XCoS: Explainable Code Search based on Query Scoping and Knowledge Graph

CHONG WANG*, Fudan University, China
XIN PENG*, Fudan University, China
ZHENCHANG XING, CSIRO's Data61 & Australian National University, Australia
YUE ZHANG*, Fudan University, China
MINGWEI LIU*, Fudan University, China
RONG LUO*, Fudan University, China
XIUJIE MENG*, Fudan University, China

When searching code developers may express additional constraints (*e.g.,* functional constraints and nonfunctional constraints) on the implementations of desired functionalities in the queries. Existing code search tools treat the queries as a whole and ignore the different implications of different parts of the queries. Moreover, these tools usually return a ranked list of candidate code snippets without any explanations. Therefore, the developers often find it hard to choose the desired results and build confidence on them. In this paper, we conduct a developer survey to better understand and address these issues and induct some insights from the survey results. Based on the insights, we propose XCoS, an explainable code search approach based on query scoping and knowledge graph. XCoS extracts a background knowledge graph from general knowledge bases like Wikidata and Wikipedia. Given a code search query, XCoS identifies different parts (*i.e.,* functionalities, functional constraints, nonfunctional constraints) from it and use the expressions of functionalities and functional constraints to search the codebase. It then links both the query and the candidate code snippets to the concepts in the background knowledge graph and generates explanations based on the association paths between these two parts of concepts together with relevant descriptions. XCoS uses an interactive user interface that allows the user to better understand the associations between candidate code snippets and the query from different aspects and choose the desired results. Our evaluation shows that the quality of the extracted background knowledge and the concept linkings in codebase is generally high. Furthermore, the generated explanations are considered complete, concise, and readable and the approach can help developers find the desired code snippets more accurately and confidently.

CCS Concepts: • **Software and its engineering → Software development techniques**.

Additional Key Words and Phrases: code search, explainability, knowledge, concept

1 INTRODUCTION

Developers often accomplish their development tasks by searching code using queries in the form of natural language [57, 71]. Many code search methods have been proposed to help developers, which can be divided

*C. Wang, X. Peng, M. Liu, Y. Zhang, R. Luo and X. Meng are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.

Authors' addresses: Chong Wang, Fudan University, China; Xin Peng, Fudan University, China; Zhenchang Xing, CSIRO's Data61 & Australian National University, Australia; Yue Zhang, Fudan University, China; Mingwei Liu, Fudan University, China; Rong Luo, Fudan University, China; Xiujie Meng, Fudan University, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1049-331X/2023/4-ART $15.00
https://doi.org/10.1145/3593800

Just Accepted

into two categories: methods based on information retrieval (IR) and methods based on deep learning (DL). The IR-based methods [8, 15, 51] treat code as text and retrieve the code with the highest lexical similarities to the query by combining the bag-of-words model with statistical features such as TF-IDF [70] and BM25 [67]. IR-based methods are very fast and can practically support code search on large-scale codebases. The DL-based methods [22, 25, 26, 38, 47, 69, 78] rely on different DL models to learn representations of queries and code snippets with semantics and measure the relevance based on the semantic representations. There are also researches on query expansion [28, 29, 44, 50, 52, 62]. These methods improve the search accuracy by expanding the words in the query with dictionaries or crowd-sourced resources, so as to bridge the lexical gap between the query and the code.

However, there are two major issues that have not been addressed or resolved. First, when searching code developers may express additional constraints (*e.g.,* functional constraints and nonfunctional constraints) on the implementations of desired functionalities in the queries. Existing code search methods treat the queries as a whole and ignore the different implications of different parts of the queries. This may blur the focus of the search for some search methods (especially IR-based methods), resulting in noisy code snippets that are not relevant to the desired functionality. Second, existing methods usually return a ranked list of candidate code snippets without any explanations. It is time-consuming and labor-intensive for developers to understand search results and they often find it hard to choose the desired results [48]. Moreover, due to the lack of background knowledge and relevant explanations, developers may lack confidence in the code they choose, and even choose problematic (*e.g.,* vulnerable) code without realizing it.

To better understand and address the issues, we conduct a developer survey with 101 developers to investigate practitioners' practices and perspectives on code search, covering the aspects of professional background, search requirements, code choosing, code understanding, and knowledge needs. From the survey results, we obtain some insights, such as: **I1:** reading and understanding code is very important for choosing desired code snippets; **I2:** code search services should not be affected by the expressed nonfunctional concerns; **I3:** background knowledge should be provided for developers to make them understand code snippets better and more confidently; **I4:** code snippets should be filtered conveniently rather than just being browsed and checked linearly.

Our key ideas to address two issues are two folds. We can solve the first issue by query scoping, a commonly used technique in automatic question answering and recommendation systems [66]. It includes query segmentation and query tagging, which are responsible for dividing the query into different parts and determining the implication scope of each part respectively. Through query scoping, we can determine which part of the query should be used to directly search the code, and which part should be used to help filter the results. For the second issue, the key idea is to bridge the knowledge gap between the code and the query using a background knowledge graph extracted from general knowledge bases. We observe that background knowledge related to software development is contained in general knowledge bases like Wikidata [12] and Wikipedia [13], which can help developers understand the conceptual associations between the code and the query, and provide explainability for the searching results.

Based on these two ideas, in this paper we propose XCoS, an e**X**plainable **Co**de **S**earch approach based on query scoping and knowledge graph. XCoS includes an offline phase for background knowledge graph construction and an online phase for code search. In the offline phase, XCoS first builds the skeleton of the background knowledge graph with software development related concepts and basic relationships identified from general knowledge bases and then enriches it with extended knowledge extracted from related Wikipedia articles. In the online phase, XCoS first identifies different parts (*i.e.,* functionalities, functional constraints, nonfunctional constraints) from the given search query and uses the expressions of functionalities and functional constraints to search the codebase with existing search tools (*e.g.,* one built based on Elasticsearch [4]). After that, XCoS links both the query and the candidate code snippets to the concepts in the background knowledge graph and generates explanations based on the association paths between these two parts of concepts together with relevant descriptions.

Just Accepted

XCoS uses an interactive user interface (UI) that allows the user to better understand the associations between candidate code snippets and the query from different aspects. These aspects include: query parts and their corresponding scopes, explanatory information (*i.e.,* conceptual association paths, related descriptions and additional suggestions) organized in the form of trees, and concept-annotated code snippets. Based on this interactive interface, XCoS helps the user choose desired results better.

To evaluate the quality and usefulness of XCoS, we conduct a series of experiments with the Java code snippets in CodeSearchNet dataset [30]. An empirical evaluation of the intrinsic quality reveals that XCoS constructs high quality of background knowledge graph and concept linkings in the codebase. The effectiveness assessment for generating explanations shows that XCoS can produce complete, concise, and readable explanatory information. More so, an empirical study with 14 participants reveals that XCoS can significantly improve the accuracy by 40.5% and increase confidence in choice when used to support participants in code search tasks.

2 MOTIVATING EXAMPLE

As reported in previous studies [42, 76] on Q&A forums such as Stack Overflow (SO), when accomplishing development tasks developers often have funcational requirements (*e.g.,* specific functionalities) and nonfunctional concerns (*e.g.,* performance and security). The funcational requirements and nonfunctional concerns can be reflected in the titles, bodies, answers, and discussion threads of questions. In the similar vein, developers may also have both functional requirements and nonfunctional concerns when searching code and express them into search queries as constraints. Below, we demonstrate the impact of constraints on search and the importance of explainability through a concrete example.

Hash calculation is a commonly used functionality in development tasks and Fig. 1 shows a question [11] on Stack Overflow about it. Besides the desired basic functionality *calculate hash*, the questioner has an additional funcational constraint (*i.e., to verify file integrity*). Moreover, according to the statements in the question body (as shown in the red box), the questioner also concerns about the security aspect of different hash functions. It means that the questioner also has a demand for some nonfunctional constraints, *e.g., not vulnerable to attacks*, attached to the basic functionality. For better exploration, we take all the Java code snippets (about 500K Java methods) in the CodeSearchNet [30] dataset as the codebase, and use Elasticsearch to build a code search service.

For the above example, when only the basic functionality (*i.e., calculate hash*) is used to search, the first few candidate code snippets returned have nothing to do with calculating file hash until the 8th and 9th ones, which are computing the hash of file by using a hash tree and SHA-1, respectively. When searching by combining the functionality and functional constraint like *calculate hash to verify file integrity*, the first candidate code snippet returned is about downloading a file, and part of this code is using MD5 to verify the integrity of the downloaded file. Compared to searching with only functionality, it can be seen that adding functional constraints to basic functionality can help clarify the intent of the search and reduce the search scope. On the other hand, if we attach the nonfunctional constraint to the functionality (*e.g., calculate hash not vulnerable to attacks*) to search code, of the top 10 candidate code snippets, only the 8th one is related to hash, and the rest are all related to vulnerabilities or attacks. It means that adding nonfunctional constraints brings noise to the search. To this end, given a search query, we need to analyze the query and select appropriate parts of it to search code.

Even if we can analyze the query well to make the search results more relevant to the query, developers may still find it difficult to pick the desired code snippet. For the above example, assume that *calculate hash to verify file integrity* is used for searching. Although the first candidate code snippet contains code for using MD5 to verify a file's integrity, developers still need to read and understand this code, and combine their own background knowledge about MD5 to determine whether the code snippet meets their needs. For example, from the description on Wikipedia shown in Fig. 2, we know that the MD5 hash algorithm can be used to verify data integrity, which means that the code snippet can satisfy the functional constraint of verifying file integrity. But

Just Accepted

Signature/Hash Choice for File Integrity Verification

Asked 9 years, 3 months ago Active 8 years, 9 months ago Viewed 1k times

1

For a file repository, I need to select a hashing algorithm that will reasonably ensure the integrity of files.

I need an algorithm that anyone (with a bit of effort) would be able to easily use to verify the integrity given the hash. In short, the file may be transferred to the user, along with a hash, and they must be able to verify that the hash comes from the file.

My first choice would be MD5 because there seems to be widely available utilities to verify MD5 hashes, but I'm concerned with the MD5 algorithm being cryptographically broken (ref Wikipedia/US-CERT: http://en.wikipedia.org/wiki/MD5)

Fig. 1. A SO Question about Calculating Hash

MD5

From Wikipedia, the free encyclopedia

The **MD5 message-digest algorithm** is a cryptographically broken but still widely used hash function producing a 128-bit hash value. Although MD5 was initially designed to be used as a cryptographic hash function, it has been found to suffer from extensive vulnerabilities. It can still be used as a checksum to verify data integrity, but only against unintentional corruption.

Fig. 2. Descriptions in the Wikipedia Article of MD5

the description also mentions that MD5 is extensively vulnerable, which means that this code may not satisfy the security concerns of the developers. If the developers themselves do not know these relevant background knowledge, the result selection process may require consulting external materials, which is very time-consuming and labor-intensive. In some cases, *e.g.,* some developers are unaware of the security issues of the hash algorithm, they may even choose problematic code snippets that lead to some bad consequences (*e.g.,* introducing potential vulnerabilities). At the same time, due to the above reasons, developers usually lack confidence in the selected code snippets when they do not have relevant background knowledge. The main cause for the above problems is that existing code search tools lack the associations between search results and query parts and the explainability of search results.

3 DEVELOPER SURVEY

We further design a developer survey to investigate practitioners' practices and perspectives on code search requirements and code understanding.

To ensure the quality and participation rate of the survey, we invite active developers on GitHub to participant in our online survey. We first obtain the list of active GitHub developers using a commit statistics tool[1] and then get the corresponding email addresses of these developers via GitHub APIs[2]. We then email each of these developers to briefly introduce our research topic on code search and kindly invite them to participate in our online questionnaire survey. Finally, 4,208 emails are sent in total. As shown in Table 1, the survey include 12 questions

[1]https://github.com/lauripiispanen/most-active-github-users-counter, the statistics data was generated at Oct. 30, 2022
[2]https://api.github.com/

Just Accepted

Table 1. Survey Questions

Q1	How many years of development experience do you have?
Q2	How often do you search code snippets using natural language queries to accomplish your development tasks?
Q3	What methods or tools do you use to search code snippets?
Q4	Besides the basic functionality requirements, what other nonfunctional concerns do you have about the code snippets?
Q5	Do you want to express these concerns in queries when searching code snippets? Why?
Q6	How do you choose the desired code snippets that meet your requirements or concerns in the search results?
Q7	Is it efficient to linearly browse and check the code snippets in the list of search results?
Q8	How much time will be spent understanding the search results during the choosing process?
Q9	What kinds of information/knowledge are necessary to understand the search results?
Q10	How do you consult and summarize the necessary information/knowledge?
Q11	How do the information/knowledge and the understanding of search results affect your confidence in the code snippets you choose?
Q12	Would it be helpful to develop a tool that can provide related information/knowledge for understanding the search results?

Just Accepted

that cover the aspects of professional background, search requirements, code choosing, code understanding, and knowledge needs. It takes participants about 5-10 minutes to complete. For the open questions, three authors categorize the answers separately and a group discussion is conducted to reach consensus. In seven days, there are 101 developers coming from 67 countries/regions responsing to our invitation emails and finishing the questionnaire (*i.e.,* a participation rate of 2.4%).

Professional Background (Q1–Q3). ***Q1:*** Of all the 101 participants, 39.6% have more than 10 years of development experience, 14.8% have 5 to 10 years, and 21.8% have 3 to 5 years, and 23.8% have less than 3 years. ***Q2:*** 45.5% of the participants always or often search code to accomplish development tasks, 8.9% as often as not search code, 22.8% sometimes search code, and 21.8% rarely search code. ***Q3:*** Of the participants, 96 give their common search services, including Google (84.4%), Stack Overflow (32.3%), GitHub (11.5%), DuckDuckGo (5.2%), etc.

Search Requirements (Q4–Q5). ***Q4:*** There are 65 participants listing their nonfunctional concerns when searching code, mainly including performance/efficiency (40.0%), security/vulnerability (18.5%), robustness/reliability/stability (15.4%), compatibility (*e.g.,* platforms, languages, and dependencies) (15.4%), understandability (13.8%), style/cleanliness/readability (10.8%), copyright/license (8.8%), recentness/timeliness (7.7%), idiomaticness/patterns (7.7%), etc. ***Q5:*** Of the 65 participants, 35.4% respond that they want to express their nonfunctional concerns in search queries, while 49.2% respond that they do not. For the participants wanting to express the nonfunctional concerns, the main reason is that they expect to narrow down the search scope and get better search results. For the participants not wanting to, there are three main reasons: search services cannot give the better (even worse) results when they add the nonfunctional concerns (35.0%); they want to manually check whether the searched code meets the nonfunctional concerns (30.0%); they do not know how to express the nonfunctional concerns (15.0%).

Choosing of Desired Code Snippets (Q6–Q7). ***Q6:*** There are 77 participants giving that how they choose the desired code snippets that meet their requirements. The most commonly used ways include reading code and comments/documentation (48.1%), testing and trying the code (27.3%), checking some extra metrics (*e.g.,* website authority, GitHub stars, open/close issues, Stack Overflow upvotes, and lisences) (24.7%), and comparing different solutions (3.9%), etc. ***Q7:*** Of the 77 participants, 50.6% think that it is efficient to linearly (*i.e.,* one-by-one) browse and check the code snippets, and the rest 49.4% think the opposite.

Time of Code Understaning (Q8). Of the participants, 28.3% spend most of the time on code understaning in the code choosing process, 38.0% spend about a half of the time, and 33.7% spend little time.

Information/Knowledge Needs for Code Understaning (Q9–Q10). ***Q9:*** There are 78 participants listing the information/knowledge they think necessary to understand code snippets, including basic technical background knowledge (*e.g.,* the terms/concepts used in code and the understanding of security and performance) (42.3%), language/library specific knowledge (*e.g.,* language syntax and library APIs) (33.3%), explanations of the code (*e.g.,* brief descriptions and related documentation) (11.5%), experience (*e.g.,* reading-comprehension and source criticism skills) (7.7%), etc. ***Q10:*** Of the 78 participants, 70 respond the two common ways they consult and summarize the necessary information/knowledge, *i.e.,* searching online resources (*e.g.,* Google, Documentation, Wikipedia, Forums, and Blogs) (71.4%) and asking other developers for help (10.0%).

Impact of Information/Knowledge (Q11–Q12). ***Q11:*** Of the participants, 68.1% think that the information/-knowledge makes them more confident on code understaning, 22.9% think no effect, and 8.5% think less confident. ***Q12:*** 74.5% of the participants think it is helpful to develop a tool to provide related information/knowledge for understanding code snippets, and the rest 25.5% think the opposite.

Insights. Based on the survey results, we obtain the following insights. **I0:** In practice, developer still use information retrieval based code search services (*e.g.,* GitHub Search) to search code, instead of deep learning based methods (*e.g.,* DeepCS [25]). **I1:** Reading and understanding code occupies a very important position in the process of choosing desired code snippets and takes a lot of time, tools are needed to help developers understand the code snippets in search results. **I2:** Such tools should allow developers to express their functional requirements and nonfunctional concerns simultaneously, and the code search services should not be affected by the nonfunctional concerns. **I3:** Such tools should provide necessary information/knowledge for developers which can make them understand code snippets better and more confidently, and background knowledge is the most important kind of the information/knowledge. **I4:** Developers should be able to conveniently filter code snippets with such tools, rather than just browsing and checking the code snippets linearly in the order returned by the search services.

4 APPROACH

Based on the insights **I1**, **I2**, and **I3**, we propose an explainable code search approach named XCoS based on query scoping and knowledge graph. With query scoping, XCoS solves the problem of a query having different implication parts. By extracting the background knowledge graph from the general knowledge bases and using the knowledge graph to establish the associations between the search results and the query, XCoS can increase the explainability of existing code search methods and help developers choose the desired code snippets better and more confidently. In this section, we first present an overview of XCoS, and then describe each step of it in detail.

4.1 Overview

As shown in Fig. 3, XCoS includes an offline phase for background knowledge graph construction and an online phase for code search.

The offline phase extracts software development related background knowledge from Wikidata [12] and Wikipedia [13]. Wikidata is a free and open knowledge base for general knowledge and includes many software development related concepts such as “MD5”, “JSON”, “Computer network” [46]. Wikipedia is a free online encyclopedia which is closely associated with Wikidata. For many concepts in Wikidata we can find articles that describe them in detail in Wikipedia. The background knowledge extraction includes two steps, *i.e.,* skeleton knowledge extraction and extended knowledge extraction. In the first step, XCoS extracts software development related concepts together with their basic relationships and definitions from Wikidata and Wikipedia. These concepts and relationships constitute the skeleton of the background knowledge graph. In the second step, XCoS extracts additional descriptions and relationships for the concepts obtained in the first step by analyzing the

Just Accepted

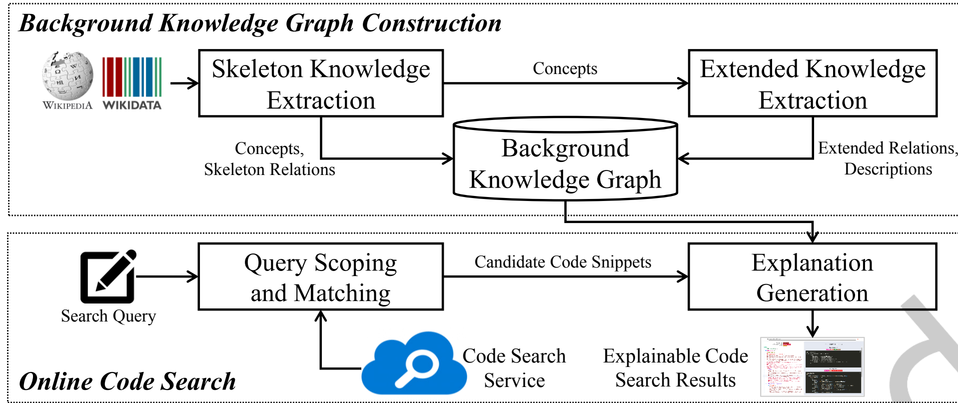


Fig. 3. An Overview of XCoS

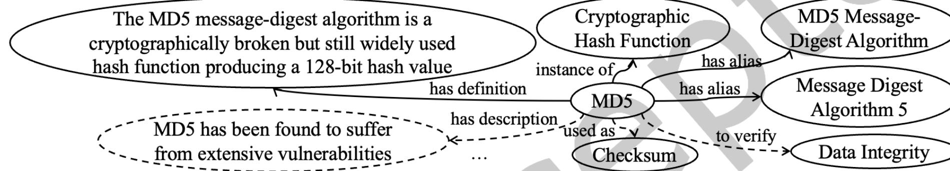


Fig. 4. Background Knowledge Extracted for "MD5"

corresponding Wikipedia articles. Fig. 4 shows a part of the extracted background knowledge for "MD5". Solid line ellipses and arrows represent skeleton knowledge extracted in the first step, including: related concepts and relationships defined in Wikidata, *e.g.,* "MD5", "Data Integrity", "Checksum", ("MD5", "instance of", "cryptographic hash function"), ("MD5", "has alias", "Message Digest Algorithm 5"); a concept definition given by Wikipedia, *e.g.,* "The MD5 message-digest algorithm is a cryptographically broken but still widely used hash function producing a 128-bit hash value". Dashed ellipses and arrows represent extended knowledge extracted in the second step, including: description sentences extracted from Wikipedia articles, *e.g.,* "MD5 has been found to suffer from extensive vulnerabilities"; additional relationships extracted from Wikipedia articles, *e.g.,* ("MD5", "to verify", "Data Integrity").

The online phase produces explainable code search results for a code search query based on the background knowledge graph. It includes two steps, *i.e.,* query scoping and matching, explanation generation. In the first step, XCoS extracts different parts (*i.e.,* functionalities, functional constraints, nonfunctional constraints) from the search query and uses an existing code search service (*e.g.,* one built on Elasticsearch [4] or code search approaches like DeepCS [25]) to obtain a set of candidate code snippets based on the functionalities and functional constraints. In the second step, XCoS links both the query and the candidate code snippets to the concepts in the background knowledge graph and generates explanations based on the association paths between these two parts of concepts together with relevant descriptions.

4.2 Skeleton Knowledge Extraction

The key for skeleton knowledge extraction is the identification of software development related concepts from Wikidata.

Just Accepted

XCoS extracts all Wikidata concepts from the dump of Wikidata [2] and the corresponding Wikipedia articles from the dump of Wikipedia [3]. It associates Wikidata concepts and Wikipedia articles based on the Wikipedia hyperlinks of the concepts, and then filters out the concepts that have no corresponding Wikipedia articles. For each remaining concept, XCoS takes the title and the first sentence of the corresponding Wikipedia article as its name and definition respectively. In this way, we obtain 5,420,321 Wikidata concepts that have the corresponding name and definition.

To identify the software development related concepts from all the Wikidata concepts, XCoS then trains a classifier based on the concept definitions using BERT [18], a pre-trained text representation model that can achieve very high accuracy in text classification tasks. To avoid manual annotation, we design a method to automatically construct the training data based on the associations between Wikidata concepts and SO tags. There are 3,181 Wikidata concepts (*e.g.,* RGB color model[3]) having corresponding SO tags (*e.g.,* rgb[4]) which can be obtained from their "Stack Exchange tag" attributes. We take all these concepts as positive samples and randomly select twice the number of positive samples from remaining Wikidata concepts as negative samples, resulting in a relatively class-balanced dataset. Class imbalance of training data is a serious problem for learning based classification algorithms, which will decrease the effectiveness of the trained classifier [43]. Artificially data balancing is a commonly used method to address the problem. We divide the dataset into training set and validation set by 9:1.

Based on the trained classifier, we identify software development related Wikidata concepts. For identified concepts, we obtain their aliases and relationships between them from Wikidata. After randomly sampling and manually validating, the relationships are confirmed high-quality. This is thanks to the fact that Wikidata is a reliable crowdsourced knowledge graph and only the relationships between two identified concepts are fetched. These concepts and relationships together with the names and definitions of the concepts constitute the skeleton knowledge of the background knowledge graph. For the example shown in Fig. 4, all the solid line concepts, definitions, and relationships are extracted in this step.

4.3 Extended Knowledge Extraction

For a software development related concept extracted from Wikidata, Wikipedia articles often provide various descriptions about it. These descriptions may provide useful knowledge explaining the relationships between queries and desired code snippets. XCoS extracts these descriptions and the concept relationships in them using the following process. First, it extracts and normalizes description sentences from Wikipedia articles. Second, it extracts open relationships from the description sentences. Third, it identifies candidate mentions of software development related concepts in the description sentences. Fourth, it links the identified mentions to the corresponding software development related concepts. Fifth, it creates additional relationships based on concept linking results.

4.3.1 Description Sentence Extraction and Normalization. For each concept, we obtain the first paragraph of the corresponding Wikipedia article for further analysis. We use a coreference resolution tool (*e.g.,* neuralcoref [6] in our implementation) to resolve the references of pronouns in the paragraph. As the coreference resolution tool does not work well on sentences starting with "this/the" followed by nouns (*e.g.,* "this algorithm"), we further use some heuristic rules to handle such cases. We first use a natural language processing (NLP) tool (*e.g.,* spaCy [9] in our implementation) to segment the paragraph into sentences. Then for each sentence starting with "this/the" followed by a noun, we determine whether the noun is a suffix of the article title, and if so replace "this/the" and the noun with the title. After coreference resolution, we identify all the hyperlinks that point to other Wikipedia articles in the sentences and replace the anchor text of them with the titles of the target articles. Through the

[3]https://www.wikidata.org/wiki/Q166194
[4]https://stackoverflow.com/tags/rgb

Just Accepted

above process, we can obtain a set of normalized description sentences for each concept. For example, for the concept "MD5" a description sentence "MD5 can still be used as a checksum to verify data integrity" can be obtained from the original sentence "It can still be used as a checksum to verify data integrity".

4.3.2 Open Relationship Extraction. To facilitate the association analysis between search queries and code snippets, we further extract structured concept relationships from the description sentences. We use an open information extraction tool (*e.g.,* OpenIE6 [35] in our implementation) to extract triples of concept relationships from the sentences. For example, two concept relationship triples ("MD5", "used as", "checksum") and ("MD5", "to verify", "data integrity") can be extracted from the description "MD5 can still be used as a checksum to verify data integrity". Note that the conceptual expressions in these triples such as "checksum" and "data integrity" are still textual expressions that are not linked to the corresponding concepts at this time. There is no standard list of relationships since the relationships built by open information extraction are in open domain (usually the predicates in description sentences).

4.3.3 Candidate Concept Mention Identification. To obtain a highly structured background knowledge graph, we need to further link the extracted description sentences and open relationships to the related concepts. To this end, we need to first identify candidate concept mentions in the description sentences by analyzing noun phrases. Given a concept description, we use an NLP tool (*e.g.,* spaCy in our implementation) to identify the noun phrases in it. For each identified noun phrase, we remove stop words at the beginning and end of it, and then use all its possible sub-phrases (including the phrase itself) to match possible concepts. If a sub-phrase is a part of the names or aliases of some concepts, the sub-phrase is treated as a candidate concept mention and all the matching concepts are treated as candidate concepts for the mention. To ensure the quality of candidate mentions and concepts, following rules are applied.

1) Longer Concept Mentions Preferred. If a noun phrase contains multiple overlapping sub-phrases that can be treated as candidate mentions, we will only consider the longest sub-phrase as a candidate mention.

2) Concepts Referred to by Hyperlinks Preferred. Some concepts are referred to by hyperlinks in the current Wikipedia article. If a sub-phrase matches such a concept, we will consider the concept as the only candidate concept for the mention. For example, if "buffer" appears in a description and there is a hyperlink to "Data Buffer" in the current article but no hyperlink to "Optical Buffer", "Data Buffer" will be considered as the only candidate concept for the mention "buffer".

3) Single-Word Sub-Phrase Specially Considered. A candidate mention that contains only one word may be just a common word. To alleviate the ambiguity brought by common words, we treat the corresponding common word as a special candidate concept besides other candidate concepts. For example, for the candidate mention "buffer" we consider the common word "buffer" as a special candidate concept besides "Data Buffer".

4.3.4 Concept Linking. Given a set of candidate mentions in a description sentence, we link each of them to one of its candidate concepts or none (*i.e.,* no relevant concept). The concept linking here can be treated as an optimization problem with the following three objectives: maximize the lexical similarity between the mentions and linked concepts; maximize the semantic similarity between the context of mentions and linked concepts; and maximize the conceptual coherence of the linked concepts of all mentions.

The lexical similarity between a mention m and a candidate concept c can be calculated using a smoothed Jaccard coefficient as Eq. 1 (Eq. is short for Equation), where $wb(p)$ is the bag of lemmatized words (excluding stop words) in a given phrase p and $names(c)$ returns the name and aliases of c. We consider the maximum lexical similarity between the mention and the concept names/aliases and use the square root to smooth the origin Jaccard coefficient.

$$lexS(m,c) = \max_{n_c \in names(c)} \sqrt{\frac{|wb(m) \cap wb(n_c)|}{|wb(m) \cup wb(n_c)|}} \tag{1}$$

Just Accepted

The semantic similarity between the context of the mentions ctx (*i.e.,* the description sentence) and a candidate concept c can be measured by the cosine similarity between their text embeddings as Eq. 2. In Eq. 2, $def(c)$ is the definition of c and the function $emb_s(s)$ returns the sentence embedding of a sentence s by averaging the embeddings of all the words in $wb(s)$. We use a pre-trained Wikipedia2Vec model [14] to generate the word embeddings.

$$ctxS(ctx, c) = cosine(emb_s(def(c)), emb_s(ctx)) \tag{2}$$

The conceptual coherence of two candidate concepts c_1 and c_2 can be measured by the cosine similarity of their embeddings as Eq. 3. Note that besides concepts there may be a common word in the set of candidate concepts of a mention. Therefore, we need an embedding model to map both concepts and words into the same vector space when calculating the conceptual coherence. We use Wikipedia2Vec model [14] as the model, as it can simultaneously guarantee that concepts and words with similar meanings are close in the vector space.

$$coh(c_1, c_2) = cosine(emb_w(c_1), emb_w(c_2)) \tag{3}$$

Given a set of candidate mentions M in a description sentence, we need to find a set of concepts C where each mention m_i in M has a linked concept c_i from its candidate concepts. We adopt the widely used pairwise strategy in entity linking literatures [36] to define the fitness function of concept linking as Eq. 4, which combines the fitness of the linked concepts for all pairs of different mentions (m_i, m_j) as defined in Eq. 5. For a pair of mention (m_i, m_j) and the corresponding pair of linked concepts (c_i, c_j), Eq. 5 combines the similarities between the concepts and the mentions with the context of the mentions and the conceptual coherence of the concepts.

$$fitness(M, C, ctx) = \sum_{\substack{m_i, m_j \in M (i \neq j) \\ c_i, c_j \in C}} pairFit(m_i, m_j, c_i, c_j, ctx) \tag{4}$$

$$\begin{aligned} pairFit(m_i, m_j, c_i, c_j, ctx) = & \; lexS(m_i, c_i) + lexS(m_j, c_j) \\ & + ctxS(ctx, c_i) + ctxS(ctx, c_j) \\ & + coh(c_i, c_j) + coh(c_j, c_i) \end{aligned} \tag{5}$$

To optimize the fitness function defined in Eq. 4, we need to enumerate all possible groups of candidate concepts and find a best group which can maximize the fitness. This problem is NP-hard, so we choose to adopt a strategy that greedily links the mentions to the corresponding concepts in a pairwise way: 1) choose a pair of mentions (m_i, m_j) with at least one of them unlinked and a pair of concepts (c_i, c_j) from their candidate concepts and if m_i or m_j is linked only consider the linked concept for c_i or c_j; 2) consider all combinations meeting the above conditions and select a combination (m_i, m_j, c_i, c_j) with the highest pairwise fitness as defined by Eq. 5, link m_i and m_j to c_i and c_j respectively if they are not linked yet; 3) repeat step 1 and 2 until all mentions are linked or the highest pairwise fitness is lower than a threshold (0.8 in our implementation). When the process ends, all candidate mentions that are not linked or linked to a common word are omitted.

4.3.5 Additional Relationship Creation. Based on the results of concept linking, we create the following three kinds of relationships for related concepts.

1) Description Relationship. For each linked concept of a mention in a description sentence, create a "has description" relationship from the concept to the description sentence. For example, the relationship from "MD5" to "MD5 has been found to suffer from extensive vulnerabilities" in Fig. 4 is created in this way.

2) Extracted Open Relationship. For each open relationship $(head, rel, tail)$ extracted in Substep 2, try to map the element (*i.e., head* or *tail*) to a linked concept of the corresponding description sentence in the following way: find a linked concept c that makes the highest Jaccard coefficient between the element and the mention of c in the sentence; if the Jaccard coefficient is larger than a predefined threshold (0.6 in our implementation), then map the element to c. If both *head* and *tail* can be mapped to a concept (denoted by c_h and c_t respectively),

Just Accepted

create a relationship *rel* from c_h to c_t. For example, the relationship ("MD5", "used as", "Checksum") and ("MD5", "to verify", "Data Integrity") in Fig. 4 are created in this way.

3) Concept Association Relationship. For each pair of concepts c_1 and c_2, if the cosine similarity of their Wikipedia2Vec embeddings [14] is higher than a threshold (0.8 in our implementation), then create an "associated to" relationship between them.

4.4 Query Scoping and Matching

Query scoping extracts different parts, *i.e.,* functionalities, functional constraints, nonfunctional constraints, from a code search query. The first two parts represent basic funcationalities and additional funcational requirements respectively, and the third part reflects the nonfunctional concerns. These parts are used in different ways in subsequent code snippets matching and explanation generation. Given a code search query we first remove the starting words for question, *e.g.,* "how to" and "how can I". Then we use an NLP tool (*e.g.,* spaCy in our implementation) to analyze the part of speech and dependence tree of the remaining query. After that we extract different parts from the query using linguistic rules. Applying linguistic rules on the NLP analysis results (*i.e.,* part of speech and dependence tree) is a common way to extract functionality and constraints [17, 31, 58, 77]. We design the linguistic rules based on these previous works and make some adaptations on search queries. For example, adverb or adverb clauses are connected to nonfunctional constaints based on the mined rules by Dalpiaz et al. [17] and resultant decision tree by Hussain et al. [31] We randomly sample 50 question titles (not involved in subsequent evaluation) as a validation dataset to iteratively refine and validate the rules by observing the extraction results. In the rules, *VERB*, *DOBJ*, *PREP*, *POBJ*, and *MOD* denote verb, direct object, preposition, preposition object, and modifier, respectively.

- Functionality: *VERB* or *VERB DOBJ*;
- Functional Constraint: *PREP POBJ*, *to VERB DOBJ*, or *using DOBJ*;
- Nonfunctional Constraint: adverbs, adverb clauses, relative clauses, or phrases such as *in MOD way* that are used to modify or qualify the functionalities or functional constraints.

Using the above rules, for example, we can obtain the following results of query scoping, where the pink, cyan, and orange parts represent functionality, functional constraint, and nonfunctional constraint, respectively.

calculate hash *to verify file integrity*
read large file *in memory efficient way*

Note that, there is no predefined and fixed set of non-functional requirements. The non-functional intents are extracted from a given query using linguistic heuristics and based on the query content. Based on the results of query scoping for a code search query, we compose the functionality part and functional constraint part together to retrieve code snippets using an existing code search service, *e.g.,* "*calculate hash* *to verify file integrity*" and "*read large file*" are used to retrieve code snippets for above two queries respectively.

4.5 Explanation Generation

For the candidate code snippets returned by the code search service, XCoS generates the corresponding explanations based on the conceptual associations between the code search query and the code snippets. These explanations later are displayed in a structured way on the code search UI (see Sec. 5) to help the user to choose the desired code snippets. For each candidate code snippet, the generated explanations include the following three parts.

- **Conceptual Association Paths**: one or multiple paths in the background knowledge graph that can potentially associate the query and the code snippet.
- **Related Descriptions**: description sentences that may help explain the association between the query and the code snippet.

Just Accepted

- **Additional Suggestions**: description sentences that are not so relevant to the query but the user may be interested in.

4.5.1 Identification of Conceptual Association Path. To identify the association paths between a query and a code snippet, we need to first link the query and the code snippet to the concepts in the background knowledge graph and then identify conceptual paths between the linked concepts of them.

The concept linking for the query is conducted in a similar way to the candidate concept mention identification in the extended knowledge extraction (see Sec. 4.3.3). We identify the noun phrases in the query and use all their possible sub-phrases (including the phrases themselves) to match possible concepts. As the query is usually very short and contains very little context, we do not conduct entity linking as Sec. 4.3.4 and consider all the candidate concepts for further analysis of association paths.

The concept linking for the code snippet is conducted by analyzing the code identifiers. We use a static code analysis tool (*e.g.,* javalang [5] in our implementation) to extract all the identifiers in the code snippet, and then use a tool called Sprial [10] to split the identifiers into words. After that, we use a tool called POSSE [27] (re-implemented in Python) to analyze the part of speech of the words in each identifier and obtain the noun phrases in it. To ensure the quality of concept linking, we filter out stop words such as single letters and common verbs like "update" based on a predefined list. For all the remaining noun phrases, we conduct mention identification and concept linking in a similar way to Sec. 4.3.3 and 4.3.4. During the process, all noun phrases obtained from the code snippet are combined together as the context for concept linking. Note that the concept linking for code snippets is independent of the query, so it can be done for all the code snippets offline if the codebase is available to improve the efficiency of online code search.

For a set of linked concepts C_{query} of the query and a set of linked concepts C_{code} of the code snippet, we identify all the pairs (c_q, c_c) $(c_q \in C_{query}, c_c \in C_{code})$ that satisfy one of the following three conditions.

- c_c and c_q are the same concept;
- c_c and c_q are neighbors in the background knowledge graph;
- c_c and c_q are connected by a path in which all the edges are hypernym/hyponym relationships (*e.g.,* "instance of" or "subclass of") in the background knowledge graph.

For each pair of concepts (c_q, c_c) that satisfies one of the above conditions, we treat c_q and c_c as the ***anchor concepts*** and the path between them as an association path. Then, we can link the query part that c_q appears in to the association path. The association path provides a potential explanation for how the code snippet is relevant to the query. For example, the path $[MD5 - to\ verify \rightarrow Data\ integrity]$ can explain the association between the code snippets containing MD5 and the functional constraint *to verify file integrity* of the query *calculate hash to verify file integrity.*

4.5.2 Identification of Related Descriptions. Description sentences of relevant concepts often can provide additional explanations for choosing or not choosing a candidate code snippet. For example, the description "MD5 has been found to suffer from extensive vulnerabilities" can help users to exclude code snippets that use MD5 if they seek a solution that is not vulnerable to attacks. For an association path, we identify related descriptions from the description sentences of the anchor concept of the code snippet by measuring their relevance to different parts (*i.e.,* functionalities, functional constraints, nonfunctional constraints) of the query. To this end, we use Sentence-BERT [64], which is suitable for measuring semantic textual similarity, to obtain the vector representations of the candidate descriptions and different parts of the query and calculate their cosine similarity as Eq. 6 where des is a candidate description, q_part is a part of the query, $sbert(text)$ returns the vector representation of $text$ using Sentence-BERT. For the functionality part of the query, we select two candidate descriptions that have the highest similarities calculated using Eq. 6 as related descriptions and use the similarities as their scores. For each constraint part (functional or nonfunctional) of the query, we also select two candidate descriptions that have the highest scores calculated using Eq. 7 which combines the similarities between the description and both

Just Accepted

the constraint part (*i.e., con*) and the functionality part (*i.e., fun*) of the query. The two weights 0.7 and 0.3 are experimentally determined based on a small validation dataset.

$$sim(des, q_part) = cosine(sbert(des), sbert(q_part)) \tag{6}$$

$$score(des, con, fun) = 0.7 \times sim(des, con) + 0.3 \times sim(des, fun) \tag{7}$$

We decide to select the two descriptions having the highest similarity scores based on the observation on some validation examples and the discussion among the authors and some developers (not involved in evaluation). Two description sentences can ensure good completeness and conciseness of the explanations, which is confirmed by the results of RQ2 (see Section 6.2). We do not set a threshold for the similarity scores of the description sentences, as we find that the difference between the scores is generally small so that it is difficult to choose a suitable threshold.

4.5.3 Identification of Additional Suggestions. In some cases the user may hope to see diversified explanations that cover some concerns not directly mentioned in the query. Therefore, for an anchor concept of the code snippet, we identify some additional suggestions from its description sentences that are not identified as related descriptions. We consider all the remaining description sentences as the candidates and calculate their average cosine similarity (as Eq. 6) with all the identified related descriptions of the association paths ended with the anchor concept. Based on the results we select two additional description sentences that have the median similarities as the additional suggestions. The rationale for the selection is that the additional suggestions should be similar to the related descriptions and at the same time reflect diversified perspectives of explanations. For example, if users do not realize vulnerability issues when calculating hash to verify file integrity, an additional suggestion "MD5 has been found to suffer from extensive vulnerabilities" can prompt them to notice the problem. We decide to select the two descriptions having the highest similarity scores based on the observation on some validation examples (randomly sampled 50 question titles that are not involved in subsequent evaluation) and the discussion among the authors and some developers. Two description sentences can ensure good completeness and conciseness of the explanations, which is confirmed by the results of RQ2 (see Section 6.2). We do not set a threshold for the similarity scores of the description sentences, as we find that the difference between the scores is generally small so that it is difficult to choose a suitable threshold.

5 UI DESIGN

XCoS supports explainable code search through an interactive UI as shown in Fig. 5, which can support the insight **I4** from our survey results. The UI displays the results of query scoping, the identified conceptual association paths and descriptions in a structured way and supports the user to interactively examine the candidate code snippets based on related concepts and descriptions.

The UI includes three areas, *i.e.,* **Query Parts**, **Explanations**, and **Code Snippets**. The **Query Parts** area shows different parts of the query, *i.e.,* functionalities, functional constraints, nonfunctional constraints (see Sec. 4.4). For example, "calculate hash" and "to verify file integrity" shown in this area are the functionality and the functional constraint of the query. These parts link to the conceptual association paths (see Sec. 4.5.1). This area includes a special query part "See also", which links to the additional suggestions (see Sec. 4.5.3). The **Explanations** area shows the explanations generated for the current candidate code snippets. It organizes the explanations in conceptual trees as follows. We collect all the anchor concepts of code snippets (see Sec. 4.5.1) and try to build the hierarchical structure in trees between them. That is, if an anchor concept c_1 is connected to another anchor concept c_2 though one or multiple hyponym relationships (*e.g.,* "instance of" or "subclass of") in the background knowledge graph, then c_1 and c_2 are inserted to the same conceptual tree and the hyponym relationships between them are reserved as the hierarchical structure in tree. For example, "Cryptographic hash function", "MD5", and "SHA-1" are three anchor concepts of the code snippets, they are in the same conceptual

Just Accepted

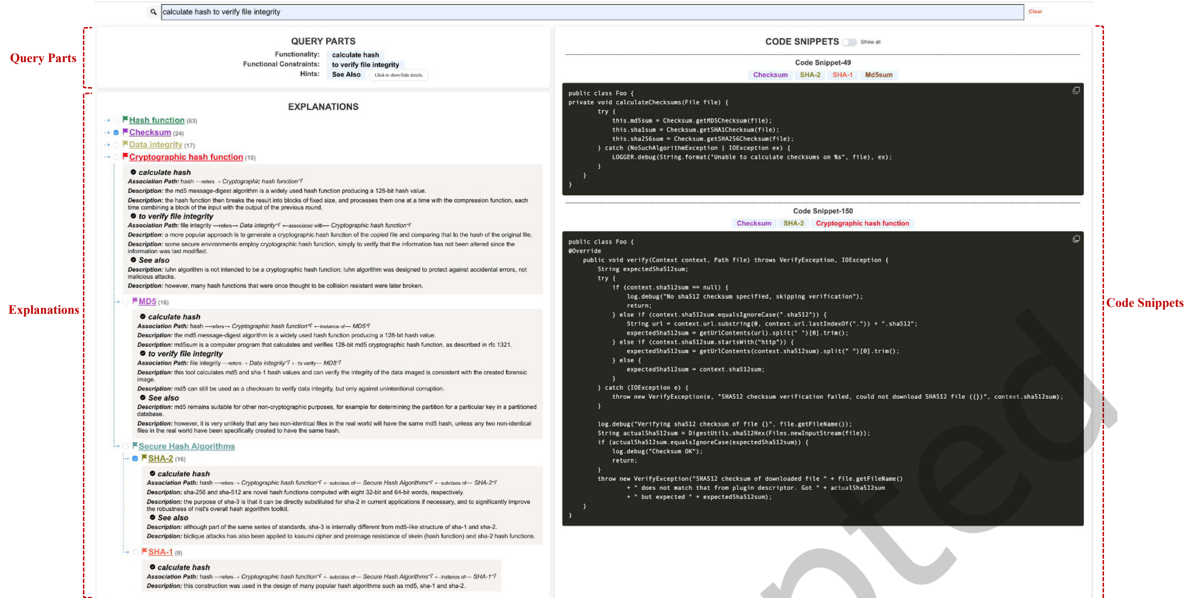


Fig. 5. User Interface of XCoS

tree. “MD5” is a child of “Cryptographic hash function” in the tree since “MD5” is a “instance of” “Cryptographic hash function”, while “SHA-1” is a superchild of “Cryptographic hash function” since “Secure Hash Algorithms” is a “subclass of” “Cryptographic hash function” and “SHA-1” is a “instance of” “Secure Hash Algorithms”. Here, some concepts (e.g., “Secure Hash Algorithms”) are not anchor concepts of code snippets, they appear in the trees as they can help developers better understand the relationships between the anchor concepts. For each anchor concept in the conceptual trees, the association paths ended with it and the related descriptions of the paths are listed below it, categorized by the query parts linking to the paths. The additional suggestions of it is also listed below it with the special query part “See also”. For example, below the anchor concept “MD5”, the association paths and related descriptions are displayed with the two query parts “calculate hash” and “to verify file integrity”, and the additional suggestions are shown with “See also”. Note that, displaying all anchor concepts of code snippets and the corresponding explanations may lead to information overload, we only keep the 15 anchor concepts with the highest relevance to the query. The relevance is measured by averaging the scores (see Sec. 4.5.2) of the related descriptions below the anchor concepts. The **Code Snippets** area lists the candidate code snippets. The anchor concepts of each code snippet are listed on the top of it. For example, the anchor concepts “Checksum”, “SHA-2”, and “Cryptographic hash function” are listed on the top of Code Snippet 150.

The UI provides interaction features based on the links between the contents of different areas. The conceptual trees in **Explanations** area are built based on the anchor concepts of code snippets (the colors of the anchor concepts in the **Explanations** area and the **Code Snippets** area are consistent), they can be naturally used to group and filter the code snippets. For each anchor concept in a conceptual tree, XCoS shows the number of the corresponding related code snippets after the name of the concept. The user can determine whether a concept is relevant to the query by reading the explanations below the concept. The explanations can be folded/unfoled by clicking the name of the concept. Because the explanations are categorized by different query parts, thus using the query parts to control the display of the explanations. The user can click a query part in the **Query Parts** area to show or hide the corresponding explanations. If the user think an anchor concept is relevant to the query, he/she can select the concept by checking the corresponding checkbox in front of it. The **Code Snippets** area

Just Accepted

will be updated accordingly to show only the code snippets that are related to all the currently selected concepts. For example, after reading the explanations the user know that checksum is often used to verify file integrity and SHA-2 is more secure than MD5. He/she can then select "checksum" and "SHA-2" and check all the candidate code snippets related to both the concepts.

6 EVALUATION

The constructed background knowledge graph in offline phase contains 50,472 software development related concepts, 310,451 relationships (85,913 relationships from Wikidata, and 224,538 additional open relationships and association relationships from Wikipedia), and 239,368 concept descriptions. We encapsulate the concepts and relationships in the knowledge graph with a set of data structures (i.e., Python classes) and store/load the concepts/relationships to/from files via object serialization/deserialization (i.e., dump/load APIs in Python pickle package). We do not apply semantic web technology or graph database (e.g., Neo4j), since we do not use complex graph queries and it is more efficient to directly operate on objects in memory. In online phase, we take all the Java code snippets (about 500K Java methods) in the CodeSearchNet [30] dataset as the codebase, and use Elasticsearch to build a code search service.

Based on the background knowledge graph and code search service, we conduct a series of experiments to evaluate the quality of the key steps of XCoS and the effectiveness and usefulness of XCoS by answering the following research questions. All the data and results can be found in our replication package [7].

RQ1 (Quality): What is the intrinsic quality of the results of the key steps of XCoS?

RQ2 (Effectiveness): How effective is XCoS in generating explanations for code search in terms of completeness, conciseness, and readability?

RQ3 (Usefulness): How useful is XCoS in helping developers during code search tasks?

6.1 Quality of Key Steps (RQ1)

We evaluate the quality of the results of the main steps of the approach (see Section 4), including software development related **concept identification**, **concept linking for description**, and **concept linking for code**. These three steps are the key for offline knowledge graph construction and concept linking in codebase. We don't evaluate query scoping because it can only be used online and its quality is reflected in the evaluation of the quality of generated explanations (see Section 6.2).

6.1.1 Protocol. As these steps involve large numbers of results (*i.e.,* software development related concepts, concept linkings of descriptions, and concept linkings of code snippets), we randomly sample some from all the results. For **concept identification**, we randomly sample *S1* software development related concepts; For **concept linking for description**, we randomly sample *S2* descriptions; For **concept linking for code**, we randomly sample *S3* code snippets from the codebase. The sample sizes *S1* and *S2* are both determined to 384 using a statistical sampling method [72], which ensures the estimated accuracy is in 0.05 error margin at 95% confidence level (In practice, the sample sizes can be calculated by using Sample Size Calculator[5]). The sample size *S3* is 50. There are two reasons why we don't apply statistical sampling method for *S3*: 1) examining the linkings in code is much more difficult and time-consuming; 2) the concept linking algorithm for code is the same as for description.

We invite two MS students (who are not involved in this study and have more than 3 years Java development experience) to independently examine the accuracy of the three steps. For **concept identification**, the annotators are asked to examine whether the identified concepts are related to software development. They are provided the names, definitions, and corresponding Wikipedia pages of the concepts to make the binary decisions (*i.e.,*

[5]https://www.surveysystem.com/sscalc.htm

Just Accepted

related to or not related to software development). For **concept linking for description**, the annotators are asked to examine whether the identified mentions and linked concepts for descriptions are correct. They are provided the descriptions in which the mentions are identified, the Wikipedia pages containing the descriptions, and the Wikipedia pages of the linked concepts to make the binary decisions (*i.e.,* correct or not correct). For **concept linking for code**, the annotators are asked to examine whether the identified mentions and linked concepts for code are correct. They are provided the code snippets in which the mentions are identified and the Wikipedia pages of the linked concepts to make the binary decisions (*i.e.,* correct or not correct). We compute Cohen's Kappa [53] to evaluate the inter-rater agreement. For the samples that the two annotators disagree, they discuss and come to a consensus. Based on the consensus annotations, we evaluate the quality of the results of the key steps.

6.1.2 Results. Table 2 shows the evaluation results. The column *Accuracy* is the accuracy after resolving the disagreements. The column *Kappa* is the Kappa inter-rater agreement. The accuracies of the three key steps are 87.2%, 95.2% and 72.3% respectively. The Kappa agreements are all above 0.80, which indicates substantial agreement between the two annotators. It can be seen that we achieve high accuracy in concept identification and description concept linking, which ensure that the quality of the background knowledge graph we construct is high. The accuracy of concept linking for code is also reasonably high, but is much lower than that of concept linking for description.

The same concept linking algorithm performs much better on descriptions than on code snippets, which is mainly due to the following reasons. First, Wikipedia is a high-quality crowdsourced encyclopedic knowledge base, and many of the concepts mentioned in the descriptions we extracted from it are directly marked with hyperlinks and linked to the corresponding concepts. We fully consider this prior information (see Section 4.3.3) when performing mention recognition and candidate matching on the descriptions. But this information is not available for the code, so the candidate concepts of the mentions in code are usually more, which makes the concept linking more difficult. For example, as exemplified in Section 4.3, for the "buffer" in the description, we can just keep the concept of "Data buffer" when matching candidates. But for the "buffer" that appears in the code, we have to keep all possible 26 candidate concepts. Second, the concept and word embedding model we currently use is the pre-trained Wikipedia2Vec model trained on Wikipedia corpus. It may not be entirely appropriate for the calculation of $ctxS(c, ctx)$ for code since the ctx is composed using code identifiers. For example, "path" and "file" often appear together in code and should have a high similarity, but the similarity calculated using the Wikipedia2Vec embeddings is only 0.623. This is because in general corpus, "path" often means "road" and more often appears together with other words like "mountain" and "woods" (their similarities with "path" are 0.731 and 0.720 respectively). In future, we will consider to train an embedding model that can jointly map words, concepts and code identifiers into the same vector space to overcome the problem.

In fact, the incorrectly linked concepts in code snippets do not have a large negative impact on the whole approach. First, the incorrect concepts usually have no conceptual association paths with the query, so they will not appear in the generated explanations. For example, if "buffer" in a candidate code snippet for the query *read file* is linked to "Optical buffer", the incorrect linking does not introduce noise since "Optical buffer" has no associated path with the query. Second, since incorrect linkings may cause some useful code snippets not to be placed in the right group and the user can't find the code they want, we let the code search service return a large number (*e.g.,* 200) of code snippets to avoid this problem.

6.1.3 Summary. Our evaluation shows that the quality of the extracted background knowledge and the concept linkings in codebase is generally high.

Just Accepted

Table 2. Accuracy of the Three Key Steps

Step	Accuracy	Kappa
Concept Identification	87.2%	0.887
Concept Linking for Description	95.2%	0.898
Concept Linking for Code	72.3%	0.834

6.2 Effectiveness for Generating Explanations (RQ2)

We perform an empirical evaluation for assessing the explanations generated by XCoS, similar to previous research [46, 56, 73].

6.2.1 Tasks. Since the queries in the CodeSearchNet dataset are code comments and cannot represent real development tasks and search queries, we construct experimental tasks and corresponding search queries from SO questions. On the other hand, many concerns of developers are not only contained in the question titles, but also reflected in the question bodies [41, 45]. Thus, instead of using SO question titles directly as tasks, we construct them as follows. We select questions tagged with at least one of the six most popular tags (*i.e.,* java, python, javascript, php, c#, android) from the SO data dump [1] as the corpus. In order to ensure the quality of selected questions, we only keep questions with at least one upvote, leading to a corpus with 304,976 questions. Then, we use an NLP tool (spaCy) to analyze the titles of these questions and extract the *VERB OBJECT* structures as candidate functionalities. For example, from the question title "how to read file in Java" we extract "read file". We keep those candidate functionalities that appear more than 3 times, and obtain final code search tasks based on the remaining candidate functionalities.

We define three difficulty levels (*i.e.,* easy, medium, and hard) for tasks based on constraints, respectively indicating no constraints, one functional or nonfunctional constraint, and multiple constraints. We hope that the code search tasks can cover different difficulty levels. The easy tasks are obtained by randomly sampling from the candidate functionalities and the sampled functionalities are taken as search queries directly. For medium and hard task, we randomly sample some candidate functionalities and try to obtain search queries by completing them with constraints. For example, we can add a nonfunctional constraint "in memory efficient way" to "read large file" since memory issues are mentioned in the bodies of some SO questions on reading large file. The constrains are obtained through: 1) SO question titles; 2) SO question bodies; 3) Google's query autocompletion API (https://suggestqueries.google.com/). For the first way, we directly obtain constraints from existing question titles. For the second way, we try to express the constraints (e.g., "in a memory efficient way") with the words/phrases (e.g., "memory efficient") used in the original question bodies. For the third way, since the autocompletion API may suggest multiple autocompletions for a functionality, we first filter out the irrelevant/useless suggestions (e.g., language restrictions like "in python") for our development tasks and then select the first suggestion as the constraint. The above processes are relatively objective thus can avoid introducing biases. Note that during the process, the candidate functionalities that are similar to the existing task topics and those that are not successfully completed are skipped. For each task, we find an SO question that corresponds to its functionality and select one description from the question body as its context. In this way, we obtain 10 code search tasks (4 easy, 4 medium, 2 hard) as shown in Table 3. The 10 queries cover different problem topics (*e.g.,* reading file, converting byte array), constraints, and difficulties, thus can relatively realistically reflect the different code search scenarios. In the above process, we do not sample a fixed number of candidate functionalities (extracted from Stack Overflow queries) through statistical methods (as in RQ1) in advance, because many candidate functionalities cannot become a task (*e.g.,* skipped due to having duplicated problem topic with existing tasks) so that we cannot guarantee

Just Accepted

Table 3. Ratings for 10 Tasks on Completeness, Conciseness and Readability

ID	Task	Difficulty	Completeness				Conciseness				Readability			
			1	2	3	4	1	2	3	4	1	2	3	4
T1	get absolute url	easy	0	1	6	5	0	7	3	2	0	2	4	6
T2	draw circle	easy	0	0	6	6	0	1	2	9	0	0	4	8
T3	add cookie	easy	0	0	5	7	0	1	9	2	0	0	5	7
T4	post json	easy	0	1	2	9	0	1	5	6	0	0	3	9
T5	read XML using XPath	medium	0	2	7	3	0	1	8	3	0	0	5	7
T6	get icon from website	medium	0	2	5	5	0	2	7	3	0	2	3	7
T7	convert byte array to BASE64	medium	0	2	6	4	0	4	6	2	0	2	5	5
T8	read large file in memory efficient way	medium	0	0	2	10	0	0	6	6	0	0	0	12
T9	check url for malware in a safe way	hard	0	0	3	9	0	2	5	5	0	0	3	9
T10	calculate hash to verify file integrity and least vulnerable to attacks	hard	1	0	4	7	0	0	4	8	0	0	3	9
Sum			1	8	46	65	0	19	55	46	0	6	35	79

Just Accepted

how many tasks can be obtained from the sampled functionalities. To this end, we keep sampling the candidate functionalities until 10 tasks are obtained.

6.2.2 Protocol. We invite 12 MS students (experienced in Java development) to evaluate the quality of the explanations generated by XCoS. Their programming expertise is assessed through a survey administered to 50 graduate students. The 12 most experienced students are selected and they have at least 3 years's Java development experience. The participants are from the same institution as the authors but have no academic relationship with the authors. In addition, they are not involved in this work before invited to participant in the experiments, thus have no potential conflicts of interest. For each task we generate explanations by searching the query of the task in XCoS. The participants are asked to evaluate the generated explanations in terms of completeness, conciseness, and readability on a 4-points Likert scale [40] (1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree) by the following statements.

1) **Completeness**. The explanations contain all the necessary information for explaining search results.

2) **Conciseness**. The explanations contain no unnecessary or redundant information for explaining search results.

3) **Readability**. The explanations are well-organized and easy to understand.

Note that in order to reduce bias, the second statement is phrased negatively to maintain the interpretation of the answers similar to all three statements. After the participants finish the evaluation, we ask them to explain the low ratings (1 or 2).

6.2.3 Results. The results of the ratings for each task are shown in Table 3 and the ratings for the three statements across all tasks are shown in Figure 6. For *completeness* of XCoS, **54.2**% of the answers are 4 (agree), **38.3**% are 3 (somewhat agree), **6.7**% are 2 (somewhat disagree), and **0.8**% are 1 (disagree). For *conciseness* of XCoS, **38.3**% of the answers are 4 (agree), **45.8**% are 3 (somewhat agree), **15.8**% are 2 (somewhat disagree), and there are no 1 (disagree) answers. For *readability* of XCoS, **65.8**% of the answers are 4 (agree), **29.2**% are 3 (somewhat agree), **5.0**% are 2 (somewhat disagree), and there are no 1 (disagree) answers. We use one sample T-test [68] to verify the statistical significance of the difference between the participants' ratings and random ratings. The null hypothesis is that the ratings for completeness, conciseness, and readability are random and the mean of the ratings for each

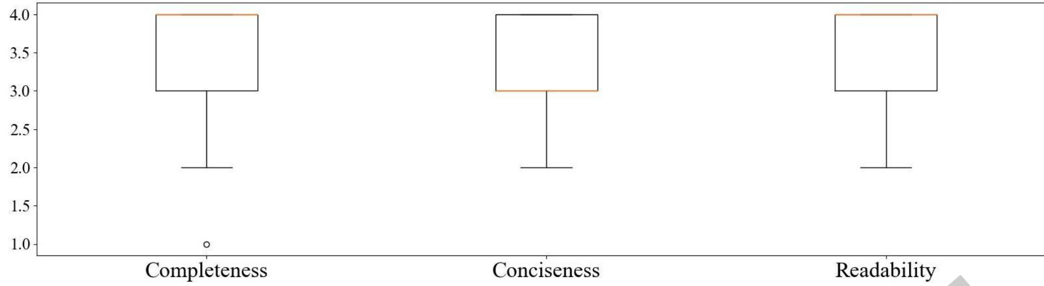


Fig. 6. Ratings of Completeness, Conciseness and Readability of Explanations

property is 2.5. The results show that for each property the statistical difference is significant ($p << 0.01$), so we reject the null hypothesis.

For *conciseness*, there are 19 answers 2 (somewhat disagree). The reasons given by the participants are mainly that simple tasks usually do not require much explanation, and the explanation information we provide is redundant for these tasks. For example, the explanations of "circle" for **T2** is unnecessary.

6.2.4 Summary. The participants consider that the explanations generated by XCoS are complete, concise, and readable.

6.3 Usefulness of XCoS (RQ3)

We evaluate the usefulness of XCoS in code search tasks, that is, choosing the most suitable code snippet for a given task. Note that, we focus on helping developers by providing explanations for search results, instead of proposing a stronger search algorithm. To this end, we evaluate the usefulness of XCoS through whether the explorations can help the developers better complete code search tasks. The effectiveness of query scoping can bee also reflected in this evaluation, since the explanations generated by XCoS are organized by the corresponding query parts, which greatly influence the choosing process of code snippets (see Section 5).

6.3.1 Baseline. For comparison, we implement a baseline that directly displays the results returned by the code search service. The baseline has a UI similar to XCoS, but only shows the code snippets without explanation information. Other state-of-the-art code search techniques are not compared, since the key to comparison is whether there are explanations for the search results returned by the same search service, instead of comparing the effectiveness of different search techniques. We build the search service using Elasticsearch instead of state-of-the-art code search techniques, as most practical code search services are built based on information retrieval based search engines (see the insight **I0** of our survey) and Elasticsearch is the most popular one of such engines.

6.3.2 Protocol. We use the same tasks in Section 6.2 to evaluate the usefulness of XCoS, and randomly divide the tasks into two roughly equivalent group T_A and T_B. In each group, the number of tasks in easy, medium, hard level are 2, 2, 1 respectively. We invite 14 MS students (not involved in RQ2) with 1-5 years' Java programming experience to participate this experiment. Their programming expertise is assessed through a survey distributed to 50 graduate students and they are divided into three experience levels (*i.e.,* beginners, intermediate, advanced) according to the survey. The invited students have different programming experience, which ensure the diversity of the participants. In addition, most of them (except beginners) have participated in the development of some medium/large scale projects (> 10,000 LOC), and some participants have professional programming experience such as working or interning in some companies for a period of time. We further divide them into two "equivalent" groups (G_A and G_B) and each group with 2 beginner, 3 intermediate, and 2 advanced.

Just Accepted

Table 4. Average Accuracy, Time and Ranking

	Average Accuracy	**Average Time**	**Average Ranking**
XCoS	74.3%	172.9s	32.3
Baseline	52.9%	147.7s	16.3

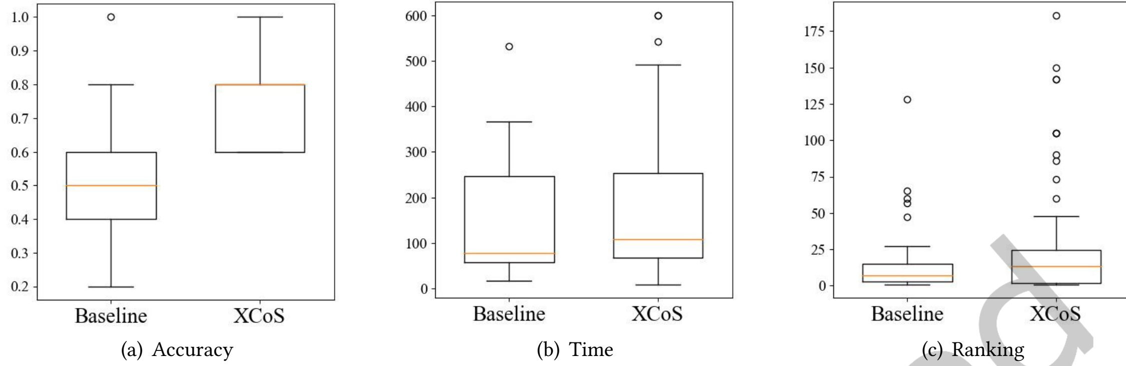
We ask the participants to complete code search tasks with XCoS and the baseline by adopting a balanced treatment distribution for the groups. Participants in G_A are asked to complete the tasks in T_A with XCoS and complete the tasks in T_B with the baseline. Conversely, participants in G_B are asked to complete the tasks in T_B with XCoS and complete the tasks in T_A with the baseline. Overall, each participant is asked to complete all the 10 tasks, 5 with XCoS and 5 with the baseline. Before starting the tasks, we give these participants a 20-minute training session and provide some examples to familiarize them with XCoS and the baseline. When completing the tasks with baseline, the participants are allowed to use external search engines like Google and Bing. The reason is that in practice some developers will consult these external resources to check whether the code snippets meet their requirements (see the results of our developer survey in Section 3). For each participant, the tasks are done by interleaving XCoS and the baseline. At the same time, the order in which tasks are completed is completely random and has nothing to do with difficulty.

For each task, a participant is asked to select the most suitable code snippet from search results and submit it as the answer. Moreover, we ask the participants to record the answer's ranking in the search results and the reason for choice. If participants cannot find a suitable code snippet, they can submit an empty answer. The participants have a time limit of 15 minutes on each task, after which they are considered to submit an empty answer. When all participants have completed their tasks, we assess the correctness of the answers. Note that it is impossible to build an oracle to specify a unique correct answer for each task, as the code base contains about 500K code snippets and a task may be correctly completed by different code snippets. To this end, for each task we mix together the answers submitted by all participants (completed with XCoS and the baseline), and then two authors independently assess the correctness of each answer comparatively. A correct answer must satisfy both functionality and constraints of the task query, but is allowed for redundant parts (such as a few extra operations). We compute Cohen's Kappa [53] to evaluate the inter-rater agreement between the two authors. For the answers that the two annotators disagree, a third author resolves the conflicts based on the major-win strategy. Finally, we also conduct interviews to collect participants' feedback on XCoS. The participants are asked to answer whether XCoS is useful and why.

Note that, we do not use the ranking-based metrics (*e.g.,* top-k or ranked position) to evaluate XCoS, as the ranked lists are returned by the code service and the rankings of the correct answers are meaningless for evaluating XCoS. Through the explanations generated by XCoS, users can quickly group and filter the code snippets based on their requirements or concerns (see Section 5), rather than must browse and check the results in the ranked list linearly (*i.e.,* check the searched code snippets one-by-one). Therefore, the original ranking information cannot reflect the effectiveness of completing tasks with XCoS. For example, for the query "calculate hash to verify file" shown in Figure 5, when the users check the checkboxes in front of the concepts "Checksum" and "SHA-2", the **Code Snippets** area will update and only display the code snippets containing the two concepts. Then, the users may directly choose the first code snippet (*i.e.,* Code Snippet-49) in the updated area as the answer. Although the code snippet is ranked 49-th in the original ranked list, the users can skip the 48 code snippets ranked in front of it and quickly choose it when determining to use "Checksum" and "SHA-2".

6.3.3 Results. Figure 7 and Table 4 show the accuracy (*i.e.,* the ratio that the right code snippets were selected by a participant for a task group), the completion time, and the answer's ranking over the tasks when completed with XCoS and without XCoS respectively. The Kappa agreement is 0.82, which indicates substantial agreement

Just Accepted



(a) Accuracy (b) Time (c) Ranking

Fig. 7. Usefulness Evaluation for Code Search Tasks

Table 5. Accuracy Analysis

	Easy	Medium	Hard	Beginner	Intermediate	Adavanced
XCoS	96.4%	75.0%	28.5%	65.0%	76.7%	80.0%
Baseline	82.1%	46.4%	7.1%	50.0%	53.3%	55.0%

between the two annotators. Here, because 51 (18 with XCoS and 33 with the baseline) of the total 140 submitted answers are assessed as incorrect, we exclude them when counting the time and ranking. According to the results, compared to without XCoS, using XCoS the participants complete the tasks 40.5% more accurately (21.4% on average) in comparable time by choosing the answers from a wider range of code snippets. We use Welch's T-test [83] for verifying the statistical significance of the differences. The improvement of accuracy of XCoS is statistically significant (p = 0.007, *effect size* = 1.503), while the extra time spent is not significant (p = 0.417, *effect size* = 0.176). In addition, Table 5 gives detailed accuracy from different aspects. It shows that XCoS can improve accuracy for tasks of different difficulty levels and developers with different programming experience levels.

The improvement of XCoS in accuracy mainly comes from two aspects. First, users can accurately judge which concepts are related to the query by reading the association paths and descriptions we provide. For example, for **T10**, some participants who use XCoS write that "From the provided information, it is known that SHA-512 is a more secure hash algorithm", and for this reason they choose the code snippets using SHA-512 instead of MD5. Second, our grouping of code snippets allows users to efficiently filter out a lot of top-ranked but irrelevant code snippets, allowing them to find the low-ranked correct answers. By checking the reasons given by the participants for their answers, we find that many mention that they filter the code snippets using the concept checkboxs while using XCoS. Without the grouping and filtering, some lower-ranked correct answers might not be discovered by users at all. This can also be seen from the ranking data in Figure 7 and Table 4, for tasks done correctly with XCoS the answers are chosen from a wider range of rankings.

Of course, we also find some problems. For few beginner participants, the explanations we provide do not improve the likelihood that they will choose the correct answer. We analyze the reasons they give for their answers and find that although we provide a lot of explanations, they still lack the relevant background knowledge to judge whether a code snippet is related to the query. For example, for **T10**, a beginner participant choose an incorrect code snippet because "I filter the code snippets using both *Hash function* and *Data integrity* and choose

Just Accepted

one from the remaining code snippets". But in fact this code snippet has nothing to do with the functionality of the task.

After analyzing the feedback, we find that all participants think our tool to be useful in most cases. The usefulness is mainly reflected in the following facts: it can quickly help filter out irrelevant code and greatly reduce the number of code snippets that need to be examined; it can help familiarize themselves with some unknown concepts and rationales, and can greatly reduce the time to seek the information from external resources (*e.g.,* Google); it can help them focus on some key information, and can significantly improve the confidence of choice. Some participants mention that the explanations provided by XCoS are not very helpful on simple tasks that do not need too much background knowledge.

6.3.4 Summary. Our approach significantly improves the accuracy and confidence for code search tasks and is considered useful.

6.4 Threats to Validity

Data sampling and data annotation are involved in RQ1 and RQ3. Thus common threats to the internal validity include randomness of the sampling and subjective judgment of the involved annotators. To minimize such threats, we follow commonly used sampling and data analysis techniques, such as involving multiple annotators and conflict resolution steps and reporting agreement coefficients. A threat to the external validity is that we only use a limited number of tasks (*i.e.,* 10 tasks) in the evaluation. Therefore, our findings may not be generalizable. To minimize such threats, we sample the tasks from Stack Overflow and try to ensure the diversity of tasks as much as possible by covering different topics and difficulties.

7 RELATED WORK

Early research on code search mainly use information retrieval technologies to establish the association between code and query [15, 51]. Thanks to the efficiency, the information retrieval technologies are often used to build Internet-scale code search engines [24]. Some studies take into account code characteristics by organizing code into directed graphs and transform a code search task into a graph search task. For example, McMillan *et al.* [54] propose Portfolio, a code search engine that combines keyword matching with PageRank and SAN scores based on the API call graph to return a chain of functions. Li *et al.* [39] further propose a relation-based code search framework for JavaScript RACS, which can structurally match the action graph parsed from the query with the call graph parsed from the code to improve the accuracy. However, these works do not consider the conceptual association between query and code, nor provide explanations for the search results.

Recently, some studies use deep learning models to match query with code at the semantic level [22, 25, 26, 38, 47, 69, 78]. Gu *et al.* [25] propose DeepCS, which relies on a LSTM-based deep neural network to encode the code snippet and query into a same vector space and use the cosine similarity between query vector and the code vector as the matching score. Similarly, Sachdev *et al.* [69] also propose a code search tool named NCS, which combines Word2Vec and TF-IDF to generate vector representations for code snippets and query and calculates the distance between their vectors as a relevance score. However, these deep learning based approaches cannot provide explainability.

An important factor affecting the accuracy of code search is the lexical gap between query and code. Many existing studies focus on query expansion and reformulation [28, 29, 44, 50, 52, 62, 79] to solve the lexical gap problem. Hill *et al.* [29] reformulate queries with natural language phrasal representations of method signatures. Haiduc *et al.* [28] propose a machine learning based approach to reformulate queries, which can automatically recommend a reformulation strategy based on the query properties. Lu *et al.* [50] also extend queries with synonyms based on WordNet to improve the hit rate of code search. Lv *et al.* [52] propose CodeHow that can expand the query with potential relevant APIs and retrieve code by considering both text similarity and potential

Just Accepted

APIs. Wang *et al.* [79] propose a learning based approach to mine domain glossary and show that the mined glossary can be used to expand search queries. Recently, Liu *et al.* [44] propose NQE model to predict the keywords related to the query from the corpus, so as to expand the query. However, these methods cannot bridge background knowledge gap between query and code.

Feature location, concept location, and bug localization are special code search scenarios that aim to retrieve code in a single project. According to the types of used analysis, the proposed approaches can be classified into different categories [19], including textual [16, 33, 55], structural [21, 37, 65], historical [34, 81, 84] and dynamic types. Besides applying individual analysis, recent approaches [20, 81] combine different types of analysis to compensate for the limitations.

There are some works focusing on interactive and explanatory code search [48, 49, 80]. Wang *et al.* [80] propose an approach to improve feature location and code search with multi-faceted interactive exploration. This approach automatically extracts and mines multiple syntactic and semantic facets from candidate program elements. Lu *et al.* [49] propose INQRES, a code search tool that helps user improve queries in an interactive way. Liu *et al.* [48] propose CodeNuance, which utilizes differencing and visualization to solve 1) many pieces of online code are largely similar but subtly different; 2) several pieces of code may form complex relations through their differences. These methods lack explanatory information on the background knowledge level of the search results.

Explainable artificial intelligence (XAI) is a trending research direction and some researchers have explored the explainability problem for software engineering, especially for software defect prediction. There are some research works to make the defect prediction models more practical, explainable, and actionable [74, 75], including investigating practitioners' needs of explainability of defect prediction models [32], developing line-level just-in-time defect prediction approaches and leveraging model-agnostic techniques (*e.g.,* LIME) to improve the explainability [23, 59–61, 82], and applying the more explainable models to software quality assurance [63].

8 CONCLUSIONS AND FUTURE WORK

In this paper, we propose an approach (called XCoS) to support explainable code search based on query scoping and knowledge graph. It can bridge the background knowledge gap between the query and the code. Our evaluation confirms the intrinsic quality of the constructed background knowledge graph and the generated explanations for search results. It also shows the usefulness of XCoS in code search tasks. In the future, we will improve and extend the approach from several aspects. First, we will train an embedding model that simultaneously considers the semantic relationships of concepts, words, and code identifiers. Second, we will incorporate more knowledge resources such as Stack Overflow to construct a more informative background knowledge graph. Third, we will design our UI interaction to be easier to use based on user feedback. Fourth, we will explore the possibility of using the background knowledge graph to directly improve code search algorithms.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant 61972098.

REFERENCES

[1] 2021. *Stack Overflow data dump version from March 4, 2021.* Retrieved September 4, 2021 from https://archive.org/download/stackexchange/
[2] 2021. *Wikidata data dump version from November 24, 2021.* Retrieved November 24, 2021 from https://dumps.wikimedia.org/wikidatawiki/entities/
[3] 2021. *Wikipedia data dump version from December 20, 2021.* Retrieved December 20, 2021 from https://dumps.wikimedia.org/enwiki/
[4] 2022. *Elasticsearch.* Retrieved March 5, 2022 from https://www.elastic.co/elasticsearch/
[5] 2022. *javalang.* Retrieved March 5, 2022 from https://github.com/c2nes/javalang
[6] 2022. *neuralcoref.* Retrieved March 5, 2022 from https://github.com/huggingface/neuralcoref

Just Accepted

[7] 2022. *Replication Package.* Retrieved March 5, 2022 from https://xcos-replicationpackage.github.io/
[8] 2022. *searchcode.* Retrieved March 5, 2022 from https://searchcode.com/
[9] 2022. *spaCy.* Retrieved March 5, 2022 from https://spacy.io/
[10] 2022. *Sprial.* Retrieved March 5, 2022 from https://github.com/casics/spiral
[11] 2022. *Stack Overflow Question 13269606.* Retrieved March 5, 2022 from https://stackoverflow.com/questions/13269606
[12] 2022. *Wikidata.* Retrieved March 5, 2022 from https://www.wikidata.org/
[13] 2022. *Wikipedia.* Retrieved March 5, 2022 from https://en.wikipedia.org/
[14] 2022. *Wikipedia2Vec.* Retrieved March 5, 2022 from https://wikipedia2vec.github.io/wikipedia2vec/
[15] Sushil Krishna Bajracharya, Trung Chi Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Videira Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. ACM, 681–682.
[16] Lauren R. Biggers, Cecylia Bocovich, Riley Capshaw, Brian P. Eddy, Letha H. Etzkorn, and Nicholas A. Kraft. 2014. Configuring latent Dirichlet allocation based feature location. *Empir. Softw. Eng.* 19, 3 (2014), 465–500. https://doi.org/10.1007/s10664-012-9224-x
[17] Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. 2019. Requirements classification with interpretable machine learning and dependency parsing. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 142–152.
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.
[19] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process.* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567
[20] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. 2013. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empir. Softw. Eng.* 18, 2 (2013), 277–309. https://doi.org/10.1007/s10664-011-9194-4
[21] Brian P. Eddy, Nicholas A. Kraft, and Jeff Gray. 2018. Impact of structural weighting on a latent Dirichlet allocation-based feature location technique. *J. Softw. Evol. Process.* 30, 1 (2018). https://doi.org/10.1002/smr.1892
[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.
[23] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *Proceedings of 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 608–620. https://doi.org/10.1145/3524842.3528452
[24] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. 2009. Internet-Scale Code Search. In *Proceedings of 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. 49–52. https://doi.org/10.1109/SUITE.2009.5070022
[25] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 933–944.
[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ
[27] Samir Gupta, Sana Malik, Lori L. Pollock, and K. Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 3–12.
[28] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 842–851.
[29] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 524–527.
[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
[31] Ishrar Hussain, Leila Kosseim, and Olga Ormandjieva. 2008. Using linguistic knowledge to classify non-functional requirements in SRS documents. In *Natural Language and Information Systems: 13th International Conference on Applications of Natural Language to*

Just Accepted

Information Systems, NLDB 2008 London, UK, June 24-27, 2008 Proceedings 13. Springer, 287–298.
[32] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John C. Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *Proceedings of 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 432–443. https://doi.org/10.1109/MSR52588.2021.00055
[33] Huzefa H. Kagdi, Malcom Gethers, and Denys Poshyvanyk. 2013. Integrating conceptual and logical couplings for change impact analysis in software. *Empir. Softw. Eng.* 18, 5 (2013), 933–969. https://doi.org/10.1007/s10664-012-9233-9
[34] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Trans. Software Eng.* 39, 11 (2013), 1597–1610. https://doi.org/10.1109/TSE.2013.24
[35] Keshav Kolluru, Vaibhav Adlakha, Samarth Aggarwal, Mausam, and Soumen Chakrabarti. 2020. OpenIE6: Iterative Grid Labeling and Coordination Analysis for Open Information Extraction. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 3748–3761.
[36] Phong Le and Ivan Titov. 2018. Improving Entity Linking by Modeling Latent Relations between Mentions. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 1595–1604.
[37] Bixin Li, Xiaobing Sun, and Hareton Leung. 2012. Combining concept lattice with call graph for impact analysis. *Adv. Eng. Softw.* 53 (2012), 1–13. https://doi.org/10.1016/j.advengsoft.2012.07.001
[38] Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. CodeRetriever: Unimodal and Bimodal Contrastive Learning. *arXiv preprint arXiv:2201.10866* (2022).
[39] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, November 13-18, 2016, Seattle, WA, USA*. ACM, 690–701.
[40] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).
[41] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. 2019. Pattern-based mining of opinions in Q&A websites. In *Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 548–559.
[42] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. 2019. Pattern-based mining of opinions in Q&A websites. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 548–559.
[43] Charles X. Ling and Chenghui Li. 1998. Data Mining for Direct Marketing: Problems and Solutions. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98), New York City, New York, USA, August 27-31, 1998*, Rakesh Agrawal, Paul E. Stolorz, and Gregory Piatetsky-Shapiro (Eds.). AAAI Press, 73–79. http://www.aaai.org/Library/KDD/1998/kdd98-011.php
[44] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural query expansion for code search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 29–37.
[45] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. 2021. API-Related Developer Information Needs in Stack Overflow. *IEEE Transactions on Software Engineering* (2021).
[46] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, August 26-30, 2019, Tallinn, Estonia*. 120–130.
[47] Shangqing Liu, Xiaofei Xie, Lei Ma, Jing Kai Siow, and Yang Liu. 2021. GraphSearchNet: Enhancing GNNs via Capturing Global Dependency for Semantic Code Search. *CoRR* abs/2111.02671 (2021).
[48] Wenjian Liu, Xin Peng, Zhenchang Xing, Junyi Li, Bing Xie, and Wenyun Zhao. 2018. Supporting exploratory code search with differencing and visualization. In *Proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 300–310.
[49] Jinting Lu, Ying Wei, Xiaobing Sun, Bin Li, Wanzhi Wen, and Cheng Zhou. 2018. Interactive Query Reformulation for Source-Code Search With Word Relations. *IEEE Access* 6 (2018), 75660–75668.
[50] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *Proceedings of 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. IEEE Computer Society, 545–549.
[51] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2008. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proceedings of 15th Working Conference on Reverse Engineering, WCRE 2008, October 15-18, 2008, Antwerp, Belgium*. IEEE Computer Society, 155–164.

Just Accepted

[52] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 260–270.

[53] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[54] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 111–120.

[55] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. 2017. Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks. *ACM Trans. Softw. Eng. Methodol.* 26, 1 (2017), 3:1–3:45. https://doi.org/10.1145/3078841

[56] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and Vijay Shanker. 2013. Automatic Generation of Natural Language Summaries for Java Classes. In *21st IEEE International Conference on Program Comprehension (ICPC'13)*. IEEE, San Francisco, USA, 23–32.

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

[58] Rahul Pandita, Kunal Taneja, Laurie A. Williams, and Teresa Tung. 2016. ICON: Inferring Temporal Constraints from Natural Language API Descriptions. In *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 378–388. https://doi.org/10.1109/ICSME.2016.59

[59] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *Proceedings of 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 369–379. https://doi.org/10.1109/MSR52588.2021.00049

[60] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Trans. Software Eng.* (2022).

[61] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *Proceedings of 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 407–418. https://doi.org/10.1109/ASE51524.2021.9678763

[62] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. 2019. Automatic query reformulation for code search using crowdsourced knowledge. *Empir. Softw. Eng.* 24, 4 (2019), 1869–1924.

[63] Dilini Rajapaksha, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Christoph Bergmeir, John Grundy, and Wray L. Buntine. 2022. SQAPlanner: Generating Data-Informed Software Quality Improvement Plans. *IEEE Trans. Software Eng.* 48, 8 (2022), 2814–2835. https://doi.org/10.1109/TSE.2021.3070559

[64] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990.

[65] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2011. Using structural and textual information to capture feature coupling in object-oriented software. *Empir. Softw. Eng.* 16, 6 (2011), 773–811. https://doi.org/10.1007/s10664-011-9159-7

[66] Knut Magne Risvik, Tomasz Mikolajewski, and Peter Boros. 2003. Query Segmentation for Web Search. In *Proceedings of the Twelfth International World Wide Web Conference - Posters, WWW 2003, Budapest, Hungary, May 20-24, 2003*, Irwin King and Tamás Máray (Eds.). http://www2003.org/cdrom/papers/poster/p052/xhtml/querysegmentation.html

[67] Stephen E. Robertson and Steve Walker. 1994. Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. In *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, W. Bruce Croft and C. J. van Rijsbergen (Eds.). ACM/Springer, 232–241.

[68] Amanda Ross and Victor L Willson. 2017. One-sample T-test. In *Basic and advanced statistical tests*. Springer, 9–12.

[69] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 31–41.

[70] Gerard Salton and Chris Buckley. 1988. Term-Weighting Approaches in Automatic Text Retrieval. *Inf. Process. Manag.* 24, 5 (1988), 513–523.

[71] Janice Singer, Timothy C. Lethbridge, Norman G. Vinson, and Nicolas Anquetil. 1997. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*. IBM, 21.

[72] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of Survey Sampling*. Vol. 15. Springer Science & Business Media.

[73] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. Antwerp,

Just Accepted

Belgium, 43–52.

[74] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2021. Actionable Analytics: Stop Telling Me What It Is; Please Tell Me What To Do. *IEEE Softw.* 38, 4 (2021), 115–120. https://doi.org/10.1109/MS.2021.3072088

[75] Chakkrit Kla Tantithamthavorn and Jirayus Jiarpakdee. 2021. Explainable AI for Software Engineering. In *Proceedings of 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1–2. https://doi.org/10.1109/ASE51524.2021.9678580

[76] Christoph Treude, Ohad Barzilay, and Margaret-Anne D. Storey. 2011. How do programmers ask and answer questions on the web?. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 804–807.

[77] Christoph Treude, Martin P Robillard, and Barthélémy Dagenais. 2014. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering* 41, 6 (2014), 565–581.

[78] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 13–25.

[79] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. 2019. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 97–108.

[80] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 762–771.

[81] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: putting them together for improved bug localization. In *Proceedings of 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 53–63. https://doi.org/10.1145/2597008.2597148

[82] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Trans. Software Eng.* 48, 5 (2022), 1480–1496. https://doi.org/10.1109/TSE.2020.3023177

[83] Bernard L Welch. 1947. The generalization of Student's problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[84] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2016. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. *IEEE Trans. Software Eng.* 42, 4 (2016), 379–402. https://doi.org/10.1109/TSE.2015.2479232

Just Accepted