGA algorithm to find maximal matching library	✓				✓										✓										✓				✓	Java
	SWIM [419]	Leverage clickthrough data from normal engine to refine query	✓		✓		✓								✓						✓							✓	✓			C#
	APIREC [385]	Leverage the predictiveness of repeated code changes to provide API recommandation	✓														✓				✓	✓									✓	Java
	Lee et al. [282]	Discovers API usage pattern via keyword mining and semantical similarity			✓		✓								✓						✓										✓	Java
	AUSearch [12]	Approach that attempts to allow type constraints of the required API			✓		✓											✓								✓			✓			Java
Binary code search	Tracelets [111]	Code search over binary exploiting the execution traces	✓				✓			✓					✓							✓		✓					✓			Binary (x86)
	Rendezvous [254]	Uses mnemonics, CFG and data constants from binaries of different optimized compiler to compare against given query	✓			✓				✓		✓				✓					✓			✓					✓			C, C++
	BINGO [79]	Uses partial traces on a binary function that matches different function semantics in agnostic manner	✓			✓									✓						✓		✓					✓				Binary C
	BINGO-E [578]	Extension of BINGO [79] leverages partial traces of different lengths	✓			✓										✓							✓		✓				✓				Binary C
	Gemini [577]	Trains Neural network that embeds CFG of binary functions; leverages Struct2Vec										✓				✓																
GUI code search	GUIFetch [39]	Takes input of Sketch file and finds similar GUI's		✓			✓				✓			✓	✓						✓								✓			Java
	SUSIE [431]	Extract keywords from sketches and use existing search engines to find similar sketch code		✓			✓								✓		✓				✓								✓			Java
	Xie et al. [574]	Converts visual representation as ordered tree and compares it against others		✓		✓				✓		✓		✓							✓	✓									✓	Java
Exception handling	CodeNuance [322]	Leverages code clone technique that provides additional information for ranked list of methods taken from other search engines	✓				✓			✓						✓							✓						✓				Java
	Rahman and Roy [421]	Searches for exception handling code snippets using lexical and structural context	✓				✓			✓							✓					✓	✓									✓	Java
	MUSE [373]	Mines and ranks code examples for specific method			✓	✓											✓						✓									✓	Java

3.4.6.7 Different research directions and ideas

Recently, Nguyen et al. [388] explores a new research direction namely personalized code search which focuses on personal code patterns of developers. The research prototype, PCR consists of three components: (1) *History Extractor* extracts personalized object usage instances, (2) *Model Learner* trains Occurrence Ranking Model (ORM) for declaration types and initialization method sequences, and (3) *Code Recommender* retrieves code snippets by leveraging the ranking model from the previous step. To extract object usage instances of a user, PCR uses GROUM [394] which is a graph that represents the object usages in the source code. Then, the ORM captures declaration types and initialization method sequences by leveraging fuzzy set theory [265]. Finally, it uses union operation of fuzzy set theory to compute correlation scores for both types and method sequences to rank code snippets and recommend.

Huang et al. [211] presents a different research direction of code search that focuses on similar code snippets for smart contracts. The approach takes two similar contracts which illustrates how a software feature is implemented and retrieves differentiated code which is defined as the source code except the cloned ones. Overall architecture is composed of two phases: (1) smart contract similarity analysis and differentiated code recommendation. In the first phase, it measures the similarity between smart contracts through the syntax and semantic analysis. Beginning with extracting the syntax tokens and identifiers from the source code of smart contracts, it extracts the code syntactic information from code syntax tokens and extracts semantic information from the code identifiers, respectively. The differentiated code recommendation phase employs K-means clustering [178] and computes the similarity to retrieve code snippets for the update of target smart contract.

3.5 Code Search Infrastructures

Code search infrastructure is a higher concept than a code search engine (i.e., some code search engines are a part of code search infrastructures) as it can be used for building applications or be leveraged to evaluate a code search engine. A code search infrastructure can consist of many different components: a crawler, repository, database, indexer, search engine, code miner, and ranker. Each of the component can be changed or used depending on the requirements of developers. For instance, employing a different ranker component to increase the quality of retrieved results. Code search infrastructure is an important aspect that is used by academic as well as commercial entities to study the code search problems. This section covers the different code search infrastructures from researchers (most of which are publically available) and the commercial ones.

Several researchers have put effort in this direction. An early example of code search infrastructure is built by Inoue et al. [216–218]; an infrastructure for a component retrieval system named SPARS-J. It analyses Java source files and stores the classes and interfaces as components and the use-relations between the components into its database.

The repository of SPARS-J contains all types of the use-relations; (1) software component (i.e., a building unit of a software system such as a class or a method), (2) use-relation (e.g., a method call or a field access), (3) software component graph where a node represents a component, and an edge represents a use-relation, and (4) In-degree and out-degree (i.e., the number of incoming relations to a node and outgoing relations from a node). Specifically, SPARS-J's database called SPARS_DB includes libraries such as JDK and several software systems checked out from open source repositories (e.g., Apache Software Foundation [555], Sourceforge [488], Eclipse [556], and NetBeans [559]).

Another code search infrastructure developed by Keivanloo et al. [246, 250] named SE-CodeSearch contains a source code repository and a Resource Description Framework (RDF) repository for its data. It also consists of other features such as the Crawler, an ASTBuilder, and an Ontologizer for Back-end and a StaticSlicer, an API, and a Reasoners for Front-end. The base approach of this

infrastructure relies on the ontological representation and modeling of the source code's information named as facts (e.g., the connection between call statement and message receiver). To support various scenarios in the infrastructure, it extracts all the possible facts by leveraging existing code analysis techniques. They model source code using Description Logic (DL) and use Semantic Web reasoners to complete or deal with missing parts from the code snippets. The infrastructure has an ontologizer as the major component that transforms extracted facts to a Web Ontology Language (OWL) representation. They introduced a novel ontology; *sicsont* that addresses static code analysis knowledge representation in OWL and DL.

The largest data-centric architectural infrastructure for code search has been built by Bajracharya et al. [17, 19, 310] named Sourcerer. It has specific tools to crawl source code, create and manage repository, extract features, import the data to database, and index the code. Given the tools, Sourcerer infrastructure builds *File Repository*, *Sourcerer Database*, and *Code Index*. Sourcerer applications (e.g., Sourcerer Code Search [14, 16, 17], CodeGenie [278–280, 289], Sourcerer API Search [18], Structural Semantic Indexing [20]) have been built based on such stored contents by using the supported services such as *Repository Access*, *Relational Query*, *Dependency Slicing*, *Code Search*, and *Similarity Calculation*. Particularly, Sourcerer infrastructure can automatically collect and store the code in a repository with a standardized directory structure and perform an in-depth structural analysis of the available source code (e.g., dependence analyses, and store the code structure information in a centralized database. Furthermore, with this infrastructure, it is possible to perform several mining tasks within the target repository, including code search.

An option-centric architectural infrastructure provides ease of swapping features like code miners or ranking algorithms. As an example, Sameness [349] has an infrastructure that lets the code search system easily swap out ranking algorithms without changing anything else. The infrastructure relies on four major components; (1) List Server, (2) Code Miner, (3) Sameness Algorithm Ranker, and (4) Automatic Reformulator. The List Server contains a list of URLs of repositories (only Java-related) on GitHub. It passes a repository URL to the Code Miner. Once the Code Miner receives a URL, it clones and analyzes the repository to create a technical summary of each source code. It extracts various information such as method and variable names. Then, the extracted information goes into the search engine database. The search engine uses Apache Solr framework [468] to index and retrieve. Then, Sameness Algorithm Ranker leverages their specific ranking algorithm to re-rank.

An existing commercial code search infrastructure provides more general information rather than supporting only source code retrieval. For example, Krugle [271,272] (early version) had its infrastructure providing three types of information; (1) web pages, (2) source code, and (3) project descriptions. Each has its collection, processing, and searching infrastructure. The infrastructure leveraged an early version of Hadoop [473] (a framework that allows distributed processing of large data sets) to process web pages and source code. It utilized Nutch [560] (a scalable web crawler developed by Apache) to crawl data. Nutch's search support (built on top of Lucene [558]) was used to support searching these web pages, source code, and projects.

3.6 Datasets and Benchmarks for Code Search

A collection of source code files from various software repositories forms the basis of a dataset. The dataset may also contain additional information such as Q&A posts of open developer forums and software metadata, while benchmarks can incorporate other features like fixed queries and metrics. This section explains the source code dataset, non-code dataset, and benchmarks used within the code search domain to create search bases or evaluate the code search engines. Figure 3.5 illustrates the rate for use of each dataset in the code search domain.

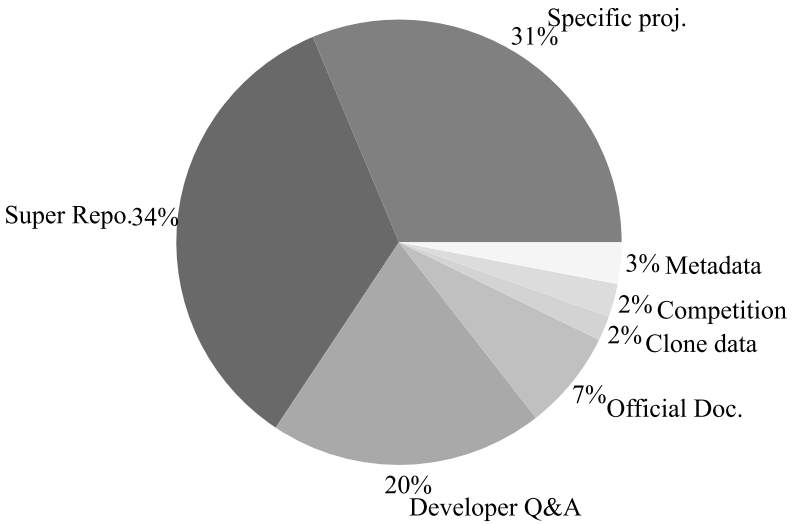


Figure 3.5: Dataset used by code search approaches.

3.6.1 Specific open-source Projects

In an earlier time before the rise of open-source software, only a limited number of projects and their source codes were actively maintained and accessible to the public. As a result, researchers intending to apply code search ideas had to leverage these sparsely available open-source projects. These projects ranged from different communities such as Apache HTTP Server [522] and Linux operating system [151]. The Portfolio [356, 359] and Coogle [452] are examples of approaches that utilize specific open-source projects to demonstrate their results.

3.6.2 Data Sources from Super Repositories

In the last decade, there has been widespread use and adaptation of open-source software. This has created thriving open-source software development communities that leverage collaborative coding systems and are made centrally available for everyone to use. Similarly, researchers within the code search have adopted the data sources that form a large set of different open-source projects when testing their approaches [14, 523].

The leveraged data source, known as the dataset, is in a file archive, where most of the source code files are extracted from different software projects. These repositories are often accessible online on code hosting services such as `Github` [557], `Sourceforge` [488], and `Bitbucket` [53]. The hosting services are built upon version control systems (VCSs) such as `Git` [311], `Mercurial` [360], `GNU Bazaar` [36], `Darcs` [109]. Software repositories are a rich source of code snippets created and curated by developers around the globe. Furthermore, the curated source code snippets in the form of datasets provide opportunities to investigate and research new ways for code search techniques.

Many of commercial code search approaches [96,155,271,384,399,469,489] and research prototypes [261, 335, 346, 349, 368, 376, 419–422, 482, 497, 553, 585] have utilized source code files collected from the code repository platform `Github` as their search base. `Github` is one of the largest super-repositories built on top of the `Git` VCS [27]. It is the most prominent hosting service [426] with more than 100 million repositories hosted as of January 2020. `Github` also hosts huge and popular projects such as Homebrew [350] (a package manager), Django (a web framework) [199], Bootstrap (a front-end framework) [490], and jQuery (a cross-platform javascript library) [159].

There are multiple ways in which the Github's data can be accessed and collected to create the datasets. The most popular way is by using GHTorrent [148], a scalable, queriable, and curated metadata database of the Github. It allows its users to download a snapshot for the entire data of the publicly available repositories. Github also has its REST API [149] to provide the dataset.

Sourceforge is another super repository used for constructing code search datasets. As of January 2020, it hosts more than 500 thousand projects. Similar to GHTorrent for Github, FLOSSMole [200] project supports in creating a dataset [371] from Sourceforge. Exemplar [352], Lukins et al. [331], and Hsu and Lin [202] crawled and utilized the data from Sourceforge.

Despite the similarity among Github, Sourceforce and the Bitbucket; Bitbucket has never been used for data collection. Because Github is more popular and hosts more software projects, provides its search API [150], and includes integrated social features, it is more attractive for software engineers and researchers [235]. However, many code search studies [261, 419, 482, 553, 569, 599] indicate that Bitbucket's data can be used for code search approaches.

For Android-centric code search approaches, there exists a website named F-droid[12]. It is a repository of Android applications of various sizes and categories. The dataset[13] contains meta-data (name, description, and version), the source code of each major version, and its most recent 'apk' file. Moreover, the version control information such as the committer user name, commit time, and commit messages can be extracted for the search base of the code search approach. Some code search approaches [225, 395] leveraged such dataset for android code search.

Although these super repositories have been used frequently to construct various code search approaches, their use has several disadvantages. For example, users tend to upload (commit and push) files unrelated to source code, such as security-related tokens or documents containing personal information. These can affect the quality of the search base. Such accidental exposure of information can have dire consequences (e.g., disclosing personal information to the public) [515] even through the code search engines. Another disadvantage of super repositories for research is that the projects' and developers' quality is hard to quantify. A significant amount of time needs to be dedicated to curating good candidate projects.

3.6.3 Other Data Sources

There exist other resources that have been leveraged for different approaches towards the code search problems. Resources such as the developer Q&A forums, benchmarks, metadata, and API documents are applied for some approaches. Such additional resources have demonstrated an increase in the code search techniques' efficiency and accuracy.

3.6.3.1 Software developer Q&A forum

Question and Answer (Q&A) forums are community-driven platforms that allow users to share knowledge with other users who participate in them. Q&A forums offer social mechanisms to evaluate and improve the quality of both the question and answer that implicitly leads to brevity in questions and qualitative answers, potentially with source code snippets [261]. Many of the Q&A forums determine a user's reputation by scores awarded by others, based on the clarity and correctness of the answers [1]. These forums have recently gained immense popularity having over 10 million users as of January 2020. The posts within such forms tend to include code snippets within the question and its answer. It makes them ideal for forming the part of the dataset for some code search approaches.

A majority of the code search approaches [11, 221, 248, 261, 284, 301, 335, 345, 368, 395, 397, 421, 482, 494, 497, 498, 536, 537, 594] have utilized the developer Q&A forum called Stackoverflow. Stackoverflow

[12] https://f-droid.org
[13] https://gitlab.com/fdroid/fdroiddata

is the largest Q&A forum that mainly contains questions and answers on programming-related topics [159]. Answers from `Stackoverflow` often are an alternative explanation for corresponding official product documentation wherein the documentation is either insufficient, does not exist, or lacks in-depth information [32]. As of January 2020, the public data dump [491] lists include a total of 176 communities (categories such as LaTeX, Linux, etc.), 58 million posts, and over 10 million registered users. A significant portion of answers includes code snippets that demonstrate the solution for the corresponding programming problem. These solutions may even contain information about the usage of a particular function from either a library or a framework [261, 528, 581].

Recently, large and systematically mined dataset using Stackoverflow are proposed and learning-based code search approaches [114, 177, 204, 542, 600] leveraged these in their evaluation. CoNaLa dataset [591] by Yin et al. consists of two parts, a manually curated parallel corpus of 2,379 training and 500 test examples, and systematically mined 600k pair examples. The pairs include Python code snippets and their corresponding annotations in natural language. Unlike CoNaLa, StaQC [587] is specifically intended for code search (i.e., not for code generation or summarization) and it consists of 148K systematically mined Python and SQL question-code pair dataset. It is mainly composed of straightforward "How-to-do-it" questions and one-to-one alignment to raise usability.

`CodeProject` is similar a platform to the `Stackoverflow` that is used by some approaches for evaluations. It contains articles on diverse topics related to programming posts. Thummalapenta et al. [525] used code search to narrow down the scope of a vast search base by mining the entire base and extracting methods. They used the QuickGraph [95], a .NET application, from a CodeProject article as one of the evaluation subjects to validate their approach. Later McMillan et al. [356] used CodeProject to experiment with Portfolio's proposed code search approach-. They picked one task (e.g., Implement a module for reading and playing midi files) from CodeProject for a fair comparison with its competitors (i.e., [82, 271, 523]).

Even though such data from Q&A forums have advantages for code search, they are not primarily designed for code reuse or the search for particular code snippets [32]. Thus, not all the posts include source code, as some are mere questions and answers either with incomplete code or devoid of any code. For example, there are multiple instances where the code is written with ellipses (i.e., "...") [482]. It implies that some people prefer to express natural language expressions that make it hard to leverage the data. Furthermore, pervasive unqualified names [505] and ambiguity in enclosing class names of method calls [105] for code snippets can affect the accuracy of the overall source code approach.

3.6.3.2 Language/API documentation

Some code search studies [82, 258, 345, 352] leveraged well-known Application Programming Interface (API) documentation. Generally, developers refer to the API documentation for examples [301] of a concerned project where such documents are the official source of information, such as the `JavaDoc` [400] or `MSDN` Library [363]. These documentations are known and followed by all vendors for consistencies usually written by multiple people, undergoing multiple reviews based on feedback from other developers [116]. The API documentation also serves as the natural bridge between the natural language query and API methods invoked in code snippets [301].

For Java API code search, `Javadoc` is frequently used within the community. Using `Javadoc` basically mitigates a well-known external threat to validity, "when the documentation is provided by a third-party, its content and format may vary" [353] about software documentation. Kim et al. [258] used `Javadoc` for evaluating the performance of their approach when augmenting code examples compared to that from the `Javadoc`. Annotating `Javadoc` permits Sniff [82] to add meaningful comments pruned to remove stop-words allowing Sniff to retrieve better code snippets. Javadoc is also leveraged to describe the functionality of API calls in various code search approaches [167,308,345,353], evaluate an API-related embedding-based approach [258], and extract more precise code snippets [609].

Code search approaches that focus on the Microsoft products need to consider the MSDN Library [363]. MSDN is an official library of technical documentation from Microsoft for developers working on Microsoft applications by using C/C++, C#, and .NET as the programming languages. It includes sample codes, technical articles, and programming information. A code search approach, CodeHow [335], leverages such library to enrich APIs for their API understanding phase. This allows the system to understand the queries by identifying the APIs appearing in them. Mcmillan et al. [352] proved that the combination of application descriptions and API documents yields the most meaningful search results.

3.6.3.3 Code clone benchmark

Several researchers have tried to use a benchmark designed explicitly for code clone detection to evaluate their code search approaches. For example, BigCloneBench [9, 511, 512] is a benchmark that clearly distinguishes between the true java clones mined from a cross-project dataset, IJaDataset 2.0 [8, 9]. FaCoY [261], a semantic-based code search approach uses BigCloneBench in a different perspective (i.e., code clone detection using code search approach). Its evaluation consists partially from the BigCloneBench's data as its search base and evaluated against the state-of-the-art clone detection tools [173, 226, 238, 444, 454]. Later Zhang et al. [597] proposed a code search approach that uses BigCloneBench as its corpus dataset. They tried to identify the API class names for a given free-form text query from the corpus expanded from the initial query.

3.6.3.4 Challenge/Competition data

Coding competitions are events where its participants try to code a program based on specific problems. Researchers have leveraged the problems and the different solutions submitted by the users to evaluate the code search engines. In these competitions, the participants submit source code, and the hosting system determines the correctness of the given code for particular problems.

For example, Google Code Jam [154] is an annual online coding competition hosted by Google. This competition data is leveraged to evaluate a code search engine. Each submission for a defined problem is expected to perform semantically similar even though they may syntactically be different. DyCLINK [502] leveraged such a dataset for the clustering software community because the ground truth is that the solving programs for the same problem are likely to be similar. FaCoY [261] has been applied to codes written for Code Jam to identify code relatives at the granularity of methods.

The latest challenge published within the code search is from the industrial researchers (affiliated with Microsoft and Github) known as CodeSearchNet [215]. This challenge aims to encourage researchers and practitioners to study and propose new approaches towards code search. Several learning-based approaches [166, 309, 507] leveraged this challenge dataset for evaluating their code search engines as its corpus has text-code paired dataset. Moreover, it consists of over 2 million data points that enable training high-capacity deep neural models on the task.

3.6.3.5 Metadata

Software project metadata includes information such as the project name, owner, description, homepage, rating of the project, creation date, code commit dates, license of files, the number of developers, etc. [44, 51, 100, 126, 172, 450]. Some open-source project repositories provide keyword matching searches over the projects' metadata (e.g., using the project name, description, programming language, etc.) [14]. Code search empirical studies [145, 171] have leveraged project metadata by claiming that metadata can play an important role in the design and implementation of code search approaches. It can be used to restrict the result sets [60, 245], supplement searching accuracy along with user query [356], change the search range in the sense of iterative search [346], and give its users a better

understanding with more information of projects [261, 482]. Also, social and human trust ability factors (e.g., project's rating, the number of developers and users, etc.) from project metadata can increase user confidence in the retrieved search results [171].

Metadata from Q&A forums supports better keyword matching on the code search approaches. Researchers concerned such metadata as a keyword search index. For example, Zagalsky et al. [594] used both the code snippets and the metadata that accompanied the code snippets at `Stackoverflow`. It supports developers to find desired code snippets that may not include the query keyword. The keyword appears in the contextual data and indicates that it has been used in the context. `Stackoverflow` allows its users to denote the best solution to a question based on whether the answer helped. Such metadata provides insight into the quality and correctness of the code included in the answer. Thus, these code snippets represent high-quality source code examples [504].

3.6.4 Existing Benchmarks

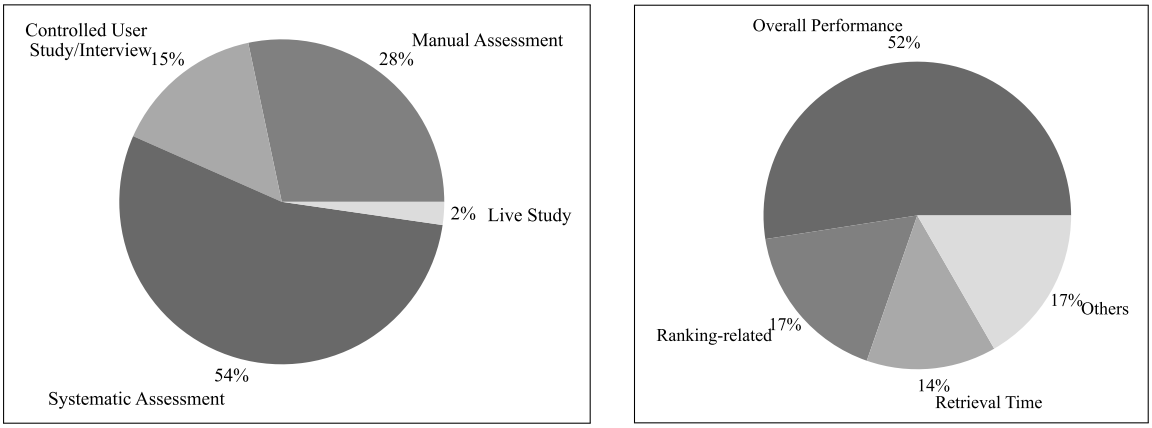
A benchmark is a package that serves as a basis for evaluation or comparison. Many studies on code search collected their datasets from various sources and incorporated several evaluation metrics. Approaching this direction leads to inconsistencies for fair comparison because their search base has different snippets, and different metrics may indicate different results. Furthermore, different queries for the same task make the retrieved results vary. These problems motivate researchers to build benchmarks for code search using common search bases and metrics or to have the same queries and answer code snippets. Therefore, the benchmarks provide the dataset, metrics, and sample results that lead the users to a fair comparison in the code search evaluation phase. There are several code search benchmarks proposed recently.

For instance, Li et al. [292] introduced a benchmark in the field of code search by providing the results of their neural code search approaches [71, 449]. The benchmark consists of a dataset as a natural language query (from Stackoverflow) and code snippet pairs about Android software development (from Github). They further employed two neural code search engines named NCS [449] and UNIF [71] to build the benchmark providing the results. To evaluate the usefulness of the benchmark, Aroma [328] is leveraged as a similarity computation method. The similarity score aims to mimic manually assessing the correctness of search results.

Another benchmark CosBench [580] combines a search base (from GitHub), query and answer pairs (from Stackoverflow), and a set of metrics. To show the usefulness of the benchmark, they employed four IR-based [326, 335, 395, 558] and two learning-based [167, 589] approaches to compare in the evaluation. For the results, the benchmark's empirical study discovered that code-specific information enhancement helps improve code search, IR-based approaches are faster in indexing and search, learning-based generally outperforms the others in terms of accuracy, and most of the existing approaches perform better on keyword queries than on phrase queries. A deep-learning approach [298] has leveraged this benchmark to study user query and source code interactions.

3.7 Evaluation

Evaluating any proposed approach's performance provides an overview of how efficient and accurate it is against others or over a particular dataset. The methodology used by various researchers differs significantly: some approach manually assess by checking the result of the approach against what is an expected outcome, others compare the performance against other state-of-the-art approaches, some approaches undertake an independent study using developers to evaluate the performance, and finally some approach focused on having a study over public forums. This section investigates the different evaluation methods and the metrics used within the code search approaches.



(a) Evaluation methods. **(b)** Evaluation metrics.

Figure 3.6: Evaluation methods and metrics.

3.7.1 Evaluation Methods

Figure 3.6a and Table 3.8 presents how many approaches take which evaluation methods. Evaluation of classical search engines has been limited towards manual relevancy checks by domain experts [341]. Similarly, code search engines' performance was manually measured in the early stage until researchers found out systematic ways due to removing the subjective bias. Since then, systematic evaluation methods (e.g., comparison against the state-of-the-art approaches and models) were used to provide a more precise evaluation for the proposed approach. Moreover, other methods such as controlled user study/interview provide opinions and insights from experts, developers, or a more significant portion of people capable of providing adequate evaluation. Recently, with the development forum's high growth, the live study method is being applied to leverage crowd knowledge.

3.7.1.1 Manual internal assessment

One of the practical evaluation methods known as *Manual internal assessment* requires the researchers' manual efforts. In this method, the researchers manually validate the results returned by the code search engines, determining their relevance to the user query and assigning a score for each of them.

Table 3.8: Evaluation assessment

Evaluation Type	Approaches
Manual internal assessment	[11,12,29,30,39,78,79,90,111,119,132,132,135,143,161,163,185, 194,196,198,211,220,222,230,241,248,261,287,300,314,322,336, 339,342,353,358,368,377,378,389,393,401,415,419,452,453,464, 478,480,482,484,486,518,523,536,551,574,577,578,582,594,606]
Systematic assessment	[3, 4, 14, 37, 71, 72, 78, 82, 84, 85, 88, 90, 97, 114, 122, 124, 166–168, 177, 183, 186, 189, 194, 196, 198, 201, 203–210, 222, 225, 231, 241, 248, 254, 261, 278–280, 282, 284, 286, 287, 289, 296, 298–300, 302, 308,309,314,316,325,328,334,347,354,356,359,377,385,388,390, 395, 398, 401, 421–425, 427, 428, 430, 432, 449, 453, 462, 466, 472, 480, 482, 484, 486, 495–498, 502, 507, 523, 538, 541, 542, 546, 550, 564, 568, 574, 577, 578, 582, 585, 586, 588, 590, 594, 600, 606–609]
Controlled user study/interview	[39, 72, 97, 161, 163, 168, 278–280, 289, 322, 325, 334, 346, 347, 353, 354, 356, 359, 373, 395, 398, 401, 409, 453, 462, 480, 482, 486, 547, 549, 552, 566]
Live study	[346, 480, 482, 550, 566]

However, this method is known as inefficient and expansive (i.e., the evaluation's manual process is very time-consuming). Furthermore, as the evaluation subjects are usually the authors, it may have a subjective bias.

3.7.1.2 Systematic assessment

To overcome the potential issues of bias in manual assessment, researchers undertook a systematic evaluation of the results. The systematic evaluation consists of a comparison against other state-of-the-art approaches and models. For example, some researchers utilize the dataset (e.g., queries and search space) and configure similar environments (e.g., Computing power, parameters) used in other state-of-the-art. This direct comparison aims to bring about a fair comparison of the search engine and its performance. Yet, a significant challenge when undertaking the systematic assessment lies in the lack of a replication package of the state-of-the-art. Another challenge is the non-agnostic nature of many search engines, i.e., and some specific target programming languages or different software paradigms. Similarly, the learning-based approaches utilize specific learning models or a combination of models from different approaches to bring new insights into the learning algorithm's characteristics.

3.7.1.3 Controlled user study/interview

Practical evaluation of a new approach can be challenging considering that there may not be a comparative state-of-the-art available. To tackle this, researchers undertake a *Controlled user study/interview* session that provides researchers with opinions and insights on the approach from real-world users who are likely to be domain experts. A controller user study/interview consists of solid criteria such as limited time, topics, tools that can be used to evaluate corresponding code search engines. As there is no standard or defined process rule on the number or occupation of involved users, type of queries, or topics, researchers have tried their best to involve as many users as possible to understand the generic opinions. This type of evaluation can be subjective, and its results are acceptable if the evaluation method is well-designed (for instance, the tasks and questions are non-ambiguous and precise), and its test subjects are experts or professionals within the domain of software engineering. Large industrial entities such as Microsoft tend to leverage this evaluation method to evaluate their proposed approach.

3.7.1.4 Live study

The goal of the *Live study* is to evaluate the approach in the real world where any user (professional or otherwise) would use and evaluate the results. These users generally are in the form of utilizing the users of popular developer forums. For example, Sirres et al. [482] took the questions from StackOverflow as their testing queries and posted the code snippets from their code search engine CoCaBu as the answer. The public (i.e., developers from the StackOverflow) judge the questions and answers by voting and commenting in the system. Researchers utilize this method on various platforms (e.g., StackOverflow and GitHub).

3.7.2 Evaluation Metrics

There exist several metrics to access code search engines. We split them into four dimensions: overall performance, ranking, retrieval time, and others (e.g., simple counting, statistical metrics, user satisfaction). These metrics are adopted for various purposes, such as to measure how the results of a code search engine are relevant for a given query, how fast it can perform, or how much query range it can cover. Figure 3.6b shows the distribution of the metrics for assessing code search approaches.

Table 3.9 illustrates the list of performance metrics that have mainly been used in the code search domain. The overall performance metrics are primarily based on utilizing the confusion matrix.

Table 3.9: Metrics for Overall Performance

Metric	Sub-Metric	Approaches
Precision	Precision	[11, 12, 78, 97, 124, 161, 163, 168, 186, 220, 248, 254, 336, 353, 354, 356, 359, 421, 464, 486, 495–497, 536, 549, 564, 582, 606, 609]
	Precision@k	[3, 4, 79, 122, 167, 205, 206, 210, 225, 261, 282, 296, 314, 377, 395, 401, 462, 480, 482, 484, 495–498, 542, 568, 585]
MAP	MAP	[4, 119, 241, 484, 498, 600]
	MAP@k	[208, 209, 231, 298, 421–425, 608]
Recall	Recall	[11, 14, 78, 124, 168, 186, 194, 196, 198, 254, 261, 284, 287, 336, 378, 421, 464, 486, 495–497, 518, 536, 549, 564, 582, 606, 609]
	Recall@k	[4, 166, 167, 177, 183, 203, 204, 210, 296, 314, 328, 401, 424, 425, 472, 484]
Accuracy	Accuracy	[84, 111, 316, 421, 466]
	Accuracy@k	[85, 114, 325, 385, 388, 390, 401, 419, 590]
SuccessRate	SuccessRate	[322, 478, 507, 552]
	SuccessRate@k	[167, 205, 298, 300, 302, 308, 309, 377, 422–425, 432, 541, 550, 606, 607]
NDCG	NDCG	[161, 163, 206, 353, 498, 542, 547, 550, 568, 585, 588, 600]
	NDCG@k	[3, 122, 208–210, 225, 231, 299, 302, 395, 398, 422, 423, 547, 608]
F-Measure		[78, 124, 168, 185, 186, 254, 486, 549, 564]
ROC Curve		[111, 230, 577]
FP & FN Rate†		[229, 498]

† False-Positive and Negative Rate.

Precision concerns multiple codes relevant to a query, while *Mean Average Precision (MAP)* measures the average. *Recall* quantifies the number of correct predictions made out of all positive examples in the dataset. *F-measure* is the harmonic mean of the precision and recall[14]. Similarly, *NDCG* measures the performance with the consideration of the recommended candidates' order (weight-based). *Accuracy* is the proportion of correct predictions (both true positives and true negatives) among the total number of cases. *SuccessRate* measures the percentage of queries for which more than one correct result could exist in the results. *ROC curve* is created by plotting the true positive rate against the false-positive rate with various thresholds. Furthermore, two approaches leveraged *False positive and negative rates* to check results' search engine errors.

Table 3.10: Ranking Metrics

Metric	Sub-Metric	Approaches
Simple Rank		[79, 211, 339, 368]
Rank of the First Correct (RFC)		[79, 230]
FRank	FRank	[82, 203, 296, 358, 377, 419, 432, 453, 523, 594]
	FRank@k	[71]
Expected Reciprocal Rank (ERR)		[398]
Mean Reciprocal Rank (MRR)	MRR	[71, 85, 88, 114, 166, 167, 177, 183, 194, 196, 198, 203–205, 222, 296, 299, 308, 309, 314, 316, 334, 377, 472, 480, 482, 498, 507, 541, 542, 550, 586, 588, 590, 606, 607, 609]
	MRR@k	[298, 422–425, 432]
Quality of a Candidate Ranking†		[39, 342]

†Ranking based on significance, density, and cohesiveness.

[14]F1-score is the harmonic mean, and F2-score is weighted version

Table 3.11: Various Other Metrics for Code Search Evaluation

Metric	Sub-Metric	Approaches
Retrieval Time†		[37, 78, 84, 85, 90, 111, 143, 177, 213, 213, 224, 224, 230, 241, 254, 278–280, 289, 302, 308, 314, 322, 328, 339, 346, 377, 385, 401, 419, 427, 428, 430, 453, 478, 495–498, 502, 507, 538, 546, 549, 551, 552, 566, 577, 578, 607]
Counting	Absolute matching	[39, 213, 224, 230, 284, 368, 415, 466, 502]
	Top k recommendation	[90, 316, 325, 389, 393, 449, 578]
Statistical	Correlation analysis*	[4, 39, 574]
	Mean Squared Error (MSE)	[574]
	Hypothesis test (p-value)	[97,161,163,185,209,222,284,286,287,346,347, 353,354,356,359,373,398,401,484,538,547,608]
User Satisfaction	Experience score	[72, 143, 189, 334, 346, 347, 373, 409, 538]
	Mouse click	[594]
Others	External library‡	[452]
	Rate of passing test cases	[85, 427, 428, 430]
	BLEU	[88, 586]
	METEOR	[88]
	QC (Query Coverage)	[248, 287]

† Time to search/implement/train.
* ICC, KCC, SCC, and PCC.
‡ i.e., org.eclipse.compare-Plug-in.

Table 3.10 shows the ranking metrics related to code search field. Such metrics primarily consider the location of the correct answers among the results. Several approaches are evaluated by checking the rank of the correct answers manually. *FRrank* (also known as *best hit rank* [295]) is the rank of the first hit result in the result list [419]. *Rank of the first correct (RFC)* and *Expected Reciprocal Rank (ERR)* [81] are measured to check if the user should review a specific number of results. *Mean Reciprocal Rank (MRR)* is the average of the reciprocal ranks of results of a set of queries. The reciprocal rank of a query is the inverse of the rank of the first hit result [164]. Differently, a paper [39] measured the quality of a candidate ranking against users' opinions to prove the ranking performance.

On the other hand, there exist other metrics to evaluate code search approaches. Table 3.11 introduces various other metrics. *Retrieval Time* can be taken for one of the assessment metrics. The approaches should reply on time since code search is also based on information retrieval. In the early stage, researchers often count the absolute number of correct answers from the results by themselves. These metrics are not used anymore more the main evaluation since they can be biased depending on the perspectives. There exist many studies evaluated with several statistical tests. One study leveraged the *Mean Squared Errors (MSE)*, a risk function that corresponds to the expected value of the squared error loss. More relevancy score-based metrics such as *Kendall's Correlation Coefficient (KCC)*, *Spearman Correlation Coefficient (SCC)*, and *Pearson Correlation Coefficient (PCC)* are also leveraged. As famous relevancy measuring metric, the Hypothesis test (e.g., *Mann-Whitney U test*) has been leveraged 17 times by the approaches. Approaches that employed user study as their evaluation method tended to rely on user satisfaction metrics such as experience score and mouse clicks. The experience score is graded by the employed users. For the mouse click, the researchers deploy their code search engines online and trace the users' activity to see if they click the results. A study [594] showed that it is not necessary to do further clicks when the users get the results. Other metrics includes *METEOR* [28] score, *BLEU-4* [404], *Rate of passing the test cases*, *Query Coverage (QC)*, and metrics based on an External library (specifically, `org.eclipse.compare-plug-in`). For example, regarding time metrics, it is an essential factor of a search engine, and 40 studies measured their performing time, such as searching code snippets, training the retrieval models, and implementing

code search engines. The majority of them compared with the state-of-the-art and showed their improved time performance. Two particular studies [88, 222] based on code summarization evaluated the summarization part of their approaches with *METEOR* and *BLEU-4*, together with a human study for naturalness and informativeness of the output. *METEOR* is recall-oriented, and it measures how well a model captures content from the references in the output. At the same time, *BLEU-4* is a correspondence score between a machine's output and a human. Despite evaluating code search engines with test cases is rare and difficult to conduct, there are two studies [85, 430] with the Rate of passing test cases. Researchers design a set of test cases for prepared queries and see if the retrieved results pass such cases. For a special case, a study [452] employed an external library called *org.eclipse.compare* which measures the similarity of the classes in Java.

3.8 Discussion and Open Issues

As this survey reveals, code search has experienced rapid growth, and it has been treated as an essential activity for many software developers. To avoid serving a poor quality code search engine that can drag developers out, researchers have put tremendous efforts into the field and published many studies. Despite these numerous proposed approaches and advancements, open issues remain and impede rapid and efficient software development.

Recommendation of Code Search: Given a wide variety of existing approaches to each code search procedure, developers who search for source code can be confused because they cannot be sure which one is the best for their specific tasks. For instance, developers who are new to their field may need a code search engine that can adequately reformulate their queries because of their lack of ability to formulate a query due to lack of knowledge. Some developers need a feedback-driven code search engine when they have many requirements and considering points. They can leverage the interaction that the engine provides to address the task step by step. Code-to-code search approach can be utilized to refactor the existing code by finding either syntactically or semantically similar code snippets. These task-characteristic pairs are analyzed, and as a result, the best way to address a specific task remains an open issue. This is a new and novel research direction.

Vocabulary Mismatch Problem: An issue that affects information retrieval in general and specifically within code search is the vocabulary mismatch problem (VMP). VMP states that the likelihood of two people choosing the same keyword for a familiar concept is only between 10-15% [141]. As the code search is based on the use of keywords, VMP is relevant in this context. Automatic query reformulation approaches have been proposed to circumvent such problems by providing synonyms, but they still suffer from the characteristic difference between natural language and the source code. The VMP problem within the code search needs further research to improve tasks' efficiency, such as the tokenization and translation of query and the source code.

Benchmarks: Despite researchers and practitioners all know that a more general and trustworthy conclusion for the code search will be drawn with a high-quality benchmark [14], there is still a lack of a benchmark for internet-scale code search engines. During the review process, we observed that comparison of existing approaches is a common means at evaluating code search engines. However, many of them collected their dataset, re-implemented the target approaches, and applied their dataset for comparison. As we reviewed in the Section 3.6, two benchmarks [292, 580] for code search purposes have been proposed recently. These benchmarks leverage queries from the most famous developer Q&A forum, Stackoverflow and code snippets from the biggest code repository, Github. They both retain and provide the dataset along with their evaluation results to the public so that researchers can download the dataset and compare it with the existing results. However, these recent benchmarks also have open issues: 1) these are not validated yet by code search researchers; 2) they focus on learning-based approaches, i.e., classical and learning-based approaches are different; and 3) they still need concrete common metrics to test various approaches. Furthermore, other considerations such as different NL queries for a specific set of tasks should provide a similar set of results. Every user has a

unique way of describing a problem, especially when it comes to using NL. Thus, generating a set of techniques that would consider different NL representations for the same set of questions would provide a more normalized set of results. This also requires a careful extension towards query topics that would be relevant to the search.

Extensibility: Software development consists of using multiple different programming languages for developing one full product. Consequently, code search approaches do not always prove to be a good solution because of their limitation on specific programming languages. The extensibility of code search approaches towards all viable programming languages is an important issue in the domain. Being able to apply a particular search approach towards different programming languages would provide more usefulness and convenience.

Consensus: The result from a code search engine is not always what a user expected. Different approaches rarely consider different contextual requirements of the users. For example, a user looking for a code snippet that is memory efficient might not consider just a normal code snippet as an answer as it fails to satisfy the requirement of being memory-optimized.

Usability: Specific code search approaches such as binary code search are limited in their usability in a real-life scenario. While finding similar binary code is useful for certain domains such as vulnerability detection, code search is rather severely limited. This is mainly because a majority of the approaches rely on an important assumption that the binary codes are not obfuscated. Obfuscated executables would make it difficult to perform any code abstraction that would make matching easier. Furthermore, the rise in compiler level optimization or even hardware-specific optimization would prevent disassembly techniques that would limit to searching capabilities of code search approaches.

Replicability: Even though this survey encompasses many approaches that support code search, most do not have publicly available replication packages. This poses an obstacle for developers who need to apply such an approach. A shared replication package helps developers in two significant ways: First being a time-efficient way to deploy and test the approach, and second, a reference implementation to check for errors and issues. Re-implementation of an approach is a time-consuming task even if the approach is well-explained. It can derive into potential errors where the performance is different from the reported one. Therefore, sharing source code can improve efficiency in the field of code search.

3.9 Conclusion

Through a comprehensive review process of 136 code search approaches, our survey presents an operational taxonomy that aims to assist newcomers within the domain. The taxonomy classifies all the approaches that permit a fair comparison and identifies potential future research areas that can be explored. Newcomers within the field can leverage this survey as an exhaustive introduction, while practitioners can extend their understanding to identify techniques based on the context of their exploration. Moreover, existing researchers can identify the different contributions and advances made within the different categories. This survey shed lights on the existing open issues within the code search, such as the lack of standard benchmark that provide a fair evaluation between various code search engines. Finally, our procedure-driven systematic literature review provides analysis on each phase, and it should deliver the insights for the overall field.

4 Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search

Source code terms such as method names and variable types are often different from conceptual words mentioned in a search query. This vocabulary mismatch problem can make code search inefficient. In this chapter, we present COde voCABUlary (CoCaBu), an approach to resolving the vocabulary mismatch problem when dealing with free-form code search queries. Our approach leverages common developer questions and the associated expert answers to augment user queries with the relevant, but missing, structural code entities in order to improve the performance of matching relevant code examples within large code repositories. To instantiate this approach, we build GitSearch, a code search engine, on top of GitHub and Stack Overflow Q&A data. We evaluate GitSearch in several dimensions to demonstrate that (1) its code search results are correct w.r.t user accepted answers; (2) the results are qualitatively better than those of existing Internet-scale code search engines; (3) our engine is competitive against web search engines, such as Google, in helping users complete solve programming tasks; and (4) GitSearch provides code examples that are acceptable or interesting to the community as answers for Stack Overflow questions.

This chapter is based on the work published in the following research paper:

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

Contents

4.1 Introduction

Code search is an important activity in software development since developers are regularly searching [451] for code examples dealing with diverse programming concepts, APIs, and specific platform peculiarities. Such examples can indeed help them practice programming against a library and platform, or they can immediately be used for inspiration in software development tasks. Because contemporary programmers often implement most of the program elements (e.g., classes and methods) based on existing programs already written by other programmers [355], an effective code search engine is a critical factor for programming productivity.

Open source project hosting platforms, such as `GitHub`, SourceForge, and BitBucket now offer an opportunity for students, researchers and developers to access real-world software projects for improving their work. It is, however, challenging to locate relevant source code due to the enormous size of existing code repositories. For instance, as of August 2015, `GitHub` is hosting more than 25 millions private and public code repositories[1]. To help developers search for source code, several Internet-scale code search engines [146], such as OpenHub [399] and Codota [96] have been proposed. The advantage of these engines is that users can express their queries in a list of keywords (i.e., free-form queries) rather than specific program elements such as API classes and methods.

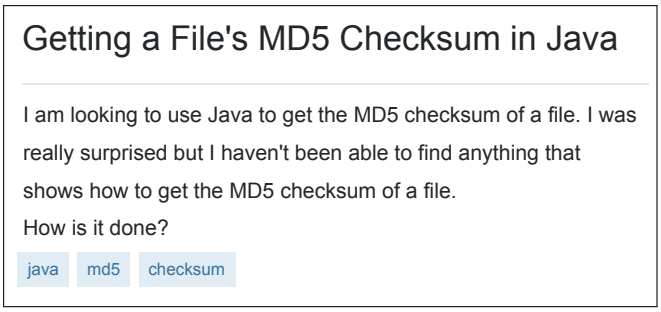
Unfortunately, these Internet-scale code search engines have an accuracy issue since they treat source code as natural language documents. Source code, however, is written in a programming language while query terms are typically expressed in natural language. As a result, searching source code with query keywords in natural language often leads to irrelevant and low-quality search results unless the keywords exactly correspond to program elements. According to Hoffmann *et al.* [193], however, around 64% of programmer web queries for code are merely descriptive but do not contain actual names of APIs, packages, types, etc.

As in any search engine, the terms in a code search query must be mapped with an index built from the code. Unfortunately, the construction of such an index as well as the mapping process are challenging since "no single word can be chosen to describe a programming concept in the best way" [140]. This is known in the literature as the vocabulary mismatch problem: user search queries frequently mismatch a majority of the relevant documents [140, 174, 602, 604]. This problem occurs in various software engineering research work such as retrieving regulatory codes in product requirement specifications [94], identifying bug files based on bug reports [386], and searching code examples [174, 175, 188].

The vocabulary mismatch problem is further exacerbated in code search engines where the source code may be poorly documented or may use non-explicit names for variables and method names [262]. To work around the translation issue between the query terms and the relevant code, one can leverage a developer community. Actually, developers often resort to web-based resources such as blogs, tutorial pages, and Q&A sites. `StackOverflow` is one of such leading discussion platforms, which has gained popularity among software developers. In `StackOverflow`, an answer to a question is typically short texts accompanied by code snippets that demonstrate a solution to a given development task or the usage of a particular functionality in a library or framework. `StackOverflow` provides social mechanisms to assess and improve the quality of posts that leads implicitly to high-quality source code snippets. Figure 4.1 shows an example of the vocabulary mismatch problem.

While code snippets found in Q&A sites certainly accelerate the software development process, they fail to explore the potential of large code repositories. Typically, those code snippets are manually crafted by developers rather than being actual examples from source code repositories. Thus, snippets often omit context information (e.g., variable types and initialization values) that might be necessary to understand interactions with other relevant components. On the other hand, actual examples in source code repositories can provide different views on how a single functionality

[1] `https://github.com/about/press` (verified 14.08.2015)



(a) Example of questions in `StackOverflow`.

There's an input stream decorator, `java.security.DigestInputStream`, so that you can compute the digest while using the input stream as you normally would, instead of having to make an extra pass over the data.

```
MessageDigest md = MessageDigest.getInstance("MD5");
try (InputStream is = Files.newInputStream(Paths.get("file.txt"));
     DigestInputStream dis = new DigestInputStream(is, md))
{
  /* Read decorated stream (dis) to EOF as normal... */
}
byte[] digest = md.digest();
```

(b) Potential solution for the question in (a).

Figure 4.1: Example of the vocabulary mismatch problem. Regarding the question shown in (a), a user of `StackOverflow` posted a potential answer as shown in (b). The keywords of the question include `MD5, checksum, Java, file`. However, the snippet in the answer contains a different set of keywords such as `MessageDigest, InputStream, DigestInputStream, digest, MD5, file`. Using the keywords of the question when searching for code examples has significantly low possibility to locate code fragments similar to the snippet in the answer.

can be implemented by different APIs. Source code repositories also contain concrete code that demonstrates the interaction between various modules and APIs of interest. Besides, usually, in Q&A sites, an acceptable answer only exists when the question, or a very similar one, has been asked before. Otherwise, the questioner must wait for other experienced developers to provide answers.

Our work focuses on building an approach to automatically expand developer code search queries. Specifically, we aim at translating free-form queries to augment them with relevant program elements. To augment a user query, we consider first finding similar (in terms of natural language words) queries for which we have some sketched answers. Then we can collect from these answers some important code keywords. Finally, such code keywords are simply used to enrich the user's initial free-form terms. This query expansion is effective in retrieving relevant code search results even when the user has not provided in his query terms essential information such as API names.

Contributions We propose a novel approach to augmenting user queries in a free-form code search scenario. This approach aims at improving the quality of code examples returned by Internet-scale code search engines by building a Code voCaBulary (CoCaBu). The originality of CoCaBu is that it addresses the vocabulary mismatch problem, by expanding/enriching/re-targeting a user's free-form query, building on similar questions in Q&A sites so that a code search engine can find highly relevant code in source code repositories.

Overall, this paper makes the following contributions:

- CoCaBu **approach to the vocabulary mismatch problem:** We propose a technique for finding relevant code with free-form query terms that describe programming tasks, with no a-priori knowledge on the API keywords to search for. In this regard, we differ from several

state-of-the-art techniques, which perform by searching relevant usage examples of APIs that the user can already list as relevant for his task [83, 249, 338, 374].

- GitSearch **free-form search engine for GitHub:** We instantiate the CoCaBu approach based on indices of Java files built from `GitHub` and Q&A posts from `StackOverflow` to find the most relevant source code examples for developer queries.
- **Empirical user evaluation**: We present the evaluation results implying that GitSearch accurately extends user queries to produce correct (i.e., relevant) results. Comparison with popular code search engines further shows that GitSearch is more effective in returning acceptable code search results. In addition, Comparison against web search engines indicates that GitSearch is a competitive alternative. Finally, via a live study, we show that users on Q&A sites may find GitSearch's real code examples acceptable as answers to developer questions.

The remainder of this paper is organized as follows. Section 4.2 motivates our work further, listing some limitations in the state-of-the-art and introducing the key ideas behind our approach. Section 4.3 then overviews the CoCaBu approach. We provide evaluation results in Section 4.5 and discuss related work in Section 4.6. Finally, Section 4.7 concludes the paper.

4.2 Motivation

The literature contains a large body of approaches that attempt to solve the vocabulary mismatch problem. They either 1) use a *controlled vocabulary* [321] maintained by experts in specific and restricted domains; or 2) automatically derive a *thesaurus* [128], e.g., word co-occurrence statistics in an exhaustive corpus; or 3) *interactively expand* user queries [448], e.g., by recommending other terms from previous query logs; or 4) *automatically expand* queries [74] by adding derived words from the terms included in the original query, e.g., add the `integer` word in a query with `int`; or 5) completely *rewrite* the query automatically [152]. Most of these approaches are not suitable in the settings of a code search engine, since i) the domain is not restricted, ii) the corpus is not finite, iii) query logs are not always available, iv) code terms and query terms may not share any stem words, and v) query terms remain valuable to be matched against identifiers in the code.

Furthermore in practice, implementing a *code* search engine has its own additional tasks: (1) relevant data is hidden in the deep web and unlinked; (2) the variety of concepts in programming languages, APIs, platforms or development environment challenges indexing; (3) the vocabulary mismatch problem complicates query processing; and (4) granularity of search output (e.g., code snippets, files, or applications) is also challenging to determine and satisfy.

Among the above tasks, query processing is one of the key components since search engine must match the query terms with relevant keywords from the index. The indexing step itself can improve speed and performance in finding relevant documents (source code files in our case) corresponding to a given search query. It often uses the salient keywords in a document. In code search, however, such keywords may not include API names since a single programming concept can be translated and implemented by several different classes and methods. This mismatch may degrade the quality of code search results.

4.2.1 Limitations of the state-of-the-art

Online code search engines such as OpenHub [399] and Codota [96] perform basic string matching between user free-form queries and the code (which is then strictly considered as a text document, with no distinction between code and documentation). This, however, produces very low-quality results since programming language terms do not always match natural language words [499].

Figure 4.2 shows an example of OpenHub's search results for the query "Generating random words in Java?"[2]. This top result from the search engine is not relevant: the returned snippet is for a program that *randomly selects a word from an array of words* rather than generating random words. This inaccurate search result occurs because the words used in the query are not appropriate for direct match with source code terms; "random *string* in Java" is the correct terminology that would have matched a more relevant program. Following results from the search engine were found irrelevant as well. The described example shows the limitation of the current practice in face of the vocabulary mismatch problem.

File: RecyclerTest.java Project: OHA-Android-2.2_r1.1

```
80      // Read in dictionary of words
81      mWords = new ArrayList<String>(98568);      // count of words in words file
82      StringBuilder sb = new StringBuilder();
83      try {
84          Log.v(TAG, "Loading dictionary of words");
85          FileInputStream words = context.openFileInput("words");
...
102         Log.e(TAG, "can't open words file at /data/data/com.android.mms/files/words");
103         return;
104     }
105
106     // Read in list of recipients
107     mRecipients = new ArrayList<String>();
108     try {
109         Log.v(TAG, "Loading recipients");
110         FileInputStream recipients = context.openFileInput("recipients");
...
133     int wordsInMessage = mRandom.nextInt(9) + 1;   // up to 10 words in the message
134     StringBuilder msg = new StringBuilder();
135     for (int i = 0; i < wordsInMessage; i++) {
136         msg.append(mWords.get(mRandom.nextInt(mWordCount)) + " ");
137     }
```

Figure 4.2: Top result provided by OpenHub for the free-form code search query "*Generating random words in Java?*"

Our goal is to resolve this vocabulary mismatch problem in order to allow code search engines to return highly relevant code snippets for user free-form queries. Indeed, if we can appropriately transform words used in a search query to keywords found in source code, the search result would be more accurate as shown in Figure 4.3; this is an actual search result of our approach described in Section 5.3. The produced code snippet, extracted from real world code, is practically identical to the manually crafted accepted answer for the question in the Q&A post.

Note that state-of-the-art approaches in the literature, such as Muse [374] and MAPO [572], focus on finding usage examples of API methods whose names must be explicitly indicated in the query. Thus, they may not be suitable for development tasks where users do not know the source code keywords of the relevant APIs. In particular, novice programmers may fail to get relevant code usage examples without knowing exactly necessary class or method names.

Other techniques such as Sourcerer [15] have proposed infrastructures to collect and model open source code data that users can query programmatically (e.g., SQL query statements). The Portfolio [357] search engine returns output relevant functions and their usage scenarios. However, these approaches also simply match query terms with function names in the code base.

In summary, because of the vocabulary mismatch problem, current state-of-the-art approaches to code search fail to support entirely free-form and complex queries such as the ones developers are asking to other experienced developers on Q&A sites (cf. query in the caption of Figure 4.2).

[2]This is a real question asked by a user in this post: `http://stackoverflow.com/questions/4951997/generating-random-words-in-java`

```
24  public class RandomString {
25
26      private static final char[] symbols = new char[36];
27      private static final Random random = new Random();
28
29      static {


37      {
38          char[] buf = new char[length];
39          for (int idx = 0; idx < length; ++idx)
40              buf[idx] = symbols[random.nextInt(symbols.length)];
41          return new String(buf);
42      }
```

Figure 4.3: Top result provided by a CoCaBu-based search engine (see Section 5.3) for the same query used in Figure 4.2. This code snippet was found in class `org.neo4j.vagrant.RandomString` of *simpsonjulian/neophyte* project from `GitHub`.

4.2.2 Key Intuition

Q&A posts contain a wealth of information that can be automatically leveraged by a code search engine. A typical Q&A post is a developer question accompanied with answers provided by experienced developers:

- In Q&A sites, developer questions, which are also often rewritten to make them explicit and limit the opportunities for duplicate questions, are good summaries of typical developer query terms.
- Code snippets embedded in experienced developer answers are a good starting point to systematically list relevant source code information related to developer question.

Thus, by leveraging developer questions from Q&A sites, and the associated code snippets, we can document concept mappings, i.e., the mappings between human concepts, which are expressed in questions, and program elements, which can be identified in code snippets. Once a large corpus of such mappings become available, the vocabulary mismatch problem can be alleviated. Indeed, any developer query, written in natural language, can be translated into a program query that explicitly makes references to specific program elements such as method and class names. This new query can then be directly matched against any source code file.

4.3 Our Approach

CoCaBu is about retrieving most relevant source code snippets to answer a free-form query given by a user. To resolve the vocabulary mismatch problem illustrated in Section 4.2.1, our approach leverages the intuition described in Section 4.2.2. Figure 4.4 provides an overview of our approach.

The search process begins with a free-form query from a user, i.e., a sentence written in a natural language:

(a) – For a given query, CoCaBu first searches for relevant posts in Q&A forums. The role of the Search Proxy is then to forward developer free-form queries to web search engines that can collect and rank entries in Q&A with the most relevant documents for the query.

(b) – CoCaBu then generates an augmented query based on the information in the relevant posts. To that end, it mainly leverages code snippets in the previously identified posts. Since these snippets are

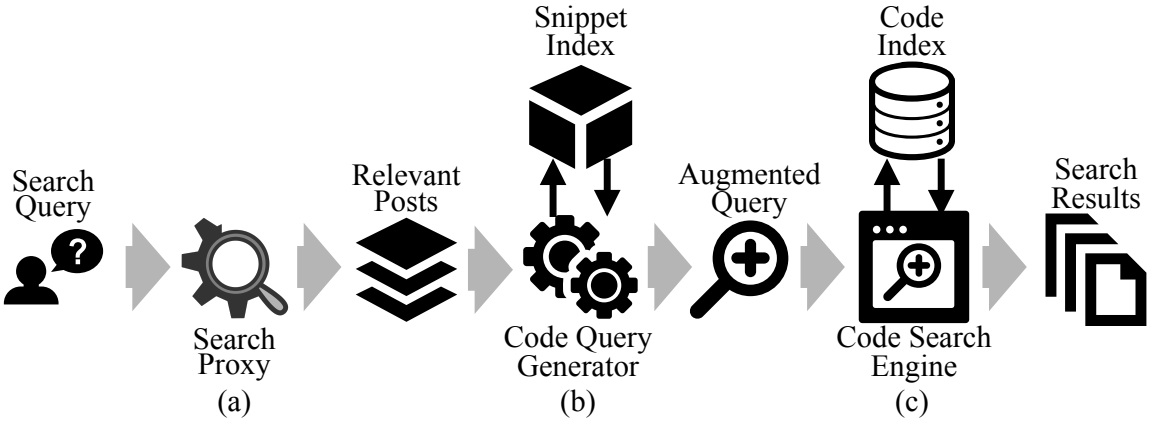


Figure 4.4: Overview of CoCaBu.

approved by developers as acceptable code examples from the posted question, CoCaBu can consider them translations of human concepts into program elements. CoCaBu's Code Query Generator then creates another query which includes not only the initial user query terms but also program elements, such as method and class names, from the extracted snippets. To accelerate this step in the search process, CoCaBu builds upfront a snippet index for Q&A posts.

(c) – Once the augmented query is constructed, CoCaBu searches source code files for code locations that match the query terms. For this step, we can crawl a large number of public code repositories and build upfront a code index for program elements in the source code. It then leverages the code index to produce search results for a given augmented query. This search result can be presented to a user at different granularity level (e.g., relevant source code file, or code snippet).

To efficiently search source code in repositories for relevant code locations that match information from Q&A posts, CoCaBu makes indices of structural code entities in code snippets and source code files. Previous studies on code search and recommendation systems [21, 22, 89, 324] have already proposed to take advantage of structural code information (e.g., method identifiers and class types) to improve query results. Indeed, if provided by user query, this information enables to map source code based on specific program elements. We use similar structural entities to those leveraged in many of previous work [21, 22, 89, 324]. Since structural code entities extraction process is specific to a programming language, we report such details in Section 4.4.2 when overviewing the GitSearch implementation case study.

The remainder of this section details the design of CoCaBu components (Sections 4.3.1 – 4.3.3).

4.3.1 Search Proxy

The search proxy takes a free-form query as an input and returns a set of relevant posts collected from developer Q&A sites as an output. The goal of this component is **to collect sufficient data so that the search engine can later find out how natural language concepts can be translated into program elements.** Indeed, code snippets in answers of Q&A posts can provide potential translation rules from concepts written in natural languages to program elements such as API methods or classes. As discussed in Section 5.2, such translation rules facilitate the subsequent code search process by alleviating the vocabulary mismatch problem that exists between user queries and source code elements.

Relying on general purpose engines such as Google Web Search, Bing, and Yahoo Search, CoCaBu can search several different forums and rank the search results according to their relevancy to the query. Thus, in practice, once a user submits a code search query, the search proxy forwards it to a

general-purpose web search engine to obtain related questions on the web. Since these search engines are specialized for text search, we assume that they are better than other built-in search engines in Q&A forums. Web search results are then filtered by the search proxy to eliminate URLs not related to Q&A posts. For example, if we want to consider only `StackOverflow` posts, the search proxy would try to match the following pattern to collect relevant posts:

```
http://stackoverflow.com/questions/<ID>/<TITLE>
```

The ranking of relevant posts is directly preserved from the sorting order proposed by the general-purpose search engine. If we consider for example the question "Generating random words in Java?" described in Section 5.2, the search proxy supported by Google Web Search returns the relevant posts[3] as listed in Table 4.1.

Table 4.1: List of Q&A posts relevant to 'Generating random words in Java?'

Q&A site	Post title	Post ID
`StackOverflow`	Generating random words of a certain length in java?	27429181
`StackOverflow`	Random word from array list	20358980
`dummies.com`	How to Generate Words Randomly in Java	-
`java2notice.com`	How to create random string with random characters?	-
`coderanch.com`	Random string generation	374794

4.3.2 Code Query Generator

The Code Query Generator creates a code search query that augments and structures the free-form query taken by the search proxy (Section 4.3.1). This augmented query is a list of program elements, such as class and method names (e.g., `Math.random`), as well as natural language terms which can be used to match the documentation.

To generate the augmented query, CoCaBu must extract structural code entities from code snippets embedded in the answers to the questions in the relevant posts returned by the search proxy (Figure 4.5(b)). The code query generator component only considers accepted answers, i.e., answers approved by the Q&A site community.

The augmented query produced by the code query generator is illustrated in Figure 4.5(c) based on the Lucene search engine query format. The reader can observe the following from the illustrated example query whose field semantics are previously described in Table 4.3:

- terms, excluding stop words, in the user free-form query (i.e., Figure 4.5(a)) are kept, after stemming, in the augmented query (e.g., `code:gener`).
- structural code entities collected from Q&A snippets (i.e., Figure 4.5(b)) are mentioned with their type (e.g., non-qualified/partially qualified method invocation, or class) in the augmented query (e.g., `pq_method_invocation:Random.nextInt`).

To accelerate code query generation, CoCaBu builds an index of posts. Typically, Q&A forums provide archives of their posts. These posts are often formatted by a structural language such as XML. For example, in `StackOverflow` posts, code snippets are enclosed in `<code> ...</code>`. As shown in Figure 4.6, our approach takes pre-downloaded posts from a Q&A site and extracts metadata (post ID, question title) and code snippets for each post. Each code snippet is then analyzed to retrieve the structural code entities. This phase presents challenges that will be addressed in Section 4.4.2.

[3] In this illustrative example, we excluded the actual post (`http://stackoverflow.com/questions/4951997/generating-random-words-in-java`) where this question is asked. To eliminate bias, in all experiments described in Section 3.7, in which we selected a question of a Q&A site as a subject, we removed the corresponding posts from the list of relevant posts to be used for augmenting the query.

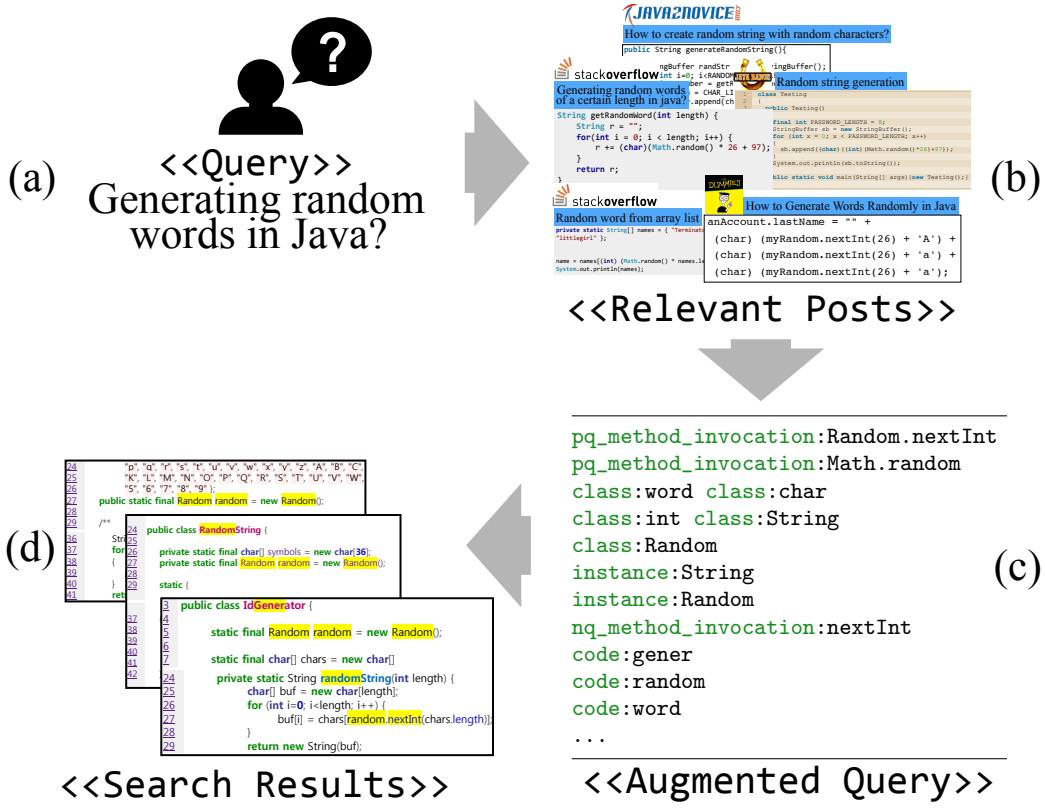


Figure 4.5: Illustrative input, intermediate results, and output of a CoCaBu-based code search engine.

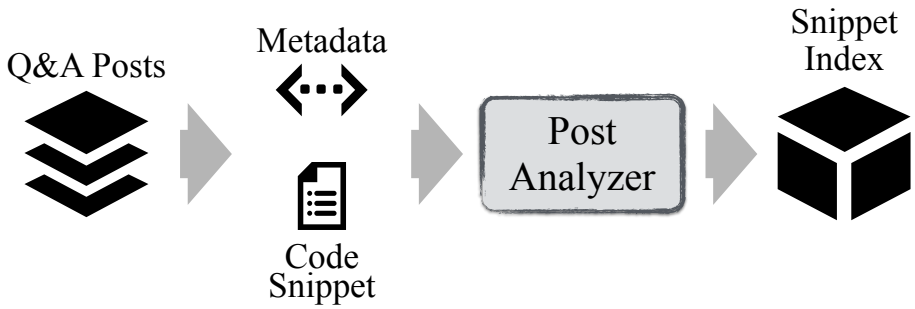


Figure 4.6: Creating an index for metadata and code snippets of Q&A posts.

Building an index upfront reduces the query generation time when the target post is already indexed. For new posts collected, the component follows the process shown in Figure 4.6 to insert it into the index.

4.3.3 Code Search Engine

The code search engine takes an augmented query from the code query generator and provides a list of search results to the user who issued the original query. The search results are of two granularity levels:

- In case the query is augmented, granularity is further controlled since the structural code entities matched within a source file and the search result can focus on showing only the excerpt

with code lines where a match occurred as in the illustrative example of Figure 4.3.
- When the query has not been augmented (i.e., the search proxy did not find any Q&A post link within the top ten web search results set), the search engine returns for each result a whole file.

To efficiently provide answers for augmented queries, the code search engine builds an index of source code files found in repositories (cf. Figure 4.7). The matching then becomes straightforward as the structural entities in the augmented queries, as well as the NLP terms, are directly search for using the index which will list the most relevant files.

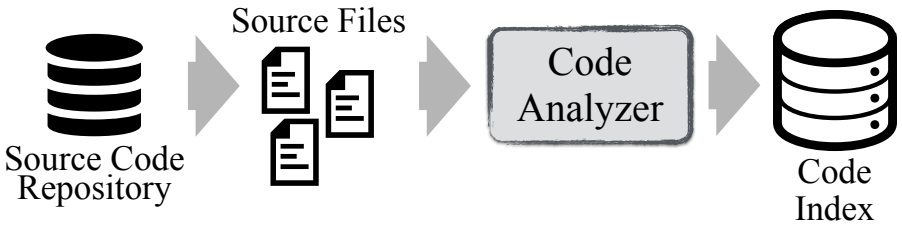


Figure 4.7: Creating an index for source code in code repositories up front.

Since the snippet index and the code index (shown in Figure 4.6 and 4.7, respectively) store indices in the same format, full-text search can be effective to obtain search results. Source code files are then the documents while structural code entities represent the search terms.

Once search results are retrieved, the code search engine computes rankings of the source code files based on a scoring function that measures the similarity between the matched files and query terms. The current implementation of CoCaBu uses the scoring function implemented in the Lucene library. This function combines the Boolean Model (BM) and the Vector Space Model (VSM) to determine the relevancy of a document given for a user query[4]. BM is used for reducing the amount of documents that need to be scored by using Boolean logic in the query specification. Each document is represented as a vector $d = (w_1, w_2, ..., w_n)$ where w_i corresponds to the weight of a term occurring in that document. To compute these weights, we use the TF-IDF weighting scheme implemented in Lucene. With these weights, VSM computes the similarity between the documents by using the cosine similarity measure[5].

Since displaying the entire content of a source code file is often ineffective for users to understand code examples, the code search engine shows the files after summarizing the content and highlighting lines of code relevant to a given query [340]. To summarize and highlight search results, CoCaBu uses a query-dependent approach that displays segments of code based on the query terms occurring in the source file. Specifically, the component displays a set of N adjacent lines (the default value is $N = 3$ lines) of code containing the matching query keyword. Finally, we highlight query words occurring in the summarized file to ease their identification.

4.4 The GitSearch Code Search Engine

This section describes an example instantiation of the CoCaBu approach. We build GitSearch, a code search engine on top of `GitHub` and `StackOverflow` to explore the large amounts of source code and Q&A posts. In the remainder of this section, we detail the implementation choices that were made in GitSearch.

[4]https://goo.gl/MqETzP (last accessed 12.07.2015)
[5]`https://goo.gl/VPvxnX` (last accessed 12.07.2015)

4.4.1 Data collection

To build GitSearch, we selected `StackOverflow` as the Q&A site where to retrieve relevant developer-approved code snippets. `StackOverflow` was selected as it is popular among the developer community and it enforces several rules and strategies (e.g., no duplication of question, response voting, marking of accepted answer, rewriting of developer questions for precision and concision, etc.) which make it a fairly representative and reliable dataset of developer questions and answers. For the search proxy, our implementation directly leverages Google web search[6]. User queries are sent to Google Search for retrieving all relevant Q&A posts (i.e., text similarity matching). Note that it is possible for other implementations to use other web search engines including built-in search services of Q&A sites.

We used a dump of `StackOverflow` posts between July 2008[7] and March 2015 containing 1,363,002 Java and Android tagged questions to build the snippet index. Java was selected in this instantiation since it is one of the most popular programming languages and represents a large developer base [52]. In this work, we made use of the `posts.xml` documents that have an actual post (i.e., question and answer pair) and other associated metadata such as tags, creation date, question ID, view count of the post, and the score of answers. We then collected posts with snippets in their answers as specified in Section 4.3.2. In addition, we extracted snippets from answers that were accepted and had a positive score to ensure high quality of code examples. To account for updates in posts, we leveraged the StackExchange REST API[8] with which we could extract metadata and snippets. Users of CoCaBu may collect and use posts from other multiple Q&A forums to extend the opportunity to search for more code snippets.

For the code index, we leverage `GitHub`, the largest repository of open source projects [236]. `GitHub` project data is further widely used by software engineering researchers and practitioners [48, 49, 526]. We considered `GitHub` projects that were forked at least once, to avoid toy and/or inactive projects. Since we focused on Java and Android, we collected `GitHub` projects in which its major language is "Java" and then removed all non-Java files from the projects when building the code index. As a result, Table 4.2 shows the statistics of `GitHub` projects we collected in this work.

Table 4.2: Statistics of collected projects from `GitHub`.

Feature	Value
Number of projects	7,601
Number of files	1,705,677
Number of duplicate files	182,043
Numver of non-Java files	212,680
LOCs	> 297 M

4.4.2 Processing Code Artifacts

Table 4.3 enumerates structural code entities that GitSearch collects when parsing snippets from Q&A posts and source code files from code repositories. These fields are used for indexing documents (source code files and code snippets in Q&A posts) by using Lucene. GitSearch leverages these field to make an augmented query as well.

The `import` field contains tokens indexed as import declarations, which can be found at the beginning of Java files (e.g., `java.io.InputStream`). The `super` field represents tokens used as superclass and interface implementation such as type names after `extends` or `implements`. All type names used in a Java file are indexed by the `class` field. For example, `Writer` and `BufferedWriter` are

[6] `www.google.com`
[7] We use the dump that contains the oldest data available since the launch of `StackOverflow` in 2008.
[8] `https://api.stackexchange.com/`

indexed as `used_class` field from this statement `Writer writer = new BufferedWriter(...);`. Names used in methods declarations (e.g., `getValue` in `void` **`getValue`**`(...)`) are indexed in the `method_declaration` field. `nq_method_invocation` and `pq_method_invocation` fields contain indexed tokens from non-qualified and qualified method calls, respectively. For example, in this statement: `obj.nextInt();`, we use `nq_method_invocation` if we cannot resolve type information of `obj`. Otherwise, we use `pq_method_invocation` to make an index of `Random.nextInt` if `obj` is resolved as `Random`. The `instance_creation` field is used for making an index of tokens in new object creation such as `Type1` in `Object o = new Type1();`. The `literal` field contains tokens from string constants in a Java file. In addition to other fields, we also use the `code` field to make indices of all tokens in a source code file as plain text after preprocessing. The remainder of this section details our index creation process.

Table 4.3: Structural Code Entities.

Field	Description
import	Name of import declarations
super	Direct superclass and implemented interfaces
used_class	Name of used classes
method_declaration	Name of method declarations
nq_method_invocation	Non-qualified method invocations
pq_method_invocation	Partially qualified method invocations
instance_creation	Class instance creations
literal	String Literals
code	tokens in source code after preprocessing

Wrapping code snippets: While source code from public repositories is mostly compilable, code snippets from Q&A posts are inherently incomplete since they only include the necessary statements to convey expert responder explanations on a question. Although few code snippets may contain a complete class declaration, in most cases a code snippet consists of a block of code statements. Snippet authors furthermore frequently use ellipses (i.e., "...") before and after code blocks. Thus, GitSearch removes ellipses and wraps code snippets by using a custom dummy class and method templates to make it able to parse by standard Java parsers.

Qualifying non-qualified names: In addition to wrapping snippets, our approach reasons about qualified names in code snippets. Enclosing class names of methods in snippets are often ambiguous [106] (i.e., method name qualification). For example, Subramanian et al. [506] found that there are unqualified method name `getId()` more than 27,000 times in their oracle containing 1.6 million types (i.e., classes and method/field signatures) whereas partially qualified name `Node.getId()` can be identified only a few times. This implies that name qualification (even if it is partial) can reduce the size of search space. With a smaller search space, code search tools can more accurately locate and rank the search results. Thus, recovering unqualified names can improve the accuracy of code search.

To recover qualified names of methods, GitSearch transforms unqualified names to partially qualified names using structural information collected during AST traversal. Specifically, it converts variable names on which methods are called through their respective classes. Figure 4.8 illustrates this processing step with an example of code snippet before and after the method qualification.

Text processing: In addition to structural entities, our approach collects textual information as well. By treating source code as text, the approach conducts pre-processing such as tokenization (e.g., splitting camel case), stop word removal[9] [340], and stemming.

[9]Lucene's (version 4) English default stop word set.

```
URL url = new URL(urlToRssFeed);
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
XMLReader xmlreader = parser.getXMLReader();
RssHandler theRSSHandler = new RssHandler();
xmlreader.setContentHandler(theRSSHandler);
InputSource is = new InputSource(url.openStream());
xmlreader.parse(is);
return theRSSHandler.getFeed();
```

(a) Snippet before recovering name qualification.

```
URL url = new URL(urlToRssFeed);
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = SAXParserFactory.newSAXParser();
XMLReader xmlreader = SAXParser.getXMLReader();
RssHandler theRSSHandler = new RssHandler();
XMLReader.setContentHandler(theRSSHandler);
InputSource is = new InputSource(URL.openStream());
XMLReader.parse(is);
return RssHandler.getFeed();
```

(b) Snippet after recovering name qualification.

Figure 4.8: Recovery of qualification information.

Indexing: With the collected set of information, CoCaBu can build an index of text terms as well as structural code entities found in the source code. To create an index, we build our approach on top of Lucene[10]. Lucene stores data as an index, each consisting of a set of fields, where each field value represents a basic code element for search in our case. Fields are populated with the structural and textual information, produced by the above process, along with the index specific metadata. Further details on this process are provided in Section 4.3.3.

Table 4.4 provides a summary of the resulting indices (i.e., the snippet and code indices shown in Figures 4.6 and 4.7) built from `StackOverflow` posts and `GitHub` open source code repositories.

Table 4.4: Descriptive statistics of the snippet index and code index built from `StackOverflow` posts and `GitHub` projects, respectively.

Feature	Code Index from `GitHub`	Snippet Index from `StackOverflow`
# of Documents	1,310,954	298,595
import	8,463,814	86,629
super	1,011,254	35,057
method_declaration	9,056,046	268,929
used_class	23,973,254	1,288,683
nq_method_invocation	771,583	347,917
pq_method_invocation	13,869,622	593,099
instance_creation	2,895,284	220,574
literal	4,079,608	203,762
code	1,101,516	607,070

4.5 Evaluation

This section describes our evaluation design and reports its results. Our evaluation consists of four studies: a manual verification, online survey, controlled user study, and live study, focusing on answering the following research questions, respectively:

- **RQ1**: Can GitSearch effectively produce relevant code examples for developer queries?
- **RQ2**: Does GitSearch outperform existing code search engines with more acceptable results?
- **RQ3**: Is GitSearch competitive against general search engine for helping to solve programming tasks?
- **RQ4**: Can `StackOverflow` users accept the search results of GitSearch as answers?

[10] `http://lucene.apache.org`

4.5.1 RQ1: Verification against a community ground truth

First, we investigate the relevance of the results yielded by GitSearch. To evaluate the relevance, we consider comparing the output code examples against the ground truth of code snippets in answers accepted by the `StackOverflow` community. This type of verification, which is commonly used in the literature [15, 333], is essential since developers can be quickly deterred by search engine producing many irrelevant results.

Study Design: We collect well-known developer questions from `StackOverflow` posts based on two requirements: (i) a question in a post must relate to "Java" and (ii) its answer must include code snippets. We select the top 10 posts (Q1 – Q10) with the highest 'view count' values (for their questions) to ensure that the study focuses on representative and popular developer tasks. In addition, we randomly collect another 10 more questions (Q11 – Q20) out of top 5,000 `StackOverflow` posts to avoid the bias of popularity. Table 4.5 lists the queries used in this study. Note that this process does not bias in favor of our approach. Indeed, for the fair comparison, the actual post where the question is asked is filtered out from the relevant posts, returned by the search proxy that GitSearch uses to augment user queries.

Table 4.5: Free-form queries used for RQ1 and RQ2.

ID	Query Terms
Q1	How to add an image to a JPanel?
Q2	How to generate a random alpha-numeric string?
Q3	How to save the activity state in Android?
Q4	How do I invoke a Java method when given the method name as a string?
Q5	Remove HTML tags from a String
Q6	How to get the path of a running JAR file?
Q7	Getting a File's MD5 Checksum in Java
Q8	Loading a properties file from Java package
Q9	How can I play sound in Java?
Q10	What is the best way to SFTP a file from a server?
Q11	Using java.net.URLConnection to fire and handle HTTP requests
Q12	How do I create a file and write to it in Java?
Q13	How do servlets work? Instantiation, sessions, shared variables and multithreading
Q14	Get current stack trace in Java
Q15	Difference between HashMap, LinkedHashMap and TreeMap
Q16	How to convert a char to a String?
Q17	How do I convert a String to an InputStream in Java?
Q18	When do you use Java's @Override annotation and why?
Q19	How to append text to an existing file in Java?
Q20	Java URL encoding of query string parameters

We first execute GitSearch by giving the title of each question listed in Table 4.5 as a user query to obtain search results. Since the title often has concise and precise representations of user queries, we focus on the titles instead of descriptions of the posts. Then, we evaluate the top 5 code search examples by GitSearch. To assess the relevancy of a GitSearch code example, two authors of this paper compared it against the accepted answer on `StackOverflow` for the associated query. We consider that the example is indeed relevant when it includes the necessary API methods and classes required in the `StackOverflow` answer's code snippet. To increase confidence, both authors must unanimously agree on the relevance of a GitSearch result.

Results: Figures 4.9 and 4.10 shows that GitSearch results are largely relevant to the user query, indirectly demonstrating the accuracy of the query expansion approach.

We also evaluate the effectiveness of GitSearch using the *Precision@k* metric:

$$Precision@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{|relevant_{i,k}|}{k} \tag{4.1}$$

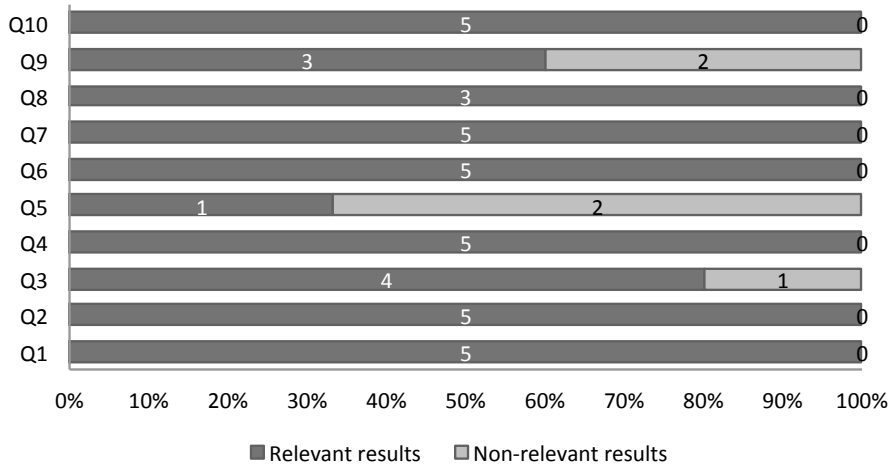


Figure 4.9: Relevance of top 5 GitSearch results for popular queries (Q1 – Q10) listed in Table 4.5.

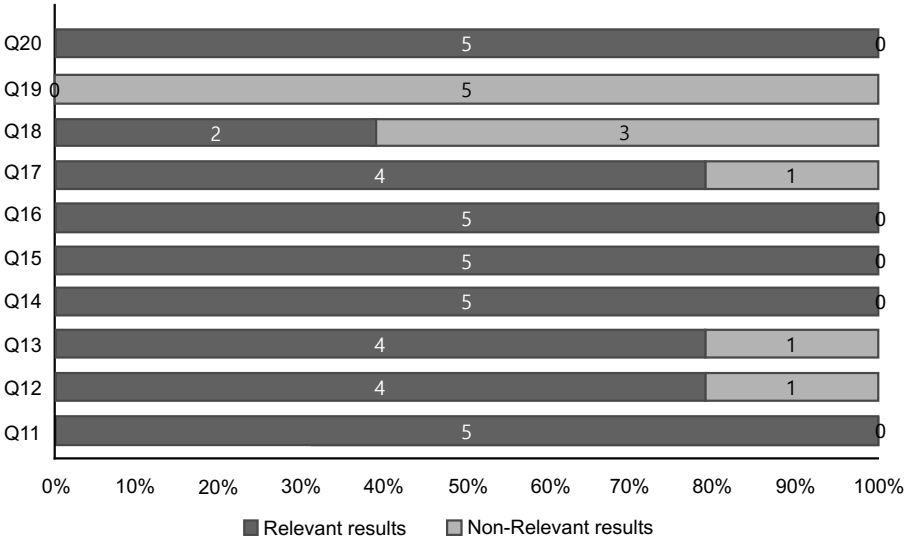


Figure 4.10: Relevance of top 5 GitSearch results for random queries (Q11 – Q20) listed in Table 4.5.

where $relevant_{i,k}$ represents the relevant code search results for query i in the top k returned results, and Q is a set of queries. $Precision@k$ takes an average of all queries whose relevant answers could be found by inspecting the top k (k = 1, 2, 5) of the returned code examples. An effective code search engine should allow developers to find the relevant code examples by examining fewer returned results. Thus, the higher $Precision@k$, the better code search performance. We found that GitSearch achieves 90%, 90% and 88% scores for $Precision@1$, $Precision@2$, and $Precision@5$, respectively, when applying to the top questions (Q1 – Q10). In addition, for randomly selected questions (Q11 – Q20), $Precision@1$, $Precision@2$, and $Precision@5$ are 80%, 80%, and 78%, respectively. We could not define $recall@k$ because it is impossible to compile the "complete" set of all possible correct answers for a code search query.

To verify the effectiveness of query expansion in GitSearch, we conducted another experiment in which our tool has been applied without query expansion. For the same 20 questions in Table 4.5, the results of code search without query expansion are shown in Figures 4.11 and 4.12. GitSearch

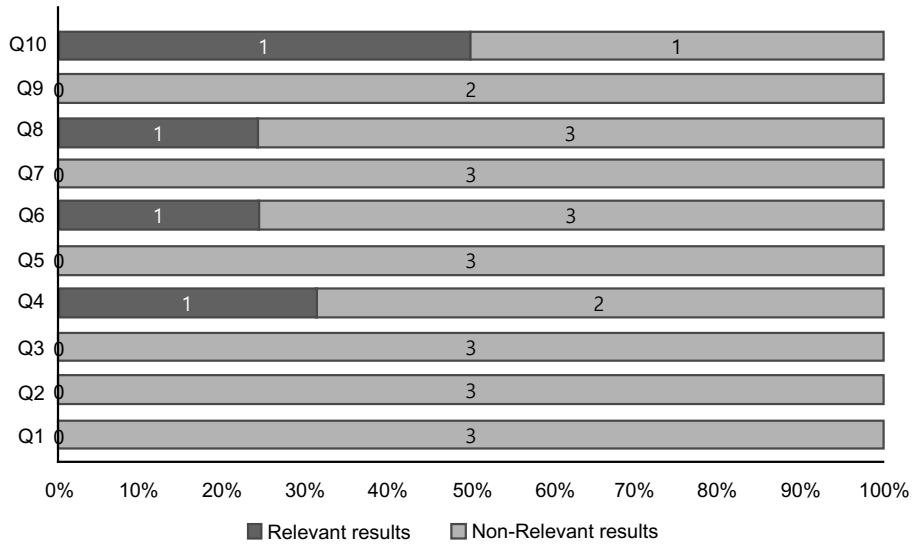


Figure 4.11: Relevance of top 5 GitSearch results, without query expansion for popular queries (Q1 – Q10) listed in Table 4.5.

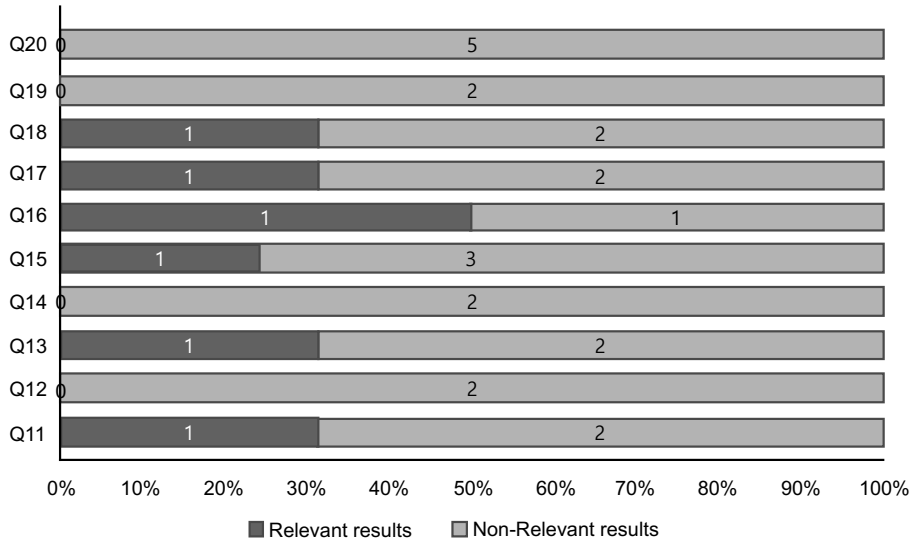


Figure 4.12: Relevance of top 5 GitSearch results, without query expansion for random queries (Q11 – Q20) listed in Table 4.5.

without query expansion results in *Precision*@1, *Precision*@2, and *Precision*@5 as 20%, 22.5%, and 18.6% for all questions (Q1 – Q20). The precision scores are 20%, 15%, and 13.3% for the top 10 questions (Q1 – Q10), while the scores are 20%, 30%, and 24.1% for the random 10 questions (Q11 – Q20). This implies that query expansion in GitSearch significantly improves the search quality.

4.5.2 RQ2: Comparison against other code search engines

First, we applied the same queries for assessing GitSearch (cf. Table 4.5) to other code search engines: OpenHub [399] and Codota [96]. These code search engines were selected since they are state-of-the-art Internet-scale code search engines and currently available online. On the other hand, we could not compare GitSearch against other recent state-of-the-art approaches from the literature

because of the reasons listed in Table 4.6.

Table 4.6: Unavailability of code search tools and techniques.

Portfolio [357] is not available (anymore) and supports only C++.
Exemplar [355] is no longer available.
Sourcerer [15]'s team did not reply about the use of their SAS code search engine.
Muse [374] is not relevant: - focuses on API - cannot be queried for snippets.
SNIFF [83] engine could not work (issue with the Eclipse plugin).
Keivanloo et al's [249]' tool is no longer available (lead developer left the project).
CodeHow [333] is not available - only a demo video online.

The *Precision*@k values of those two search engines are lower than that of GitSearch. OpenHub resulted in 60%, 60%, and 38% scores for *Precision*@1, *Precision*@2, and *Precision*@5, respectively. For Codota, these values are 10%, 10%, and 12%, respectively.

Second, we conduct a user study where we ask developers to check the effectiveness of different code search engines, to assess the usefulness of GitSearch from the perspective of practitioners.

Study Design: For this study, we recruited participants by posting online survey invitations in software developer communities (750 GitHub, Mozilla, and Eclipse developers, and developers in a Korean company). In all survey invitations, we clearly stated that only developers/students who have Java experience are invited. To facilitate the study, we built a web-based survey tool displaying the code search results from OpenHub, Codota, and GitSearch in three anonymized columns. To avoid the bias of people toying with the tool, we only consider the entries of participants who entirely completed the study using the queries in Table 4.5.

Participants can select code examples based on their preference (i.e., they select their favorite "answers" as it is done with the voting mechanism of `StackOverflow`). They can select several favorite search results (up to three). The idea, however, being to select only the relevant results, we clearly ask participants to select no result if none is satisfying them. In addition to anonymization, the survey tool excludes the source `StackOverflow` posts listed in Table 4.5 from the training data of GitSearch to avoid any bias.

Results: At the end of the study, we had 47 participants who tried the tool (at least one response). Some of them did not complete the study. Among them, 14 participants completed this study. We randomly sampled 10 answers selected by the participants and manually verified that they were indeed appropriate for resolving the query indicated.

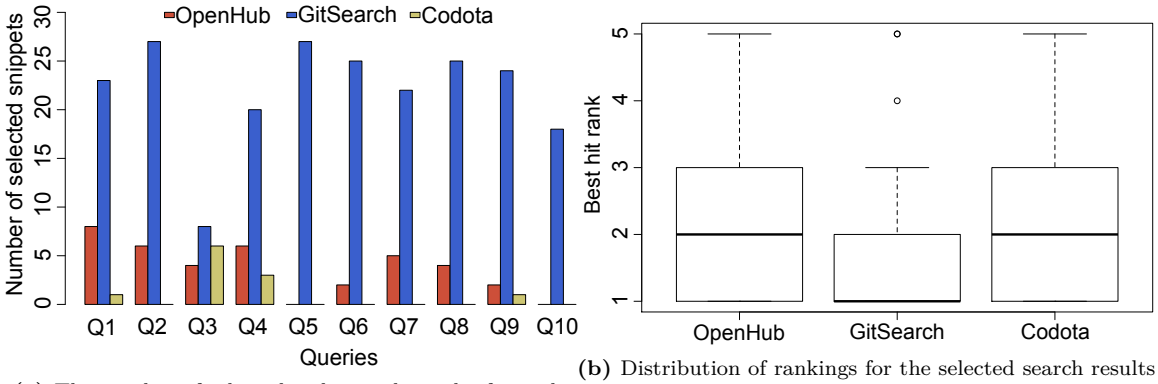
Figure 4.13(a) shows the number of selected search results for each code search engine. Participants selected more code examples returned by GitSearch than other engines for all queries. In particular, the number of selected results was more than double compared to others except for Query Q3.

In addition, we computed the distribution of rankings for the selected search results. If multiple search results of an engine were selected by a user, we counted the highest ranked result only. As shown in Figure 4.13(b), the median value of GitSearch is equal to 1 while the values of other engines are 2.

Discussion: Although we could not compare against the most recent CodeHow tool [333], note that its authors reported that it produces about 20% more relevant results than OpenHub[11] while Figure 4.13(a) indicates that GitSearch provides 50% more relevant results[12] than OpenHub.

[11] Ohloh is now OpenHub.

[12] Despite different queries, our query sets are similar to those of [333] and representatives of common developer search queries.



(a) The number of selected code search results for each search engine.

(b) Distribution of rankings for the selected search results (the lower the better).

Figure 4.13: Comparison between GitSearch, Codota and OpenHub.

4.5.3 RQ3: Comparison against general search engines

We conducted a comparative study between GitSearch and general web search engines. Since many developers rely on general search engines to find solutions to programming tasks, we evaluate the competitiveness of GitSearch in comparison to such engines.

Study Design: For this study, we recruited 20 graduate students from three universities (Pierre and Marie Curie University in France, University of Luxembourg, and Zhejiang University in China). No author of this paper took part in the study. Each student was asked to find code examples for solving the following two programming tasks from a previous code search study [333]:

- *Task 1:* Sending emails - write a Java program to read a list of email addresses from a text file, and then send an email with an attachment file to all the email addresses.
- *Task 2:* Image format conversion - write a Java program to read an image in JPEG format, rotate it 180, and then convert it to PNG format.

Participants to the controlled study have been asked to solve one task with GitSearch and the other task a general search engine (Google for participants in Europe and Baidu for participants in China). We consider in this experiment that Google and Baidu are equivalent, and thus we report Google and Baidu results together; indeed, the purpose of the evaluation scenario was not to implicitly compare Google and Baidu. We specify the combinations (task, tool) for every participant by ourselves in order to ensure an even distribution. The tasks were assigned so that every combination (task, tool) is assigned to at least one participant from each university, and all combinations are evenly distributed across the participant pool. Each participant fills a form indicating the different free-form queries used for code search as well as the rank of the returned results that he/she found relevant for the task. We specified that only top 10 results returned by the tools could be examined.

We assess the efficiency of the engines through the Mean Reciprocal Rank (MRR), a statistical metric used to evaluate a process that produces a list of possible responses to a query [162]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q. MRR is computed by using the formula:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{4.2}$$

where $rank_i$ represents the rank of the first search results that users find satisfying for query i. MRR values range between 0 and 1, and the higher MRR value the better the performance.

Results: Participants to the study entered 77 (37 for Task 1 and 40 for Task 2) distinct free-form queries.

Table 4.7 shows the percentage of relevant search results that participants in the study marked for the different search engines. GitSearch provides more satisfying results for Task 1 while users found more satisfying results with Google/Baidu for Task 2. In contrast, for Task 2, GitSearch outperforms Google/Baidu in terms of MRR, returning in higher ranks the satisfying results. On the other hand, GitSearch has a lower MRR for Task 1 results. These results suggest that GitSearch is competitive against web search engines. We do not, however, take into account the effort required in web search to follow link redirections and parse web pages to find potentially incomplete code snippets. GitSearch, on the other hand, provides immediately real-world working code examples.

Table 4.7: Performance of GitSearch vs. general search engine.

	Percentage of successful queries[†]		**MRR**	
	GitSearch	Google/Baidu	GitSearch	Google/Baidu
Task 1	**93.76**%	90.00%	0.83	**0.96**
Task 2	75.00%	**100.00%**	**0.89**	0.84

[†] We compute the ratio of queries having produced satisfying results vs. the total number of queries entered per task.

Discussion: We investigated the 77 queries entered by participants in this study. We figured out an interesting pattern: queries entered on web search engines appeared to be more "complete" and more redundant across participants than queries entered on GitSearch. Participants to the study admitted that they followed auto-completion suggestions by web search engines.

We perform a cross-validation experiment by randomly sampling 10 queries entered by participants on web search engines and use them on GitSearch. Similarly, we randomly sample 10 queries entered by participants on GitSearch and use them on Google search engine. We record improved MRR values of 0.94 and 0.90 with Task 1 and Task 2 respectively for GitSearch. In contrast, MRR values for the web search engine has decreased to 0.72 and 0.65 with Task 1 and Task 2 respectively.

These results suggest a future work on GitSearch where we must include logging and feedback mechanisms to record successful queries and propose them to auto-complete queries of future requesters.

4.5.4 RQ4: Live study into the wild

To assess the usefulness of code search engines in Q&A forums, we posted code search results as answers to `StackOverflow` questions. This study investigates how developers interpret working code examples when they have programming issues. Although GitSearch is not designed to directly answer developers' questions, it might help them find a starting point of a programming task. In particular, GitSearch can be a good first responder in the context of `StackOverflow` since there are many unanswered questions (not only "no answer selected by questioners" but also literally "no answer") in `StackOverflow`.

Study Design: We monitored questions with **Java** tags and selected 25 out of them based on the following criteria:

- Questions about Java programming.
- "How-To" questions such as "List all files in resources directory in java project".
- No tool usage questions such as "How to create a project in Eclipse?"
- No conceptual questions such as "What is the difference between **A** or **B**" and "why this class is so slow?".
- Questions not answered by anyone yet.

For each question, we extracted its title and put it into GitSearch to obtain code search results. We took the topmost result among the search results and posted it as an answer. The answer consists of 1) the most relevant code fragments selected by GitSearch and 2) hyperlink for the original source code where GitSearch found the fragments from. The latter is important since developers can figure out more context about the working code examples. In addition, we repeated the same procedure with OpenHub to compare its effectiveness with our technique. We do not post Codota's results on `StackOverflow` to avoid "spamming" requesters, as we could see ourselves that its topmost result was irrelevant for most questions.

Results: GitSearch could answer more questions than OpenHub as shown in Table 4.8 ("Resp" of "#Ans"). GitSearch responded 25 questions by using its search results while OpenHub did only for 18 out of 25 questions. For the other seven questions, OpenHub could not produce any search result for reasons that are unknown to us (perhaps, due to an issue of the engine's query matching implementation). In addition, at the end of the study, `StackOverflow` users eventually answered only 8 out of 25 questions. Note that three answers were accepted by the questioners among the 25 answers by GitSearch. None of 18 answers by OpenHub were accepted. For human answers, questioners accepted four out of 8 answers.

Table 4.8: Results of our live study between GitSearch, OpenHub, and Human. "Resp" in "#Ans" is the number of questions answered by each technique while "Acc" is the number of answers accepted by the questioners. "Up and down" votes are the number of votes given by `StackOverflow` users. "Pos." and "Neg. Comm." comments are positive and negative comments made by the users for each answer. "Most voted?" represents the number of answers that received the most number of upvotes. $|x|$ indicates the number of answers with at least one up/down vote and positive/negative comment. Σ is the sum of occurrences while the numbers in parentheses are average (i.e., $\Sigma/Resp$).

	#Ans		Upvotes		Downvotes		Pos. Comm.		Neg. Comm.		Most
	Resp	Acc	$\|x\|$	Σ	$\|x\|$	Σ	$\|x\|$	Σ	$\|x\|$	Σ	voted?
GitSearch	25	3	6	7 (0.28)	5	10 (0.40)	7	7 (0.28)	3	5 (0.20)	4 (0.16)
OpenHub	18	0	2	3 (0.17)	2	3 (0.17)	2	2 (0.11)	2	3 (.17)	0
Human	8	4	4	9 (1.13)	1	1 (0.13)	3	6 (0.75)	2	2 (0.25)	3 (0.38)

Our technique received more up and downvotes than OpenHub while human answers took more upvotes and less downvotes. Six out of 25 answers by GitSearch received at least one upvote while other five of them took at least one downvote (six upvotes and 10 downvotes in total). Four of the six were the most-upvoted answers in their posts. OpenHub's answers had only two upvotes and three downvotes, respectively. Human answers took 9 upvotes (from four different answers) and 1 downvote (note that a single answer took 4 upvotes) where three answers were most-voted. In `StackOverflow`, votes imply that those users would encourage (or discourage) the answer. While its up and downvotes were almost tied with human results, it is obvious that GitSearch had more interest from users than OpenHub.

In addition, GitSearch initiated user discussions more frequently. We counted comments made by `StackOverflow` users and examined whether each comment is positive and negative. Our answers took 7 positive and 5 negative comments while OpenHub's results were followed by two and three, respectively. GitSearch does not explicitly outperform human answers (6 positive and 2 negative) but note that there were 17 of out 25 questions unanswered yet by human users. For the 17 questions, our technique answered them and received one upvote and three downvotes as well as three positive and two negative comments.

Discussion: The results of this study implies that GitSearch can be a better first responder than OpenHub. As shown in Table 4.8, many questions in `StackOverflow` remain unanswered for several days. Our technique can provide a starting point for questions even if they are not complete answers as many users would follow up the answers by giving their votes and adding comments. Once users are interested in a question, there might be more probability to discuss solutions for the question.

In addition, code search results by GitSearch can be selected by `StackOverflow` users as accepted answers, which implies that the results are highly relevant and appropriate to the questions. For three out of 25 questions, the questioners accepted our results even though the answers have only code excerpt from real source code without any additional explanation. Note that a questioner can select only one answer as the accepted one. This may indicate that questioners would take advantage of code search results to deal with their problems shown in the question. Furthermore, this can imply that code search engines would be an automatic answer generator for some questions in `StackOverflow` if their accuracy is improved.

4.5.5 Threats to Validity

The design of CoCaBu and the implementation of GitSearch raises a number of threats to validity that we have tried to mitigate. We list them below:

Internal validity: the user study was performed with a limited total number of 34 (=14+20) participants compared to the large number of participants used by Muse [374] authors for their API example search engine. However, among free-form code search works, some do not perform user studies (e.g., [15]), while others use fewer participants than us (e.g., CodeHow (20), Portfolio (19), SNIFF (undisclosed)). We have attempted to reach representativity by inviting professional developers as well as graduate students.

In addition, throughout the live study (Section 4.5.4), we tried to take feedback from `StackOverflow` overflow users in the loop of problem-solving. This implies that an additional number of participants were involved in our evaluation.

External validity: we used only English as a query language, focused on Java-related questions, and explored only `StackOverflow` and `GitHub` in our implementation. This threat should be limited by the fact that (1) English is a popular language in the programming community, (2) Java is one of the most popular programming languages, and furthermore, (3) `GitHub` and `StackOverflow` are the largest code hosting site and Q&A forum respectively.

Construct validity: we only focus on queries without the exact name of APIs. This threat, however, is limited since for new tasks, developers often do not know the name of the relevant APIs [193].

4.6 Related Work

There are several lines of research work that relate to our approach. We list their main contributions in each category.

4.6.1 API usage examples search

Recently, there have been a number of code search techniques [21, 170, 249, 338, 374, 524], focusing on locating API usage examples. Searching for specific API usages is a subset of code search activities. Compared to general code search, developers tend to be aware of the exact (or similar) name of a target API, which facilitates the search. Thus, these techniques focus on creating an index of API call sites only.

Moreno et al. [374] proposed Muse, an approach to mining and ranking code examples that show how to use a given method. Muse and CoCaBu differ on three main aspects. First, CoCaBu supports free-form queries, while Muse takes as input an API method signature. Second, Muse provides a code snippet for a specific method. CoCaBu, on the other hand, is not attached to a single API, and shows a set if APIs used to solve the task at hand. Lastly, Muse requires fully compilable client

projects in order to apply static slicing. Note that CoCaBu is able to handle incomplete source code.

Chatterjee et al. presented SNIFF [83], a technique that combines API documentation with publicly available Java code. SNIFF annotates each method call statement with its corresponding API documentation. This allows free-form English queries about the task at hand, which relaxes the need to know the appropriate API beforehand. Although SNIFF returns usage code examples as well, it requires a fully compilable code unit and the accompanying API documentation as well as external libraries. Additionally, the before-mentioned code intersection is not suitable for Internet-scale code search, because it has a complexity of O(n^2), where n is the number of hits.

4.6.2 Source code search

There have been several approaches to code search, which are relevant to CoCaBu. CodeHow, Sourcerer and Portfolio constitute the state-of-the-art of such approaches in the literature. CodeHow [333] leverages code documentation to recognize the potential APIs a query refers to and expands the query with these APIs to improve the accuracy of the search results. In contrast, CoCaBu assumes that 1) documentation is not always available, and 2) leveraging independent API documentation may create noise in a query whose answer requires a specific set of related APIs. Furthermore, CoCaBu augments queries based on information of code terms in source code snippets.

Sourcerer [15] is an infrastructure that facilitates the collection and analysis of large-scale open-source repositories. On top of that infrastructure, Sourcerer provides programmatic access to all the artifacts stored and managed through a set of services. Sourcerer crawls Java projects from several types of code repositories such open code repositories (e.g. Sourceforge and Apache) and web sites. Similar to CoCaBu, Sourcerer leverage structural code information to perform fine-grained code search. However, the construction of the search index requires a complete compilation unit (i.e., all dependencies must be resolved). Moreover, we exploit high-quality code snippets from `StackOverflow` to improve the quality of code search results.

Portfolio [357] retrieves and visualizes relevant functions and their usage scenarios to highlight a chain of function invocations. To realizes their objective, Portfolio computes the textual similarity between a user query and the function signatures. Subsequently, a function call graph is employed to locate functions which are relevant to a task, even if those function signatures do not include any keywords of the query. Compared to Portfolio, CoCaBu focuses on usage examples that answer complex queries by leveraging `StackOverflow` code snippets.

OpenHub Code Search [399] (formerly ohloh.net) is a free web-based code search engine. Although OpenHub has indices of more than 21 billion lines of code collected from open source projects on the Internet, it directly matches query terms with terms in source files. This is a common limitation of several Internet-scale search engines, including Codota [96]. Contrary to them, we resolve the vocabulary mismatch problem by augmenting user queries.

4.6.3 Query Reformulation

The literature of software engineering research in general, and code search in particular, includes a number of approaches dealing with query reformulations. Haiduc et al. [174] proposed a query reformulation strategy leveraging machine learning on a set of historical queries and associated relevant results. In contrast, CoCaBu does not require labelled data (which can be expensive to build for a large scale engine). Exemplar [162] uses help documents to expand queries while CoCaBu accounts for the fact that a number of relevant code examples in repositories are actually not documented. Finally, CodeExchange [348] further refines textual methods by exploiting relationships between

successive user queries. Such an approach can complement CoCaBu-based implementations of search engines by further improving the selection of expansion tokens.

There are broadly two ways of reformulating a query: a global approach would use a thesaurus, like WordNet, to enumerated related words and synonyms of the query terms; a more local approach, however, iteratively tries to expand the query by considering extra terms appearing in initial results obtained with the original query and which are marked as relevant by the searcher. Query expansion has been shown to be effective in many natural language processing (NLP) tasks [74, 576]. In code search research, query expansion has been intensively used in recent years: Wang et al. [548] consider human intervention to rank search results. Sisman and Kak [485] also proposed to leverage user feedback for searching code in the context of bug localization. CONQUER [190, 441, 471] refines the queries by suggesting the most highly co-occurring words that appear in the source code as alternative words. More recently, Lu et al. [327] developed a query reformulation technique based on part-of-speech of each word in queries and WordNet. Lemos et al. [285] proposed AQE, an automatic query expansion approach, which uses test cases as inputs, and leverages WordNet and SwordNet [584], a code-related thesaurus, to expand queries.

To deal with the search on structured data in the web (e.g., databases of IT consumables), Gollapudi et al. [152] have proposed to rewrite web search queries expanding them to include tags on the nature of each token (e.g., brand, display size, etc.), thus creating a structured query. Likewise, CoCaBu implements a query structuration approach, to improve accuracy. Closely related to our work, QECK [396], concurrently developed with CoCaBu, automatically extracts software-specific expansion words from Q&A posts on Stack Overflow: the idea is to identify what are the most recurrent code terms that are often associated with specific query terms. Our approach, however, is more refined as we focus on Q&A posts which are mostly related to the user input query.

4.6.4 Miscellaneous

Code recommendation: Recommendation engines assist developers in their use of complex libraries or frameworks by presenting them with reusable code fragments in other locations of their code, with documentation, or with pointers to blogs and Q&A sites. Strathcona [197] is an approach in which a query is generated from a user's source code and matched with an example repository that uses a target library of the framework. They thus require prior knowledge of the relevant library.

Prompter [412], on the contrary, does not provide code snippets but matches the current code context with relevant `StackOverflow` posts. The technique relies on different features to capture the similarity between `StackOverflow` discussions and the current code context. On the other hand, our approach does not recommend discussions but use `StackOverflow`'s code snippets to search for similar usage examples in a large code repository.

Stack Overflow: Several studies have explored `StackOverflow` questions and answers [33, 337, 381, 529]. However, to the best of our knowledge, its data has never been leveraged to improve code search engine results.

4.7 Conclusion

We have presented CoCaBu, a novel approach to addressing the vocabulary mismatch problem in code search. CoCaBu augments free-form queries by leveraging code snippets in answers of related posts from Q&A sites. The key insight from our work is that it is possible to map human concepts expressed in queries (which are often written with similar terms by developers) with structural code entities (which are the most relevant terms for matching source code with high relevance). We implemented a code search engine, GitSearch, following the CoCaBu approach for the `GitHub` super-repository

of projects. To that end, we leveraged `StackOverflow` posts to find the best mappings between developer query terms and structural code entities. Our evaluation with user studies demonstrated that GitSearch outperforms Internet-scale code search engines and is competitive against established web search engines for resolving programming tasks. We also found with a live study that users in Q&A forums show interest in the real-world code examples yielded by GitSearch.

Availability

We make all our data available: source code of GitSearch, search indices, user study results. See `https://github.com/serval-snt-uni-lu/cocabu`. A prototype implementation of cocabu-based search engine, GitSearch, is live at `http://www.cocabu.com`.

5 FaCoY – A Code-to-Code Search Engine

Code search is an unavoidable activity in software development. Various approaches and techniques have been explored in the literature to support code search tasks. Most of these approaches focus on serving user queries provided as natural language free-form input. However, there exists a wide range of use-case scenarios where a code-to-code approach would be most beneficial. For example, research directions in code transplantation, code diversity, patch recommendation can leverage a code-to-code search engine to find essential ingredients for their techniques. In this chapter, we introduce FaCoY*, a novel approach for statically finding code fragments which may be* semantically *similar to user input code.* FaCoY *implements a* query alternation *strategy: instead of directly matching code query tokens with code in the search space,* FaCoY *first attempts to identify other tokens which may also be relevant in implementing the functional behavior of the input code. With various experiments, we show that (1)* FaCoY *is more effective than online code-to-code search engines; (2)* FaCoY *can detect more semantic code clones (i.e., Type-4) in BigCloneBench than the state-of-the-art; (3)* FaCoY*, while static, can detect code fragments which are indeed similar with respect to runtime execution behavior; and (4)* FaCoY *can be useful in code/patch recommendation.*

This chapter is based on the work published in the following research paper:

- Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY – A Code-to-Code Search Engine. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 946–957, 2018

Contents

5.1 Overview

In software development activities, source code examples are critical for understanding concepts, applying fixes, improving performance, and extending software functionalities [24, 266, 347, 569, 573]. Previous studies have even revealed that more than 60% of developers search for source code every day [193, 497]. With the existence of super-repositories such as `GitHub` hosting millions of open source projects, there are opportunities to satisfy the search need of developers for resolving a large variety of programming issues.

Oftentimes, developers are looking for code fragments that offer similar functionality than some other code fragments. For example, a developer may need to find Java implementations of all sorting algorithms that could be more efficient than the one she/he has. We refer to such code fragments which have similar functional behavior even if their code is dissimilar as *semantic clones*. The literature also refers to them as *Type-4* clones for consistency with the taxonomy of code clones [42, 445]. Besides the potential of helping developers collect relevant examples to improve their code, finding similar code fragments is an important endeavor, at they can provide essential ingredients for addressing challenges in various software engineering techniques. Among such challenges, we can enumerate automated software transplantation [31], software diversification [34], and even software repair [243].

Finding semantically similar code fragments is, however, challenging to perform statically, an essential trait to ensure scalability. A few studies [129, 228] have investigated program inputs and outputs to find equivalent code fragments. More recently, Su et al. [501] have proposed an approach to find code relatives relying on instruction-level execution traces (i.e., code with similar execution behaviors). Unfortunately, all such dynamic approaches cannot scale to large repositories because of their requirement of runtime information.

Our key insight to statically find code fragments which are semantically similar is first to undertake a description of the functionality implemented by any code fragment. Then, such descriptions can be used to match other code fragments that could be described similarly. This insight is closely related to the work by Marcus and Maletic on *high-level concept clones* [344] whose detection is based on source code text (comments and identifiers) providing an abstract view of code. Unfortunately, their approach can only help to identify high-level concepts (e.g., abstract data types), but is not targeted at describing functionalities per se.

Because of the vocabulary mismatch problem [140, 174, 602, 604] between code terms and human description words, it is challenging to identify the most accurate terms to summarize, in natural language, the functionality implemented by a code fragment.

To work around the issue of translating a code fragment into natural language description terms, one can look[1] up to a developer community. Actually, developers often resort to web-based resources such as blogs, tutorial pages, and Q&A sites. `StackOverflow` is one of such leading discussion platforms, which has gained popularity among software developers. In `StackOverflow`, an answer to a question is typically short texts accompanied by code snippets that demonstrate a solution to a given development task or the usage of a particular functionality in a library or framework. `StackOverflow` provides social mechanisms to assess and improve the quality of posts that leads implicitly to high-quality source code snippets on the one hand as well as concise and comprehensive questions on the other hand. Our intuition is that information in Q&A sites can be leveraged as a collective knowledge to build an intermediate translation step before the exploration of large code bases.

[1] Because software projects can be more or less poorly documented, we do not consider source code comments as a reliable and consistent database for extracting descriptions. Instead, we rely on developer community Q&A sites to collect descriptions.

This paper. We propose FACOY (Find a Code other than Yours) as a novel, static, scalable and effective code-to-code search engine for finding semantically similar code fragments in large code bases.

Overall, we make the following contributions:

• **The FACOY approach for code-to-code search**: We propose a solution to discover code fragments implementing similar functionalities. Our approach radically shifts from mere syntactic patterns detection. It is further fully static (i.e., relies solely on source code) with no constraint of having runtime information. FACOY is based on *query alternation*: after extracting structural code elements from a code fragment to build a query, we build alternate queries using code fragments that present similar descriptions to the initial code fragment. We instantiate the FACOY approach based on indices on Java files collected from GitHub and Q&A posts from `StackOverflow` to find the best descriptions of functionalities implemented in a large and diversified set of code snippets.

• **A comprehensive empirical evaluation**. We present evaluation results demonstrating that FACOY can accurately help search for (syntactically and semantically) similar code fragments, outperforming popular online code-to-code search engines. We further show, with the BigCloneBench benchmark [513], that we perform better than the state-of-the-art on static code clone detectors identifying semantic clones; our approach identifies over 635,000 semantic clones, while others detect few to no semantic clones. We also break down the performance of FACOY to highlight the added-value of our *query alternation* scheme. Using the DYCLINK dynamic tool [501], we validate that, in 68% of the cases, our approach indeed finds code fragments that exhibit similar runtime behavior. Finally, we investigate the capability of FACOY to be leveraged for repair patch recommendation.

5.2 Motivation and Insight

Finding similar code fragments beyond syntactic similarities has several uses in the field of software engineering. For example, developers can leverage a code-to-code search tool to find alternative implementations of some functionalities. Recent automated software engineering research directions for software transplantation or repair constitute further illustrations of how a code-to-code search engine can be leveraged.

Despite years of active research in the field of code search and code clones detection, few techniques have explicitly targeted semantically similar code fragments. Instead, most approaches focus on textually, structurally or syntactically code fragments. The state-of-the-art techniques on static detection of code clones leverage various intermediate representations to compute code similarity. Token-based [25, 239, 304] representations are used in approaches that target syntactic similarity. AST-based [35, 227] representations are employed in approaches that detect similar but potentially structurally different code fragments. Finally, (program dependency) graph-based [270, 312] representations are used in detecting clones where statements are not ordered or parts of the clone code are intertwined with each other. Although similar code fragments identified by all these approaches usually have similar behavior, the contemporary static approaches still miss finding such fragments which have similar behavior even if their code is dissimilar [233].

To find similarly behaving code fragments, researchers have relied upon dynamic code similarity detection which consists in identifying programs that yield similar outputs for the same inputs. State-of-the-art dynamic approaches generate random inputs [228], rely on symbolic [297] or concolic execution [273] and check abstract memory states [256] to compute function similarity based on execution outputs. The most recent state-of-the-art on dynamic clone detection focuses on the computations performed by the different programs and compares instruction-level execution traces to identify equivalent behavior [501]. Although these approaches can be very effective in finding semantic code clones, dynamic execution of code is not scalable and implies several limitations for practical usage (e.g., the need of exhaustive test cases to ensure confidence in behavioral equivalence).

To search for relevant code fragments, users turn to online code-to-code search engines, such as Krugle [271], which statically scan open source projects. Unfortunately, such Internet-scale search engines still perform syntactic matching, leading to low-quality output in terms of semantic clones.

On the key idea Consider the code fragments shown in Figure 5.1. They constitute variant implementations for computing the hash of a string. These examples are reported in BigCloneBench [513] as type-4 clones (i.e., semantic clones). Indeed, their syntactic similarity is limited to a few library function calls. Textually, only about half of the terms are similar in both code fragments. Structurally, the first implementation presents only one execution path while the second includes two paths with the try/catch mechanism.

```
public String getHash(final String password)
        throws NoSuchAlgorithmException, UnsupportedEncodingException {
    final MessageDigest digest = MessageDigest.getInstance("MD5");
    byte[] md5hash;
    digest.update(password.getBytes("utf-8"), 0, password.length());
    md5hash = digest.digest();
    return convertToHex(md5hash);
}
```

(a) Excerpt from `MD5HashHelperImpl.java` in the `yes-cart` project.

```
public static String encrypt(String message) {
    try {
        MessageDigest digest = MessageDigest.getInstance("MD5");
        digest.update(message.getBytes());
        BASE64Encoder encoder = new BASE64Encoder();
        return encoder.encode(digest.digest());
    } catch (NoSuchAlgorithmException ex) {
    ...
```

(b) Excerpt from `Crypt.java` in the `BettaServer` project.

Figure 5.1: Implementation variants for hashing.

To statically identify the code fragments above as semantically similar, a code-to-code search engine would require extra hint that (i) *getHash* and *encrypt* deal with related concepts and that (ii) *BASE64Encoder* API is offering similar functionality as *convertToHex*. Such hints can be derived automatically if one can build a comprehensive collection of code fragments with associated descriptions allowing for high-level groupings of fragments (based on natural language descriptions) to infer relationships among code tokens. The inference of such relationships will then enable the generation of **alternate queries** displaying related, but potentially syntactically different, tokens borrowed from other code fragments having similar descriptions than the input fragment. Thus, given the code example of Figure 5.1(a), the system will detect similar code fragments by matching not only tokens that are syntactically[2] similar to the ones in this code fragment (i.e., `gethash`, `messagedigest`, and `converttohex`), but also others similar to known related tokens (e.g., `base64encoder`, `encrypt`, etc.). Such a system would then be able to identify the code fragment of Figure 5.1(b) as a semantically similar code fragment (i.e., a semantic clone).

We postulate that Q&A posts and their associated answers constitute a comprehensive dataset with a wealth of information on how different code tokens (found in code snippets displayed as example answers) can be matched together based on natural language descriptions (found in questions). Figure 5.2 illustrates the steps that could be unfolded for exploiting Q&A data to find semantic clones in software repositories such as Github.

Given a code fragment, the first step would consist to infer natural language terms that best describe the functionality implemented. To that end, we must match the code fragment with the closest code example provided in a Q&A post. Then, we search in the Q&A dataset all posts with similar descriptions to collect their associated code examples. By mixing all such code examples, we can

[2]Naive syntactic matching may lead to false positives. Instead, there is a need to maintain qualification information about the tokens (e.g., the token represents a method call, a literal, etc.) cf. Section 5.3.

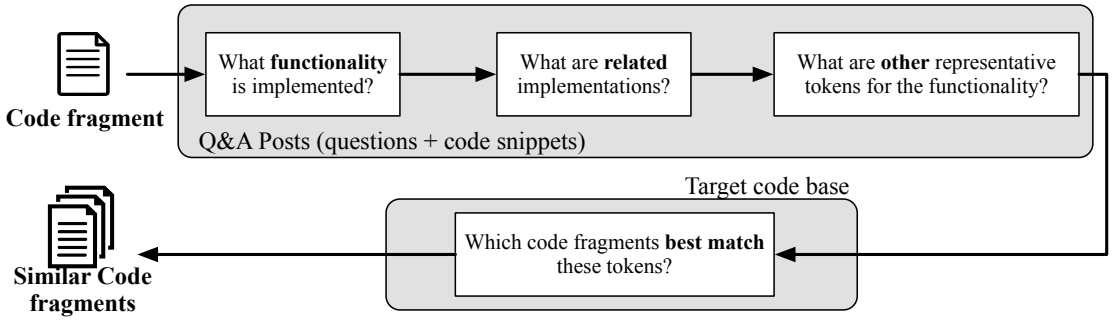


Figure 5.2: Conceptual steps for our search engine.

build a more diverse set of code tokens that could be involved in the implementation of the relevant functionality. Using this set of tokens, we can then search real-world projects for fragments which may be syntactically dissimilar while implementing similar functionality.

Basic definitions

We use the following definitions for our approach in Section 5.3.

• **Code Fragment**: A contiguous set of code lines that is fed as input to the search engine. The output of the engine is also a list of code fragments. We formalize it as a finite sequence of tokens representing (full or partial) program source code at different granularities: e.g., a function, or an arbitrary sequence of statements.

• **Code Snippet**: A code fragment found in Q&A sites. We propose this terminology to differentiate code fragments that are leveraged during the search process from those that are used as input or that are finally yielded by our approach.

• **Q&A Post**: A pair $p = (q, A)$ also noted p^q_A, where q is a question and A is a list of answers. For instance, for a given post p^q_A, question q is a document describing what the questioner is trying to ask about and $a \in A$ is a document that answers the question in q. Each answer a can include one or several code snippets: $S = snippets(a)$, where S is a set of code snippets.

We also recall for the reader the following well-accepted definitions of clone types [42, 445, 501]:

•**Type-1**: Identical code fragments, except for differences in whitespace, layout, and comments.

•**Type-2**: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.

•**Type-3**: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences.

•**Type-4**: Syntactically dissimilar code fragments that implement the same functionality. They are also known as *semantic clones*.

Disclaimer. In this work, we refer to a pair of code fragments which are semantically similar as *semantic clones*, although they might have been implemented independently (i.e., no cloning, e.g., copy/paste, was involved). Such pairs are primary targets of FaCoY.

5.3 Approach

FACOY takes a code fragment from a user and searches in a software projects' code base to identify code fragments that are similar to the user's input. Although the design of FACOY is targeted at taking into account functionally similar code with syntactically different implementations, the search often returns fragments that are also syntactically similar to the input query.

Figure 5.3 illustrates the steps that are unfolded in the working process of the search:

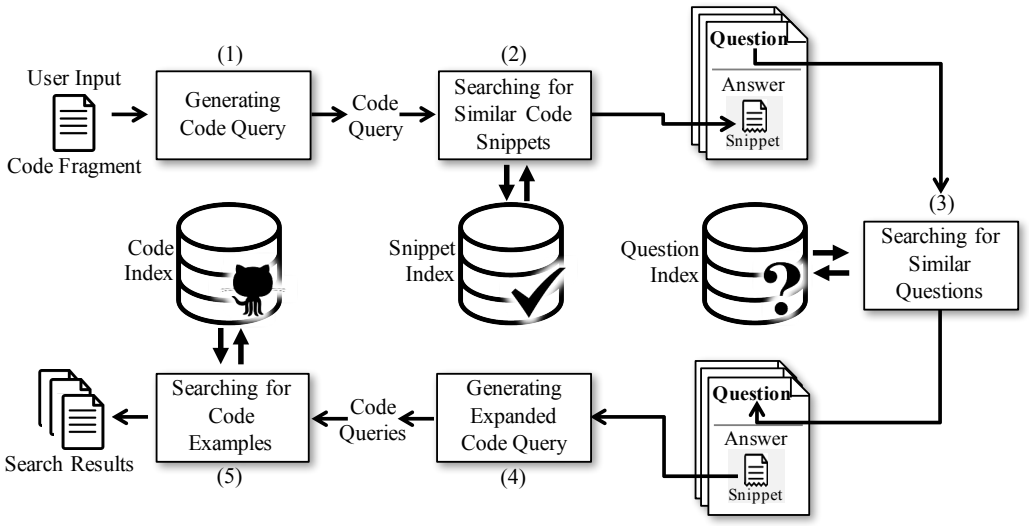


Figure 5.3: Overview of FACOY.

(1) When FACOY receives a code fragment, it generates a structured query called *Code Query* based on the code elements present in the fragment (Section 5.3.2.1).
(2) Given a code query, FACOY searches for Q&A posts that include the most syntactically similar code snippets. To that end, the query is matched against the *Snippet Index* of Q&A posts (Section 5.3.2.2).
(3) Once the relevant posts are identified, FACOY collects natural language descriptive terms from the associated questions and matches them with the *Question index* of Q&A posts to find other relevant posts. The objective is to find additional code snippets that could implement similar functionalities with a diversified set of syntactic tokens (Section 5.3.2.3).
(4) Using code snippets in answers of Q&A posts collected by previous steps, FACOY generates code queries that each constitutes an alternative to the first code query obtained from user input in Step (1) (Section 5.3.2.4).
(5) As the final step, FACOY searches for similar code fragments by matching the code queries yielded in Step (4) against the *Code Index* built from the source code of software projects (Section 5.3.2.5).

5.3.1 Indexing

Our approach constructs three indices, *snippet, question, and code*, in advance to facilitate the search process as well as ensuring a reasonable search speed [340]. To create these indices, we use Apache Lucene, one of the most popular information retrieval libraries [351]. Lucene indices map tokens into instances which in our cases can be natural language text, code fragments or source code files.

5.3.1.1 Snippet Index

The Snippet Index in FACOY maintains the mapping information between answer document IDs of Q&A posts and their code snippets. It is defined as a function:
$Inx_S : S \rightarrow 2^P$, where S is a set of code snippets, and P is a set of Q&A posts. 2^P denotes the power set of P (i.e., the set of all possible subsets of P). This index function maps a code snippet into a subset of P, in which the answer in a post has a similar snippet to the input. Our approach leverages the Snippet Index to retrieve the Q&A post answers that include the most similar code snippets to a given query.

To create this index, we must first collect code examples provided as part of a Q&A post answer. Since code snippets are mixed in the middle of answer documents, it is necessary to identify such regions containing the code snippets. Fortunately, most Q&A sites, including `StackOverflow`, make posts available in a structured document (e.g., HTML or XML) and explicitly indicate source code elements with ad-hoc tags such as <code> ··· <code> allowing FACOY to readily parse answer documents and locate code snippets.

After collecting a code snippet from an answer, FACOY creates its corresponding index information as a list of index terms. An index term is a pair of the form '**token_type**:*actual_token*" (e.g., **used_class**:*ActionBar*). Table 5.1 enumerates examples of token types considered by FACOY. The complete list is available in [434].

Table 5.1: Examples of token types for snippet index creation.

Type	Description
typed_method_call	(Partially) qualified name of called method
unresolved_method_call	Non-qualified name of called method
str_literal	String literal used in code

Figure 5.4 shows an example of code fragment with the corresponding index terms.

```
public class BaseActivity extends AppCompatActivity{
    public static final int IMAGE_PICK_REQUEST_CODE =
      5828;
    public static final int MUSIC_PICK_REQUEST_CODE =
      58228;
    protected ActionBar actionBar;
    @Override
    protected void onCreate(Bundle savedInstanceState)
      {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_based);
}   }
```

(a) Example code fragment.

```
extends:AppCompatActivity
used_class:BaseActivity
used_class:R
used_class:R.layout
used_class:ActionBar
used_class:Bundle
methods_declaration:onCreate
typed_method_call:AppCompatActivity.onCreate
typed_method_call:AppCompatActivity.setContentView
```

(b) Corresponding index terms.

Figure 5.4: Extraction of index terms from a code fragment.

To generate index terms, FACOY must produce the abstract syntax tree (AST) of a code snippet. Each AST node that corresponds to a token type in Table 5.1[3] is then retained to form an index

[3]Our approach focuses on the types in the table since they represent most of the characteristics in a code snippet.

term. Finally, FACOY preserves, for each entry in the index, the answer document identifier to enable reverse lookup.

The main challenge in this step is due to the difficulty of parsing *incomplete code*, a common trait of code snippets in Q&A posts [481]. Indeed, it is common for such code snippets to include partial statements or excerpts of a program, with the purpose to give only some key ideas in a concise manner. Often, snippets include ellipses (i.e., "...") before and after the main code blocks. To allow parsing by standard Java parsers, FACOY resolves the problem by removing the ellipses and wrapping code snippets with a custom dummy class and method templates.

Besides incompleteness, code snippets present another limitation due to *unqualified names*. Indeed, in code snippets, enclosing class names of method calls are often ambiguous [106]. A recent study [506] even reported that unqualified method names are pervasive in code snippets. Recovering unqualified names is, however, necessary for ensuring accuracy when building the Snippet Index. To that end, FACOY transforms unqualified names to (partially) qualified names by using structural information collected during the AST traversal of a given code snippet. This process converts variable names on which methods are invoked through their corresponding classes. Figure 5.5 showcases the recovering of name qualification information. Although this process cannot recover all qualified names, it does improve the value of the Snippet Index.

```
SAXParserFactory ft;
InputSource is ;
URL ul = new URL(feed)
ft = SAXParserFactory.newInstance();
SAXParser pr = factory.newSAXParser();
XMLReader rd = pr.getXMLReader();
RssHandler hd = new RssHandler();
rd.setContentHandler(hd);
is = new InputSource(url.openStream());
xmlreader.parse(is);
return hd.getFeed();
```

(a) Fragment before recovering name qualification.

```
SAXParserFactory ft;
InputSource is;
URL ul = new URL(feed)
ft = SAXParserFactory.newInstance();
SAXParser pr = _SAXParserFactory_.newSA...
XMLReader rd = _SAXParser_.getXMLReader();
RssHandler hd = new RssHandler();
_XMLReader_.setContentHandler(hd);
is = new InputSource(_URL_.openStream());
_XMLReader_.parse(is);
return _RssHandler_.getFeed();
```

(b) Fragment after recovering name qualification.

Figure 5.5: Recovery of qualification information [481]. Recovered name qualifications are highlighted by red color.

Disclaimer. FACOY is compliant with the Creative Commons Attribute-ShareAlike license [10, 103] of `StackOverflow`: we do not redistribute any code from Q&A posts. We only mine developer code examples in `StackOverflow` to learn relationships among tokens.

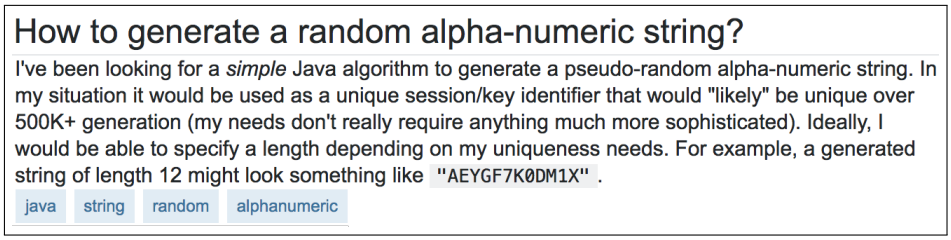
5.3.1.2 Question Index

The Question Index maps a set of word tokens into Q&A posts. The mapping information serves to identify Q&A posts where the questions are similar to the questions retrieved in Step (2) (cf. Figure 5.3) whose answers contain code snippets that are similar to the user input. FACOY leverages this index to construct alternate queries to the one derived from user input. These alternate queries are necessary to increase the opportunities for finding semantically similar code fragments rather than only syntactically similar fragments. The following equation defines the Question Index function: $Inx_Q : Q \rightarrow 2^P$, where Q is a set of questions, and P is a set of Q&A posts. This function maps a set of words from the input into a subset of posts ($P_E \in P$) that are similar to the input.

To build the Question Index, FACOY takes the question part (q) of each Q&A post and generates index terms. To that end a pre-processing of the question text is necessary. This pre-processing includes tokenization (e.g., splitting camel case), stop word removal[4] [340], and stemming. From the pre-processing output, FACOY builds index terms in the form of "`term`:*token*". Each index term

[4]Using Lucene's (version 4) English default stop word set.

is further mapped with the question where its token is originated from, to keep an inverse link. Figure 5.6 illustrates how, given a question text, index terms are generated.



(a) Description text in a question.

term:simpl, term:java, term:algorithm, term:generat, term:pseudo, term:random, term:alpha, term:numer, term:string, term:situat, term:uniq, term:session, term:key, term:identifi, term:uniq, term:500k, term:requir, term:sophist, term:ideal, term:length, term:depend, term:string, term:length, term:12, term:aeygf7k0dm1x

(b) Resulting index terms generated from (a).

Figure 5.6: Example of question index creation.

5.3.1.3 Code Index

The *Code Index* maintains mapping information between tokens and source code files. FACOY leverages this index to search for code examples corresponding to a code query yielded at the end of Step (4) (cf. Figure 5.3). This index actually defines the search space of our approach (e.g., F-droid repository of Android apps, or Java projects in Github, or a subset of Mozilla projects). The Code Index function is defined as: $Inx_C : S \rightarrow 2^F$, where S is a set of code snippets and F is a set of code fragments. F actually defines the space of FACOY.

The creation process of the Code Index is similar to the process for building the Snippet Index that is described in Section 5.3.1.1. FACOY first scans all available source code files in the considered code base. Then, each file is parsed[5] to generate an AST from which FACOY collects the set of AST nodes corresponding to the token types listed in Table 5.1. The AST nodes and their actual tokens are used for creating index terms in the same format as in the case of the Snippet Index. Finally, each index term is mapped to the source code file where the token of the term has been retrieved.

5.3.2 Search

Once the search indices are built, FACOY is ready to take a user query and search for relevant code fragments. Algorithm 1 formulates the search process for a given user query. Its input also considers the three indices described in Section 5.3.1 and stretch parameters used in the algorithm. The following sections detail the whole process.

5.3.2.1 Generating a Code Query from a User Input

As the first step of code search, FACOY takes a user input and generates a code query from the input to search the snippet index (Line 2 in Algorithm 1). The code query is in the same form of index terms illustrated in Figure 5.4 so that it can be readily used to match the index terms in the Snippet Index.

[5]This step also recovers qualified names by applying, whenever necessary, the same procedure described in Section 5.3.1.1.

Input : c: code fragment (i.e., user query).
Input : $Inx_S(q_s)$: a function of a code snippet, s, to a sorted list of posts, P_s.
Input : $Inx_Q(q_q)$: a function of a question, q, to a sorted lis of questions, Q_q.
Input : $Inx_C(q_s)$: a function of a code snippet, s, to a sorted list of code fragments, F_s.
Input : n_s, n_q, n_c: stretch parameters for snippet, question, and code search, respectively (e.g., consider Top n similar snippets).
Output : F: a set of code fragments that are similar to c.

```
1  Function SearchSimilarCodeExamples(c, Inx_S, Inx_Q, Inx_C, n_s, n_q, n_c)
2      q_in ⇐ genCodeQuery(c);
3      P_s ⇐ Inx_S(q_in).top(n_s);
4      Q_s ⇐ P_s.foreach(p_i => takeQuestion(p_i));
5      P_e ⇐ Q_s.foreach(q_i => Inx_Q(q_i).top(n_q));
6      let F ⇐ ∅;
7      foreach p_i ∈ P_e do
8          s ⇐ takeSnippet(getAnswer(p_i));
9          q_ex ⇐ genCodeQuery(s);
10         F ⇐ F ∪ Inx_C(q_ex).top(n_c));
11     end
12     return F;
13 end
```

Algorithm 1: Similar code search in FACOY.

To generate a code query, our approach follows the process described in Section 5.3.1.1 for generating the index terms of any given code snippet. If the user input is also an incomplete code fragment (i.e., impossible to parse), FACOY seamlessly wraps the fragment using a dummy class and some method templates after removing ellipses. It then parses the code fragment to obtain an AST and collect the necessary AST nodes to generate index terms in the form of **token_type**:*actual_token*.

5.3.2.2 Searching for Similar Code Snippets

After generating a code query from a user input, our approach tries to search for similar snippets in answers of Q&A posts (Line 3 in Algorithm 1). Since the code query and index terms in the snippet index are in the same format, our approach uses full-text search (i.e., examining all index terms for a code snippet to compare with those in a code query). The full-text function implemented by Lucene is utilized.

Our approach computes rankings of the search results based on a scoring function that measures the similarity between the code query and matched code snippets. FACOY integrates two scoring functions, Boolean Model (BM) [276] and Vector Space Model (VSM) [456], which are already implemented in Lucene. First, BM reduces the amount of code snippets to be ranked. Our approach transforms the code query of the previous step, q_in, into a normal form and matches code snippets indexed in the snippet index. We adopt best match retrieval to find as many similar snippets as possible. Then, for the retained snippets, VSM computes similarity scores. After computing TF-IDF (Term Frequency - Inverse Document Frequency) [458] of terms in each snippet as a weighting scheme, it calculates Cosine similarity values between the code query and indexed snippets.

From the sorted list of similar snippets, FACOY takes top n_s snippets (i.e., those that will allow to consider only most relevant natural language descriptions to associate with the user input). By default, in all our experiments in this paper, unless otherwise indicated, we set the value of n_s (stretch parameter) to 3.

5.3.2.3 Searching for Similar Questions

In this step (Line 4 in Algorithm 1), our approach searches for questions similar to the questions of Q&A posts retrieved in the previous step (cf. Section 5.3.2.2). The result of this step is an additional set of Q&A posts containing questions that are similar to the given questions identified as describing best the functionality implemented in the user input. Thanks to this **search space enrichment**

approach, FACOY can include more diverse code snippets for enumerating more code tokens which are semantically relevant.

To search for similar questions, we use the Question Index described in Section 5.3.1.2. Since all questions are indexed beforehand, the approach simply computes similarity values between questions as the previous step does (cf. Section 5.3.2.2), i.e., filtering questions based on BM and calculating cosine similarity based on VSM.

Once similarity scores are computed, we select the top n_q posts based on the scores of their questions, as the goal is to recommend most relevant questions rather than listing up all similar questions. Since it takes n_s posts for each of n_q questions retrieved in Line 3 of Algorithm 1, the result of this step consists of $n_s \times n_q$ posts when using the same stretch parameter for both steps. FACOY can be tuned to consider different stretch values for each step.

5.3.2.4 Generating Alternate Code Queries

This step (Line 5 in Algorithm 1) generates code queries from code snippets present in newly retrieved Q&A posts at the end of the previous step (cf. Section 5.3.2.3). Our approach in this step first takes Q&A posts identified in Line 4 and extracts code snippets from their answer parts. It then follows the same process described in Section 5.3.2.1 to generate code queries. Since the result of the previous step (Line 4) is $n_s \times n_q$ posts (when using the same value for stretch parameters), this step generates at most $n_s \times n_q$ code queries, referred to as *alternate queries*.

5.3.2.5 Searching for Similar Code fragments

As the last step, FACOY searches the Code Index for similar code fragments to output (Lines 6–12 in Algorithm 1). Based on the alternate code queries generated in the previous step (cf. Section 5.3.2.4), and since code queries and index terms are represented in the same format, FACOY can leverage the same process of Step (2) illustrated in Section 5.3.2.2 to match code fragments. While the step described in Section 5.3.2.2 returns answers containing code snippets similar to a user query, the result of this step is a set of source code files containing code fragments corresponding to the alternate code query from the previous step. Note that FACOY provides at most $n_s \times n_q \times n_c$ code fragments as Line 10 in Algorithm 1 uses n_c to take top results.

Delimitating code fragments: Since displaying the entire content of a source code file will be ineffective for users to readily locate the identified similar code fragment, FACOY specifies a code range after summarizing the content [340]. To summarize search results into a specific range, FACOY uses a query-dependent approach that displays segments of code based on the query terms occurring in the source file. Concretely, the code fragment starts from k lines preceding the first matched token and spreads until k lines following the last matched token.

5.4 Evaluation

In this section, we describe the design of different assessment scenarios for FACOY and report on the evaluation results. Specifically, our experiments aim to address the following research questions:

- **RQ1**: How relevant are code examples found by FACOY compared to other code-to-code search engines?
- **RQ2**: What is the effectiveness of FACOY in finding semantic clones based on a code clone benchmark?
- **RQ3**: Do the semantically similar code fragments yielded by FACOY exhibit similar runtime behavior?

- **RQ4**: Could FACOY recommend correct code as alternative of buggy code?

To answer these research questions, we build a prototype version of FACOY where search indices are interchangeable to serve the purpose of each assessment scenario. We provide in Section 5.4.1 some details on the implementation before describing the design and results for the different evaluations.

5.4.1 Implementation details

Accuracy and speed performance of a search engine are generally impacted by the quantity of data and the quality of the indices [406]. We collect a comprehensive dataset from `GitHub`, a popular and very large open source project repository, as well from `StackOverflow`, a popular Q&A site with a large community to curate and propose accurate information on code examples. We further leverage the Apache Lucene library, whose algorithms have been tested and validated by researchers and practitioners alike for indexing and searching tasks.

For building the Code Index representing the search space of the code base where to code fragments, we consider the `GitHub` repository. We focus on Java projects since Java remains popular in the development community and is associated with a large number of projects in `GitHub` [50]. Overall, we have enumerated 2,993,513 projects where Java is set as the main language. Since there are many toy projects on `GitHub` [237], we focused on projects that have been forked at least once by other developers and dropped out projects where the source code include non-ascii characters. Table 5.2 summarizes the collected data.

Table 5.2: Statistics on the collected GitHub data.

Feature	Value	Feature	Value
Number of projects	10,449	Number of duplicate files	382,512
Number of files	2,163,944	LOCs	>384M

For building the Snippet and Question indices, we downloaded a dump file from the `StackOverflow` website containing all posts between July 2008 and September 2016 in XML format. In total, we have collected and indexed 1,856,592 posts tagged as about Java or Android coding. We have used a standard XML parser to extract natural language elements (tagged with `<p>...</p>` markers) and code snippets (tagged with `<code>...</code>`). It should be noted that we first filter in code snippets from answers that have been accepted by the questioner. Then we only retained those accepted answers that have been up-voted at least once. These precautions aim at ensuring that we leverage code snippets that are of high quality and are really matching the questions. As a result, we eventually used 268,264 Q&A posts to build the snippet and question indices. By default, we set all three stretch parameters to $n_s = n_q = n_c = 3$. The stretch for delimitating output code fragments is also set to $k = 3$.

5.4.2 RQ1: Comparison with code search engines

Design: In this experiment, we compare the search results of FACOY with those from online code search engines. We focus on Krugle [271] and searchcode [469] since these engines support code-to-code search. As input code fragments, we consider code examples implementing popular functionalities that developers ask about. To that end, we select snippets from posts in `StackOverflow`. The snippets are selected following two requirements: (1) the associated post is related to "Java" and (2) the answer include code snippets. We select code snippets in the top 10 posts with the highest view counts (for their questions). Table 5.3 lists the titles of `StackOverflow` posts whose code snippets are used in our experiment. Note that, for a fair comparison and to avoid any bias towards FACOY, the actual posts (including the code snippets in their answers) shown in the table are removed from the

Table 5.3: Top 10 `StackOverflow` Java posts with code snippets.

Query #	Question title
Q1	How to add an image to a JPanel?
Q2	How to generate a random alpha-numeric string?
Q3	How to save the activity state in Android?
Q4	How do I invoke a Java method when given the method name as a string?
Q5	Remove HTML tags from a String
Q6	How to get the path of a running JAR file?
Q7	Getting a File's MD5 Checksum in Java
Q8	Loading a properties file from Java package
Q9	How can I play sound in Java?
Q10	What is the best way to SFTP a file from a server?

snippet and question indices; this prevents our tool from leveraging answer data in advance, which would be unfair.

Figure 5.7 shows an example of input code fragments collected from `StackOverflow` that is used in our evaluation. 10 code snippets[6] are then used to query FACOY, `Krugle`, and `searchcode`.

```
1 import java.security.SecureRandom;
2 import java.math.BigInteger;
3 public final class SessionIdentifierGenerator {
4   private SecureRandom random = new SecureRandom();
5   public String nextSessionId() {
6     return new BigInteger(130, random).toString(32);
7   }
8 }
```

Figure 5.7: Code snippet associated to Q2 in Table 5.3.

On each search engine, we consider at most the top 20 search results for each query and manually classify them into one of the four clone types defined in Section 5.2.

Table 5.4: Statistics based on manual checks of search results.

	FACOY				Searchcode			Krugle	
Query	**# outputs**	**Type-2**	**Type-3**	**Type-4**	**# outputs**	**Type-1**	**Type-3**	**# outputs**	**Type-1**
Q1	18		5(27.7%)	4(22.2%)	0			0	
Q2	21		6(28.5%)	2(9.5%)	0			0	
Q3	18		9(50%)		0			0	
Q4	0				20	20(100%)		1	1(100%)
Q5	19	1(5.2%)	2(10.5%)	6(31.5%)	3	2(66.6%)	1(33.3%)	0	
Q6	9	1(11.1%)	3(30%)	1(11.1%)	20	20(100%)		0	
Q7	17				0			0	
Q8	17		7(41.1%)	7(41.1%)	0			0	
Q9	0				2		2(100%)	0	
Q10	9		1(11.1%)	7(77.7%)	0			0	

Result: Table 5.4 details statistics on the search results for the different search engines. FACOY, `Krugle`, and `searchcode` produce search results for eight, four and one queries respectively. Search results can also be false positives. We evaluate the efficiency of FACOY using the *Precision*@k metric defined as follows:

$$Precision@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{|relevant_{i,k}|}{k} \tag{5.1}$$

where $relevant_{i,k}$ represents the relevant search results for query i in the top k returned results, and Q is a set of queries.

FACOY achieves 57.69% and 48.82% scores for *Precision*@10 and *Precision*@20 respectively.

[6]Code snippets available on project web page [434].

We further categorize the true positive code fragments based on the clone type. `Krugle` appears to be able to identify only Type-1 clones. `searchcode` on the other hand also yields some Type-3 code clones for 2 queries. Finally, FACOY mostly successfully retrieves Type-3 and Type-4 clones.

Unlike online code-to-code search engines, FACOY can identify (1) similar code fragments for a more diverse set of functionality implementations. Those code fragments can be syntactically dissimilar to the query while implementing similar functionalities.

5.4.3 RQ2: Finding similar code in IJaDataset

Design: This experiment aims at evaluating FACOY against an existing benchmark. Since our code-to-code search engine is similar to a code clone detector in many respects, we focus on assessing which clones FACOY can effectively identify in a code clone benchmark. A clone benchmark contains pairs of code fragments which are similar to each other.

We leverage BigCloneBench [513], one of the biggest (8 million clone pairs) code clone benchmarks publicly available. This benchmark is built by labeling of pairs of code fragments from the IJaDataset-2.0 [165]. IJaDataset includes approximately 25,000 open-source Java projects consisting of 3 million source code files and 250 millions of lines of code (MLOC). BigCloneBench maintainers have mined this dataset focusing on a specific set of functionalities. They then record metadata information about the identified code clone pairs for the different functionalities. In this paper, we use a recent snapshot of BigCloneBench including clone pairs clustered in 43 functionality groups made available for the evaluation of SourcererCC [455].

We consider 8,345,104 clone pairs in BigCloneBench based on the same criteria used in [455]: both code fragments in a clone pair have at least 6 lines and 50 tokens in length, a standard minimum clone size for benchmarking [42, 514].

Clone pairs are further assigned a type based on the criteria in [455]: Type-1 and Type-2 clone pairs are classified according to the classical definitions recalled in Section 5.2. Type-3 and Type-4 clones are further divided into four sub-categories based on their syntactical similarity: Very Strongly Type 3 (VST3), Strongly Type 3 (ST3), Moderately Type 3 (MT3), and Weakly Type 3/Type 4 (WT3/4). Each clone pair (unless it is Type 1 or 2) is identified as one of four if its similarity score falls into a specific range; VST3: $[90\%, 100\%)$, ST3: $[70\%, 90\%)$, MT3: $[50\%, 70\%)$, and WT3/4: $[0\%, 50\%)$.

For this experiment, we adapt the implementation described in Section 5.4.1. Since the experiment conducted in [455] detected clones only from IJaDataset, the `GitHub`-based code index in our tool is replaced by a custom index generated from IJaDataset for a fair comparison. This makes FACOY search only code fragments in IJaDataset. In addition, the stretch parameters (see Algorithm 1) are set to $n_s = n_q = 3, n_c = 100$, making FACOY yield as many snippets, posts and fragments as possible in each step.

We feed FACOY with each code fragment referenced in the benchmark in order to search for their clones in the IJaDataset. We compare each pair, formed by an input fragment and a search result, against the clone pairs of BigCloneBench. We then compute the recall of FACOY following the definition proposed in the benchmark [513]:

$$Recall = \frac{D \cap B_{tc}}{B_{tc}} \tag{5.2}$$

where B_{tc} is the set of all true clone pairs in BigCloneBench, and D is the set of clone pairs found by FACOY.

To quantify the improvement brought by the two main strategies proposed in this work, namely *query alternation* and *query structuring*, we define four different search engine configurations:

- Baseline SE: The baseline search engine does not implement any query structuring or query alternation. Input code query, as well as the search corpus, are treated as natural language text documents. Search is then directly performed by matching tokens with no type information.
- FaCoY$_{noQA}$: In this version, only query structuring is applied. No query alternation is performed, and thus only input code query is used to match the search corpus.
- FaCoY$_{noUQ}$: In this version, query alternation is performed along with query structuring, but initial input query is left out.
- FaCoY: This version represents the full-feature version of the code-to-code search engine: queries are alternated and structured, and initial input code query also contributes in the matching process.

Result: Table 5.5 details the recall[7] scores for the baseline SE, FaCOY$_{noQA}$, FaCOY$_{noUQ}$ and FaCoY. Recall scores are summarized per clone type with the categories introduced above. Since we are reproducing for FaCoY the experiments performed in [455], we directly report in this table all results that the authors have obtained on the benchmark for state-of-the-art Nicad [101], iClones [173], SourcererCC [455], CCFinderX [239], Deckard [227] clone detectors.

Table 5.5: Recall scores on BigCloneBench [513].

	Clone Types					
	T1	T2	VST3	ST3	MT3	WT3/T4
(# of Clone Pairs)	(39,304)	(4,829)	(6,171)	(18,582)	(90,591)	(6,045,600)
FaCoY	65	90	67	69	**41**	**10** (635,844)*
FaCoY $_{noUQ}$	35	74	45	55	37	10
FaCoY $_{noQA}$	66	26	56	54	20	2
Baseline SE	66	26	54	50	20	2
SourcererCC	**100**	98	93	61	5	0
CCFinderX†	**100**	93	62	15	1	0
Deckard†	60	58	62	31	12	1
iClones†	**100**	82	82	24	0	0
NiCad†	**100**	**100**	**100**	**95**	1	0

* Cumulative number of WT3/4 clones that FaCoY found.
† The tools could not scale to the entire files of IJaDataset [165].

We recall that FaCoY is a code-to-code search engine and thus the objective is to find semantic clones (i.e., towards Type-4 clones). Nevertheless, for a comprehensive evaluation of the added value of the strategies implemented in the FaCoY approach, we provide comparison results of recall values across all clone types.

Overall, FaCoY produces the highest recall values for moderately Type-3 as well as Weakly Type-3 and Type-4 clones. The recall performance of FaCoY for MT3 clones is an order of magnitude higher than that of 4 out the 5 detection tools. While most tools detect little to no WT3/T4 code clone pairs, FaCoY is able to find over 635,000 clones in the IJADataset. Furthermore, apart from SourcererCC, the other tools could not cover the entire IJaDataset as reported in [455].

Benefit of query structuring. The difference of performance between Baseline SE and FaCOY$_{noQA}$ indicates that query structuring helps to match more code fragments which are not strictly, syntactically, identical to the query (cf. VST3 & ST3).
Benefit of query alternation. The difference of performance between FaCoY and FaCOY$_{noQA}$ definitively confirms that query alternation is the strategy that allows collecting semantic clones: recall for WT3/T4 goes from 2% to 10% and recall for MT3 goes from 20 to 41.
Benefit of keeping input query. The difference of performance between FaCoY and FaCOY$_{noUQ}$ finally indicates that initial input code query is essential for retrieving some code fragments that are more syntactically similar, in addition to semantically similar code fragments matched by alternate queries.

[7] It should be noted that Recall is actually Recall@MAX.

With 10% recall for semantic clones (WT3/T4), FACOY achieves the best performance score in the literature. Although this score may appear to be small, it should be noted that this corresponds to the identification of 635,844 clones, a larger set than the accumulated set of all clones of other types in the benchmark. Finally, it should also be noted that, while the dataset includes over 7.7 million WT3/T4 code clones, state-of-the-art detection tools can detect only 1% or less of these clones.

We further investigate the recall of FACOY with regards to the functionalities implemented by clone pairs. In BigCloneBench, every clone pair is classified into one of 43 functionalities, including "Download From Web" and "Decompress zip archive". For each clone type, we count the number of clones that FACOY can find, per functionality. Functionalities with higher recall tend to have implementations based on APIs and libraries while those with low recall are more computation intensive without APIs. This confirms that FACOY performs better for programs implemented by descriptive API names since it leverages keywords in snippets and questions. This issue is discussed in Section 5.5 in detail. Because of space constraints, we refer the reader to the FACOY project page for more statistics and information details on its performance.

Double-checking* FACOY*'s false positives: Although it is one of the largest benchmarks available to the research community, BigCloneBench clone information may not be complete. Indeed, as described in [513], BigCloneBench is built via an incremental additive process (i.e., gradually relaxing search queries) based on keyword and source pattern matching. Thus, it may miss some clones despite the manual verification. In any case, computing precision of a code search engine remains an open problem [455]. Instead, we chose to focus on manually analysing sampled false positives.

We manually verify the clone pairs that are not associated in BigCloneBench, but FACOY recommended as code clones, i.e., false positives. Our objective is then to verify to what extent they are indeed false positives and not misses by BigCloneBench. We sample 10 false positives per clone type category for a manual check. For 32 out of 60 cases, it turns out that BigCloneBench actually missed to include them. Specifically, it missed 25 Type-4, 2 Type-3, 1 Type-2, and even 4 Type-1 clones. Among the 28 cases, for 26 cases, FACOY points to the correct file but another location than actual clones. In only two cases FACOY completely fails. We provide this data in the project web page [434] as a first step towards initiating a curated benchmark of semantic code clones, which can be eventually integrated into BigCloneBench.

FACOY can find more Type-3, Weakly Type-3 and Type-4 clones than the state-of-the-art, thus fulfilling the objective for which it was designed.

5.4.4 RQ3: Validating semantic similarity

Design: Since FACOY focuses on identifying semantically similar code snippets rather than syntactic/structural clones, it is necessary to verify whether the search results of the approach indeed exhibit similar functional behavior (beyond keyword matching with high syntactic differences implied in BigCloneBench). The datasets used in Sections 5.4.2 and 5.4.3 are however not appropriate for dynamic analysis: the code must compile as well as execute, and there must be test cases for exercising the programs.

To overcome these challenges, we build on DYCLINK [501], a dynamic approach that computes the similarity of execution traces to detect that two code fragments are relatives (i.e., that they behave (functionally) similarly). The tool has been applied to programs written for *Google Code Jam* [157] to identify code relatives at the granularity of methods. We carefully reproduced their results with the publicly available version of DYCLINK. Among the 642 methods in the code base, DYCLINK matches 411 pairs as code relatives[8]. We consider all methods for which DYCLINK finds a relative

[8]We did our best to reproduce the results of DYCLINK. We checked with the authors that the found 411 relatives are consistent with the released tool version.

and use FACOY to search for its clones in *codejam*, and we check that the found clones are relatives of the input.

Since FACOY provides a ranked list of code examples for a given query, we measure the hit[9] ratio of the top N search results. Here, we use the default stretch parameters specified in Section 5.4.1 and thus $N = 27$. In addition to hit ratio, we compute the Mean Reciprocal Rank (MRR) of the hit cases. To calculate MRR for each clone pair, we use the following formula:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{5.3}$$

where $rank_i$ is the rank position of the corresponding code fragment for the given peer in a clone pair. Q is the number of all queries.

Result: As a result, FACOY can identify 278 out of 411 code relatives and the hit ratio is 68%. As for efficiency, FACOY achieves 45% and 88% scores respectively for *Precision*@10 and *Precision*@20, and exhibits an MRR of 0.18, which means FACOY recommends the code relatives into lower rankings.

On the one hand, since many programs in Google Code Jam often use variables with no meaning (such as `void s(int a){}`), FACOY cannot find related code in `StackOverflow` and thus cannot build alternate queries, limiting the hit ratio. On the other hand, since DYCLINK also uses a similarity metric to decide on code relativeness, the MRR score of FACOY could be higher with a more relaxed threshold (currently set at 82%) in DYCLINK.

FACOY can indeed find alternative fragments that exhibit similar runtime behavior with input code fragment.

5.4.5 RQ4: Recommending patches with FACOY

Design: This experiment attempts to use FACOY to search for correct code that can help fix buggy code. Code search has indeed been proposed recently as a potential step for patch recommendation [147], and even automated repair [242]. Since FACOY can find code snippets that are semantically similar to a given query code, we conjecture that it can be used for helping find alternative implementations which may turn out to be more correct than the user's code. We assess such a potential application by leveraging the Defects4J benchmark [234].

Defects4J include 395 real bugs: for each bug, the buggy code and the associated fixed version are made available, along with test suites for execution. For each bug, we take buggy code fragment (generally a function) and query FACOY. By going through the search results from the top, we manually compare each result with the actual paired fixed code. Our objective is to check whether FACOY's output code fragments can help build a patch that would have been correct w.r.t. to the benchmark fix. We perform the same experiments using `Krugle` [271] and `searchcode` [469].

For each bug, one of the authors of this paper examined at most top 15 search results from each search engine. When the author marks a result as a good candidate for patch recommendation, two other authors double check, and the final decision is made by majority voting. Note that, since Defects4J projects are also available in GitHub, the fixed code may be in FACOY corpus. Thus, we have filtered out from the search results any code fragment that is collected from the same project file as the buggy code used as query. Figure 5.8a shows an example buggy function that we used as query to FACOY. Fig. 5.8b shows one of the similar code fragments returned by FACOY and which we found that it was a good candidate for recommending the patch that was actually applied (cf. Fig. 5.8c).

[9] A "hit" indicates that the corresponding code fragment is in the top N results for a given query.

```
public static boolean equals(CharSequence cs1, CharSequence cs2)
    { return cs1 == null ? cs2 == null : cs1.equals(cs2); }
```

(a) Defects4J buggy code fragment from Commons-LANG†.

```
public static boolean equals(CharSequence a, CharSequence b) {
    if (a == b) return true;
    int length;
    if (a != null && b != null && (length = a.length()) == b.length()) {
        if (a instanceof String && b instanceof String) {
            return a.equals(b);
        }
        for (int i = 0; i < length; i++) {
            if (a.charAt(i) != b.charAt(i)) return false;
        }
        return true;
    }
    return false;
}
```

(b) Code fragment found in GitHub by FACOY as similar to fragment in (a)*.

```
 public static boolean equals(CharSequence cs1, CharSequence cs2) {
-   return cs1 == null ? cs2 == null : cs1.equals(cs2);
+   if (cs1 == cs2) {
+       return true;
+   }
+   if (cs1 == null || cs2 == null) {
+       return false;
+   }
+   if (cs1 instanceof String && cs2 instanceof String) {
+       return cs1.equals(cs2);
+   }
+   return CharSequenceUtils.regionMatches(...);
 }
```

(c) Actual patch‡ that was proposed to fix the buggy code in (a).

† https://goo.gl/5kn6Zr * https://goo.gl/URdriN ‡ https://goo.gl/PD6KL5

Figure 5.8: Successful patch recommendation by FACOY.

Result: Out of 395 bugs in Defects4J, our preliminary results show that FACOY found similar fixed code examples for 21 bugs. In contrast, `searchcode` located a single code example, while `Krugle` provided no relevant results at all. Specifically, project-specific results are as follows. Lang: 6/65, Mockito: 3/38, Chart: 3/26, Closure: 2/133, Time: 2/27, and Math: 5/106. searchcode was successful only for 1/38 Mokito bug. All details are available in [434].

FACOY-based search of semantically similar code fragments can support patch/code recommendation, software diversification or transplantation.

5.5 Discussions

Exhaustivity of Q&A data: The main limitation of FACOY comes from the use of code snippets and natural language descriptions in Q&A posts to enable the generation of alternate queries towards identifying semantically similar code fragments. This data may simply be insufficient with regards to a given user input fragment (e.g., uncommon functionality implementation).

Threats to Validity: As threat to *External validity*, we note that we only used Java subjects for the search. However, the same process can be developed with other programming languages by changing the language parser, the indices for related Q&A posts and project code. Another threat stems from the use of `StackOverflow` and `GitHub` which may be limited. We note however that their data can be substituted or augmented with data from other repositories.

Internal validity: We use subjects from BigCloneBench and DyCLINK datasets to validate our work. Those subjects may be biased for clone detection. Nevertheless, these subjects are commonly used and allow for a fair comparison as well as for reproducibility.

5.6 Related Work

Code search engines. Code search literature is abundant [15,130,219,244,335,357,398,413,461,481]. CodeHow [335] finds code snippets relevant to a user query written in natural language. It explores API documents to identify relationships between query terms and APIs. Sourcerer [15] leverages structural code information from a complete compilation unit to perform fine-grained code search. Portfolio [357] is a code search and visualization approach where a chain of function calls are highlighted as usage scenario. CodeGenie [284, 288] expands queries for interface-driven code search (IDCS). It takes test cases rather than free-form queries as inputs and leverages WordNet and a code-related thesaurus for query expansion. Sirres et al. [481], also use `StackOverflow` data to implement a free-form code search engine.

Clone detection and search. Clone detection has various applications [64, 281, 467] such as plagiarism detection. However, most techniques detect syntactically similar code fragments in source code using tokens [25,239,304], AST trees [35,227], or (program dependency) graphs [270,312]. Only a few techniques target semantically similar source code clones [228, 256, 267]. Komondoor and Horwitz search for isomorphic sub-graphs of program dependence graphs using program slicing [267]. Jiang and Su compare program execution traces using automated random testing to find functionally equivalent code fragments [228]. MeCC detects semantically-similar C functions based on the similarity of their abstract memory states [256]. White et al. [565] propose to use deep learning to find code clones. Their approach is more effective for Type-1/2/3 clones than Type-4.

Code recommendation systems [194,197,352,412] support developers with reusable code fragments from other programs, or with pointers to blogs and Q&A sites. Strathcona [197] generates queries from user code and matches them against repository examples, Prompter [412] directly matches the current code context with relevant Q&A posts. Although several studies have explored `StackOverflow` posts [33, 147, 337, 381, 481, 529], none, to the best of our knowledge, leveraged `StackOverflow` data to improve clone detection.

Program repair [47, 158, 255] can also benefit from code search. Gao et al. [147] proposed an approach to fix recurring crash bugs by searching for similar code snippets in `StackOverflow`. SearchRepair [242] infers potential patch ingredients by looking up code fragments encoded as SMT constraints. Koyuncu et al. [269] showed that patching tools yield recurrent fix actions that can be explored to fix similar code. Liu et al. [318] explore the potential of fix patterns for similar code fragments that may be buggy w.r.t. FindBugs rules.

API recommendation is a natural application of code search. The Baker approach connects existing source code snippets to API documentation [506]. MUSE [374] builds an index of real source code fragments by using static slicing and code clone detection, and then recommends API usage examples. Keivanloo et al. [249] presented an Internet-scale code search engine that locates working code examples. Buse and Weimer [69] proposed an approach to recommend API usage examples by synthesizing code snippets based on dataflow analysis and pattern abstraction. Bajracharya [20] proposed Structural Semantic Indexing which examines the API calls extracted in source code to determine code similarity.

5.7 Conclusion

We have presented FaCoY, a code-to-code search engine that accepts code fragments from users and recommends semantically similar code fragments found in a target code base. FaCoY is based on **query alternation**: after generating a structured code query summarizing structural code elements in the input fragment, we search in Q&A posts other code snippets having similar descriptions but which may present implementation variabilities. These variant implementations are then used to generate alternate code queries. We have implemented a prototype of FaCoY using `StackOverflow` and `GitHub` data on Java. FaCoY achieves better accuracy than online code-to-code search engines and finds more semantic code clones in BigCloneBench than state-of-the-art clone detectors. Dynamic analysis shows that FaCoY's similar code fragments are indeed related execution-wise. Finally, we have investigated a potential application of FaCoY for code/patch recommendation on buggy code in the Defects4J benchmark.

Availability

We make available all our data: source code of FaCoY, search indices, experimental results. See `https://github.com/FalconLK/facoy`. A prototype implementation of FaCoY search engine is live at `http://http://code-search.uni.lu/facoy`.

6 Conclusions and Future Work

6.1 Conclusions

In this dissertation, we targeted investigating and improving semantic code search. To this end, we explored all over the approaches and proposed two semantic code search approaches.

Regarding the large-scale investigation on the field of code search, our objective was to provide the overview and to further deepen the knowledge on techniques used in the field. Through a comprehensive review process of 136 code search approaches, we found that there are opportunities for code search to target at semantic level that have not yet been investigated. We also provided a taxonomy based on the comparison of existing approaches that can irradiate the future research directions. People who are newly approaching this field can leverage this study as a stepping stone to understand the overview while the practitioners can use this work as a technique identifier to select an approach according to their needs. Moreover, the researchers can figure out the further contribution and advancement points by checking out the trend and our analysis. We further discussed the open issues and future works on the code search such as lack of information for decision, standard benchmarks, semantic level approaches, etc.

For the second part, we presented a novel approach for NL-to-code search addressing the vocabulary mismatch problem. Our search engine augments free-form queries by leveraging code snippets in answers of related posts from Q&A sites. The key insight from our work is that it is possible to map human concepts expressed in queries (which are often written with similar terms by developers) with structural code entities (which are the most relevant terms for matching source code with high relevance). Our research prototype in this study leveraged StackOverflow posts to find the best mappings between developer query terms and structural code entities. We found that the approach can outperforms the state-of-the-art and it was competitive against established web search engines for solving programming tasks.

As a final part, we presented a novel approach for code-to-code (i.e., accepting code fragments as a query) search. It was based on **query alternation**: after generating a structured code query summarizing structural code elements in the input fragment, we search in Q&A posts other code snippets having similar descriptions but which may present implementation variabilities. These variant implementations are then used to generate alternate code queries. We used StackOverflow for this research prototype again and Java source code data from GitHub. Through the experimental results, we found that our approach can achieve better accuracy than online code-to-code search engines and finds more code snippets that are semantically similar comparing to the state-of-the-art. We further proved that the approach has potential for code/patch recommendation on buggy code by using a well-known defects benchmark Defects4J.

6.2 Future Work

We now summarize potential future research directions that are in line with this dissertation.

1. **Recommendation of code search.** Given a wide variety of existing approaches in the field of code search, researchers and practitioners may suffer or confused when they do not have the full knowledge on the field. For example, newbie practitioners need a code search engine that can adequately reformulate their queries by their needs. Some other need a feedback from the search engine when they have many requirements to consider. They can leverage interaction to address the requirements one by one. On the other hand, code-to-code rather than NL-to-code (which is a usual way), can be utilized to refactor the existing code by finding either syntactically or semantically similar code snippets. The deepened analysis of these task-characteristic pairs should be conducted to support the decision of approach.
2. **Investigating semantic levels.** Issues like vocabulary mismatch prevent the code search engines from retrieving good quality code snippets. These problems are naturally occurred by either essential difference between NL and source code or different writing styles of each developer. Further research tasks such as translation of tokens of the NL query to tokens of the source code, should be conducted to understand the semantics from both side.
3. **Creating and improving benchmarks.** Despite researchers and practitioners all know that a more general and trustworthy conclude for the code search will be drawn with a high quality benchmark, there is still a lack of a benchmark for internet-scale code search engines. In particular, we observed that the comparisons of existing approaches are common in evaluating code search engines. However, many of them collected their dataset and they re-implemented the target approaches and applied their dataset to compare. To this end, a few researchers proposed benchmarks for code search recently but these benchmarks also have open issues: 1) these are not validated, yet, by the research community; 2) they focus on learning-based approaches, i.e., classical approaches cannot be compared; 3) they still need concrete common metrics to test various approaches. Furthermore, other considerations such as different NL queries for a specific set of tasks should provide a similar set of results (which current benchmarks fails). Every user has a unique way of describing a problem, especially when it comes to using NL. Thus, generating a set of techniques that would consider different NL representations for the same set of questions would provide a more normalized set of results. This also requires a careful extension towards query topics that would be relevant to the search
4. **Extending programming languages in a code search.** Software development consists of using multiple different programming languages for developing one full product. Consequently, code search approaches do not always prove to be a good solution because of their limitation towards specific programming languages. The extensibility of code search approaches towards all viable programming languages is an important issue in the domain. Being able to apply a particular search approach towards different programming languages would provide more usefulness and convenience.
5. **Enhancing the consensus.** The result from a code search engine is not always what a user expected. Different approaches rarely consider different contextual requirements of the users. For example, a user looking for a code snippet that is memory efficient might not consider just a normal code snippet as an answer as it fails to satisfy the requirement of being memory-optimized.
6. **Considering more usabilities.** Binary code search has limited application only within the security domain, only when making a strong assumption about the level of obfuscation and compiler or hardware-level optimization. Any form of obfuscation or optimization makes the techniques of binary search fail. This severely limits the use of binary code search even within the security domain. An approach that considers the different levels of obfuscation and even the compiler level optimization would make it more useful for real-world scenarios.
7. **Ensuring the replicability.** Even though dissertation encompasses many approaches that support code search, most do not have publicly available replication packages. This poses an obstacle for developers who need to apply such an approach. A shared replication package helps developers in two significant ways: First being a time-efficient way to deploy and test the approach, and second, a reference implementation to check for errors and issues. Re-

implementation of an approach is a time-consuming task even if the approach is well-explained. It can derive to potential errors where the performance is different from the reported one. Therefore, sharing source code can improve efficiency in the field of code search.

List of papers, tools & services

Papers included in this dissertation:

- Kisub Kim, Sankalp Ghatpande, Dongsun Kim, Tegawendé F. Bissyandé, Jacques Klein, Kui Liu, and Yves Le Traon. Big Code Search – a Bibliography. *A Technical Report*, 2021
- Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY – A Code-to-Code Search Engine. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 946–957, 2018
- Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering (EMSE)*, 23(5):2622–2654, 2018

Papers not included in this dissertation:

- Kisub Kim, Sankalp Ghatpande, Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. DigBug – Pre/Post-processing Operator Selection for Accurate Bug Localization. *Journal of Systems and Software (JSS)*, 0(0):0–0, 2021. ***Under Submission***
- Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 615–627, 2020
- Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12. IEEE, 2019
- Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662. IEEE, 2018

Tools and Datasets:

- **CoCaBu Dataset** - `https://github.com/serval-snt-uni-lu/cocabu`
- **CoCaBu Search Engine** - `http://code-search.uni.lu/cocabu`
- **FaCoY Dataset** - `https://github.com/FalconLK/facoy`
- **FaCoY Search Engine** - `http://code-search.uni.lu/facoy`
- **Systematic Literature Review** - `https://github.com/FalconLK/Bibliography`

Services (Reviewer & External Reviewer):

- Empirical Software Engineering (EMSE) ('20)
- International Conference on Software Engineering (ICSE) ('16, '18, '20, '21)
- Automated Software Engineering (ASE) ('18, '19, '20, '21)
- International Symposium on Software Testing and Analysis (ISSTA) ('18, '19)
- Conference on Software Analysis, Evolution and Reengineering (SANER) ('18)
- International Conference on Software Maintenance and Evolution (ICSME) ('18, '19)
- Journal of Software and Systems (JSS) ('20)

Bibliography

[1] Tanveer Ahmed and Abhishek Srivastava. Understanding and evaluating the behavior of technical users. A study of developer interaction at StackOverflow. *Hum. Cent. Comput. Inf. Sci.*, 7(1):8, December 2017.

[2] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[3] Lei Ai, Zhiqiu Huang, Weiwei Li, Yu Zhou, and Yaoshen Yu. SENSORY: Leveraging code statement sequence information for code snippets recommendation. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 27–36, 2019.

[4] Shayan Akbar and Avinash Kak. SCOR: Source code retrieval with semantics and order. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 1–12, 2019.

[5] M. Akhin, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. Search by example in TouchDevelop: Code search made easy. In *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, pages 5–8, 2012.

[6] Gerald Albaum. The likert scale revisited. *Market Research Society. Journal.*, 39(2):1–21, 1997.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419, 2018.

[8] Ambient Software Evoluton Group. http://secold.org/ projects/seclone, Dec. 2020. last accessed 01.09.2020.

[9] Ambient Software Evoluton Group. https://github.com/clonebench/bigclonebench, Dec. 2020. last accessed 01.09.2020.

[10] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. Stack Overflow: A Code Laundering Platform? In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293. IEEE, February 2017.

[11] A. Arwan, S. Rochimah, and R.J. Akbar. Source code retrieval on StackOverflow using LDA. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 295–299, 2015.

[12] M. H. Asyrofi, F. Thung, D. Lo, and L. Jiang. AUSearch: Accurate API usage search in GitHub repositories with type resolution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 637–641, 2020.

[13] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 111–120, 2009.

[14] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 681–682. ACM, 2006.

[15] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 681–682. ACM, October 2006.

[16] Sushil Bajracharya, Trung Ngo, Erik Linstead, Paul Rigor, Yimeng Dou, Pierre Baldi, and Cristina Lopes. A study of ranking schemes in internet-scale code search. Technical report, Technical report, UCI Institute for Software Research, 2007.

[17] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4. IEEE Computer Society, 2009.

[18] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Searching API usage examples in code repositories with sourcerer API search. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, SUITE '10, pages 5–8. Association for Computing Machinery, 2010.

[19] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79(Supplement C):241–259, January 2014.

[20] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 157–166. Association for Computing Machinery, 2010.

[21] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 157–166. ACM, November 2010.

[22] Sushil Krishna Bajracharya. *Facilitating Internet-scale Code Retrieval*. PhD thesis, University of California Irvine, Long Beach, CA, USA, 2010. AAI3422111.

[23] Sushil Krishna Bajracharya and Cristina Videira Lopes. Analyzing and mining a code search engine usage log. *Empir Software Eng*, 17(4-5):424–466, September 2010.

[24] Sushil Krishna Bajracharya and Cristina Videira Lopes. Analyzing and Mining a Code Search Engine Usage Log. *Empirical Software Engineering (EMSE)*, 17(4-5):424–466, August 2012.

[25] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 1992.

[26] Brenda S Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 71–80, 1993.

[27] S. Baltes, R. Kiefer, and S. Diehl. Attribution Required: Stack Overflow Code Snippets in GitHub Projects. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017.

[28] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[29] E. A. Barbosa, A. Garcia, and M. Mezini. Heuristic strategies for recommendation of exception handling code. In *2012 26th Brazilian Symposium on Software Engineering*, pages 171–180, 2012.

[30] E. A. Barbosa, A. Garcia, and M. Mezini. A recommendation system for exception handling code. In *2012 5th International Workshop on Exception Handling (WEH)*, pages 52–54, 2012.

[31] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated Software Transplantation. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*, pages 257–269. ACM, July 2015.

[32] Ohad Barzilay, Christoph Treude, and Alexey Zagalsky. Facilitating Crowd Sourced Software Engineering via Stack Overflow. In *Finding Source Code on the Web for Remix and Reuse*, pages 289–308. Springer, New York, NY, 2013. DOI: 10.1007/978-1-4614-6596-6_15.

[33] Ohad Barzilay, Christoph Treude, and Alexey Zagalsky. Facilitating Crowd Sourced Software Engineering via Stack Overflow. In *Finding Source Code on the Web for Remix and Reuse*, pages 289–308. Springer, 2013.

[34] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants. In *Proceedings of the 23th International Symposium on Software Testing and Analysis (ISSTA)*, pages 149–159. ACM, July 2014.

[35] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE, March 1998.

[36] http://bazaar.canonical.com/, Dec. 2020. last accessed 01.09.2020.

[37] S. Bazrafshan, R. Koschke, and N. Gode. Approximate code search in program histories. In *2011 18th Working Conference on Reverse Engineering*, pages 109–118, 2011.

[38] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[39] Farnaz Behrang, Steven P. Reiss, and Alessandro Orso. GUIfetch: supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 236–246. Association for Computing Machinery, 2018.

[40] Jeffrey S Beis and David G Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of IEEE computer society conference on computer vision and pattern recognition*, pages 1000–1006. IEEE, 1997.

[41] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[42] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):577–591, 2007.

[43] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2018.

[44] Sumit Bhatia, Suppawong Tuarob, Prasenjit Mitra, and C. Lee Giles. An algorithm search engine for software developers. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 13–16. ACM Press, 2011.

[45] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[46] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[47] Tegawendé F Bissyandé. Harvesting Fix Hints in the History of Bugs. *arXiv:1507.05742 [cs]*, July 2015. arXiv: 1507.05742.

[48] Tegawendé F. Bissyandé, F. Thung, D. Lo, Lingxiao Jiang, and L. Reveillere. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *Proceedings of the 37th IEEE Computer Software and Applications Conference (COMPSAC)*, pages 303–312, July 2013.

[49] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, July 2013.

[50] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *Proceedings of the 37th IEEE Computer Software and Applications Conference (COMPSAC)*, pages 303–312. IEEE, July 2013.

[51] T.F. Bissyande, F. Thung, D. Lo, Lingxiao Jiang, and L. Reveillere. Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *2013 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 242–245, July 2013.

[52] T.F. Bissyande, F. Thung, D. Lo, Lingxiao Jiang, and L. Reveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 303–312, July 2013.

[53] https://bitbucket.org/, Dec. 2020. last accessed 01.09.2020.

[54] Stephen Blott and Roger Weber. A simple vector-approximation file for similarity search in high-dimensional vector spaces. *ESPRIT Technical Report TR19, ca*, 1997.

[55] Dankmar Böhning. Multinomial logistic regression algorithm. *Annals of the institute of Statistical Mathematics*, 44(1):197–200, 1992.

[56] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[57] Christian Borgelt. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 2(6):437–456, 2012.

[58] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE international conference on data mining (ICDM'05)*, pages 8–pp. IEEE, 2005.

[59] Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1):i47–i56, 2005.

[60] Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. Searching Repositories of Web Application Models. In *Web Engineering*, Lecture Notes in Computer Science, pages 1–15. Springer, Berlin, Heidelberg, July 2010.

[61] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464, 2010.

[62] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512, 2010.

[63] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1589–1598, New York, NY, USA, 2009. ACM.

[64] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-Independent Clone Detection Applied to Plagiarism Detection. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 77–86, September 2010.

[65] Jane Bromley, James W Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(04):669–688, 1993.

[66] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[67] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

[68] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

[69] Raymond P. L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE Press, June 2012.

[70] Deng Cai, Xiaofei He, and Jiawei Han. Document clustering using locality preserving indexing. *IEEE Transactions on Knowledge and Data Engineering*, 17(12):1624–1637, 2005.

[71] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 964–974. Association for Computing Machinery, 2019.

[72] Brock Angus Campbell and Christoph Treude. NLP2code: Code snippet content assist via natural language tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 628–632, 2017.

[73] Rafael Capilla, Barbara Gallina, Carlos Cetina, and John Favaro. Opportunities for software reuse in an uncertain world: From past to emerging trends. *Journal of Software: Evolution and Process*, 31(8):e2217, 2019.

[74] Claudio Carpineto, Renato de Mori, Giovanni Romano, and Brigitte Bigi. An Information-theoretic Approach to Automatic Query Expansion. *ACM Transactions on Information Systems (TOIS)*, 19(1):1–27, January 2001.

[75] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1):1:1–1:50, 2012.

[76] S Carter, RJ Frank, and DSW Tansley. Clone detection in telecommunications software systems: A neural net approach. In *Proc. Int. Workshop on Application of Neural Networks to Telecommunications*, pages 273–287, 1993.

[77] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[78] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 1–11. Association for Computing Machinery, 2012.

[79] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689, 2016.

[80] William I Chang and Eugene L Lawler. Approximate string matching in sublinear expected time. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 116–124. IEEE, 1990.

[81] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 621–630, 2009.

[82] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, Berlin, Heidelberg, March 2009.

[83] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A Search Engine for Java using Free-Form Queries. In *Fundamental Approaches to Software Engineering (FASE)*, pages 385–400. Springer, March 2009.

[84] Chi Chen, Xin Peng, Jun Sun, Zhenchang Xing, Xin Wang, Yifan Zhao, Hairui Zhang, and Wenyun Zhao. Generative API usage code recommendation with parameter concretization. *Science China Information Sciences*, 62(9):192103, 2019.

[85] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. Holistic combination of structural and textual code information for context based API recommendation. *arXiv:2010.07514 [cs]*, 2020.

[86] Hao Chen, Shi Ying, Jin Liu, and Wei Wang. Se4sc: A specific search engine for software components. In *The Fourth International Conference onComputer and Information Technology, 2004. CIT'04.*, pages 863–868. IEEE, 2004.

[87] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[88] Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 826–831, 2018.

[89] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. Explaining Software Defects Using Topic Models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 189–198. IEEE Press, June 2012.

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

[91] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, S Yu Philip, and Haixun Wang. Fast graph pattern matching. In *2008 IEEE 24th International Conference on Data Engineering*, pages 913–922. IEEE, 2008.

[92] Kenneth Ward Church. A stochastic parts program and noun phrase parser for unrestricted text. In *International Conference on Acoustics, Speech, and Signal Processing,*, pages 695–698. IEEE, 1989.

[93] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, 1(3):215–222, 1976.

[94] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A Machine Learning Approach for Tracing Regulatory Codes to Product Specific Requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 155–164, May 2010.

[95] http://www.codeproject.com/kb/miscctrl/quickgraph.aspx, Dec. 2020. last accessed 01.09.2020.

[96] https://www.codota.com/, Dec. 2020. last accessed 01.09.2020.

[97] Collin McMillan. Finding relevant functions in millions of lines of code. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1170–1172. ACM, 2011.

[98] Allan M. Collins and Elizabeth F. Loftus. A spreading-activation theory of semantic processing. *Psychological Review*, 82(6):407–428, 1975.

[99] Allan M. Collins and M. Ross Quillian. Experiments on semantic memory and language comprehension. In *Cognition in learning and memory*, pages vii, 263–vii, 263. John Wiley & Sons, 1972.

[100] Megan Conklin. Project Entity Matching across FLOSS Repositories. In *Open Source Development, Adoption and Innovation*, IFIP — The International Federation for Information Processing, pages 45–57. Springer, Boston, MA, June 2007.

[101] J. R. Cordy and C. K. Roy. The NiCad Clone Detector. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 219–220. IEEE, June 2011.

[102] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[103] Creative Commons Attribution-ShareAlike 3.0 Unported License. `https://creativecommons.org/licenses/by-sa/3.0/legalcode`, 2016. last accessed 25.02.2017.

[104] Fabio Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11(6):453–482, 1997.

[105] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57, June 2012.

[106] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an api and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, June 2012.

[107] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[108] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[109] http://darcs.net/, Dec. 2020. last accessed 01.09.2020.

[110] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[111] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 349–360. Association for Computing Machinery, 2014.

[112] Janet E. Davidson and Robert J. Sternberg, editors. *The Psychology of Problem Solving*. Cambridge University Press, 2003.

[113] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[114] Marcelo de Rezende Martins and Marco Aurélio Gerosa. CoNCRA: A convolutional neural networks code retrieval approach. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, pages 526–531. Association for Computing Machinery, 2020.

[115] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[116] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 320–330, May 2009.

[117] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. Famix 2.1—the famoos information exchange model, 2001.

[118] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[119] T. Diamantopoulos, K. Thomopoulos, and A. Symeonidis. QualBoa: Reusability-aware recommendations of source code components. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 488–491, 2016.

[120] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[121] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[122] Donzhen Wen, Liang Yang 0003, Yingying Zhang, Yuan Lin 0001, Kan Xu, and Hongfei Lin. Multi-level semantic representation model for code search. In *Proceedings of the Joint Conference of the Information Retrieval Communities in Europe (CIRCLE 2020), Samatan, Gers, France, July 6-9, 2020.* CEUR-WS.org, 2020.

[123] Horatiu Dumitru, Marek Gibiec, Negar Hariri, Jane Cleland-Huang, Bamshad Mobasher, Carlos Castro-Herrera, and Mehdi Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 181–190, 2011.

[124] Frederico A. Durão, Taciana A. Vanderlei, Eduardo S. Almeida, and Silvio R. de L. Meira. Applying a semantic layer in a source code search tool. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 1151–1157. ACM, 2008.

[125] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431. IEEE Press, 2013.

[126] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015.

[127] Françoise Détienne and Frank Bott. *Software design—cognitive aspects*. Springer-Verlag, 2001.

[128] Kai Eckert, Heiner Stuckenschmidt, and Magnus Pfeffer. Interactive Thesaurus Assessment for Automatic Document Annotation. In *Proceedings of the 4th International Conference on Knowledge Capture (K-CAP)*, pages 103–110. ACM, October 2007.

[129] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium*, pages 303–317, 2014.

[130] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering (TSE)*, 29(3):210–224, March 2003.

[131] Alan Feuer, Stefan Savev, and Javed A Aslam. Implementing and evaluating phrasal query suggestions for proximity search. *Information Systems*, 34(8):711–723, 2009.

[132] Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th international conference on Software engineering*, pages 318–328. Citeseer, 1991.

[133] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

[134] Denis Foo Kune and Yongdae Kim. Timing attacks on pin input devices. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 678–680, 2010.

[135] W B Frakes and B A Nejmeh. Software reuse through information retrieval. *ACM SIGIR Forum*, 21(1):30–36, 1986.

[136] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003.

[137] Yuji Fujiwara, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Code-to-code search based on deep neural network and code mutation. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 1–7, 2019.

[138] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, November 1987.

[139] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[140] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Communications of the ACM*, 30(11):964–971, November 1987.

[141] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[142] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156, 2010.

[143] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663. Association for Computing Machinery, 2014.

[144] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 49–52, Washington, DC, USA, 2009. IEEE Computer Society.

[145] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 49–52. IEEE Computer Society, 2009.

[146] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. Internet-Scale Code Search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*, pages 49–52, May 2009.

[147] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318, November 2015.

[148] http://ghtorrent.org/, Dec. 2020. last accessed 01.09.2020.

[149] https://developer.github.com/v3/search/, Dec. 2020.

[150] https://developer.github.com/v3/search/, Dec. 2020. last accessed 01.09.2020.

[151] https://git.kernel.org/, Mar. 2021. last accessed 28.03.2021.

[152] Sreenivas Gollapudi, Samuel Ieong, Alexandros Ntoulas, and Stelios Paparizos. Efficient Query Rewrite for Structured Web Queries. In *Proceedings of the 20th International Conference on Information and Knowledge Management (CIKM)*, pages 2417–2420. ACM, October 2011.

[153] https://code.google.com/, Dec. 2020.

[154] https://codingcompetitions.withgoogle.com/codejam, Dec. 2020.

[155] https://cs.chromium.org/, Dec. 2020.

[156] https://www.google.com, Dec. 2020.

[157] Google Code Jam. https://code.google.com/codejam/, Jan. 2017.

[158] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72, January 2012.

[159] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: GitHub Data on Demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, New York, NY, USA, 2014. ACM.

[160] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[161] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 475–484, 2010.

[162] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, volume 1, pages 475–484, May 2010.

[163] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. Exemplar: EXEcutable exaMPLes ARchive. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 259–262, New York, NY, USA, 2010. ACM.

[164] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 475–484, 2010.

[165] Ambient Software Evoluton Group. http://secold.org/ projects/seclone, Dec. 2020. last accessed 01.09.2020.

[166] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. CRaDLe: Deep code retrieval based on semantic dependency learning. *arXiv:2012.01028 [cs]*, 2020.

[167] X. Gu, H. Zhang, and S. Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944, 2018.

[168] X. Gu, H. Zhang, and S. Kim. CodeKernel: A graph kernel based approach to the selection of API usage examples. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–601, 2019.

[169] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[170] Xiaodong Gu, HongYu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 631–642, November 2016.

[171] Florian S. Gysin. Improved Social Trustability of Code Search Results. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 513–514, New York, NY, USA, 2010. ACM.

[172] Florian S. Gysin and Adrian Kuhn. A Trustability Metric for Code Search Based on Developer Karma. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, SUITE '10, pages 41–44, New York, NY, USA, 2010. ACM.

[173] N. Göde and R. Koschke. Incremental Clone Detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 219–228, March 2009.

[174] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 842–851. IEEE Press, 2013.

[175] Sonia Haiduc, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. Query Quality Prediction and Reformulation for Source Code Search: The Refoqus Tool. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 1307–1310. IEEE Press, May 2013.

[176] Sonia Haiduc, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. Query quality prediction and reformulation for source code search: The refoqus tool. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1307–1310. IEEE Press, 2013.

[177] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. A multi-perspective architecture for semantic code search. *arXiv:2005.06980 [cs]*, 2020.

[178] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[179] http://www.harukizaemon.com/simian/, Dec. 2020.

[180] Taher H Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE transactions on knowledge and data engineering*, 15(4):784–796, 2003.

[181] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[182] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5), September 1994.

[183] Geert Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *arXiv:2008.12193 [cs]*, 2020.

[184] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration*. University of Delaware, 2010.

[185] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242. IEEE Computer Society, 2009.

[186] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 524–527. IEEE Computer Society, 2011.

[187] Emily Hill, Lori Pollock, and K Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 524–527. IEEE, 2011.

[188] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. Nl-based Query Refinement and Contextualized Code Search Results: A User Study. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 34–43, February 2014.

[189] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. NL-based query refinement and contextualized code search results: A user study. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 34–43, 2014.

[190] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. Nl-based query refinement and contextualized code search results: A user study. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 34–43. IEEE, 2014.

[191] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[192] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[193] Raphael Hoffmann, James Fogarty, and Daniel S Weld. Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers. In *Proceedings of the 20th ACM Symposium on User Interface Software and Technology (UIST)*, pages 13–22. ACM, October 2007.

[194] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006. Conference Name: IEEE Transactions on Software Engineering.

[195] Reid Holmes. Do developers search for source code examples using multiple facts? In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 13–16. IEEE Computer Society, 2009,.

[196] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125. ACM, 2005.

[197] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 117–125. ACM, May 2005.

[198] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240. ACM, 2005.

[199] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[200] James Howison, Megan Squire, and Kevin Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1:17–26, September 2008.

[201] Sheng-Kuei Hsu and Shi-Jen Lin. A block-structured model for source code retrieval. In Ngoc Thanh Nguyen, Chong-Gun Kim, and Adam Janiak, editors, *Intelligent Information and Database Systems*, Lecture Notes in Computer Science, pages 161–170. Springer, 2011.

[202] Sheng-Kuei Hsu and Shi-Jen Lin. A Block-Structured Model for Source Code Retrieval. In *Intelligent Information and Database Systems*, Lecture Notes in Computer Science, pages 161–170. Springer, Berlin, Heidelberg, April 2011.

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

[204] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, and Mengting Yuan. Neural joint attention code search over structure embeddings for software q&a sites. *Journal of Systems and Software*, 170:110773, 2020.

[205] Q. Huang, A. Qiu, M. Zhong, and Y. Wang. A code-description representation learning model based on attention. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 447–455, 2020.

[206] Qing Huang, Xudong Wang, Yangrui Yang, Hongyan Wan, Rui Wang, and Guoqing Wu. SnippetGen:enhancing the code search via intent predicting. In *SEKE*, pages 307–312, 2017.

[207] Qing Huang and Guoqing Wu. Enhance code search via reformulating queries with evolving contexts. *Automated Software Engineering*, 26(4):705–732, 2019.

[208] Qing Huang and Huaiguang Wu. QE-integrating framework based on github knowledge and SVM ranking. *Science China Information Sciences*, 62(5):52102, 2019.

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

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

[211] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng. Recommending differentiated code to support smart contract update. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 260–270, 2019.

[212] Zhiheng Huang, Geoffrey Zweig, Michael Levit, Benoit Dumoulin, Barlas Oguz, and Shawn Chang. Accelerating recurrent neural network training via two stage classes and parallelization. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 326–331. IEEE, 2013.

[213] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[214] Hamel Husain. Towards natural language semantic code search, sep 2018. *URL https://github.blog/2018-09-18-towards-natural-language-semantic-code-search/*, 2018.

[215] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv:1909.09436 [cs, stat]*, 2020.

[216] Makoto Ichii, Makoto Matsushita, and Katsuro Inoue. An exploration of power-law in use-relation of java software systems. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 422–431, 2008.

[217] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 14–24, 2003.

[218] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[219] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 331–341. IEEE Press, June 2012.

[220] M. M. Islam and R. Iqbal. SoCeR: A new source code recommendation technique for code reuse. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1552–1557, 2020.

[221] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, January 2016.

[222] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[223] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[224] Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson. Lowering the barrier to reuse through test-driven search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 21–24. IEEE Computer Society, 2009.

[225] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*, 12(1):34–46, 2016. Conference Name: IEEE Transactions on Services Computing.

[226] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[227] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105. IEEE Computer Society, May 2007.

[228] Lingxiao Jiang and Zhendong Su. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 81–92. ACM, July 2009.

[229] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 81–92. Association for Computing Machinery, 2009.

[230] Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei, Minxue Pan, and Tian Zhang. Semantics-based code search using input/output examples. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 92–102, 2018.

[231] Huan Jin and Lei Xiong. A query expansion method based on evolving source code. *Wuhan University Journal of Natural Sciences*, 24(5):391–399, 2019.

[232] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[233] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Code Similarities Beyond Copy & Paste. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 78–87. IEEE, March 2010.

[234] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, July 2014.

[235] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, New York, NY, USA, 2014. ACM.

[236] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101. ACM, May 2014.

[237] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101. ACM, May 2014.

[238] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654 – 670, July 2002.

[239] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[240] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[241] Vineeth Kashyap, David Bingham Brown, Ben Liblit, David Melski, and Thomas Reps. Source forager: A search engine for similar source code. *arXiv:1706.02769 [cs]*, 2017.

[242] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing Programs with Semantic Code Search (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, November 2015.

[243] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[244] I. Keivanloo, J. Rilling, and P. Charland. Seclone - A Hybrid Approach to Internet-Scale Real-Time Code Clone Search. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 223–224, June 2011.

[245] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling. SE-CodeSearch: A scalable Semantic Web-based source code search infrastructure. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5, September 2010.

[246] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling. SE-CodeSearch: A scalable semantic web-based source code search infrastructure. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5, 2010.

[247] Iman Keivanloo, Juergen Rilling, and Philippe Charland. SeClone - a hybrid approach to internet-scale real-time code clone search. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 223–224. IEEE Computer Society, 2011.

[248] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 664–675. ACM, 2014.

[249] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 664–675. ACM, May 2014.

[250] Iman Keivanloo, Laleh Roostapour, Philipp Schugerl, and Juergen Rilling. Semantic web-based source code search. In *Proc. 6th Intl. Workshop on Semantic Web Enabled Software Engineering*, 2010.

[251] Mik Kersten and Gail C Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005.

[252] Marcus Kessel and Colin Atkinson. Ranking software components for reuse based on non-functional properties. *Information Systems Frontiers*, 18(5):825–853, 2016.

[253] Marcus Kessel and Colin Atkinson. Integrating reuse into the rapid, continuous software engineering cycle through test-driven search. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 8–11, 2018.

[254] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338, 2013.

[255] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811, May 2013.

[256] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory Comparison-based Clone Detector. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 301–310. IEEE, May 2011.

[257] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 301–310, 2011.

[258] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an Intelligent Code Search Engine. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, July 2010.

[259] Kisub Kim, Sankalp Ghatpande, Dongsun Kim, Tegawendé F. Bissyandé, Jacques Klein, Kui Liu, and Yves Le Traon. Big Code Search – a Bibliography. *A Technical Report*, 2021.

[260] Kisub Kim, Sankalp Ghatpande, Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. DigBug – Pre/Post-processing Operator Selection for Accurate Bug Localization. *Journal of Systems and Software (JSS)*, 0(0):0–0, 2021. ***Under Submission***.

[261] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY – A Code-to-Code Search Engine. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 946–957, 2018.

[262] Suntae Kim and Dongsun Kim. Automatic Identifier Inconsistency Detection Using Code Dictionary. *Empirical Software Engineering (EMSE)*, 21(2):565–604, April 2016.

[263] Suntae Kim and Dongsun Kim. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering*, 21(2):565–604, 2016.

[264] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[265] George J Klir and Bo Yuan. Fuzzy sets and fuzzy logic: theory and applications. *Possibility Theory versus Probab. Theory*, 32(2):207–208, 1996.

[266] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering (TSE)*, 32(12):971–987, December 2006.

[267] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*, pages 40–56. Springer-Verlag, July 2001.

[268] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[269] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of Tool Support in Patch Construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–248. ACM, July 2017.

[270] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 301–309, October 2001.

[271] http://krugle.com/, Dec. 2020. last accessed 01.09.2020.

[272] Ken Krugler. Krugle code search architecture. In Susan Elliott Sim and Rosalva E. Gallardo-Valencia, editors, *Finding Source Code on the Web for Remix and Reuse*, pages 103–120. Springer, 2013.

[273] D. E. Krutz and E. Shihab. CCCD: Concolic code clone detection. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 489–490, October 2013.

[274] Daniel E. Krutz and Emad Shihab. CCCD: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 489–490, 2013.

[275] Alexander Lampropoulos, Apostolos Ampatzoglou, Stamatia Bibi, Alexander Chatzigeorgiou, and Ioannis Stamelos. React-a process for improving open-source software reuse. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 251–254. IEEE, 2018.

[276] Frederick Wilfrid Lancaster and Emily Gallup Fayen. *Information Retrieval: On-line*. Melville Publishing Company, January 1973.

[277] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 467–476, 2009.

[278] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 476–482. ACM, 2009.

[279] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294–306, 2011.

[280] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. CodeGenie: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 917–918. Association for Computing Machinery, 2007.

[281] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Instant Code Clone Search. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE)*, pages 167–176. ACM, November 2010.

[282] Shin-Jie Lee, Xavier Lin, Wu-Chen Su, and Hsi-Min Chen. A comment-driven approach to api usage patterns discovery and search. *Journal of Internet Technology*, 19(5):1587–1601, 2018.

[283] Wee Sun Lee and Bing Liu. Learning with positive and unlabeled examples using weighted logistic regression. In *ICML*, volume 3, pages 448–455, 2003.

[284] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes. Can the use of types and query expansion help improve large-scale code search? In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 41–50, September 2015.

[285] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 212–221. ACM, 2014.

[286] Otavio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Gustavo Konishi, Joel Ossher, Sushil Bajracharya, and Cristina Lopes. Using thesaurus-based tag clouds to improve test-driven code search. In *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*, pages 99–108, 2013.

[287] Otávio A. L. Lemos, Adriano C. de Paula, Felipe C. Zanichelli, and Cristina V. Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 212–221. ACM, 2014.

[288] Otávio A. L. Lemos, Adriano C. de Paula, Felipe C. Zanichelli, and Cristina V. Lopes. Thesaurus-based Automatic Query Expansion for Interface-driven Code Search. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 212–221. ACM, May 2014.

[289] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. CodeGenie: Using test-cases to search and reuse source code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 525–526. ACM, 2007.

[290] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[291] Hang Li. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems*, 94(10):1854–1862, 2011.

[292] Hongyu Li, Seohyun Kim, and Satish Chandra. Neural code search evaluation dataset. *arXiv:1908.09804 [cs]*, 2019.

[293] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

[294] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[295] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[296] R. Li, G. Hu, and M. Peng. Hierarchical embedding for code search in software q a sites. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2020.

[297] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pages 501–510. ACM, May 2016.

[298] W. Li, H. Qin, S. Yan, B. Shen, and Y. Chen. Learning code-query interaction for enhancing code searches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126, 2020.

[299] Wei Li, Shuhan Yan, Beijun Shen, and Yuting Chen. Reinforcement learning of code search sessions. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 458–465, 2019.

[300] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 690–701. Association for Computing Machinery, 2016.

[301] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware Code Search for JavaScript Frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 690–701. ACM, 2016.

[302] Yang Li, Suhang Wang, Quan Pan, Haiyun Peng, Tao Yang, and Erik Cambria. Learning binary codes with neural collaborative filtering for efficient recommendation systems. *Knowledge-Based Systems*, 172:64–75, 2019.

[303] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[304] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*. USENIX Association, December 2004.

[305] Zhixing Li, Tao Wang, Yang Zhang, Yun Zhan, and Gang Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*, Internetware '16, pages 36–44. Association for Computing Machinery, 2016.

[306] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 520–531, 2015.

[307] Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun, and Wenyun Zhao. Detecting differences across multiple instances of code clones. In *Proceedings of the 36th International Conference on Software Engineering*, pages 164–174, 2014.

[308] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 48–59, 2020.

[309] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. Deep graph matching and searching for semantic code retrieval. *arXiv:2010.12908 [cs]*, 2020.

[310] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2008.

[311] https://git-scm.com/, Dec. 2020. last accessed 01.09.2020.

[312] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 872–881. ACM, August 2006.

[313] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. Opportunities and Challenges in Code Search Tools. *arXiv:2011.02297 [cs]*, November 2020. arXiv: 2011.02297.

[314] Chao Liu, Xin Xia, David Lo, Zhiwei Liu, Ahmed E. Hassan, and Shanping Li. Simplifying deep-learning-based model for code search. *arXiv:2005.14373 [cs]*, 2020.

[315] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 234–243. Association for Computing Machinery, 2007.

[316] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. Neural query expansion for code search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 29–37. Association for Computing Machinery, 2019.

[317] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12. IEEE, 2019.

[318] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *arXiv preprint arXiv:1712.03201*, 2017.

[319] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662. IEEE, 2018.

[320] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 615–627, 2020.

[321] Li-Min Liu, Michael Halper, James Geller, and Yehoshua Perl. Controlled Vocabularies in OODBs: Modeling Issues and Implementation. *Distributed and Parallel Databases*, 7(1):37–65, January 1999.

[322] Wenjian Liu, Xin Peng, Zhenchang Xing, Junyi Li, Bing Xie, and Wenyun Zhao. Supporting exploratory code search with differencing and visualization. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 300–310, 2018.

[323] Angela Lozano, Andy Kellens, and Kim Mens. Mendel: Source code recommendation based on a genetic metaphor. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 384–387. IEEE Computer Society, 2011.

[324] Angela Lozano, Andy Kellens, and Kim Mens. Mendel: Source Code Recommendation Based on a Genetic Metaphor. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 384–387. IEEE Computer Society, November 2011.

[325] Jinting Lu, Ying Wei, Xiaobing Sun, Bin Li, Wanzhi Wen, and Cheng Zhou. Interactive query reformulation for source-code search with word relations. *IEEE Access*, 6:75660–75668, 2018. Conference Name: IEEE Access.

[326] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via WordNet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549, 2015.

[327] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 545–549. IEEE, 2015.

[328] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3:152:1–152:28, 2019.

[329] George F Luger, Peder Johnson, Carl Stern, Jean E Newman, and Ronald Yeo. *Cognitive science: The science of intelligent systems.* Academic Press, 1994.

[330] http://www.luigidragone.com/software/spectral-clusterer-for-weka/, Dec. 2020.

[331] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *2008 15th Working Conference on Reverse Engineering*, pages 155–164, October 2008.

[332] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[333] Fei Lv, HongYu Zhang, Jian guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270, November 2015.

[334] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective Code Search based on API Understanding and Extended Boolean Model. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 260–270, 2015.

[335] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective Code Search based on API Understanding and Extended Boolean Model. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE Computer Society, November 2015.

[336] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira. A source code recommender system to support newcomers. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 19–24, 2012.

[337] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design Lessons from the Fastest Q&A Site in the West. In *Proceedings of the SIG Conference on Human Factors in Computing Systems (CHI)*, pages 2857–2866. ACM, May 2011.

[338] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, June 2005.

[339] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61. ACM, 2005.

[340] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[341] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008.

[342] L. W. Mar, Y. Wu, and H. C. Jiau. Recommending proper API code examples for documentation purpose. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 331–338, 2011.

[343] Gary Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[344] A. Marcus and J. I. Maletic. Identification of High-level Concept Clones in Source Code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, pages 107–114, November 2001.

[345] L. Martie and A. van der Hoek. Toward social-technical code search. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 101–104, May 2013.

[346] L. Martie, T. D. LaToza, and A. v d Hoek. CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35, November 2015.

[347] Lee Martie, André van der Hoek, and Thomas Kwak. Understanding the Impact of Support for Iteration on Code Search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 774–785, New York, NY, USA, 2017. ACM.

[348] Lee Martie, Thomas D LaToza, and André van der Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 24–35. IEEE, 2015.

[349] Lee Martie and André van der Hoek. Sameness: An Experiment in Code Search. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 76–87. IEEE Press, 2015.

[350] https://brew.sh/, Dec. 2020. last accessed 01.09.2020.

[351] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[352] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering (TSE)*, 38(5):1069–1087, September 2012.

[353] C. McMillan, M. Grechanik, D. Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[354] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 848–858, 2012.

[355] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering (TSE)*, 38(5):1069–1087, September 2012.

[356] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 111–120. ACM, 2011.

[357] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding Relevant Functions and Their Usage. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pages 111–120. ACM, May 2011.

[358] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via API call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25. Association for Computing Machinery, 2010.

[359] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, 2013.

[360] https://www.mercurial-scm.org/, Dec. 2020. last accessed 01.09.2020.

[361] https://merobase.com/, Dec. 2020.

[362] Donald Metzler and W Bruce Croft. A markov random field model for term dependencies. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 472–479, 2005.

[363] https://msdn.microsoft.com/en-us/library/ms123401.aspx, Dec. 2020. last accessed 01.09.2020.

[364] https://www.bing.com/, Dec. 2020.

[365] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[366] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[367] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[368] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016. ACM, 2012.

[369] Bhaskar Mitra and Nick Craswell. Neural models for information retrieval. *arXiv preprint arXiv:1705.01509*, 2017.

[370] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[371] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, May 2009.

[372] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[373] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890. IEEE Press, 2015.

[374] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method? In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 880–890. IEEE Press, May 2015.

[375] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994.

[376] Ibrahim Jameel Mujhid, Joanna C. S. Santos, Raghuram Gopalakrishnan, and Mehdi Mirakhorli. A search engine for finding and reusing architecturally significant code. *Journal of Systems and Software*, 130:81–93, August 2017.

[377] Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. Searching a database of source codes using contextualized code search. *arXiv:2001.03277 [cs]*, 2020.

[378] Naoya Murakami and Hidehiko Masuhara. Optimizing a search-based code recommendation system. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 68–72. IEEE Press, 2012.

[379] Naoya Murakami, Hidehiko Masuhara, and Tomoyuki Aotani. Code recommendation based on a degree-of-interest model. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*, RSSE 2014, pages 28–29. Association for Computing Machinery, 2014.

[380] Takuma Murakami, Zhenjiang Hu, Shingo Nishioka, Akihiko Takano, and Masato Takeichi. An algebraic interface for geta search engine. In *Proceedings of Program and Programming Language Workshop, Japan*. Citeseer, 2004.

[381] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, Trento, Italy 2012.

[382] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[383] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[384] https://livegrep.com/search/linux, Dec. 2020.

[385] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522. Association for Computing Machinery, 2016.

[386] Anh Tuan Nguyen, Tung Thanh Nguyen, J. Al-Kofahi, Hung Viet Nguyen, and T.N. Nguyen. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–272. IEEE, November 2011.

[387] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 263–272. IEEE, 2011.

[388] T. Nguyen, P. Vu, and T. Nguyen. Personalized code recommendation. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 313–317, 2019.

[389] T. Nguyen, P. Vu, and T. Nguyen. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 390–393, 2019.

[390] T. Van Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen. Combining word2vec with revised vector space model for better code retrieval. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 183–185, 2017.

[391] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Recommending api usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 795–800. IEEE, 2015.

[392] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. Code search on bytecode for mobile app development. In *Proceedings of the 2019 ACM Southeast Conference*, ACM SE '19, pages 253–256. Association for Computing Machinery, 2019.

[393] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. Recommendation of exception handling code in mobile app development. *arXiv:1908.06567 [cs]*, 2019.

[394] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 383–392, 2009.

[395] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query Expansion Based on Crowd Knowledge for Code Search. *IEEE Transactions on Services Computing*, 9:771–783, September 2016.

[396] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, Sept 2016.

[397] H. Niu. Improving code search using learning-to-rank and query reformulation techniques. Master's thesis, Queen's University (Canada), 2015.

[398] Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to Rank Code Examples for Code Search Engines. *Empirical Software Engineering (EMSE)*, 22(1):259–291, February 2017.

[399] https://www.openhub.net/, Dec. 2020.

[400] https://docs.oracle.com/javase/7/docs/api/, Dec. 2020. last accessed 01.09.2020.

[401] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83:55–75, 2017.

[402] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *Acm sigops operating systems review*, 42(4):247–260, 2008.

[403] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[404] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[405] P. Pathak, M. Gordon, and Weiguo Fan. Effective information retrieval using genetic algorithms based matching functions adaptation. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 8 pp. vol.1–, 2000.

[406] Praveen Pathak, Michael Gordon, and Weiguo Fan. Effective Information Retrieval Using Genetic Algorithms based Matching Functions Adaptation. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS)*, pages 8–pp. IEEE, January 2000.

[407] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Icml*, volume 1, pages 727–734, 2000.

[408] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[409] Raphael Pham, Yauheni Stoliar, and Kurt Schneider. Automatically recommending test code examples to inexperienced developers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 890–893. Association for Computing Machinery, 2015.

[410] Nina Phan, Peter Bailey, and Ross Wilkinson. Understanding the relationship of information need specificity to search query length. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 709–710. Association for Computing Machinery, 2007.

[411] Iasonas Polakis, Georgios Kontaxis, Spiros Antonatos, Eleni Gessiou, Thanasis Petsas, and Evangelos P Markatos. Using social networks to harvest email addresses. In *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*, pages 11–20, 2010.

[412] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining Stackoverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 102–111. ACM, May 2014.

[413] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering (TSE)*, 33(6):420–432, June 2007.

[414] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[415] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1066–1082. Association for Computing Machinery, 2020.

[416] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. Evaluating the evaluations of code recommender systems: A reality check. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 111–121, 2016.

[417] Lawrence Rabiner and BiingHwang Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

[418] Luc De Raedt. *Logical and Relational Learning*. Cognitive Technologies. Springer-Verlag, 2008.

[419] M. Raghothaman, Y. Wei, and Y. Hamadi. SWIM: Synthesizing what i mean - code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 357–367, 2016.

[420] C. Ragkhitwetsagul. Measuring Code Similarity in Large-Scaled Code Corpora. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 626–630, October 2016.

[421] M. M. Rahman and C. K. Roy. On the Use of Context in Recommending Exception Handling Code Examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 285–294, September 2014.

[422] M. M. Rahman, C. K. Roy, and D. Lo. RACK: Code search in the IDE using crowdsourced knowledge. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 51–54, 2017.

[423] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. Automatic query reformulation for code search using crowdsourced knowledge. *Empirical Software Engineering*, 24(4):1869–1924, 2019.

[424] Mohammad Masudur Rahman and Chanchal Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 473–484, 2018.

[425] Mohammad Masudur Rahman and Chanchal Roy. NLP2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 714–714, 2018.

[426] Karthik Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, 8:7, February 2013.

[427] S. P. Reiss. Semantics-based code search demonstration proposal. In *2009 IEEE International Conference on Software Maintenance*, pages 385–386, 2009.

[428] S. P. Reiss. Specifying what to search for. In *Tools and Evaluation 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure*, pages 41–44, 2009.

[429] S. P. Reiss. Integrating s6 code search and code bubbles. In *2013 3rd International Workshop on Developing Tools as Plug-Ins (TOPI)*, pages 25–30, 2013.

[430] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253. IEEE Computer Society, 2009.

[431] Steven P. Reiss, Yun Miao, and Qi Xin. Seeking the user interface. *Automated Software Engineering*, 25(1):157–193, 2018.

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

[433] Steffen Rendle. Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000. IEEE, 2010.

[434] TruX research group. https://github.com/facoy/facoy, 2017.

[435] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988.

[436] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[437] Stephen E Robertson. On term selection for query expansion. *Journal of documentation*, 1990.

[438] Joseph Rocchio. Relevance feedback in information retrieval. *The Smart retrieval system-experiments in automatic document processing*, pages 313–323, 1971.

[439] Joseph John Rocchio. The smart retrieval system: Experiments in automatic document processing. *Relevance feedback in information retrieval*, pages 313–323, 1971.

[440] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails. CONQUER: A tool for NL-based query refinement and contextualizing code search results. In *2013 IEEE International Conference on Software Maintenance*, pages 512–515, 2013.

[441] Manuel Roldan-vega, Greg Mallet, Emily Hill, and Jerry Alan Fails. Conquer: A tool for nl-based query refinement and contextualizing source code search results. In *in Proc. 29th IEEE Int'l Conf. on Soft. Maintenance*. Citeseer, 2013.

[442] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[443] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[444] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.

[445] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[446] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[447] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[448] Ian Ruthven. Re-examining the Potential Effectiveness of Interactive Query Expansion. In *Proceedings of the 26th International Conference on Research and Development in Informaion Retrieval (SIGIR)*, pages 213–220, July 2003.

[449] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 31–41. Association for Computing Machinery, 2018.

[450] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, 2015. ACM.

[451] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 191–201. ACM, 2015.

[452] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 65–71. Association for Computing Machinery, 2006.

[453] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 413–430. ACM, 2006.

[454] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[455] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling Code Clone Detection to Big Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. ACM, May 2016.

[456] G. Salton, A. Wong, and C. S. Yang. A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

[457] Gerard Salton, Edward A Fox, and Harry Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, 1983.

[458] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[459] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[460] Huascar Sanchez. SNIPR: Complementing code search with code retargeting capabilities. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1423–1426. IEEE Press, 2013.

[461] Huascar Sanchez. SNIPR: Complementing Code Search with Code Retargeting Capabilities. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 1423–1426. IEEE Press, May 2013.

[462] A. Satter and K. Sakib. A search log mining based query expansion technique to improve effectiveness in code search. In *2016 19th International Conference on Computer and Information Technology (ICCIT)*, pages 586–591, 2016.

[463] Abdus Satter, M.G. Muntaqeem, Nadia Nahar, and Kazi Sakib. Retrieving self-executable and functionally correct code to improve source code search. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 749–750, 2017.

[464] Abdus Satter and Kazi Sakib. A similarity-based method retrieval technique to improve effectiveness in code search. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 1–3. Association for Computing Machinery, 2017.

[465] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[466] Max Eric Henry Schumacher, Kim Tuyen Le, and Artur Andrzejak. Improving code recommendations by combining neural and classical machine learning approaches. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 476–482. Association for Computing Machinery, 2020.

[467] Niko Schwarz, Mircea Lungu, and Romain Robbes. On How Often Code is Cloned Across Repositories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1289–1292. IEEE Press, June 2012.

[468] http://lucene.apache.org/solr/, Dec. 2020. last accessed 01.09.2020.

[469] https://searchcode.com/, July. 2017.

[470] Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered neurons: Integrating tree structures into recurrent neural networks. *arXiv preprint arXiv:1810.09536*, 2018.

[471] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

[472] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, pages 196–207. Association for Computing Machinery, 2020.

[473] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[474] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: a survey of software developers and maintainers. In *, 6th International Workshop on Program Comprehension, 1998. IWPC '98. Proceedings*, pages 180–187, June 1998.

[475] Susan Elliott Sim, Megha Agarwala, and Medha Umarji. A Controlled Experiment on the Process Used by Developers During Internet-Scale Code Search. In *Finding Source Code on the Web for Remix and Reuse*, pages 53–77. Springer, New York, NY, 2013.

[476] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How Well Do Search Engines Support Code Retrieval on the Web? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):4:1–4:25, December 2011.

[477] Herbert A. Simon and Allen Newell. Human problem solving: The state of the theory in 1970. *American Psychologist*, 26(2):145–159, 1971.

[478] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 905–908. ACM, 2006.

[479] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 432–441. IEEE, 1999.

[480] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. Le Traon. [journal first] augmenting and structuring user queries to support efficient free-form code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 945–945, 2018.

[481] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search. *Empirical Software Engineering (EMSE)*, page (to appear), 2018.

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

[483] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 945. Association for Computing Machinery, 2018.

[484] Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 309–318, 2013.

[485] Bunyamin Sisman and Avinash C Kak. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 309–318. IEEE Press, 2013.

[486] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 292–303. IEEE Press, 2019.

[487] https://www.softpedia.com/, Dec. 2020. last accessed 28.12.2020.

[488] https://sourceforge.net/, Dec. 2020. last accessed 01.09.2020.

[489] https://about.sourcegraph.com/, Dec. 2020.

[490] Jake Spurlock. *Bootstrap: Responsive Web Development.* " O'Reilly Media, Inc.", 2013.

[491] http://data.stackexchange.com/stackoverflow/query/new, Dec. 2020. last accessed 01.09.2020.

[492] http://stackoverflow.com, Dec. 2020. last accessed 01.09.2020.

[493] Jamie Starke, Chris Luce, and Jonathan Sillito. Working with search results. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 53–56. IEEE Computer Society, 2009.

[494] Kathryn Stolee and Sebastian Elbaum. Solving the Search for Suitable Code: An Initial Implementation. *CSE Technical reports*, June 2012.

[495] Kathryn T. Stolee. Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1571–1574. IEEE Press, 2012.

[496] Kathryn T. Stolee and Sebastian Elbaum. Toward semantic search via SMT solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 25:1–25:4. ACM, 2012.

[497] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the Search for Source Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26:1–26:45, May 2014.

[498] Kathryn T. Stolee, Sebastian Elbaum, and Matthew B. Dwyer. Code Search with Input/Output Queries. *Journal of System Software*, 116:35–48, June 2016.

[499] J. Stylos and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing, (VL/HCC)*, pages 195–202, 2006.

[500] J. Stylos and B.A. Myers. Mica: A web-search tool for finding API components and examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 195–202, 2006.

[501] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 702–714. ACM, November 2016.

[502] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 702–714. Association for Computing Machinery, 2016.

[503] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009, 2009.

[504] Siddharth Subramanian and Reid Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 85–88. IEEE Press, 2013.

[505] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[506] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 643–652. ACM, May 2014.

[507] Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. Pscs: A path-based neural model for semantic code search. *arXiv preprint arXiv:2008.03042*, 2020.

[508] S. Surisetty. Behavior-based code search. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 197–198, 2014.

[509] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27:3104–3112, 2014.

[510] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[511] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, September 2014.

[512] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, September 2015.

[513] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards A Big Data Curated Benchmark of Inter-Project Code Clones. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–480. IEEE, September 2014.

[514] Jeffrey Svajlenko and Chanchal K Roy. Evaluating Clone Detection Tools with Bigclonebench. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140. IEEE, September 2015.

[515] Peter P. Swire. A Theory of Disclosure for Security and Competitive Reasons: Open Source, Proprietary Software, and Government Systems. *Hous. L. Rev.*, 42:1333, 2005.

[516] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[517] Akihiko Takano, Yoshiki Niwa, Shingo Nishioka, Makoto Iwayama, Toru Hisamitsu, Osamu Imaichi, and Hirofumi Sakurai. Information access based on associative calculation. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 187–201. Springer, 2000.

[518] Watanabe Takuya and Hidehiko Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 17–20. ACM, 2011.

[519] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

[520] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[521] Tim Teitelbaum. Codesurfer. *ACM SIGSOFT Software Engineering Notes*, 25(1):99, 2000.

[522] http://svn.apache.org/repos/asf/httpd/httpd/, Mar. 2021. last accessed 28.03.2021.

[523] Suresh Thummalapenta and Tao Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213. ACM, 2007.

[524] Suresh Thummalapenta and Tao Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213. ACM, November 2007.

[525] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented Unit-test Generation via Mining Source Code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202, New York, NY, USA, 2009. ACM.

[526] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. Network structure of social coding in github. In *Software maintenance and reengineering (csmr), 2013 17th european conference on*, pages 323–326. IEEE, 2013.

[527] Sander Tichelaar. Famix java language plug-in 1.0. *Technical Report*, 1999.

[528] C. Treude and M. P. Robillard. Understanding Stack Overflow Code Fragments. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 509–513, September 2017.

[529] Christoph Treude and Martin P Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 392–403. ACM, May 2016.

[530] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering*, 41(6):565–581, 2015.

[531] Christoph Treude, Mathieu Sicard, Marc Klocke, and Martin Robillard. TaskNav: Task-based navigation of software documentation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 649–652, 2015.

[532] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[533] http://www.unicorn-engine.org/, Dec. 2020. last accessed 28.12.2020.

[534] Gabriel Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2002.

[535] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[536] Venkatesh Vinayakarao. Spotting familiar code snippet structures for program comprehension. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1054–1056. ACM, 2015. event-place: Bergamo, Italy.

[537] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. ANNE: Improving Source Code Search Using Entity Retrieval Approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 211–220. ACM, 2017.

[538] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. ANNE: Improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 211–220. Association for Computing Machinery, 2017.

[539] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28:2692–2700, 2015.

[540] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.

[541] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 13–25, 2019.

[542] Hao Wang, Jia Zhang, Yingce Xia, Jiang Bian, Chao Zhang, and Tie-Yan Liu. COSEA: Convolutional code search with layer-wise attention. *arXiv:2010.09520 [cs]*, 2020.

[543] J. Wang and J. Han. BIDE: efficient mining of frequent closed sequences. In *Proceedings. 20th International Conference on Data Engineering*, pages 79–90, 2004.

[544] Jin Wang, Liang-Chih Yu, K Robert Lai, and Xuejie Zhang. Dimensional sentiment analysis using a regional cnn-lstm model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 225–230, 2016.

[545] Quan Wang and Suya You. Fast similarity search for high-dimensional dataset. In *Eighth IEEE International Symposium on Multimedia (ISM'06)*, pages 799–804. IEEE, 2006.

[546] S. Wang, D. Lo, and L. Jiang. Code search via topic-enriched dependence graph matching. In *2011 18th Working Conference on Reverse Engineering*, pages 119–123, 2011.

[547] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14. ACM, 2014.

[548] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 677–682. ACM, 2014.

[549] Shaowei Wang, David Lo, and Lingxiao Jiang. AutoQuery: automatic construction of dependency queries for code search. *Automated Software Engineering*, 23(3):393–425, 2016.

[550] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. TranS^3: A transformer-based framework for unifying code summarization and code search. *arXiv:2003.03238 [cs]*, 2020.

[551] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 457. ACM Press, 2010.

[552] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. Hunter: Next-generation code reuse for java. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1028–1032. ACM, 2016. event-place: Seattle, WA, USA.

[553] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. Hunter: Next-generation Code Reuse for Java. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1028–1032. ACM, 2016.

[554] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. *arXiv preprint arXiv:1702.03814*, 2017.

[555] http://www.apache.org/, Dec. 2020. last accessed 01.09.2020.

[556] http://www.eclipse.org/, Dec. 2020. last accessed 01.09.2020.

[557] https://www.github.com, Dec. 2020. last accessed 01.09.2020.

[558] https://lucene.apache.org/, Dec. 2020. last accessed 01.09.2020.

[559] http://www.netbeans.org/, Dec. 2020. last accessed 01.09.2020.

[560] http://nutch.apache.org/, Dec. 2020. last accessed 01.09.2020.

[561] https://wordnet.princeton.edu/, Dec. 2020. last accessed 01.09.2020.

[562] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[563] Victor K Wei. Generalized hamming weights for linear codes. *IEEE Transactions on information theory*, 37(5):1412–1418, 1991.

[564] Markus Weimer, Alexandros Karatzoglou, and Marcel Bruch. Maximum margin matrix factorization for code recommendation. In *Proceedings of the third ACM conference on Recommender systems*, RecSys '09, pages 309–312. Association for Computing Machinery, 2009.

[565] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. ACM, September 2016.

[566] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. SnipMatch: using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, UIST '12, pages 219–228. Association for Computing Machinery, 2012.

[567] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[568] Huaiguang Wu and Yang Yang. Code search based on alteration intent. *IEEE Access*, 7:56796–56802, 2019.

[569] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Empir Software Eng*, 22(6):3149–3185, December 2017.

[570] Yingce Xia, Tao Qin, Wei Chen, Jiang Bian, Nenghai Yu, and Tie-Yan Liu. Dual supervised learning. *arXiv preprint arXiv:1707.00415*, 2017.

[571] Tao Xie and Jian Pei. MAPO: mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 54–57. Association for Computing Machinery, 2006.

[572] Tao Xie and Jian Pei. Mapo: Mining API Usages from Open Source Repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR)*, pages 54–57. ACM, May 2006.

[573] Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 54–57. ACM, 2006.

[574] Y. Xie, T. Lin, and H. Xu. User interface code retrieval: A novel visual-representation-aware approach. *IEEE Access*, 7:162756–162767, 2019.

[575] Jinxi Xu and W. Bruce Croft. Query expansion using local and global document analysis. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '96, pages 4–11. Association for Computing Machinery, 1996.

[576] Jinxi Xu and W Bruce Croft. Query expansion using local and global document analysis. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 4–11. ACM, 1996.

[577] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363–376. ACM, 2017. event-place: Dallas, Texas, USA.

[578] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. Accurate and scalable cross-architecture cross-OS binary code search with emulation. *IEEE Transactions on Software Engineering*, 45(11):1125–1149, 2019.

[579] https://www.yahoo.com, Dec. 2020.

[580] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 344–354, 2020.

[581] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 391–402, New York, NY, USA, 2016. ACM.

[582] Jinqiu Yang and Lin Tan. Inferring semantically related words from software context. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 161–170, 2012. ISSN: 2160-1860.

[583] Jinqiu Yang and Lin Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2013.

[584] Jinqiu Yang and Lin Tan. Swordnet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.

[585] Yangrui Yang and Qing Huang. IECS: Intent-Enforced Code Search via Extended Boolean Model. *Journal of Intelligent & Fuzzy Systems*, 33:2565–2576, January 2017.

[586] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. CoaCor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, WWW '19, pages 2203–2214. Association for Computing Machinery, 2019.

[587] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703, 2018.

[588] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*, WWW '20, pages 2309–2319. Association for Computing Machinery, 2020.

[589] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 404–415. Association for Computing Machinery, 2016.

[590] Hang Yin, Zhiyu Sun, Yanchun Sun, and Wenpin Jiao. A question-driven source code recommendation service based on stack overflow. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642-939X, pages 358–359, 2019.

[591] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE, 2018.

[592] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020.

[593] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

[594] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 38–42, 2012.

[595] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[596] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[597] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related API class-names. *IEEE Transactions on Software Engineering*, 44(11):1070 – 1082, 2017.

[598] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. Expanding queries for code search using semantically related API class-names. *IEEE Transactions on Software Engineering*, 44(11):1070–1082, 2018. Conference Name: IEEE Transactions on Software Engineering.

[599] T. Zhang, M. Pan, J. Zhao, Y. Yu, and X. Li. An open framework for semantic code queries on heterogeneous repositories. In *2015 International Symposium on Theoretical Aspects of Software Engineering*, pages 39–46, 2015.

[600] Jie Zhao and Huan Sun. Adversarial training for code retrieval with question-description relevance regularization. *arXiv:2010.09803 [cs]*, 2020.

[601] Le Zhao and Jamie Callan. Term necessity prediction. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 259–268, New York, NY, USA, 2010. ACM.

[602] Le Zhao and Jamie Callan. Term Necessity Prediction. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 259–268. ACM, October 2010.

[603] Le Zhao and Jamie Callan. Automatic term mismatch diagnosis for selective query expansion. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 515–524, New York, NY, USA, 2012. ACM.

[604] Le Zhao and Jamie Callan. Automatic Term Mismatch Diagnosis for Selective Query Expansion. In *Proceedings of the 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 515–524. ACM, August 2012.

[605] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.

[606] S. Zhou, H. Zhong, and B. Shen. SLAMPA: Recommending code snippets with statistical language model. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 79–88, 2018.

[607] Shufan Zhou, Beijun Shen, and Hao Zhong. Lancer: Your code tell me what you need. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1202–1205, 2019.

[608] Qun Zou and Changquan Zhang. Query expansion via learning change sequences. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 24(2):95–105, 2020.

[609] Yanzhen Zou, Chunyang Ling, Zeqi Lin, and Bing Xie. Graph embedding based code search in software project. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, Internetware '18, pages 1–10. Association for Computing Machinery, 2018.