Noname manuscript No.
(will be inserted by the editor)

Automatic Reformulation of Query for Code Search using Crowdsourced Knowledge

**Mohammad M. Rahman · Chanchal K. Roy ·
David Lo**

the date of receipt and acceptance should be inserted later

Abstract Traditional code search engines (e.g., Krugle) often do not perform well with natural language queries. They mostly apply keyword matching between query and source code. Hence, they need carefully designed queries containing references to relevant APIs for the code search. Unfortunately, preparing an effective search query is not only challenging but also time-consuming for the developers according to existing studies. In this article, we propose a novel query reformulation technique–RACK–that suggests a list of relevant API classes for a natural language query intended for code search. Our technique offers such suggestions by exploiting keyword-API associations from the questions and answers of Stack Overflow (i.e., crowdsourced knowledge). We first motivate our idea using an exploratory study with 19 standard Java API packages and 344K Java related posts from Stack Overflow. Experiments using 175 code search queries randomly chosen from three Java tutorial sites show that our technique recommends correct API classes within the Top-10 results for 83% of the queries, with 46% mean average precision and 54% recall, which are 66%, 79% and 87% higher respectively than that of the state-of-the-art. Reformulations using our suggested API classes improve 64% of the natural language queries and their overall accuracy improves by 19%. Comparisons with three state-of-the-art techniques demonstrate that RACK outperforms them in the query reformulation by a statistically significant margin. Investigation using three web/code search engines shows that our technique can significantly improve their results in the context of code search.

Keywords Code search, query reformulation, keyword-API association, crowdsourced knowledge, Stack Overflow

Mohammad Masudur Rahman, Chanchal K. Roy
University of Saskatchewan, Canada
E-mail: {masud.rahman, chanchal.roy}@usask.ca

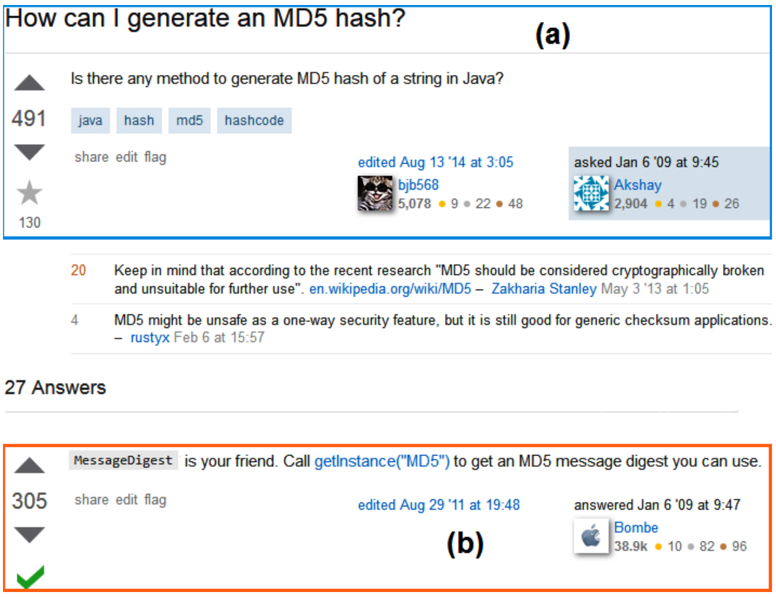
David Lo
Singapore Management University, Singapore
E-mail: davidlo@smu.edu.sg

1 Introduction

Studies show that software developers on average spend about 19% of their development time in web search. On the web, they frequently look for relevant code snippets for their tasks [16]. Online code search engines–*Open Hub, Koders, GitHub* and *Krugle*– index thousands of large open source projects, and these projects are a potential source for such code snippets [46]. However, these traditional code search engines mostly employ keyword matching. Hence, they often do not perform well with unstructured natural language (NL) queries due to vocabulary mismatch between NL query and source code [14]. They retrieve code snippets based on lexical similarity between a search query and the project source code. That means, these engines require the queries to be carefully designed by the users and to contain relevant API references. Unfortunately, preparing an effective search query that contains information on relevant APIs is not only challenging but also time-consuming for the developers [16, 38]. A previous study [38] also suggested that on average, developers regardless of their experience levels performed poorly in coming up with good search terms for code search. Thus, an automated technique that complements a natural language query with a list of relevant API classes or methods (i.e., search-engine friendly query) can greatly assist the developers in performing the code search. Our paper addresses this particular research problem–*query reformulation* with *relevant API classes*–by exploiting the crowdsourced knowledge stored at Stack Overflow programming Q & A site.

Existing studies on API recommendation accept one or more natural language queries, and return relevant API classes and methods by mining feature request history and API documentations [72], large code repositories [84], API invocation graphs [22], library usage patterns [71], code surfing behaviour of the developers and API invocation chains [46]. McMillan et al. [46] first propose *Portfolio* that recommends relevant API methods for a given code search query and demonstrates their usage from a large codebase. Chan et al. [22] improve upon *Portfolio* by employing further sophisticated graph-mining and textual similarity techniques. Thung et al. [72] recommend relevant API methods to assist the implementation of an incoming feature request. Although all these techniques perform well in different working contexts, they share a set of limitations and thus fail to address the research problem of our interest. First, each of these techniques [22, 46, 72] exploits lexical similarity measure (e.g., Dice's coefficients [22]) for candidate API selection. This warrants that the search query should be carefully prepared, and it should contain keywords similar to the API names. In other words, the developer should or must possess a certain level of experience with the target APIs to actually use these techniques [13]. Second, API names and search queries are generally provided by different developers who may use different vocabularies to convey the same concept [39]. Furnas et al. [25] named this the *vocabulary mismatch problem.* Lexical similarity based techniques often suffer from this problem. Hence, the performance of these techniques is not only limited but also subject to the identifier naming practices adopted in the codebase under study. We thus need a technique that overcomes the above limitations, and recommends relevant or appropriate APIs for natural language queries from a wider vocabulary.

One possible way to tackle the above challenges is to leverage the knowledge or experience of a large technical crowd on the usage of particular API classes and methods. Let us consider a natural language query– *"Generating MD5 hash of a Java*


How can I generate an MD5 hash?

(a)

Is there any method to generate MD5 hash of a string in Java?

java hash md5 hashcode

491

130

share edit flag

edited Aug 13 '14 at 3:05
bjb568
5,078 ● 9 ● 22 ● 48

asked Jan 6 '09 at 9:45
Akshay
2,904 ● 4 ● 19 ● 26

20 Keep in mind that according to the recent research "MD5 should be considered cryptographically broken and unsuitable for further use". en.wikipedia.org/wiki/MD5 – Zakharia Stanley May 3 '13 at 1:05

4 MD5 might be unsafe as a one-way security feature, but it is still good for generic checksum applications. – rustyx Feb 6 at 15:57

27 Answers

`MessageDigest` is your friend. Call getInstance("MD5") to get an MD5 message digest you can use.

305

share edit flag

edited Aug 29 '11 at 19:48

(b)

answered Jan 6 '09 at 9:47
Bombe
38.9k ● 10 ● 82 ● 96

Fig. 1: An example of (a) Stack Overflow question and (b) its accepted answer

string." Now, we mine thousands of Q & A posts from Stack Overflow that discuss relevant APIs for this task and then recommend APIs from them. For instance, the Q & A example in Fig. 1 discusses on how to generate an MD5 hash (Fig. 1-(a)), and the accepted answer (Fig. 1-(b)) suggests that `MessageDigest` API should be used for the task. Such usage of the API is also recommended by at least 305 technical users from Stack Overflow, which validates the appropriateness of the usage. Our approach is thus generic, language independent, project insensitive, and at the same time, it overcomes the vocabulary mismatch problem suffered by the past studies. One can argue in favour of Google which is often used by the developers for searching code on the web. Unfortunately, recent study [59] shows that developers need to spend more efforts (i.e., two times) in code search than in web search while using Google search engine. In particular, they need to reformulate their queries more frequently and more extensively for the code search. Such finding suggests that the general-purpose web search engines (e.g., Google) might be calibrated for the web pages only, and they perform sub-optimally with the source code, especially due to vocabulary mismatch issues [29, 32]. Thus, automatic tool supports in the query formulation for code search is still an open research problem that warrants further investigation.

In this paper, we propose a query reformulation technique–RACK–that exploits the associations between query keywords and different API classes used in Stack Overflow and translates a natural language query intended for code search into a set of relevant API classes. First, we motivate our idea of using crowdsourced knowledge for API recommendation with an exploratory study where we analyse 172,043 questions and their accepted answers from Stack Overflow. Second, we construct a keyword-API mapping database using these questions and answers where the keywords (i.e., programming requirements) are extracted from questions and the APIs (i.e., programming solutions) are collected from the corresponding accepted answers. Third, we propose an API recommendation technique that employs three heuristics on keyword-API associations and recommends a ranked list of API classes for a given query for its reformulation. The baseline idea is to capture and learn the responses from millions of technical users (e.g., developers, researchers, programming hobbyists) for different programming problems, and then exploit them for relevant API suggestion. Our technique (1) does not rely on the lexical similarity between query and source code of projects for API selection, and (2) addresses the vocabulary mismatch problems by using a large vocabulary (i.e., 20K) produced by millions of users of Stack Overflow. Thus, it has a great potential for overcoming the challenges faced with the past studies.

An exploratory study with 172,043 Java related Q & A threads (i.e., question + accepted answer) from Stack Overflow shows that (1) each answer uses at least two different API classes on average (RQ_1), and (2) about 65% of the classes from each of the 11 core Java API packages are used in these answers (RQ_2). Such findings clearly suggest the potential of using Stack Overflow for relevant API suggestion. Experiments using 175 code search queries randomly chosen from three Java tutorial sites–*KodeJava, Java2s* and *Javadb*–show that our technique can recommend relevant API classes with an accuracy of 83%, a mean average precision@10 of 46% and a recall@10 of 54%, which are 66%, 79% and 87% higher respectively than that of the state-of-the-art [72] (RQ_4, RQ_8). Query reformulations using our suggested API classes improve 46%–64% of the baseline queries (i.e., contain natural language only), and their overall code retrieval accuracy improves by 19% (RQ_9). Comparisons with the state-of-the-art techniques on query reformulation [51, 84] also demonstrate that RACK offers 48% net improvement in the baseline query quality as opposed to 26% by the state-of-the-art, which is 87% higher (RQ_{10}). Our investigations with Google, Stack Overflow native search, and GitHub native code search also report that our reformulated queries can improve their results by 22%–26% in precision and 12%–28% in reciprocal rank in the context of code example search (RQ_{11}).

Novelty in Contribution: This paper is a significantly extended version of our earlier work [58] which employed two heuristics (KAC and KKC, Section III-B), experimented with 150 queries, and answered seven research questions. This work extends the earlier work in various aspects. First, we improve the earlier heuristics by recalibrating their weights and thresholds (i.e., RQ_7). Second, we introduce a novel heuristic– Keyword Pair API Co-occurrence (KPAC, Section III-B)–that leverages word co-occurrences for candidate API selection more effectively. In fact, this one performs better than the earlier two. Third, we conduct experiments with a larger dataset containing 175 distinct queries, and further evaluate them in terms of their code retrieval performance (i.e., missing in the earlier work). Fourth, we extend our earlier analysis and answer 11 research questions (i.e., as opposed to seven questions answered by the earlier work). Fifth, we investigate the potential application of our approach in the context of code search using popular web search engines (e.g., Google) and code search engines (e.g., GitHub).

Thus, this journal article makes the following contributions:

– An exploratory study that suggests the potential of using Stack Overflow for relevant API suggestion against an NL query intended for code search.

Table 1: **API Packages for Exploratory Study**

Package	#Class	Package	#Class
Core Packages			
`java.lang`	255	`java.net`	84
`java.util`	470	`java.security`	148
`java.io`	105	`java.awt`	423
`java.math`	09	`java.sql`	29
`java.nio`	189	`javax.swing`	1,195
`java.applet`	05		
Non-Core Packages			
`java.beans`	62	`java.rmi`	67
`javax.xml`	327	`javax.annotation`	17
`java.text`	44	`javax.print`	123
`javax.sound`	56	`javax.management`	201
Total API Classes: 3,809			

– A keyword-API mapping database that maps 655K question keywords to 551K API classes from Stack Overflow.
– A novel technique–RACK–that exploits query keyword-API associations stored in the crowdsourced knowledge of Stack Overflow, and reformulates a natural language query using a set of relevant API classes from Stack Overflow.
– Comprehensive evaluation of the proposed technique with five performance metrics, and comparison with the state-of-the-art techniques and contemporary web search engines (e.g., Google, Stack Overflow native search) and code search engines (e.g., GitHub native code search).

Structure of the Article: The rest of article is organized as follows: Section 2 discusses design and findings of our exploratory study, and Section 3 describes our proposed technique for query reformulation. Section 4 focuses on our conducted evaluation and validation, Section 5 on threats to validity, Section 6 on related work from the literature, and finally Section 7 concludes the article with future research directions projected by this work.

2 Exploratory Study

Our technique relies on the mapping between natural language keywords from the questions of Stack Overflow and API classes from corresponding accepted answers for translating a code search query into relevant API classes. Thus, an investigation is warranted whether such answers contain any API related information and the questions contain any search query keywords. We perform an exploratory study using 172,043 Q & A threads from Stack Overflow, and analyse the usage and coverage of standard Java API classes in them. We also explore if the question titles are a potential source of suitable keywords for code search. We particularly answer three research questions as shown in Table 2.

2.1 Data Collection

We collect 172,043 questions and their accepted answers from Stack Overflow using StackExchange data explorer [3] for our investigation. Since we are interested in

Table 2: Research Questions Answered using Exploratory Study

Research Questions Targeting API Coverage
RQ_1: To what extent do the accepted answers from Stack Overflow refer to standard Java API classes?
RQ_2: To what extent are the API classes from each of the core Java packages covered (i.e., mentioned) in the accepted answers from Stack Overflow?
Research Question on Search Keyword Matching
RQ_3: Do the titles from Stack Overflow questions contain potential query keywords (i.e., technical terms) for code snippet search?

Java APIs, we only collect such questions that are annotated with *java* tag. In addition, we apply several other constraints–(1) each of the questions should have at least three answers (i.e., average answer count) with one answer being accepted as the *solution*, in order to ensure that the questions are answered substantially and successfully [45], and (2) the accepted answers should contain code like elements such as code snippets or code tokens so that API information can be extracted from them. We identify the code elements with the help of `<code>` tags in the HTML source of the answers (details in Section 2.2), and use *Jsoup*[1], a popular Java library, for HTML parsing and content extraction.

We repeat the above steps, and construct another dataset by collecting 440K Q & A threads from one of our recent works [56]. This dataset is a superset of the above collection, and it contains more recent threads from Stack Overflow. We call it the *extended dataset* in the remaining sections of the exploratory study.

We collect a total of 3,809 Java API classes for our study from 19 packages of standard Java edition 7. While 2,912 classes are taken from 11 core Java packages[2], the remaining classes have come from 8 non-core Java packages. The goal is to find out if these classes are referred to in Stack Overflow posts, and if yes, to what extent they are referred to. We first use Java Reflections [10], a runtime meta data analysis library, to collect the API classes from JDK 7, and then apply regular expressions on their fully qualified names for extracting the class name tokens. Table 1 shows class statistics of the 19 API packages selected for our investigation.

We also collect a set of 18,662 real life search queries from the Google search history of the first author over the last eight years, which are analysed to answer the third research question. Although the queries come from a single user, they contain a large vocabulary of 9,029 distinct natural language search keywords, and the vocabulary is built over a long period of time. Thus, a study using these queries can produce significant intuitions and help answer the third research question.

2.2 API Class Name Extraction

Several existing studies [12, 24, 60] extract code elements such as API packages, classes and methods from unstructured natural language texts (e.g., forum posts,

[1] https://jsoup.org/
[2] https://goo.gl/A6gEqA

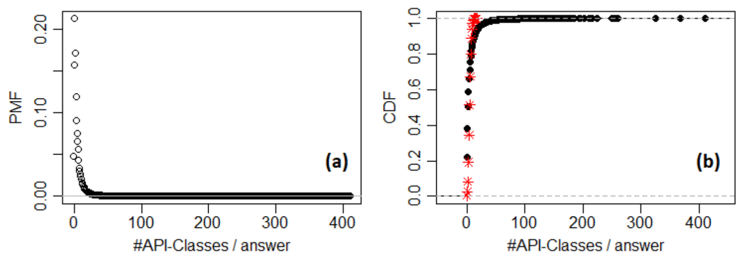


Fig. 2: Frequency distribution for core API classes – (a) API frequency PMF, (b) API frequency CDF

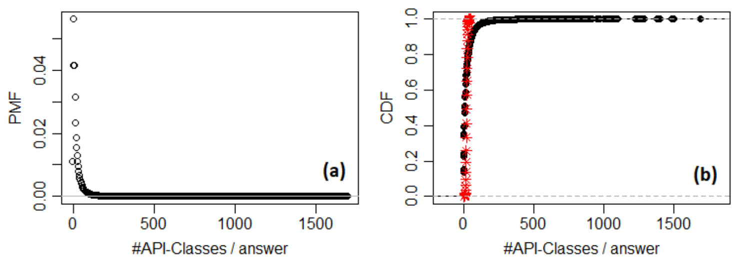


Fig. 3: Frequency distribution for core and non-core API classes over the extended dataset – (a) API frequency PMF, (b) API frequency CDF

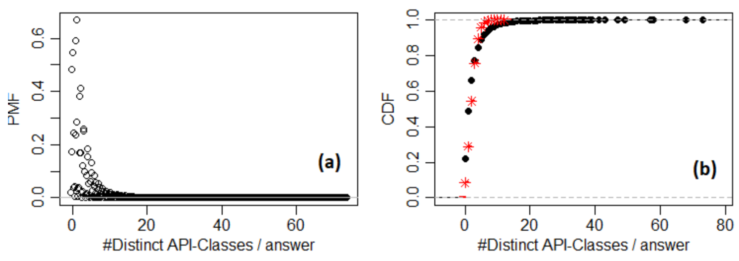


Fig. 4: Frequency distribution of unique API classes from core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF

mailing lists) using information retrieval (e.g., TF-IDF) and island parsing techniques. In the case of island parsing, they apply a set of regular expressions describing Java language specifications [27], and isolate the land (i.e., code elements) from water (i.e., free-form texts). We borrow their parsing technique [60], and apply it to the extraction of API elements from Stack Overflow posts. Since we are interested in the API classes only, we adopt a selective approach for identifying them in the post contents. We first isolate the code like sections from HTML source of each of the answers from Stack Overflow using `<code>` tags. Then we split the sections based on white spaces and punctuation marks, and collect the tokens having the camel-case notation of Java class (e.g., `HashSet`). According to the existing studies [24, 60], such parsing of code elements sometimes introduces false positives. Thus, we restrict our exploratory analysis to a closed set of 3,809 API classes from 19 Java packages (details in Table 1) to avoid false positives (e.g., camel-case tokens but not valid API classes).

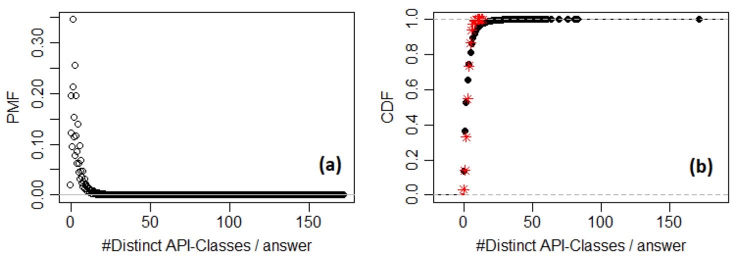


Fig. 5: Frequency distribution of unique API classes from core and non-core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF

2.3 Answering **RQ$_1$**: Use of APIs in the accepted answers of Stack Overflow

Since our API suggestion technique exploits keyword-API associations from Stack Overflow, we investigate whether the accepted answers actually use certain API classes of interest in the first place. According to our investigation, out of 172,043 accepted answers, 136,796 (79.51%) answers refer to one or more Java API class-like tokens. About 61.02% of the answers actually use API classes from 11 core Java packages whereas 9.94% of them use the classes from 8 non-core packages as a part of their solution. We analyse the HTML contents from Stack Overflow answers with tool supports and then detect the *occurrences* of 3,809 standard API classes (Table 1) in each of the accepted answers using a closed-world assumption [60]. We then examine the statistical properties or distribution of such *API occurrence frequencies* (i.e., total appearances, unique appearances) and attempt to answer our first research question.

Fig. 2 shows (a) probability mass function (PMF) and (b) cumulative density function (CDF) for the total occurrences of API classes per SO answer where the API classes belong to the core Java API packages. Both density curves suggest that the frequency observations derive from a heavy-tailed distribution, and majority of the densities accumulate over a short frequency range. That is, most of the time only a limited number of API classes co-occur in each answer from Stack Overflow. The empirical CDF curve also closely matches with the theoretical CDF [1] (i.e., red dots in Fig. 2-(b)) of a Poisson distribution. Thus, we believe that the observations are probably taken from a Poisson distribution. We get a 95% confidence interval over [5.27, 5.37] for mean frequency, $\lambda = 5.32$, which suggests that the API classes from the core packages are referred to at least *five* times on average in each of the answers from Stack Overflow. We also get 10^{th} quantile at frequency=2 and 97.5^{th} quantile at frequency=10 which suggest that only 10% of the frequencies are below 3 and only 2.5% of the frequencies are above 10. When our investigation is repeated for non-core classes, we get a mean frequency, $\lambda = 0.36$, with 95% confidence interval over [0.35, 0.37]. When 11 core and 8 non-core packages are combined and employed against the *extended dataset*, we get a 95% confidence interval over [23.62, 23.87] for the mean frequency, λ=23.75 with a similar distribution (i.e., Fig. 3). Fig. 4 shows density curves of the core API class occurrences per answer where only unique API classes are considered. These observations are also drawn from a heavy-tailed distribution. We get a 95% confidence interval over [2.35, 2.38] for the mean frequency, $\lambda = 2.37$, which

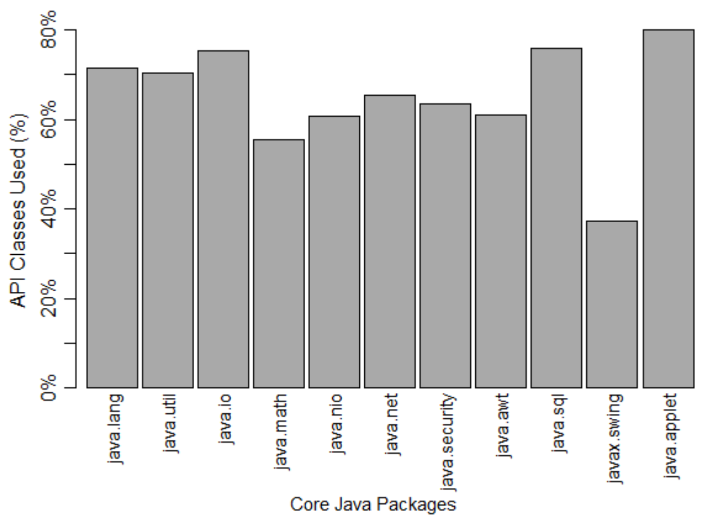


Fig. 6: Coverage of API classes from core packages by Stack Overflow answers

suggests that at least two distinct classes are used on average in each answer. 30th quantile at frequency = 1 and 80th quantile at frequency = 4 suggest that 30% of the Stack Overflow answers refer to at least one API class whereas 20% of the answers refer to at least four distinct API classes from the core Java packages under our study. In the case of non-core classes, we get 90th quantile at frequency = 1, which suggests that their frequencies are negligible. When the same investigation is repeated with 19 (11 core + 8 non-core) packages against the *extended dataset*, we get a 95% confidence interval over [3.44, 3.46] for λ=3.45 with a similar heavy tailed distribution (i.e., Fig. 5).

At least **two** different API classes from the core Java packages are referred to in each of the **61**% accepted answers that are collected from Stack Overflow. These classes are mentioned at least **five** times on average in each answer. API classes from non-core packages are discussed in ≈**10%** of the answers. Furthermore, our observations derived from 172K answers are *similar* to that derived from an extended dataset of 440K answers from Stack Overflow.

2.4 Answering **RQ$_2$**: Coverage of API classes in the accepted answers from Stack Overflow Q & A site

Since our technique exploits inherent mapping between API classes in Stack Overflow answers and keywords from corresponding questions for API suggestion, we need to investigate if such answers actually use a significant portion of the API classes from the standard packages as a part of the solution. We thus identify the occurrences of the API classes from core and non-core packages (Table 1) in Stack Overflow answers, and determine the API coverage for these packages.

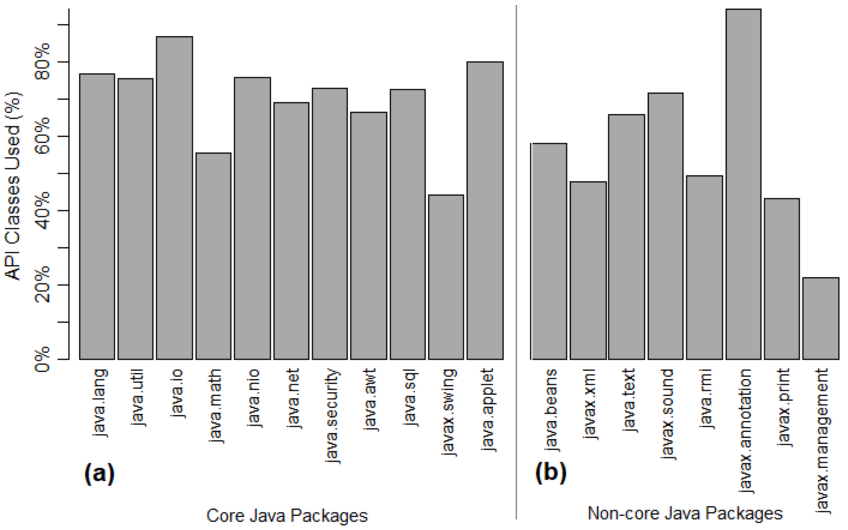


Fig. 7: Coverage of API classes from (a) core and (b) non-core packages by Stack Overflow answers (extended dataset)

Fig. 6 shows the fraction of the API classes that are used in Stack Overflow answers for each of the 11 core packages under study. We note that at least 60% of the classes are used in Stack Overflow for nine out of 11 packages. The remaining two packages–`java.math` and `javax.swing` have 55.56% and 37.41% class coverage respectively. Among these nine packages, three large packages– `java.lang`, `java.util` and `java.io` have a class coverage over 70%. Thus, on average, 65% of the classes are mentioned at least once in Stack Overflow. In Fig. 7, when our investigations are repeated using 19 (11 core + 8 non-core) packages and an *extended dataset*, we get a 95% confidence interval over [56.11, 73.01] for mean coverage, μ=64.56% with a normal distribution. We note that at least 40% of the classes from seven non-core packages are used in Stack Overflow. The remaining package, `javax.management`, has a class coverage of $\approx$ 20%. Fig. 8 shows the fraction of Stack Overflow answers (under study) that use API classes from each of the core 11 packages. We see that classes from `java.lang` package are used in over 50% of the answers, which can be explained since the package contains a number of frequently used and basic classes such as `String, Integer, Method, Exception` and so on. Two packages– `java.util` and `java.awt` that focus on utility functions (e.g., unzip, pattern matching) and user interface controls (e.g., radio button, check box) respectively have a post coverage over 20%. We also note that classes from `java.io` and `javax.swing` packages are used in over 10% of the Stack Overflow answers, whereas the same statistic for the remaining six packages is less than 10%. When our investigations are repeated using 19 (11 core + 8 non-core) packages with the extended dataset, most of the above findings on core packages are reproduced, as shown in Fig. 9-(a). However, as in Fig. 9-(b)), we see that API classes from all eight non-core packages except `javax.xml` are used in less than 5% of the Stack Overflow answers under study. Thus, although a significant amount (e.g., 40%) of the classes from non-core packages are mentioned in Stack Overflow at least for once (i.e., Fig. 7-(b)), as a whole, they are less frequently discussed compared to the core classes. Such finding can

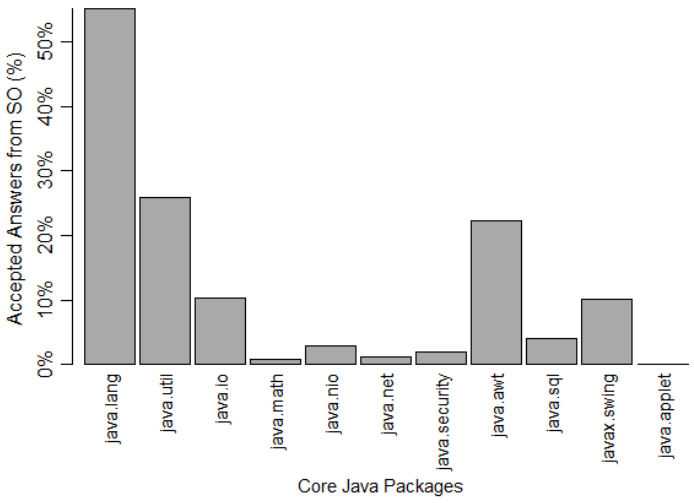


Fig. 8: Use of core API packages in the Stack Overflow answers

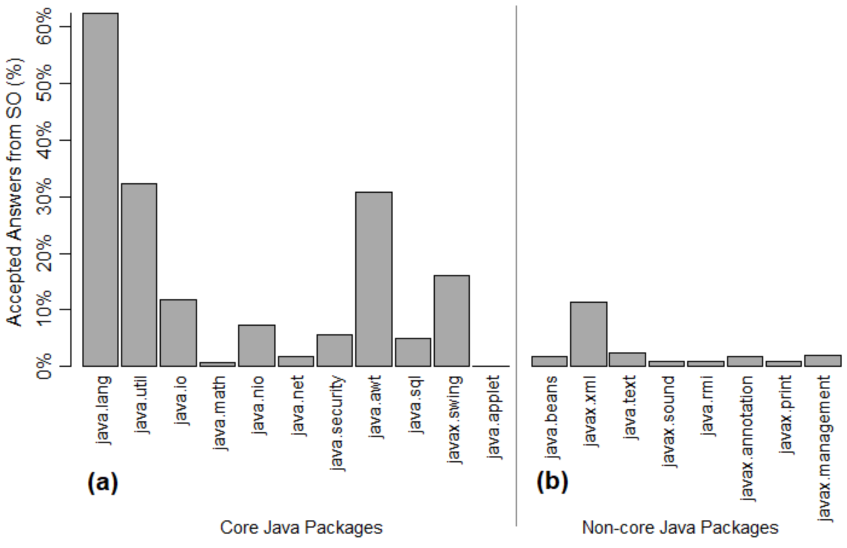


Fig. 9: Use of (a) core and (b) non-core API packages in the Stack Overflow answers (extended dataset)

also be explained by the highly specific functionalities (e.g., RMI, print) of the classes from non-core packages under study.

On average, **65%** of the API classes from each of the 19 (core + non-core) Java packages are used in Stack Overflow accepted answers. Each of these packages is referred to (using their classes) by at least **10%–12%** of the answers under our study. Such findings clearly suggest a significant presence of standard API classes in Stack Overflow posts, and thus, signal their high potential.

Table 3: **Keywords Intended for Code Search**

java	code	example
sql	server	file
string	mvc	web
add	type	lucene
android	table	programmatically

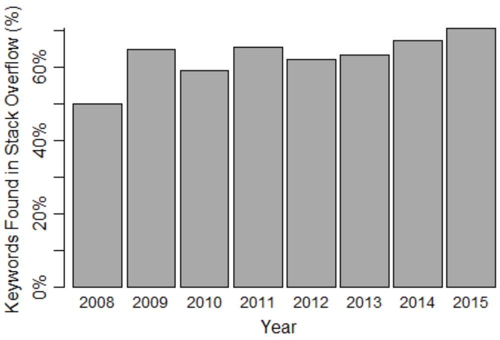


Fig. 10: Coverage of keywords from the collected queries in Stack Overflow questions

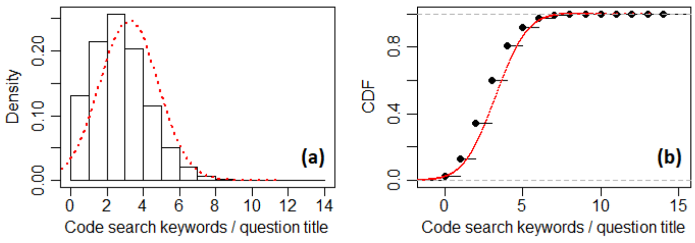


Fig. 11: Collected search query keywords in Stack Overflow– (a) Keyword frequency PMF (b) Keyword frequency CDF

2.5 Answering **RQ$_3$**: Presence of code search keywords in the title of questions from Stack Overflow

Our technique relies on the mapping between natural language terms from Stack Overflow questions and API classes from corresponding accepted answers for augmenting a code search query with relevant API classes. Thus, an investigation is warranted on whether keywords used for code search are present in the SO question texts or not. We are particularly interested in the title of a Stack Overflow question since it summarizes the technical requirement precisely using a few important words, and also resembles a search query. We analyse the titles of 172,043 Stack Overflow questions and 18,662 real life queries used for Google search (Section 2.1). Since we are interested in code related queries, we only select such queries that were intended for code search. Rahman et al. [59] recently used popular tags from Stack Overflow questions to separate code related queries from non-code queries that were submitted to a general-purpose search engine, Google. We use a subset of their selected tags (shown in Table 3) for identifying the code related queries. We discover 3,073 such queries from our query collection (Section 2.1) where the queries contain a total of 2,001 unique search keywords.

Table 4: **Code Search Keywords Found in Tutorial Sites**

Website	#Pages	#Terms	Source	Matched
Javatpoint	1,291	784	Title	20.54%
		10,099	Title+Body	60.12%
Tutorialspoint	2,219	1,292	Title	20.14%
		14,930	Title+Body	63.62%
Stack Overflow	172,043	20,391	Title	**69.22%**

Matched=Overlap between extracted terms and code search keywords

According to our analysis, 172,043 question titles contain 20,391 unique terms after performing natural language preprocessing (i.e., stop word removal, splitting and stemming). These terms match 69.22% of the keywords collected from our code search queries. Fig. 10 shows the fraction of the search keywords that match with the terms from Stack Overflow questions for the past eight years starting from 2008. On average, 62.69% of the code search keywords from each year match with Stack Overflow vocabulary derived from its question titles.

Fig. 11 shows (a) probability mass function, and (b) cumulative density function of keyword frequency in the question titles. We see that the density curve shows the central tendency like a normal curve (i.e., bell shaped curve), and the empirical CDF closely matches with the theoretical CDF (i.e., red curve) of a normal distribution with mean, $\mu = 3.22$ and standard deviation, $\sigma = 1.60$. We also draw 172,043 random samples from a normal distribution with equal mean and standard deviation, and compare with the keyword frequencies. Our Kolmogorov-Smirnov test reported a *p-value* of 2.2e-16<0.05 which suggests that both sample sets belong to the same distribution. Thus, we believe that the keyword frequency observations come from a normal distribution. We get a mean frequency, $\mu = 3.22$ with 95% confidence interval over [3.21, 3.23], which suggests that each of the question titles from Stack Overflow contains at least three code search keywords on average. Furthermore, a recent query classification model that leverages Stack Overflow tags for separating code queries from non-code queries achieves a promising accuracy of 87% precision and 86% recall [59]. Such findings further suggest the potential of Stack Overflow vocabulary for improving the code search.

We also collect all the Q & A threads from two other popular tutorial sites– *Javatpoint*[3] and *Tutorialspoint*[4], construct two *baseline vocabularies* from them, and then contrast with the vocabulary of Stack Overflow. Table 4 shows the statistics on downloaded pages and unique terms extracted from them. For example, Tutorialspoint has a total of 2,219 web pages, and they form a vocabulary of 14,930 unique terms when both title and body of the pages are considered. It encompasses various programming domains including Java, C/C++, and C#. On the contrary, when titles from only Java related questions of Stack Overflow are considered, they form a vocabulary of 20K. We also note that terms from Tutorialspoint page titles match only ≈20% of the code search keywords. On the contrary, such matching ratio is 69% for Stack Overflow which is 237% higher. Surprisingly, when analysed from a granular perspective, Stack Overflow might not be better than these two sites. For example, titles from Javatpoint and Tutorialspoint provide 15.91% and 9.08% of search keywords as opposed to <1.00% by Stack Overflow when 1000

[3] https://www.javatpoint.com
[4] https://www.tutorialspoint.com

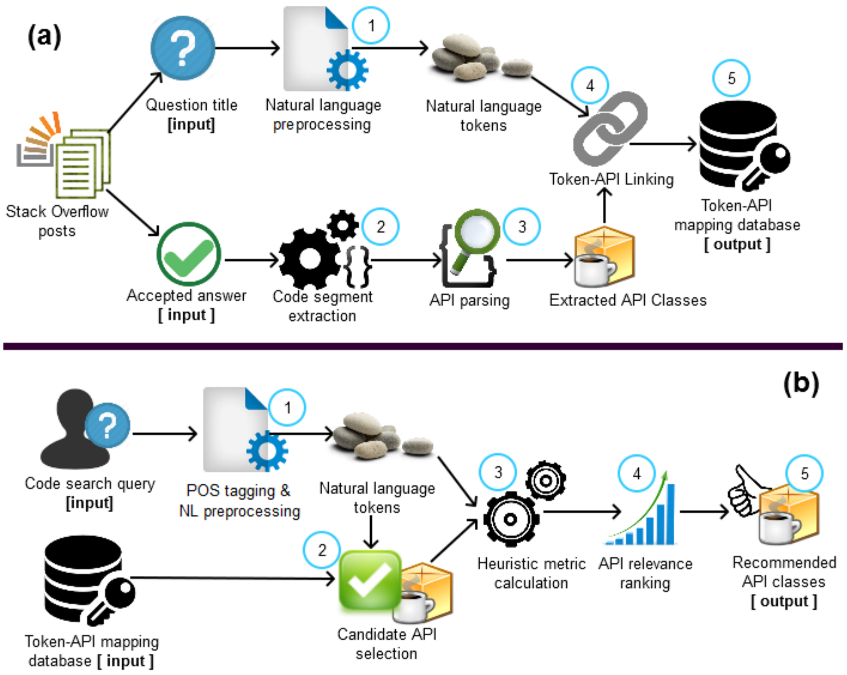

(a)
Stack Overflow posts
Question title [input]
1
Natural language preprocessing
Natural language tokens
4
Token-API Linking
5
Token-API mapping database [output]
Accepted answer [input]
2
Code segment extraction
3
API parsing
Extracted API Classes
(b)
Code search query [input]
1
POS tagging & NL preprocessing
Natural language tokens
2
Token-API mapping database [input]
Candidate API selection
3
Heuristic metric calculation
4
API relevance ranking
5
Recommended API classes [output]

Fig. 12: Proposed technique for API recommendation–(a) Construction of token-API mapping database, (b) Translation of a code search query into relevant API classes

random pages are analysed. However, Stack Overflow offers (1) a nice combination of query terms (in the questions) and API classes (in the code snippets), and (2) a much larger collection of Q & A threads compared to Javatpoint and Tutorialspoint across various domains. Thus, it has a higher potential for assisting the developers in traditional code search.

Each question title from Stack Overflow contains **three** potential keywords for code search on average. Term extracted from these titles match **69%** of the code search keywords produced in real life over the last **eight** years. Furthermore, vocabulary developed from Stack Overflow posts is *much larger* than that of any other available tutorial sites on the web.

3 RACK: Automatic Reformulation of Code Search Query using Crowdsourced Knowledge

According to the exploratory study (Section 2), at least two API classes are used in each of the accepted answers of Stack Overflow, and about 65% of the API classes from the core packages are used in these answers. Besides, the titles from Stack Overflow questions are a major source of query keywords for code search. Such findings suggest that Stack Overflow might be a potential source not only for code search keywords but also for API classes relevant to them. Since we are interested in exploiting this keyword-API association from Stack Overflow questions and answers for API suggestion (i.e., for query reformulation), we need a technique that stores such associations, mines them automatically, and then recommends the most relevant APIs. Thus, our proposed technique has two major steps – (a) Construction of token-API mapping database, and (b) Recommendation of relevant API classes for a code search query which is written in natural language (a.k.a., NL query). Fig. 12 shows the schematic diagram of our proposed technique– RACK– for API recommendation targeting query reformulation.

3.1 Construction of NL Token-API Mapping Database

Since our technique relies on keyword-API associations from Stack Overflow, we need to extract and store such associations for quick access. In Stack Overflow, each question describes a technical requirement such as *"how to send an email in Java?"* The corresponding answer offers a solution containing code example(s) that refer(s) to one or more API classes (e.g., `MimeMessage`, `Transport`). We capture both the requirement and API classes carefully, and exploit their semantic association for the development of token-API mapping database. Since the title summarizes a question using a few but important words, we only use the titles from the questions. Acceptance of an answer by the person who posted the question indicates that the answer actually meets the requirement in the question. Thus, we consider only the accepted answers from the answer collection for our analysis. The construction of the mapping database has several steps as follows:

Token Extraction from Titles: We collect title(s) from each of the questions, and apply standard natural language pre-processing steps such as stop word removal, splitting and stemming on them (Step 1, Fig. 12-(a)). Stop words are the frequently used words (e.g., *the, and, some*) that carry very little meaning for a sentence. We use a stop word list [11] hosted by Google for the stop word removal step. The splitting step splits each word containing any punctuation mark (e.g., .,?,!,;), and transforms it into a list of words. Finally, the stemming step extracts the root of each of the words (e.g., "send" from "sending") from the list, where Snowball stemmer [52, 74] is used. Thus, we extract a set of unique and stemmed words that collectively convey the meaning of the question title, and we consider them as the "tokens" from the title of a question from Stack Overflow. Finally, our database ended up with a total of 19,783 unique NL terms.

API Class Extraction: We collect the accepted answer for each of our selected questions, and parse their HTML source using Jsoup parser [6] for code segments (Step 2, 3, Fig. 12-(a)). We extract all `<code>` and `<pre>` tags from the source content as they generally contain code segments [57]. It should be noted that code segments may sometimes be demarcated by other tags or no tag at all. However, identification of such code segments is challenging and often prone to false-positives. Thus, we restrict our analysis to contents inside `<code>` tags and `<pre>` for code segment collection from Stack Overflow. We split each of the segments based on punctuation marks and white spaces, and discard the programming keywords. Existing studies [12, 60] apply island parsing for API method or class extraction where they use a set of regular expressions. Similarly, we use a regular expression for Java class [27], and extract the API class tokens having a camel case notation. Thus, we collect a set of unique API classes from each of the accepted answers. The API classes (e.g., `String, Integer, Double`) from `java.lang` package are mostly generic and frequently used in the code, which is also supported by our RQ_2. Hence, we also avoid all the API classes from this package during our API extraction from Stack Overflow answers.

Token-API Linking: Natural language tokens from a question title hint about the technical requirement described in the question, and API names from the accepted answer represent the relevant APIs that can meet such requirement. Thus, the programming Q & A site–Stack Overflow– inherently provides an important semantic association between a list of tokens and a list of APIs. For instance, our technique generates a list of natural language tokens–{*generat, md5, hash*}– and an API token– `MessageDigest`– from the showcase example on MD5 hash (Fig. 1). We capture such associations from 126,567 Stack Overflow question and accepted answer pairs, and store them in a relational database (Step 4, 5, Fig. 12-(a)) for relevant API recommendation for any code search query.

3.2 API Relevance Ranking & Reformulation of the NL-Query

In the token-API mapping database, each NL token (or term) associates with different APIs, and each API class associates with a number of NL tokens. Thus, we need a technique that carefully analyses such associations, identifies the candidate APIs, and then recommends the most relevant ones from them for a given query. It should be noted that we do not apply the traditional association rule mining [81]. Our investigations using the constructed database (Section 3.1) report that frequencies of co-occurrence between NL terms and API classes in Stack Overflow posts are not sufficient enough to form association rules for all queries. The API class ranking and recommendation targeting our query reformulation for code search involve several steps as follows:

3.2.1 Identification of Keyword Context

In natural language processing, the context of a word refers to the list of other words that co-occur with that word in the same phrase, same sentence or even the same paragraph [31]. Co-occurring words complement the semantics of one another [47]. Yuan et al. [83] analyse programming posts and tags from Stack Overflow Q & A site, and use word context for determining semantic similarity between any two software-specific words. In this research, we identify the words that co-occur with each query keyword in the thousands of question titles from Stack Overflow. For each keyword, we refer to these co-occurring words as its *context*. We then opportunistically use these contextual words for estimating semantic relevance between any two keywords.

3.2.2 Candidate API Selection

In order to collect candidate APIs for a NL query, we employ three different heuristics. These heuristics consider not only the association between query keywords and APIs but also the coherence among the APIs themselves. Thus, the key idea is to identify such programming APIs as candidates that are not only likely for the query keywords but also functionally consistent to one another.

Keyword-API Co-occurrence (KAC): Stack Overflow discusses thousands of programming problems, and these discussions contain both natural language texts (i.e., keywords) and reference to a number of APIs. According to our observation, several keywords might co-occur with a particular API and a particular keyword might co-occur with several APIs across different programming solutions. This co-occurrence generally takes place either by chance or due to semantic relevance. Thus, if carefully analysed, such co-occurrences could be a potential source for semantic association between keywords and APIs. We capture these co-occurrences (i.e., associations) between keywords from question titles and APIs from accepted answers, discard the random associations using a heuristic threshold (δ), and then collect the top API classes ($L_{KAC}[K_i]$) for each keyword (K_i) that co-occurred most frequently with the keyword at Stack Overflow.

$$L_{KAC}[K_i] = \{A_j \mid A_j \epsilon A \ \wedge \ rank_{freq}(K_i \rightarrow A_j) \leq \delta\}$$

Here, $K_i \rightarrow A_j$ denotes the association between a keyword K_i and an API class A_j, $rank_{freq}$ returns rank of the association from the ranked list based on association frequency, and δ is a heuristic rank threshold. In our research, we consider top ten (i.e., $\delta = 10$) APIs as candidates for each keyword, which is carefully chosen based on iterative experiments on our dataset (see RQ$_7$ for details).

Keyword Pair–API Co-occurrence (KPAC): While frequent co-occurrences of APIs with a query keyword are a good indication of their relevance to the query, they might also fall short due to the fact that the query might contain more than one keyword. That is, API classes relevant to (i.e., frequently co-occurred with) one keyword might not be relevant to other keywords from the query. Thus, API classes that are simultaneously relevant to multiple keywords should be selected as candidates. We consider ${}^nC_2$ keyword pairs from n keywords of a query using combination theory, and identify such APIs that frequently co-occur with both keywords from each pair in the same context (e.g., same Q & A thread). Suppose, K_i and K_j are two keywords, and they form one of the ${}^nC_2$ keyword pairs from the query. Now, the candidate API classes ($L_{KPAC}[K_i, K_j]$) are relevant if they occur in an accepted answer of Stack Overflow whereas both keywords appear in the corresponding question title. We select such relevant candidates as follows:

$$L_{KPAC}[K_i, K_j] = \{A_m \mid A_m \epsilon A \ \wedge \ rank_{freq}((K_i, K_j) \rightarrow A_m) \leq \delta\}$$

Here $(K_i, K_j) \rightarrow A_m$ denotes the association between keyword pair (K_i, K_j) from a question title and API class A_m from the corresponding accepted answer of Stack Overflow. We capture top ten (i.e., $\delta = 10$) such co-occurrences for KPAC heuristic, and the detailed justification for this choice can be found in RQ$_7$. We determine the association based on their co-occurrences in the same set of documents. In this case, each question-answer thread from Stack Overflow is considered as a document. While co-occurrences of keyword triples with APIs could also be considered for API candidacy, existing IR-based studies report that phrases of two words are more effective as a semantic unit (e.g., *"chat room"*) rather than the triples (e.g., *"find chat room"*) [47, 55].

Keyword–Keyword Coherence (KKC): The two heuristics above determine relevant API candidacy based on the co-occurrence between query keywords and

APIs in the same document. That is, multiple keywords from the query are also warranted to co-occur in the same document. However, such co-occurrences might not always happen, and yet the keywords could be semantically related to one another (i.e., co-occurred in the query). More importantly, the candidate APIs should be relevant to multiple keywords that do not co-occur. Yuan et al. [83] determine semantic similarity between any two software specific words by using their contexts from Stack Overflow questions and answers. We adapt their technique for identifying coherent keyword pairs which might not co-occur. The goal is to collect candidate APIs relevant to these pairs based on their coherence. We (1) develop a context (C_i) for each of the n query keywords by collecting its co-occurring words from thousands of question titles from Stack Overflow, (2) determine semantic similarity for each of the ${}^{n}C_2$ keyword pairs based on their context derived from Stack Overflow, and (3) use these measures to identify the coherent pairs and then to collect the functionally coherent APIs for them. At the end of this step, we have a set of candidate APIs for each of the coherent keyword pairs.

Suppose, two query keywords K_i and K_j have context word list C_i and C_j respectively. Now, the candidate APIs (L_{KKC}) that are relevant to both keywords and functionally consistent with one another can be selected as follows:

$$L_{KKC}[K_i, K_j] = \{L[K_i] \cap L[K_j] \mid cos(C_i, C_j) > \gamma\}$$

Here, $cos(C_i, C_j)$ denotes the *cosine similarity* [57] between two context lists– C_i and C_j, and γ is the similarity threshold. We consider $\gamma = 0$ in this work based on iterative experiments on our dataset (see RQ$_7$ for the detailed justification). $L[K_i]$ and $L[K_j]$ are top frequent APIs for the two keywords– K_i and K_j where K_i and K_j might not co-occur in the same question title. Thus, $L_{KKC}[K_i, K_j]$ contains such APIs that are relevant to both keywords (i.e., co-occurred with them in Stack Overflow answers) and functionally consistent with one another. Since the candidate APIs co-occur with the keywords from each coherent pair (i.e., semantically similar, $\gamma > 0$) in different contexts, they are also likely to be *coherent* for the programming task at hand. Such coherence often could be explained in terms of the dependencies among the API classes.

3.2.3 API Relevance Ranking Algorithm

Fig. 12-(b) shows the schematic diagram, and Algorithm 1 shows the pseudo code of our API relevance ranking algorithm–RACK. Once a search query is submitted, we (1) perform Part-of-Speech (POS) tagging on the query for extracting the meaningful words such as nouns and verbs [19, 79], and (2) apply standard natural language preprocessing (i.e., stop word removal, splitting, and stemming) on them to extract the stemmed words (Lines 3–4, Algorithm 1). For example, the query– *"html parser in Java"* turns into three keywords–*'html', 'parser'* and *'java'* at the end of the above step. We then apply our three heuristics–*KAC*, *KPAC* and *KKC*– on the stemmed keywords, and collect candidate APIs from the token-API linking database (Step 2, Fig. 12-(b), Lines 5–8, Algorithm 1). The candidate APIs are selected based on not only their co-occurrence with the query keywords but also the coherence (i.e., functional consistency) among themselves. We then use the following metrics (i.e., derived from the above heuristics) to estimate the relevance of the candidate API classes for the query.

Algorithm 1 API Relevance Ranking using the Proposed Heuristics

```
1: procedure RACK($Q$)                    ▷ $Q$: natural language query for code search
2:     $R \leftarrow \{\}$                      ▷ list of API classes relevant to $Q$
3:     ▷ collecting keywords using POS tagging and NL preprocessing
4:     $K \leftarrow$ preprocess(collectNounVerbs($Q$))
5:     ▷ collecting candidate API classes
6:     $L_{KAC} \leftarrow$ getKACList($K$)
7:     $L_{KPAC} \leftarrow$ getKPACList($K$)
8:     $L_{KKC} \leftarrow$ getKKCList($K$)
9:     ▷ estimating relevance of the candidate APIs
10:    for Keyword $K_i \in K$ do
11:        for APIClass $A_j \in L_{KAC}[K_i]$ do
12:            ▷ relevance of an API with single keyword
13:            $S_{KAC} \leftarrow$ getKACScore($A_j, L_{KAC}[K_i]$)
14:            $R_{KAC}[A_j].score \leftarrow R_{KAC}[A_j].score + S_{KAC}$
15:        end for
16:    end for
17:    for Keyword $K_i, K_j \in K$ do
18:        ▷ relevance of an API with multiple keywords
19:        for APIClass $A_j \in L_{KPAC}[K_i, K_j]$ do
20:            $S_{KPAC} \leftarrow$ getKPACScore($A_j, L_{KPAC}[K_i, K_j]$)
21:            $R_{KPAC}[A_j].score \leftarrow R_{KPAC}[A_j].score + S_{KPAC}$
22:        end for
23:        ▷ coherence of an API with other candidate APIs
24:        $C_i \leftarrow$ getContextList($K_i$)
25:        $C_j \leftarrow$ getContextList($K_j$)
26:        $S_{KKC} \leftarrow$ getKKCScore($C_i, C_j$)
27:        for APIClass $A_j \in L_{KKC}[K_i, K_j]$ do
28:            $R_{KKC}[A_j].score \leftarrow R_{KKC}[A_j].score + S_{KKC}$
29:        end for
30:    end for
31:    ▷ ranking of the API classes using their normalized scores and relative weights
32:    for APIClass $A_j \in \{R_{KAC}, R_{KPAC}, R_{KKC}\}$ do
33:        $R[A_j] \leftarrow \max(\alpha \times R_{KAC}[A_j],\ \beta \times R_{KPAC}[A_j],\ (1-\alpha-\beta) \times R_{KKC}[A_j])$
34:    end for
35:    $rankedAPIs \leftarrow$ sortByScore($R$)
36:    return $rankedAPIs$
37: end procedure
```

API Co-occurrence Likelihood estimates the probability of co-occurrence of a candidate API (A_j) with one (K_i) or more (K_i, K_j) keywords from the search query. It considers the rank of the API in the ranked list based on keyword-API co-occurrence frequency (i.e., KAC and $KPAC$) and the size of the list, and then provides a normalized score (on the scale from 0 to 1) as follows:

$$S_{KAC}(A_j, K_i) = 1 - \frac{rank(A_j, sortByFreq(L[K_i]))}{|L_{KAC}[K_i]|}$$

$$S_{KPAC}(A_j, K_i, K_j) = 1 - \frac{rank(A_j, sortByFreq(L_{KPAC}[K_i, K_j]))}{|L_{KPAC}[K_i, K_j]|}$$

Here, S_{KAC} and S_{KPAC} denote the API co-occurrence likelihood estimates, and they range from 0 (i.e., not likely at all for the keywords) to 1 (i.e., very much likely for the keywords). The more likely an API is for the keywords, the more relevant it is for the query. This approach might also encourage the common API classes (e.g., `List`, `String`) that are often used with most programming tasks. Such APIs might not be helpful for relevant code snippet search. We thus apply appropriate filters and thresholds to avoid such noise.

API Coherence estimates the coherence of an API (A_j) with other candidate APIs for a query. Since the query targets a particular programming task (e.g., *"parsing the HTML source"*), the suggested APIs should be logically consistent with one another. One way to heuristically determine such coherence is to exploit the semantic relevance among the corresponding keywords that co-occurred with that API (A_j). The underlying idea is that if two keywords are semantically similar, their co-occurred API sets could also be logically consistent with each other. We thus determine semantic similarity between any two keywords (K_i, K_j) from the query using their context lists (C_i, C_j) [83], and then propagate that measure to each of their candidate API classes (A_j) that co-occurred with both of the keywords (i.e., KKC) as follows:

$$S_{KKC}(A_j, K_i, K_j) = cos(C_i, C_j) \mid (K_i \rightarrow A_j) \wedge (K_j \rightarrow A_j)$$

Here, S_{KKC} denotes the API Coherence estimate, and it ranges from 0 (i.e., not relevant at all with multiple keywords) to 1 (i.e., highly relevant). It should be noted that each candidate, A_j, comes from $L[K_i]$ or $L[K_j]$, i.e., the API is already relevant to each of K_i and K_j in their corresponding contexts. S_{KKC} investigates how similar those contexts are, and thus heuristically estimates the coherence between the APIs from these contexts.

We first estimate *API Co-occurrence Likelihood* of each of the candidate APIs that suggests the likeliness of the API for one or more keywords from the given query (Lines 9–22, Algorithm 1). Then we determine *API Coherence* for each candidate API that suggests coherence of the API with other candidate APIs for the query (Lines 23–30). Once all metrics of each candidate are calculated (Step 3, Fig. 12-(b)), only the maximum score is taken into consideration where appropriate weights–α, β and $(1 - \alpha - \beta)$–are applied (Lines 31–34, Algorithm 1). These weights control how two of our above dimensions– *co-occurrence* and *coherence*– affect the final relevance ranking of the candidates. We consider a heuristic value of 0.325 for α and a value of 0.575 for β, and the detailed weight selection method is discussed in Section 4.9. The candidates are then ranked based on their final scores, and Top-K API classes from the ranked list are returned as API recommendation (Lines 35–36, Algorithm 1, Step 4, 5, Fig. 12-(b)). Such API classes are then used for NL-query reformulation.

Working Example: Table 5 shows a working example of how our proposed query reformulation technique –RACK– works. Here we reformulate our natural language query– *"HTML parser in Java"*–into relevant API classes. We first apply *KAC* heuristic, and collect the Top-5 (i.e., $\delta = 5$) candidate APIs for each of the three keywords–*'html'*,*'parser'* and *'java'*– based on co-occurrence frequencies of the candidates with the keywords. We also repeat the same step for each of the three (i.e., ${}^3C_2$) keyword pairs–*(html, parser), (html, java)* and *(parser, java)* by applying our *KPAC* heuristic. Then we estimate *co-occurrence likelihood* (with the keywords and keyword pairs) of each of the candidate APIs. For example, `Document` has a maximum likelihood of 1.00 among the candidates not only for the single search keyword but also for the keyword pairs. We then determine *coherence* of each candidate API (with other candidates) based on semantic relevance among the above three keyword pairs. For example, *'html'* and *'parser'* have a semantic relevance of 0.42 between them (on the scale from 0 to 1) based on their contexts from Stack Overflow questions and answers, and they have several common candidates such as `Document`, `Element` and `File`. Since the two keywords are semantically

Table 5: **An Example of Query Reformulation using RACK**

	html	$\mathbf{S_{KAC}}$	**parser**	$\mathbf{S_{KAC}}$	**java**	$\mathbf{S_{KAC}}$
KAC	`Document`	1.00	`Document`	1.00	`Object`	1.00
	`Jsoup`	0.80	`Element`	0.80	`ArrayList`	0.80
	`Element`	0.60	`File`	0.60	`File`	0.60
	`Elements`	0.40	`IOException`	0.40	`Class`	0.40
	`IOException`	0.20	`Node`	0.20	`IOException`	0.20
	(html, parser)	$\mathbf{S_{KPAC}}$	**(html, java)**	$\mathbf{S_{KPAC}}$	**(parser, java)**	$\mathbf{S_{KPAC}}$
KPAC	`Document`	1.00	`Document`	1.00	`Document`	1.00
	`Jsoup`	0.80	`Jsoup`	0.80	`Element`	0.80
	`Element`	0.60	`Element`	0.60	`File`	0.60
	`Elements`	0.40	`IOException`	0.40	`DocumentBuilder`	0.40
	`Parser`	0.20	`Elements`	0.20	`DocumentBuilderFactory`	0.20
	(html, parser)	$\mathbf{S_{KKC}}$	**(html, java)**	$\mathbf{S_{KKC}}$	**(parser, java)**	$\mathbf{S_{KKC}}$
KKC	`Document`	0.42	`IOException`	0.28	`File`	0.20
	`Element`	0.42	`File`	0.28	`IOException`	0.20
	`IOException`	0.42				
	`File`	0.42				
	`ArrayList`	0.42				
	Initial Query		**Reformulated Query**		**Suggested API**	**Score**
RACK	Q= *"HTML parser in Java"*		Q'={`Document, Element, File, IOException, Jsoup`} + Q		`Document`	0.79
					`Element`	0.69
					`File`	0.69
					`IOException`	0.52
					`Jsoup`	0.50

Table 6: Research Questions Answered using Experiment

Research Questions on API Suggestion
$\mathbf{RQ_4}$**:** How does the proposed technique –RACK– perform in recommending relevant APIs for a code search query?
$\mathbf{RQ_5}$**:** How effective are the proposed heuristics–*KAC, KPAC* and *KKC*–in capturing the relevant APIs for a query?
$\mathbf{RQ_6}$**:** Does an appropriate subset of the query keywords perform better than the whole query in retrieving the relevant APIs?
$\mathbf{RQ_7}$**:** How do the heuristic weights (i.e., α, β) and threshold settings (i.e., γ, δ) influence the performance of our technique?
$\mathbf{RQ_8}$**:** Can RACK outperform the state-of-the-art techniques in recommending relevant APIs for a given set of natural language queries?

Research Questions on Query Reformulation
$\mathbf{RQ_9}$**:** Can RACK significantly improve the natural language queries in terms of relevant code retrieval performance?
$\mathbf{RQ_{10}}$**:** Can RACK outperform the state-of-the-art technique in improving the natural language queries for code search?
$\mathbf{RQ_{11}}$**:** How does RACK perform compared to the popular web search engines (e.g., Google) and code search engines (e.g., GitHub code search)?

relevant, their relevance score (i.e., 0.42, S_{KKC}) is propagated to their shared candidate APIs as a proxy to the coherence among the candidates. We then gather all scores for each candidate, choose the best score, and finally get a ranked list. From the recommended list, we see that `Document`, `Element` and `Jsoup` are highly relevant APIs from *Jsoup library* for the given NL-query. Our technique returns such a list of relevant API classes as the reformulation to an original NL query.

4 Experiment

One of the most effective ways to evaluate a technique that suggests relevant API classes or methods for a query is to check their conformance with the gold set APIs of the query. Since the suggested APIs could be used to reformulate the initial query (i.e., using natural language), the quality of the automatically reformulated query could be another performance indicator for the technique. We evaluate our technique using 175 code search queries, their goldset APIs and their relevant code segments (i.e., implementing the tasks in the query) collected from three programming tutorial sites. We determine the performance of our technique using six appropriate metrics from the literature. Then we compare with two variants of the state-of-the-art technique on API recommendation [72] and a popular code search engine–*Lucene* [30]–for validating our performance. We answer eight research questions with our experiments as shown in the Table 6.

4.1 Experimental Dataset

Data Collection: We collect 175 code search queries for our experiment from three Java tutorial sites– KodeJava [7], JavaDB [5] and Java2s [4]. These sites discuss hundreds of programming tasks that involve the usage of different API classes from the standard Java API libraries. Each of these task descriptions generally has three parts–(1) a title (i.e., question) for the task, (2) one or more code snippets (i.e., answer), and (3) an associated prose explaining the code. The title summarizes a programming task (e.g., *"How do I decompress a GZip file in Java?"*) using natural language texts. It generally uses a few pertinent keywords (e.g., *"decompress"*, *"GZip"*), and also often resembles a query for code search (Section 2.5). We thus consider such titles from the tutorial sites as the code search queries, and use them for our experiment in this research.

Gold Set Development: The prose explaining the code often refers to one or more APIs (e.g., `GZipOutputStream`, `FileOutputStream`) from the code snippet(s) that are found to be essential for the task. In other words, such APIs can be considered as the most relevant ones (i.e., vital) for the target programming task. We collect such APIs from the prose against each of the task titles (i.e., code search queries) from our dataset, and develop a gold set–*API-goldset*–for the experiment. Since relevance of the APIs is determined based on working code examples and their associated prose from the publicly available and popular tutorial sites, the subjectivity associated with the relevance of the collected APIs is minimized [22]. We also collect the code segments verbatim that implement each of the selected tasks (i.e., our queries) from these tutorial sites, and develop another gold set–*Code-goldset*–for our experiments. Our goals are to (1) compare our queries containing the suggested API classes with the baseline queries containing only NL keywords and (2) compare our queries with the reformulated queries by the state-of-the-art techniques on API recommendation [51, 72, 84].

Corpus Preparation: We evaluate not only the API recommendation performance of RACK but also the retrieval performance of its reformulated queries. We collect relevant code snippets (i.e., ground truth) for each of our 175 search queries from the above tutorial sites, and develop a corpus. It should be noted that each query-code snippet pair comes from the same Q & A thread from the tutorial sites. However, this approach leaves us with a corpus of 175 documents which do not represent a real world corpus. We thus extend our code corpus by adding more code snippets from one of our earlier works [53], and this provided a corpus containing 4,170 (175+3,995) code snippets. It should be noted that the additional 3,995 code snippets were carefully collected from hundreds of open source projects hosted at GitHub (see [53] for details). This corpus is referred to as *4K-Corpus* throughout the later sections in the paper.

We also develop two other corpora containing 256,754 (175+256,399) and 769,244 (175+769,069) documents respectively. They are referred to as *256K-Corpus* and *769K-Corpus* in the rest of the sections. These corpus documents are Java classes extracted from an internet-scale and well-established dataset– IJaDataset [37, 44, 68]. The dataset was constructed using 24,666 real world Java projects across various domains, and they were collected from SourceForge[5] and Google Code[6] repositories. We analyse 1,500,000 Java source files from the dataset, and discard the ones with a size greater than 3KB. 95% of our ground truth code segments have a size less than 3KB. The goal was to avoid the large and potentially noisy code snippets in the corpus. Given the large size (i.e., 769K documents) and cross-domain nature of the collected projects, our corpora are thus likely to represent a real world code search scenario.

We consider each of these code snippets from all three corpora as an individual document, apply standard natural language preprocessing (i.e., token splitting, stop word removal, programming keyword removal) to them, and then index the corpus documents using *Apache Lucene*[7], a search engine widely used by the relevant literature [30, 38, 48]. The indexed corpus is then used to determine the retrieval performance of the initial and reformulated queries for code search.

Replication: All the experimental data, associated tools and implementations are hosted online [9] for replication or third party reuse.

4.2 Performance Metrics

We choose five performance metrics for the evaluation and validation of our technique that are widely adopted by relevant literature [22, 46, 72]. Two of them are related to recommendation systems whereas the other four metrics are widely popular in the information retrieval domain.

Top-K Accuracy/Hit@K: It refers to the percentage of the search queries for each of which at least one item (e.g., API class, code segment) is correctly returned within the Top-K results by a recommendation technique. It is also called Hit@K [75]. Top-K Accuracy of a technique can be defined as follows:

$$Top\text{–}K\,Accuracy(Q) = \frac{\sum_{q \in Q} isCorrect(q, K)}{|Q|}\%$$

Here, $isCorrect(q, K)$ returns a value 1 if there exists at least one correct API class (i.e., from the API-goldset) or one correct code segment (i.e., implements the task in query) in the Top-K returned results, and returns 0 otherwise. Q denotes the

[5] https://sourceforge.net/

[6] https://code.google.com/

[7] http://lucene.apache.org/

set of all search queries used in the experiment. Although Top-K Accuracy and Hit@K are used interchangeably in the literature [69, 75], we use Hit@K to denote recommendation accuracy in the remaining sections for the sake of consistency.

Mean Reciprocal Rank@K (MRR@K): Reciprocal rank@K refers to the multiplicative inverse of the rank (i.e., $1/rank(q,K), q \in Q$) of the first relevant API class or code segment in the Top-K results returned by a technique. Mean Reciprocal Rank@K (MRR@K) averages such measures for all search queries ($\forall q \in Q$) in the dataset. It can be defined as follows:

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank(q,K)}$$

Here, $rank(q,K)$ returns the rank of the first correct API or the correct code segment from a ranked list of size K. If no correct API class or code segment is found within the Top-K positions, then $rank(q,K)$ returns ∞. On the contrary, it returns 1 for the correct result at the topmost position of a ranked list. Thus, MRR can take a maximum value of 1 and a minimum value of 0. The bigger the MRR value is, the better the technique is.

Mean Average Precision@K (MAP@K): *Precision@K* calculates the precision at the occurrence of every single relevant item (e.g., API class, code segment) in the ranked list. *Average Precision@K (AP@K)* averages the *precision@K* for all relevant items within Top-K results for a code search query. *Mean Average Precision@K* is the mean of *Average Precision@K* for all queries (Q) from the dataset. MAP@K of a technique can be defined as follows:

$$AP@K = \frac{\sum_{k=1}^{K} P_k \times rel_k}{|RR|}$$

$$MAP@K = \frac{\sum_{q \epsilon Q} AP@K(q)}{|Q|}$$

Here, rel_k denotes the relevance function of k^{th} result in the ranked list that returns either 1 (i.e., relevant) or 0 (i.e., non-relevant), P_k denotes the precision at k^{th} result, and K refers to number of top results considered. RR is the set of relevant results for a query, and Q is the set of all queries.

Mean Recall@K (MR@K): Recall@K refers to the percentage of gold set items (e.g., API, code segment) that are correctly recommended for a code search query in the Top-K results by a technique. Mean Recall@K (MR@K) averages such measures for all queries (Q) in the dataset. It can be defined as follows:

$$MR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|result(q,K) \cap gold(q)|}{|gold(q)|}$$

Here, $result(q,K)$ refers to Top-K recommended APIs by a technique, and $gold(q)$ refers to goldset APIs for each query $q \in Q$. The bigger the MR@K value is, the better the recommendation technique is.

Query Effectiveness (QE): It refers to the rank of first relevant document in the results list retrieved by a query. The metric approximates a developer's effort in locating the first item of interest. Thus, the lower the effectiveness measure is, the more effective the query is [49, 55]. We use this measure to determine whether a given query is improved or not after its reformulation.

Normalized Discounted Cumulative Gain (NDCG): It determines the quality of ranking provided by a technique. With a *graded relevance scale* for results, the metric accumulates overall gain or usefulness from the top to the bottom of the list [35, 77]. It assumes that (1) highly relevant results are more useful when they appear earlier in the ranked list, and (2) highly relevant results are more useful than marginally relevant results. Thus, Discounted Cumulative Gain (DCG) of a ranked list returned by a query q can be calculated as follows:

$$DCG(q) = \sum_{k=1}^{K} \frac{grel_k}{log_2(k+1)} \quad \text{where} \quad grel_k = 1 - \frac{goldRank(k, gold(q))}{|gold(q)|}$$

Here, $grel_k$ refers to the graded relevance of the result at position k. $goldRank(.)$ returns the rank of the k^{th} result within the goldset items $gold(q)$. If k^{th} result is not found in the goldset, $grel_k$ simply returns 0 as a special case. Thus, $grel_k$ provides a graded relevance scale between 0 and 1 for each relevant result. Once $DCG(q)$ is calculated, the normalized DCG can be calculated as follows:

$$NDCG(q) = \frac{DCG(q)}{IDCG(q)}, \quad NDGC(Q) = \frac{1}{|Q|} \sum_{q \in Q} NCDG(q)$$

Here $IDCG(q)$ is the Ideal Discounted Cumulative Gain which is derived from the ranking of goldset items. Thus, $NDCG(q)$ is the metric for one single query q, whereas $NDCG(Q)$ averages the metric over all queries ($\forall q \in Q$). We use NDCG in order to determine the quality of code search ranking from the traditional web/code search engines (Section 4.13).

4.3 Evaluation Scenarios

Our work in this article has two different aspects– (a) relevant API suggestion and (b) automatic query reformulation. We thus employ two different setups for evaluating our approach. In the first case, we investigate API suggestion performance of RACK, calibrate our adopted parameters, and compare with the state-of-the-art approaches on API suggestion [72, 84] (RQ_4–RQ_8). In the second case, we reformulate the initial NL queries from the dataset using our suggested API classes. Then we compare our reformulated queries not only with the baseline queries but also with the queries generated by the state-of-the-art approaches on query reformulation [51, 84] (RQ_9–RQ_{10}). We also investigate the potential of our queries in the context of contemporary web and code search practices (RQ_{11}).

4.4 Statistical Significance Tests

In our comparison studies, we perform two statistical tests before claiming significance of one set of items over the other. In particular, we employ *Mann-Whitney Wilcoxon (MWW)* and *Wilcoxon Signed Rank (WSR)* for significance tests. We refer to them as MWW and WSR respectively in the remaining sections. MWW is a non-parametric test that (1) does not assume normality of the data and (2) is appropriate for small dataset [30]. We use this test for comparing any two arbitrary lists of items. WSR test is another non-parametric test that performs pair-wise

Table 7: **Performance of RACK**

Metric	**Non-weighted Version**				**Weighted Version**			
	Top-1	**Top-3**	**Top-5**	**Top-10**	**Top-1**	**Top-3**	**Top-5**	**Top-10**
Hit@K	30.29%	55.43%	68.57%	**83.43%**	38.29%	61.14%	72.00%	**83.43%**
MRR@K	0.30	0.41	0.44	**0.46**	0.38	0.48	0.48	**0.52**
MAP@K	30.29%	40.19%	**42.00%**	39.66%	38.29%	48.14%	**48.39%**	45.74%
MR@K	9.24%	22.67%	33.53%	**52.78%**	12.12%	26.41%	37.94%	**54.07%**

comparison between two lists. In our experiment, WSR was used for significance test between performance measures (e.g., Hit@K) of RACK in API/code suggestion and that of an existing approach for the same K positions (i.e., $1 \leq K \leq 10$) (RQ_8, RQ_9, RQ_{10}). We report *p-value* of each statistical test, and use 0.05 as the significance threshold. In addition to these significance tests, we also perform effect size test using *Cliff's delta* to demonstrate the level of significance. For this work, we use three significance levels – *short* ($0.147 \leq \Delta \leq 0.33$), *medium* ($0.33 \leq \Delta \leq 0.474$) and *large* ($\Delta \geq 0.474$) [62]. We use two R packages – `stats, effsize` – for conducting these statistical tests.

4.5 Matching of Suggested APIs with Goldset APIs

In order to determine performance of a technique, we apply *strict* matching between gold set APIs and the recommended APIs. That is, we consider two API classes matched if (1) they are categorically the same, and (2) they are superclass or subclass of each other. For example, if `OutputStream` is a gold set API and `FileOutputStream` is a recommended API, we consider them and their inverse as matched. If a base class is relevant for a programming task, the derived class is also likely to be relevant and thus, the recommendation is considered to be accurate. In the case of relevant code segment retrieval, we also apply exact matching between gold set segment and returned segment by a query. Since the tutorial sites clearly indicate which of the code segments implements which of our selected tasks (i.e., queries), such matching is warranted for this case. It should be noted that items (e.g., API class, code segment) outside the goldset could be also relevant to our queries. However, we stick to our gold sets for the sake of simplicity and clarity of our experiments. Our gold sets are also publicly available [9] for third-party replication or reuse.

4.6 Answering $\mathbf{RQ_4}$: How does the proposed technique perform in suggesting relevant APIs for a code search query?

Each of our selected queries summarizes a programming task that requires the use of one or more API classes from various Java libraries. Our technique recommends Top-K (e.g., $K = 10$) relevant API classes for each query. We compare the recommended items with the *API-goldset* and evaluate them using above four metrics. In this section, we answer RQ_4 using Table 7 and Fig. 13.

Table 7 shows the performance details of our technique for Top-1, Top-3, Top-5 and Top-10 API recommendations. We see that our technique recommends at least one API correctly for 83%+ of the queries with both its (a) non-weighted and (b) weighted versions. The weighted version applies a fine tuned weight to

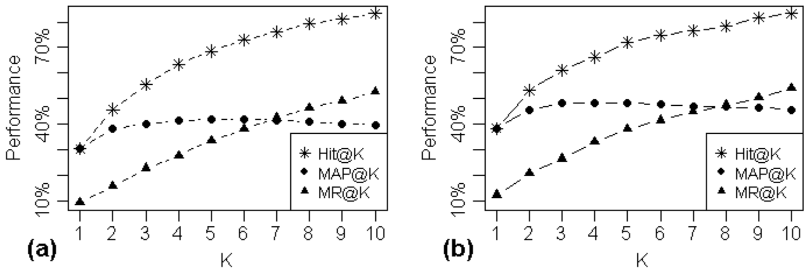


Fig. 13: Hit@K, Mean Average Precision@K, and Mean Recall@K of RACK using (a) non-weighted version (i.e., dashed line) and (b) weighted version (i.e., solid line)

each of our three heuristics–KAC, KPAC and KKC–whereas the non-weighted version treats each of the heuristics equally. Such accuracy is highly promising according to the relevant literature [22, 46]. Mean average precision and mean recall of RACK are 40%–46% and 53%–54% respectively for Top-10 results which are also promising. It also should be noted that RACK provides 55%–61% accuracy and 40%–48% precision for only Top-3 results which are good. That means, one out of the two suggested API classes is found to be relevant for the task, which could be really helpful for effective code search. Our mean reciprocal ranks are 0.46 and 0.52 for non-weighted and weighted version respectively. That is, the first correct suggestion is generally found between first to second position of our ranked list, which demonstrates the potential of our technique. Fig. 13 shows how different performance metrics – accuracy, precision and recall– change over different values of Top-K. We see that our technique reaches a high precision (i.e., 48.14%) quite early (i.e., $K = 3$) and the highest (i.e., 48.39%) at $K = 5$, and then stays comparable for the rest of the K values. However, the improvement of recall measure is comparatively slow. It is $\approx$ 10% for $K = 1$, and then increases somewhat linearly up to 54% for the last value of $K = 10$. On the contrary, the accuracy of RACK improves in a log-linear fashion, and becomes somewhat stationary for $K =$ 10 with 83%. While our accuracy and recall could further improve for increased K-values, the precision is likely to drop. Thus, we conduct our experiments using only Top-10 suggestions from a technique. Developers generally do not check items beyond the Top-10 items from the ranked list, and relevant literature [55, 69] also widely apply such cut-off value. Thus, our choice of $K = 1$ to 10 is also justified.

We also analyse the distribution of API classes from 19 (11 core + 8 non-core) Java packages (i.e., Table 1) in our ground truth, and investigate how they correlate with corresponding distributions from Stack Overflow. We found that on average, 10% of the standard Java API classes from each package overlap with our ground truth classes. On the contrary, 65% of the API classes from each package are discussed in Stack Overflow Q & A threads according to RQ_2. Thus, Stack Overflow discusses more API classes than the ground truth warrants for. In short, Stack Overflow is highly likely to deliver the relevant classes from standard API packages, and our approach harnesses that power. We also found that 51% of the ground truth classes come from the core packages whereas 10% of them come from the non-core packages. Since Stack Overflow has a good coverage (e.g., $\approx$ 65%) for both core and non-core packages (Fig. 7), RACK is also likely to perform well for such queries that require the API classes from non-core packages only.

Table 8: **Role of Proposed Heuristics– KAC, KPAC and KKC**

Heuristics	**Metric**	**Top-1**	**Top-3**	**Top-5**	**Top-10**
{Keyword-API Co-occurrence (KAC)}	Hit@K	19.43%	42.29%	58.86%	76.00%
	MRR@K	0.19	0.29	0.33	0.36
	MAP@K	19.43%	29.05%	31.94%	32.57%
	MR@K	5.97%	15.35%	25.71%	46.42%
{Keyword Pair-API Co-occurrence (KPAC)}	Hit@K	36.57%	58.86%	69.14%	**79.43**%
	MRR@K	0.37	0.46	0.49	**0.50**
	MAP@K	36.57%	**46.19**%	46.13%	43.65%
	MR@K	11.08%%	24.88%	36.20%	**52.21**%
{Keyword-Keyword Coherence (KKC)}	Hit@K	13.71%	32.57%	41.14%	55.43%
	MRR@K	0.14	.22	0.24	0.26
	MAP@K	13.71%	21.52%	23.05%	24.26%
	MR@K	4.46%	12.32%	18.07%	28.29%
{KAC + KKC} [58]	Hit@K	17.71%	40.00%	58.29%	**77.71**%
	MRR@K	0.18	0.28	0.32	0.34
	MAP@K	17.71%	27.57%	30.24%	30.84%
	MR@K	5.65%	14.66%	25.56%	**46.15**%
RACK	Hit@K	38.29%	61.14%	72.00%	**83.43**%
	MRR@K	0.38	0.48	0.48	**0.52**
	MAP@K	38.29%	48.14%	**48.39**%	45.74%
	MR@K	12.12%	26.41%	37.94%	**54.07**%

We also determine correlation between four performance measures (e.g., Hit@10, reciprocal rank, average precision, recall) of our API suggestions (against NL queries) and the coverage of their corresponding ground truth in the constructed API database (Section 3.1). We employed two correlation methods – *Pearson* and *Spearman*, and found either very weak or negligible correlations (i.e., $0.04 \leq \rho \leq 0.12$) between those two entities. That is, the API suggestion performance of RACK is not biased by the coverage of the ground truth API classes in our API database. Such finding strengthens the external validity of our results.

RACK suggests relevant API classes for about **83%** of the generic NL queries with a mean average precision@10 of **40%–46%**, a mean reciprocal rank@10 of **0.46–0.52**, and a mean recall@10 of **53%–54%**, which are highly promising.

4.7 Answering **RQ$_5$**: How effective are the proposed heuristics–KAC, KPAC and KKC– in capturing the relevant API classes for a query?

We investigate the effectiveness of our adopted heuristics– KAC, KPAC and KKC, and justify their combination in the API ranking algorithm (i.e., Algorithm 1). Table 8 and Fig. 14 demonstrate how each heuristic performs in capturing the relevant APIs for a given set of code search query as follows:

From Table 8, we see that our technique suggests correct API classes for 78.00% and 79% of the queries when KAC and KPAC heuristics are employed respectively. Both heuristics leverage co-occurrences between query keywords (in the question titles) and API classes (in the accepted answers) from Stack Overflow for such

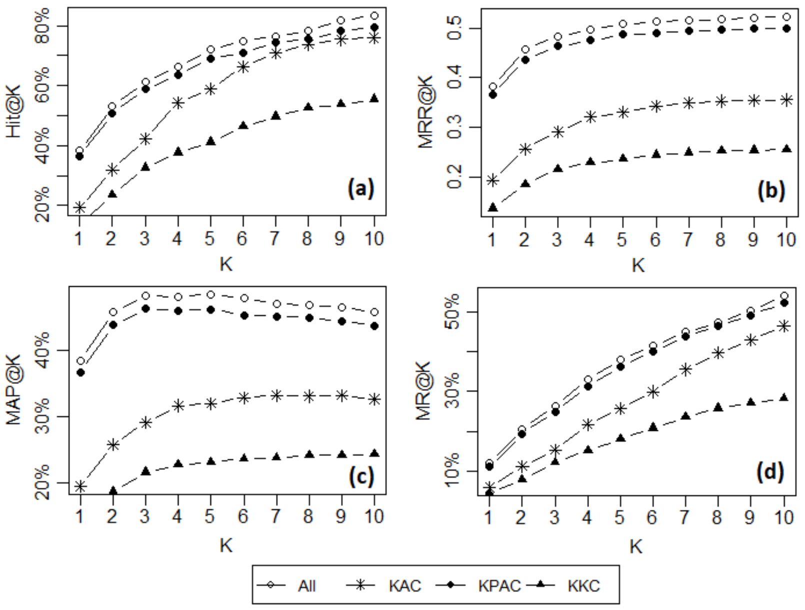


Fig. 14: (a) Hit@K of RACK, (b) Mean Average Precision@K (MAP@K) of RACK, and (c) Mean Recall@K (MR@K) of RACK for three heuristics–KAC, KPAC and KKC

recommendation. On the contrary, KKC considers coherence among the candidate API classes, and is found less effective than the former two heuristics. In fact, KPAC performs the best among all three heuristics with up to 46% precision and 52% recall. However, the weighted combination of our heuristics provides the maximum performance in terms of four metrics. It provides 83% Hit@10 with a mean reciprocal rank@10 of 0.52, a mean average precision@10 of 46% and a mean recall@10 of 54%. That is, our combination harnesses the strength from all the heuristics, and also overcomes their weaknesses simultaneously using appropriate weights. All these statistics are also highly promising according to the relevant literature [46, 72]. Thus, our combination of these three heuristics is also justified. Our earlier work combines KAC and KKC, and provides 79% Hit@10 with 35% precision and 45% recall from the experiments with 150 queries. Replication with our current extended dataset (i.e., 175 queries) reports similar performance (e.g., 78% Hit@10), which supports our earlier findings [58] as well. In this work, we introduce the new heuristic–KPAC–which improved our performance in terms of all four metrics– Hit@10 (i.e., 7% improvement), reciprocal rank (i.e., 53% improvement), precision (i.e., 48% improvement) and recall (i.e., 17% improvement). Thus, the addition of KPAC heuristic to our ranking algorithm is justified. Furthermore, we apply appropriate weights to these heuristics for controlling their influence in the API relevance ranking. Fig. 14 further demonstrates how the performance of our heuristics changes over various Top-K results. We see that KPAC is the most dominant one among the heuristics (as observed above) and achieves the maximum performance. However, the addition of the other two heuristics also improves our performance marginally (i.e., 2% – 4%) in terms of all four metrics.

Table 9: **Impact of Different Query Term Selection**

Query Terms	Metric	Top-1	Top-3	Top-5	Top-10
All terms from query	Hit@K	37.14%	60.57%	71.43%	82.86%
	MRR@K	0.37	0.48	0.50	0.52
	MAP@K	37.14%	47.29%	47.81%	45.29%
	MR@K	11.69%	26.93%	38.85%	54.80%
Noun terms only	Hit@K	33.71%	58.86%	70.29%	82.29%
	MRR@K	0.34	0.45	0.48	0.49
	MAP@K	33.71%	44.95%	45.71%	42.62%
	MR@K	10.56%	25.47%	36.67%	55.21%
Verb terms only	Hit@K	7.43%	17.71%	24.00%	35.43%
	MRR@K	0.07	0.11	0.13	0.14
	MAP@K	7.43%	11.52%	12.68%	14.02%
	MR@K	2.14%	6.69%	10.46%	17.38%
{Noun terms + Verb terms}	Hit@K	38.29%	61.14%	72.00%	83.43%
	MRR@K	0.38	0.48	0.48	**0.52**
	MAP@K	38.29%	48.14%	48.39%	45.74%
	MR@K	12.12%	26.41%	37.94%	54.07%
{Noun terms + Verb terms}-“java”	Hit@K	37.14%	60.57%	72.00%	83.43%
	MRR@K	0.37	0.47	0.47	0.52
	MAP@K	37.14%	46.90%	47.35%	45.19%
	MR@K	11.84%	26.18%	38.09%	54.08%

KPAC and **KAC** are found more effective than KKC in capturing the relevant API classes from Stack Overflow Q & A threads. However, combination of all three heuristics using appropriate relative weights delivers the maximum performance. Thus, their combination for API ranking is justified.

4.8 Answering $\mathbf{RQ_6}$: Does an appropriate subset of the query keywords perform better than the whole query in retrieving the relevant API classes?

Since the proposed technique identifies relevant API classes based on their co-occurrences with the keywords from a query, the keywords should be chosen carefully. Selection of random keywords might not return appropriate API classes. Several earlier studies choose nouns and verbs from a sentence, and report their salience in automated comment generation [79] and corpus indexing [19]. We thus also extract noun and verb terms from each query as the search keywords using Stanford POS tagger [73], and then use them for our experiments. In particular, we investigate whether our selection of keywords for code search is effective or not.

From Table 9, we see that our technique performs better with *noun*-based keywords than with *verb*-based keywords. The verb-based keywords provide a maximum of 35% Hit@10. On the contrary, RACK returns correct API classes for 82% of queries with 43% precision, 55% recall and a reciprocal rank of 0.49 when only noun-based keywords are chosen for search. However, none of the performance metrics reaches the baseline performance except recall. That is, they are lower than the performance of RACK with all query terms minus the stop words. Interestingly, when both nouns and verbs are employed as search keywords, the performance reaches the maximum especially in terms of accuracy, precision and reciprocal rank. For example, RACK achieves 83% Hit@10 with 46% precision, 54% recall and a reciprocal rank of 0.52. Although the improvement over the baseline performance (i.e., with all keywords of a query) is marginal, such performances were delivered using a fewer number of search keywords. That is, our subset of keywords not only avoids the noise but also ensures a comparatively higher performance than the baseline with relatively lower costs (i.e., fewer keywords). Thus, selection of a subset of keywords from the NL query intended for code search is justified, and our subset is also found effective.

We also investigate the impact of generic search keywords such as "java" in our query. According to our analysis, 11.43% of our queries in the dataset contain this keyword. From Table 9, we see that removal of this keyword marginally degrades most of the performance measures of our technique. Only marginal improvements can be observed in the recall measure for Top-5 and Top-10 results. Thus, our choice of retaining the generic keywords is also justified.

Important keywords from a natural language query mainly consist of its **noun** and **verb** terms. Our keyword selection approach of leveraging noun and verbs from a query is found quite effective in the relevant API suggestion.

4.9 Answering $\mathbf{RQ_7}$: How do the heuristic weights (i.e., α, β) and threshold settings (i.e., γ, δ) influence the performance of our technique?

Our relevance ranking algorithm applies two relative weights–α and β–to our proposed heuristics, and the heuristics are also constrained with two thresholds–γ and δ. While the thresholds help the heuristics collect appropriate candidate API classes, the weights control the influence of the heuristics in the API relevance ranking. In this section, we justify our chosen weights and thresholds, and investigate how they affect the performance of our technique.

We adopt a greedy search-based technique [83] (i.e., controlled iterative approach) for determining the relative weights for our heuristics. That is, we start our searches with our best initial guesses for α (i.e., 0.25) and β (i.e., 0.30), refine our weight estimates in every iteration with a step size of 0.025, and then stop when the *fitness function* [83] (i.e., performance) reaches the global maximum. We use mean average precision@10 and mean recall@10 as the fitness functions in the search for α and β. Fig. 15 shows how different values of α and β can influence the performance of RACK. Please note that when one weight is calibrated, the other one is kept constant during performance computation. We see that precision and recall of RACK reach the maximum when $\alpha \in [0.300, 0.325]$ and β=0.575. The target weights are identified using dashed vertical lines above. While α and β are considered as the relative importance of the co-occurrence based heuristics, KAC and KPAC respectively, $(1 - \alpha - \beta)$ goes to the remaining heuristic–KKC. Since KKC is found relatively weak according to our earlier investigation, we emphasize more on α and β, and chose the following heuristic weights: 0.325, 0.575 and 0.10– for KAC, KPAC and KKC respectively. Thus, all the weights sum to 1, and such weighting mechanism was also used by an earlier study [48]. The performance of RACK is significantly higher than its non-weighted version especially in terms of

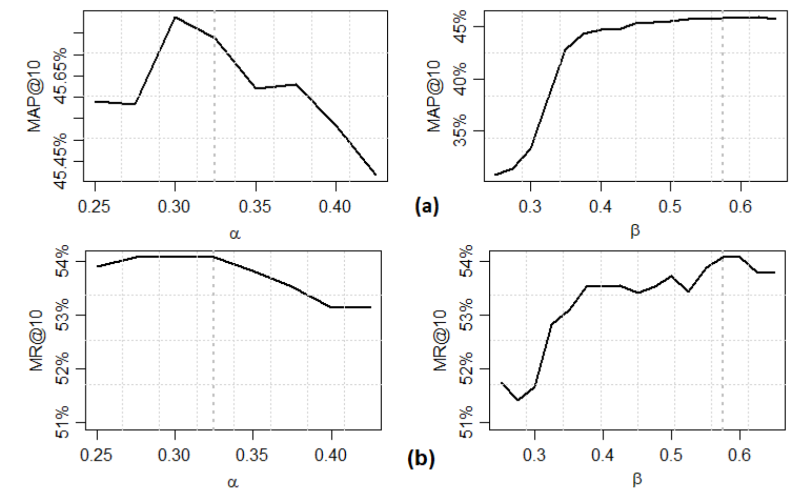


Fig. 15: (a) Mean Average Precision@10 (MAP@10), and (b) Mean Recall@10 (MR@10) of RACK for different values of the heuristic weights–α and β

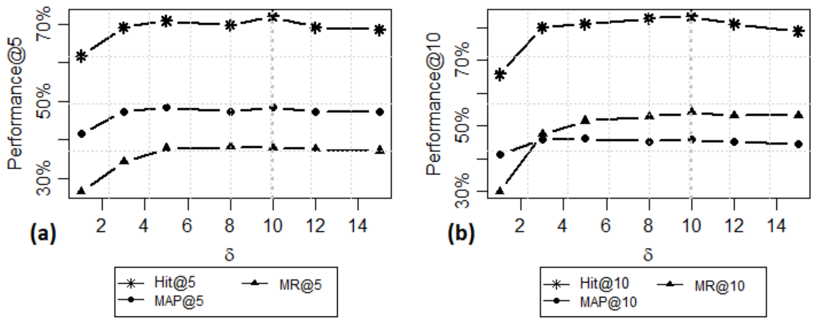


Fig. 16: Performance of RACK for different δ thresholds with (a) Top-5 results and (b) Top-10 results considered

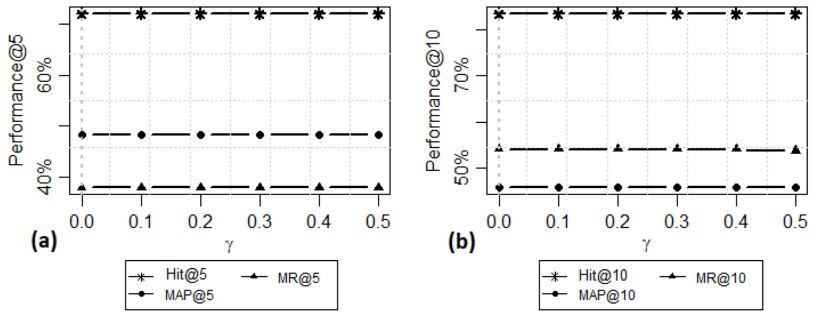


Fig. 17: Performance of RACK for different γ thresholds with (a) Top-5 results and (b) Top-10 results considered

MRR@K (i.e., WSR, *p-value*$= 0.002$, $\Delta = 1.00$ (large)) and MAP@K (i.e., WSR, *p-value*< 0.001, $\Delta = 0.84$ (large)) for Top-1 to Top-10 results. Thus, the application of relative weights to our adopted heuristics is also justified.

Both KAC and KPAC apply δ threshold for collecting candidate API classes from the token-API linking database. Fig. 16 shows how different values of δ can affect our performance. We use Hit@K, MAP@K and MR@K as the fitness functions, and determine our fitness for Top-5 and Top-10 returned results. We see that each of these performance measures reach their maximum when $\delta = 10$ for both settings. That is, collecting 10 candidate API classes for each keyword or keyword pair from the query is the most appropriate choice. Less or more than that provides comparable performance but not the best one. Thus, we chose $\delta = 10$ in our algorithm, and our choice is justified.

KKC applies another threshold, γ, for candidate API selection that refers to the degree of contextual similarity between any two keywords from the query. Fig. 17 reports our investigation on this threshold. We see that different values of γ starting from 0 to 0.5 do not change our fitness (i.e., performance) at all. Since the heuristic itself, KKC, is not strong, the variance of γ also does not have much influence on the performance of our technique. Thus, our choice of $\gamma = 0$ is also justified. That is, we consider two API classes coherent to each other when their contexts share at least one search keyword.

The performance of RACK reaches the *maximum* for certain weights and thresholds, α=**0.325**, β=**0.575**, γ=**0**, and δ=**10**. They were chosen carefully based on controlled iterative experiments, as were also done by the earlier studies [48, 83] from relevant literature.

4.10 Answering **RQ$_8$**: Can RACK outperform the state-of-the-art techniques in suggesting relevant API classes for a given set of queries?

Thung et al. [72] accept a feature request as an input and return a list of relevant API methods. Their API suggestions are based not only on the mining of feature request history but also on the textual similarity between the request texts and the corresponding API documentations. Zhang et al. [84] determine semantic distance between an NL query and a candidate API using a neural network model (CBOW) and a large code repository, and then suggest a list of relevant API classes for the query. To the best of our knowledge, these are the latest and the closest studies to ours in the context of API suggestion, and thus, we select them for comparison.

Since feature request history is not available in our experimental settings, we implement *Description-Based Recommender* module from Thung et al. We collect API documentations of 3,300 classes from the Java standard libraries (i.e., JDK 6), and develop Vector Space Model (VSM) for each of the API classes. In fact, we develop two models for each API class using (1) class header comments only, and (2) class header comments + method header comments, and implement two variants– Thung et al.-I and Thung et al.-II for our experiments. We use *Apache Lucene* [8] for VSM development, corpus indexing and for textual similarity matching between the API documentations and each of the queries from our dataset. In the case of Zhang et al., we (1) make use of *IJaDataset* [36] as a training corpus (as was done by the original authors), and (2) learn the word embeddings for both keywords and API classes using *fastText* [15], an improved version of *word2vec* implementation. We then use these vectors to determine semantic distance between

Table 10: **Comparison of API Recommendation Performance with Existing Techniques (for various Top-K Results)**

Technique	Metric	Top-1	Top-3	Top-5	Top-10
Thung et al. [72]-I	Hit@K	20.57%	30.85%	38.29%	44.00%
	MRR@K	0.21	0.25	0.26	0.27
	MAP@K	20.57%	24.57%	25.47%	24.84%
	MR@K	6.37%	11.74%	15.79%	22.19%
Thung et al. [72]-II	Hit@K	20.00%	32.57%	39.43%	**50.29%**
	MRR@K	0.20	0.26	0.27	**0.29**
	MAP@K	20.00%	25.14%	**25.85%**	25.59%
	MR@K	6.19%	13.02%	18.47%	28.95%
Zhang et al. [84]	Hit@K	19.43%	32.00%	36.00%	39.43%
	MRR@K	0.19	0.25	0.26	0.26
	MAP@K	19.43%	24.86%	25.44%	24.81%
	MR@K	6.00%	15.86%	21.54%	**29.87%**
RACK (Proposed technique)	Hit@K	38.29%	61.14%	72.00%	**83.43%**
	MRR@K	0.38	0.48	0.48	**0.52**
	MAP@K	38.29%	48.14%	**48.39%**	45.74%
	MR@K	12.12%	26.41%	37.94%%	**54.07%**

***Emboldened** items are the highest statistics for the existing and proposed techniques

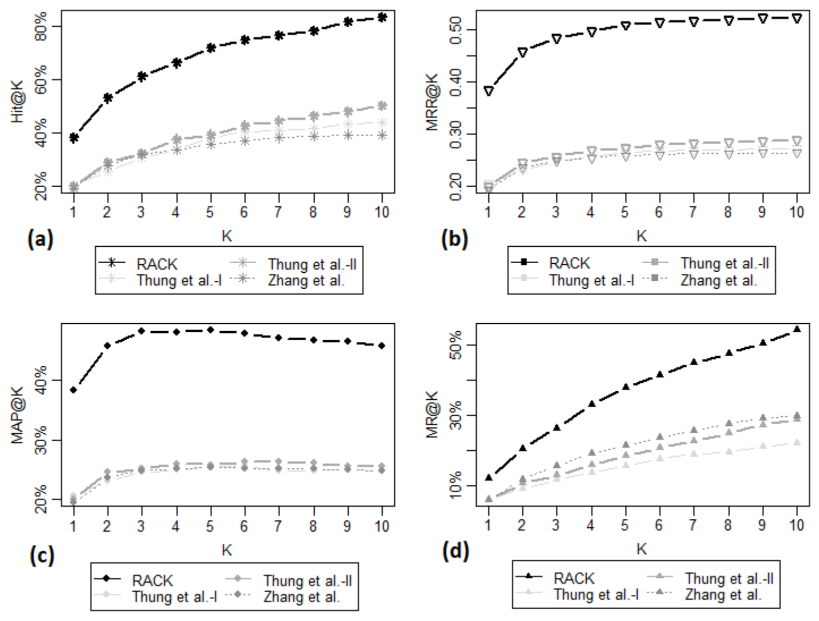


Fig. 18: Comparison of API recommendation performances with the existing techniques-(a) Hit@K, (b) Mean Reciprocal Rank@K, (c) Mean Average Precision@K, and (d) Mean Recall@K

a query and the candidate API classes using *cosine similarity* [57]. We also determine API popularity within the training corpus, and then combine with semantic distance metric to identify a set of relevant API classes for the NL query.

Table 10 summarizes the comparative analysis between our technique–RACK– and three existing techniques. Here, emboldened items refer to maximum measures provided by the existing techniques and our technique. We see that the variants of Thung et al. can provide a maximum of about 50% accuracy with about 26% precision and 29% recall for Top-10 results. On the other hand, RACK achieves a maximum accuracy of 83% with 46% precision and 54% recall which are 66%, 79% and 87% higher respectively. We investigate how the four performance measures change for different Top-K results for each of these three techniques. From Fig. 18, we see that Hit@K of RACK increases gradually up to 83% whereas such performance measures for the textual similarity based techniques stop at 50%. The MRR@K of RACK improves from 0.38 to 0.52 whereas such measures for the counterparts are as low as 0.20–0.29. It should be noted that RACK reaches its maximum precision, i.e., 48%, quite early at $K = 3$, and then its recall gradually improves up to 54% (at $K = 10$). On the contrary, such measures for the counterparts are at best 25% and 30% respectively. These demonstrate the superiority of our technique. From the box plots in Fig. 19, we see that RACK performs significantly higher than both variants in terms of all three metrics– accuracy, precision and recall. Our median accuracy is above 70% whereas such measures for those variants are close to 40%. The same goes for precision and recall measures. We perform significance and effect size tests, and compare our performance measures with the measures of the state-of-the-art for various Top-K results ($1 \leq K \leq 10$). We found that the performance of our approach is significantly higher than that of the existing techniques in terms of Hit@K (i.e., WSR, *p-value*=0.002<0.05, Δ=0.79 (large)), MRR@K (i.e., WSR, *p-value*=0.002<0.05, Δ=0.90 (large)), MAP@K (i.e., WSR, *p-value*=0.002<0.05, Δ=0.90 (large)) and MR@K (i.e., WSR, *p-value*=0.002<0.05, Δ=0.70 (large)). All these findings above suggest that (1) textual similarity between query and API signature or documentations might not be always effective for API recommendation, and (2) semantic distance between keyword and API classes should be calculated using appropriate training corpus. Our technique overcomes that issue by applying three heuristics –KAC, KPAC and KKC– which leverage the API usage knowledge of a large developer crowd stored in Stack Overflow. Performance reported for Thung et al. is project-specific, and the technique is restricted to feature requests [72]. On the contrary, our technique is generic and adaptable for any type of code search. It is also independent of any subject systems. Although Zhang et al. employ a large training corpus, they learn word embeddings for NL keywords from the source code which might not be always helpful. Source code inherently has a smaller vocabulary than regular texts [32]. On the contrary, we leverage the contexts of NL keywords and API classes more carefully from Stack Overflow Q & A site to determine their relevance. Furthermore, we harnesses the expertise of a large crowd of technical users effectively for relevant API suggestion which was not considered by the past studies from literature. Thus, our technique possibly has a greater potential.

RACK outperforms *multiple* existing studies on relevant API suggestion for NL queries, and achieves **66%** higher accuracy, **79%** higher precision and **87%** higher recall than those of the state-of-the-art.

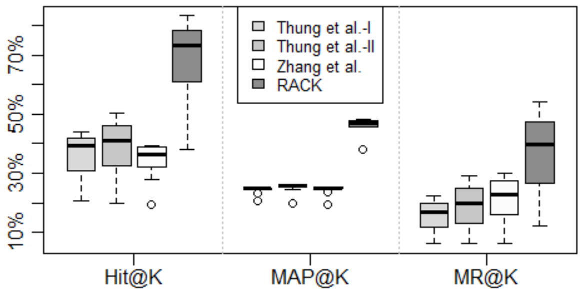


Fig. 19: Comparison of API recommendation with existing techniques using box plots

Table 11: **Comparison of Source Code Retrieval Performance with Baseline Queries**

Query	Metric	Top-1	Top-3	Top-5	Top-10
Retrieval Performance with Small Dataset (4K-Corpus)					
Baseline (NL Keywords)	Hit@K	39.43%	54.86%	62.29%	68.57%
	MRR@K	0.39	0.46	0.48	0.49
Goldset API	Hit@K	65.71%	85.71%	89.14%	91.43%
	MRR@K	0.66	0.75	0.76	0.76
Baseline + Goldset API	Hit@K	70.29%	88.00%	96.00%	97.14%
	MRR@K	0.70	0.78	0.80	0.80
RACK$_A$	Hit@K	29.71%	50.29%	56.00%	68.57%
	MRR@K	0.30	0.39	0.40	0.42
RACK$_{A+Q}$	Hit@K	**50.86%**	**73.14%**	**77.71%**	**84.00%**
	MRR@K	**0.51**	**0.61**	**0.62**	**0.63**
Retrieval Performance with Large Dataset (256K-Corpus)					
Baseline (NL Keywords)	Hit@K	22.29%	30.86%	37.71%	44.00%
	MRR@K	0.22	0.26	0.27	0.28
Goldset API	Hit@K	60.00%	78.29%	84.57%	90.29%
	MRR@K	0.60	0.69	0.70	0.71
Baseline + Goldset API	Hit@K	76.00%	89.14%	90.86%	94.86%
	MRR@K	0.76	0.82	0.82	0.83
RACK$_A$	Hit@K	14.29%	26.29%	30.86%	36.57%
	MRR@K	0.14	0.19	0.20	0.21
RACK$_{A+Q}$	Hit@K	**40.00%**	**52.57%**	**59.43%**	**66.29%**
	MRR@K	**0.40**	**0.46**	**0.47**	**0.48**
Retrieval Performance with Extra-Large Dataset (769K-Corpus)					
Baseline (NL Keywords)	Hit@K	17.14%	24.57%	0.28.57%	34.29%
	MRR@K	0.17	0.20	0.21	0.22
Goldset API	Hit@K	50.86%	69.14%	75.43%	81.14%
	MRR@K	0.51	0.59	0.61	0.62
Baseline + Goldset API	Hit@K	64.00%	80.00%	86.86%	90.29%
	MRR@K	0.64	0.71	0.73	0.73
RACK$_A$	Hit@K	10.86%	18.29%	22.29%	26.86%
	MRR@K	0.11	0.14	0.15	0.16
RACK$_{A+Q}$	Hit@K	**26.86%**	**42.29%**	**49.14%**	**56.57%**
	MRR@K	**0.27**	**0.33**	**0.35**	**0.36**

A=Suggested API classes only, **A+Q**=Reformulated query combining both suggested API classes and baseline query keywords.

4.11 Answering **RQ$_9$**: Can RACK significantly improve the natural language queries in terms of relevant code retrieval performance?

Our earlier research questions (RQ$_4$–RQ$_8$) evaluate the performance of RACK in suggesting relevant API classes for a natural language query intended for code search. Although they clearly demonstrate the potential of our technique, another way of evaluation could be the retrieval performance of our suggested queries. In this section, we investigate whether our reformulations to the baseline queries improve them or not in terms of their relevant code retrieval performances. We employ three corpora – *4K-Corpus*, *256K-Corpus*, and *769K-Corpus*– each of which includes 175 ground truth code segments (see Section 4.1 for details). We apply limited natural language preprocessing (i.e., removal of stop words and keywords, splitting of complex tokens) to each corpus document, and then index them for retrieval. We employ *Apache Lucene*[8], a popular code search engine that has been used by several earlier studies from the literature [30, 48, 52], for document indexing and for source code retrieval.

Table 11 and Fig. 20 summarize our findings on comparing our reformulated queries with the baseline queries. We consider two versions of our reformulated queries– RACK_A and RACK_{A+Q}–for our experiments. While RACK_A comprises of suggested API classes only, RACK_{A+Q} combines both the suggested API classes and the NL keywords from baseline queries. From Table 11, we see that the baseline queries (i.e., comprise of NL keywords) perform poorly especially with the large corpora. In the case of *256K-Corpus*, they return relevant code segments at the Top-1 position and within the Top-5 positions for only 22% and 38% of the queries respectively (i.e., Hit@K). On the contrary, our reformulated queries, RACK_{A+Q}, can return relevant code segments for 40% and 59% of the queries within Top-1 and Top-5 positions respectively, which are more promising. We see a notable increase in the query performance with the smaller corpus (i.e., 4K-Corpus) and a notable decrease with the bigger corpus (i.e., 769K-Corpus). Such observations can be explained by the reduced and added noise in the corpus respectively. However, our reformulated queries perform consistently higher than the baseline across all three corpora. For example, while the baseline Hit@10 reduces to 34% for 769K-Corpus, our reformulated queries deliver a Hit@10 of 57% which is 65% higher. Thus, our query reformulations offer 23%-80% improvement in Hit@K over the baseline performance across the three corpora. It should be noted that Hit@1 and Hit@5 could reach up to 60% and 85% respectively when the goldset API classes are used as the search queries. Combination of NL queries and goldset API classes performs even better. Such findings also strengthen our idea of suggesting and using relevant API classes for code search. However, we also see that reformulated queries containing both NL keywords and API classes (e.g., RACK_{A+Q}) are always better than those containing only the suggested API classes (e.g., RACK_A).

Our MRR@K measures in Table 11 are also found more promising. They suggest that on average, the relevant code segments are returned by our queries within the top three positions of the result list across all three corpora, which is promising from the perspective of practical use. Furthermore, our MRR@K measures are 29%–81% higher than the baseline counterparts across all three corpora which demonstrate the potential of our reformulated queries for code search.

Fig. 20 further demonstrates the performance of baseline queries and our reformulated queries for various Top-K results. We see that Hit@K and MRR@K of our queries are higher than those of the baseline queries by a large margin across all three corpora –*4K-Corpus*, *256K-Corpus*, and *769K-Corpus*. Non-parametric tests such as *Wilcoxon Singed Rank*, *Mann-Whitney Wilcoxon* and *Cliff's delta* tests also

[8] https://lucene.apache.org/

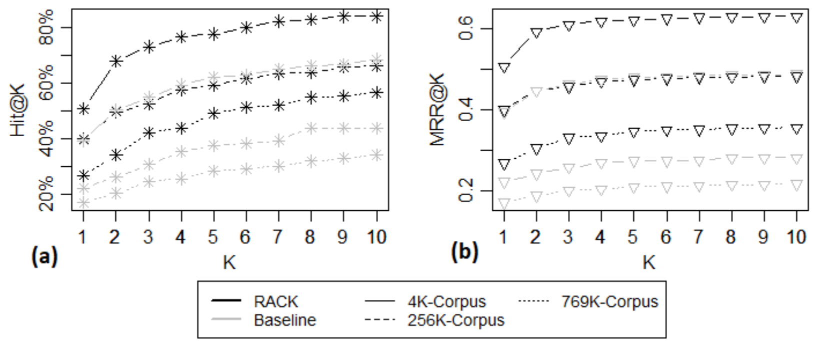


Fig. 20: Comparison of code retrieval performance with the baseline queries in terms of (a) Hit@K and (b) MRR@K

Table 12: **Improvement of Baseline Queries by RACK**

Query Pairs	Improved	Worsened	Net Gain	Preserved
Query Improvement with Small Dataset (4K-Corpus)				
Goldset API vs. Baseline	**54.29%**	13.71%	+40.58%	32.00%
RACK$_A$ vs. Baseline	42.29%	39.43%	+2.86%	18.29%
RACK$_{A+Q}$ vs. Baseline	**46.29%**	10.86%	**+35.43%**	**42.86%**
Query Improvement with Large Dataset (256K-Corpus)				
Goldset API vs. Baseline	**70.86%**	14.29%	+56.00%	14.86%
RACK$_A$	43.43%	48.00%	-4.57%	8.57%
RACK$_{A+Q}$	**61.71%**	13.14%	**+48.57%**	**25.14%**
Query Improvement with Extra-Large Dataset (769K-Corpus)				
Goldset API vs. Baseline	**74.86%**	14.86%	**+60.00%**	10.29%
RACK$_A$	44.00%	48.00%	-4.00%	8.00%
RACK$_{A+Q}$	**64.00%**	16.00%	**+48.00%**	**20.00%**

Net Gain = Gained improvement of result ranks through query reformulations

report statistical significance of our performance improvements for both Hit@K (i.e., all *p-values*$<$0.05, 0.82$\leq\Delta\leq$0.94 (large)) and MRR@K (i.e., all *p-values*$<$0.05, Δ=1.00 (large)). For the sake of simplicity, only one code segment (i.e., collected from the tutorial sites, Section 4.1) was chosen as the ground truth of each query. Thus, Hit@K and MRR@K are the most appropriate performance metrics for this case, and consequently, precision and recall were not chosen for this evaluation.

We also investigate query performance by relaxing the Top-K constraint and by analysing all the results returned by each query. Table 12 and Fig. 21 report our findings on query effectiveness [48, 49]. That is, if the first relevant code segment by a reformulated query is returned closer to the top of the result list than that of the baseline query, we consider it as *query quality improvement*, and vice versa as *query quality worsening*. If there is no change in the result ranks between baseline and reformulated queries, we call it *query quality preserving*. From Table 12, we see that 46%–64% of the baseline queries can be improved by our technique, RACK$_{A+Q}$, across all three corpora. It worsens only 11%–16% of the queries, and thus, offers a net gain of 35%–49% query improvement. While 60% net gain is possible in the best case scenario using gold set APIs directly, our technique delivers ≈ 50%, which is promising according to relevant literature [30, 55]. Fig. 21 further contrast between baseline and our reformulated queries. We see that the result ranks provided by

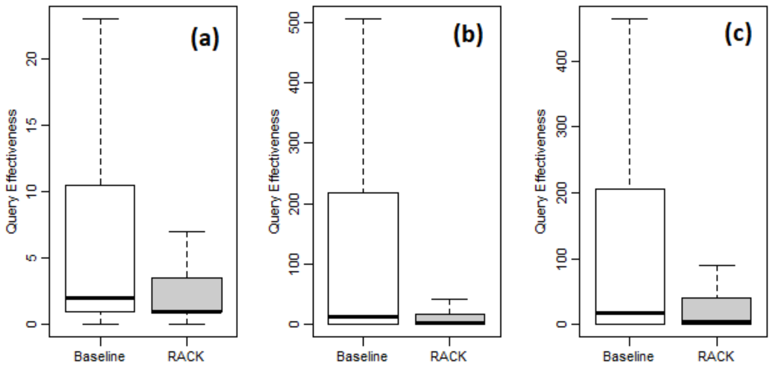


Fig. 21: Comparison of QE distribution with baseline queries across (a) 4K-Corpus, (b) 256K-Corpus and (c) 769K-Corpus

RACK are closer to zero (i.e., top of the list) across all three corpora. Such finding provides more evidence on the high potential of our suggested queries.

Reformulated queries by RACK retrieve relevant code segments with **23%–80%** higher accuracy and **29%-81%** higher reciprocal rank than those of the baseline queries. Furthermore, RACK improves **46%–64%** of the baseline queries, and they return the results closer to the top of the list.

4.12 Answering $\mathbf{RQ_{10}}$: Can RACK outperform the state-of-the-art techniques in improving the natural language queries intended for code search?

Although our reformulations improve the baseline queries significantly, we further validate them against the queries generated by existing techniques including the state-of-the-art. The study of Zhang et al. [84] is a closely related work to ours. They suggest relevant API classes for natural language queries intended for code search by analysing semantic distance between query keywords and API classes. Thung et al. [72] is another related study in the context of relevant API suggestion which was originally targeted for feature location (i.e., project-specific code search). Recently, Nie et al. [51] reformulate a query for code search by collecting pseudo-relevance feedback from Stack Overflow, and then by applying Rocchio's expansion [61] to the query. Their tool QECK suggests software-specific terms from programming questions and answers as query expansions. To the best of our knowledge, these are the most recent and the most closely related work to ours in the context of query reformulation for code search which make them the state-of-the-art. We thus compare our technique with these three existing techniques [51, 72, 84] in terms of Hit@K, MRR@K and Query Effectiveness (QE).

From Table 13, we see that the retrieval performance of RACK is consistently higher than that of the state-of-the-art techniques or their variants across all three corpora. Nie et al. [51], performs the best among the existing techniques. Their approach achieves 41%–75% Hit@5 with a MRR@5 between 0.31 to 0.59 on our dataset. However, our technique, RACK, achieves 49%–78% Hit@5 with 0.35–0.62 MRR@5 which are 4%–19% and 5%–13% higher respectively. RACK also achieves

Table 13: **Comparison of Code Retrieval Performance with Existing Techniques**

Technique	**Metric**	**Top-1**	**Top-3**	**Top-5**	**Top-10**
Retrieval Performance with Small Dataset (4K-Corpus)					
Thung et al. [72]-I	Hit@K	41.14%	58.29%	69.14%	74.29%
	MRR@K	0.41	0.49	0.51	0.52
Thung et al. [72]-II	Hit@K	44.00%	62.29%	71.43%	77.71%
	MRR@K	0.44	0.52	0.55	0.55
Nie et al. [51]	Hit@K	48.57%	69.14%	74.86%	**81.14%**
	MRR@K	0.49	0.58	0.59	**0.60**
Zhang et al. [84]	Hit@K	43.43%	64.00%	69.14%	77.71%
	MRR@K	0.43	0.53	0.54	0.55
RACK	Hit@K	**50.86%**	**73.14%**	**77.71%**	**84.00%**
	MRR@K	**0.51**	**0.61**	**0.62**	**0.63**
Retrieval Performance with Large Dataset (256K-Corpus)					
Thung et al. [72]-I	Hit@K	27.43%	40.57%	48.00%	54.86%
	MRR@K	0.27	0.33	0.35	0.36
Thung et al. [72]-II	Hit@K	33.71%	44.57%	50.29%	59.43%
	MRR@K	0.34	0.39	0.40	0.41
Nie et al. [51]	Hit@K	29.71%	44.00%	52.57%	**60.00%**
	MRR@K	0.30	0.36	0.38	**0.39**
Zhang et al. [84]	Hit@K	24.00%	34.29%	41.71%	52.57%
	MRR@K	0.24	0.29	0.30	0.32
RACK	Hit@K	**40.00%**	**52.57%**	**59.43%**	**66.29%**
	MRR@K	**0.40**	**0.46**	**0.47**	**0.48**
Retrieval Performance with Extra-Large Dataset (769K-Corpus)					
Thung et al. [72]-I	Hit@K	20.57%	29.71%	36.57%	42.86%
	MRR@K	0.21	0.24	0.26	0.27
Thung et al. [72]-II	Hit@K	25.71%	35.43%	41.14%	46.86%
	MRR@K	0.26	0.30	0.31	0.32
Nie et al. [51]	Hit@K	25.14%	36.57%	41.14%	**48.00%**
	MRR@K	0.25	0.30	0.31	**0.32**
Zhang et al. [84]	Hit@K	20.00%	28.57%	33.14%	38.29%
	MRR@K	0.20	0.24	0.25	0.26
RACK	Hit@K	**26.86%**	**42.29%**	**49.14%**	**56.57%**
	MRR@K	**0.27**	**0.33**	**0.35**	**0.36**

a Hit@10 of 57% with the extra-large corpus (i.e., 769K-Corpus) which is 18% higher than the state-of-the-art measure, i.e., 48% Hit@10 by Nie et al. While the performance measures of each technique degrade as the corpus size grows from 4K to 769K documents, our performance measures remain consistently higher than the state-of-the-art. Thus, RACK is more robust to varying sizes of corpora than any of the existing techniques under our study.

Fig. 22 further demonstrates how RACK outperforms the state-of-the-art techniques for various Top-K results in terms of Hit@K and MRR@K. We compare RACK with QECK by Nie et al. [51] for Top-1 to Top-10 performance measures using non-parametric tests. Nie et al. is clearly the state-of-the-art according to the above analysis. Our *Mann-Whitney Wilcoxon* and *Cliff's delta* tests reported statistical significance of RACK over Nie et al. with large effect sizes for both Hit@K (i.e., *p-values*$<$0.05, $0.33 \leq \Delta \leq 0.52$ (large)) and MRR@K (i.e., *p-values*$<$0.05, $0.68 \leq \Delta \leq 0.90$ (large)) across all three corpora. Thus, the findings above clearly demonstrate the superiority of our technique over the existing studies on query reformulation from the literature.

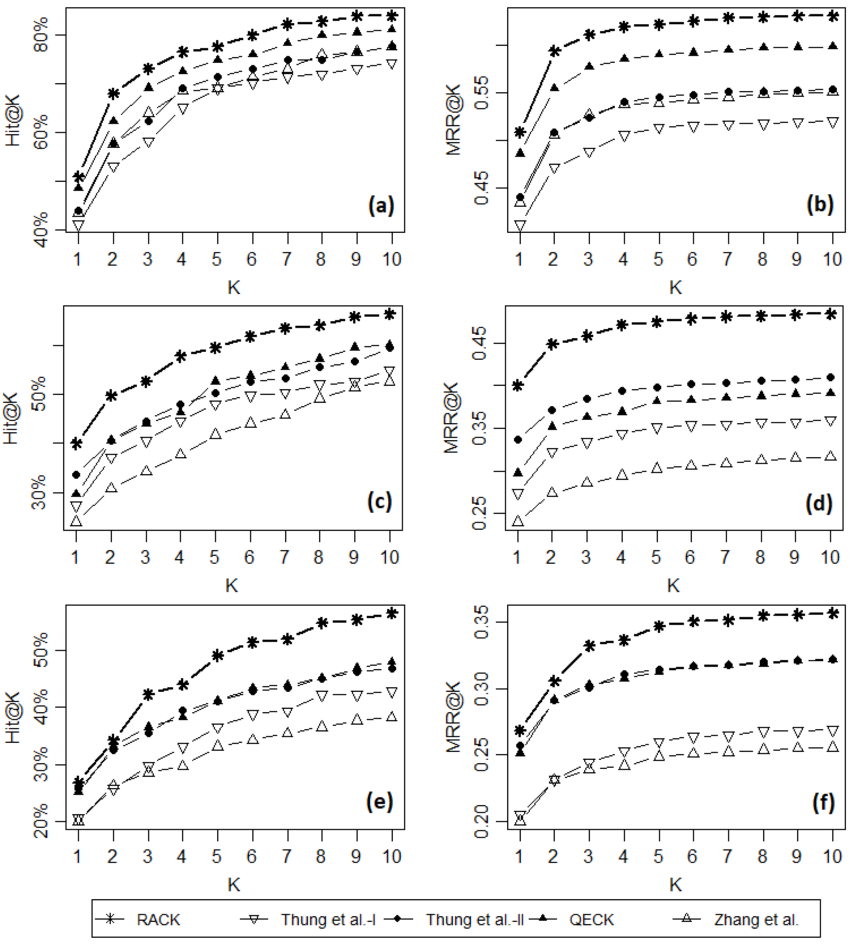


Fig. 22: Comparison of code retrieval performance with existing techniques using (a,b) 4K-Corpus, (c,d) 256K-Corpus and (e,f) 756K-Corpus

We also compare our technique with the existing techniques in terms of Query Effectiveness (QE). From Table 14, we see that Nie et al. performs the best with 4K-Corpus whereas Thung et al.-II performs the best with the remaining two corpora– 256K-Corpus and 769K-Corpus. Nie et al. improves 32% of the baseline queries whereas Thung et al.-II improves 43%–49% of the queries. On the contrary, RACK improves 46% and 62%–64% of the baseline queries in the same contexts. In particular, our technique offers 48% net gain as opposed to 26% provided by Thung et al.-II which is 87% higher. Thus, RACK clearly has a high potential for query reformulation than the state-of-the-art. It also should be noted that RACK degrades only 11%–16% of the queries across all three corpora which suggests the reliability and robustness of the technique. Fig. 23 further contrasts the result ranks of RACK with that of the state-of-the-art approaches using box plots. We see that on average, RACK provides higher ranks, and returns results closer to the top of list than the competing approaches. For example, Thung et al.-II returns 50% of its first correct results within the Top-8 positions and 75% of them within

Table 14: **Comparison of Query Improvements with Existing Techniques**

Query Pairs	**Improved**	**Worsened**	**Net Gain**	**Preserved**
Query Improvement with Small Dataset (4K-Corpus)				
Thung et al. [72]-I vs. Baseline	24.00%	11.43%	+12.57%	64.57%
Thung et al. [72]-II vs. Baseline	31.43%	10.86%	+20.57%	57.71%
Nie et al. [51] vs. Baseline	**32.00%**	8.00%	**+24.00%**	60.00%
Zhang et al. [84] vs. Baseline	28.00%	10.29%	+17.71%	61.71%
RACK vs. Baseline	**46.29%**	10.86%	**+35.43%**	**42.86%**
Query Improvement with Large Dataset (256K-Corpus)				
Thung et al. [72]-I vs. Baseline	37.71%	22.29%	+15.42%	40.00%
Thung et al. [72]-II vs. Baseline	**42.86%**	21.14%	**+21.72%**	36.00%
Nie et al. [51] vs. Baseline	41.71%	24.57%	+17.14%	33.71%
Zhang et al. [84] vs. Baseline	36.00%	26.86%	+9.14%	37.14%
RACK vs. Baseline	**61.71%**	13.14%	**+48.57%**	**25.14%**
Query Improvement with Extra-Large Dataset (769K-Corpus)				
Thung et al. [72]-I vs. Baseline	41.14%	25.71%	+15.43%	33.14%
Thung et al. [72]-II vs. Baseline	**48.57%**	22.86%	**+25.71%**	28.57%
Nie et al. [51] vs. Baseline	45.71%	24.57%	+21.14%	29.71%
Zhang et al. [84] vs. Baseline	41.14%	28.57%	12.57%	30.29%
RACK vs. Baseline	**64.00%**	16.00%	**+48.00%**	**20.00%**

Net Gain = Gained improvement of result ranks through query reformulations

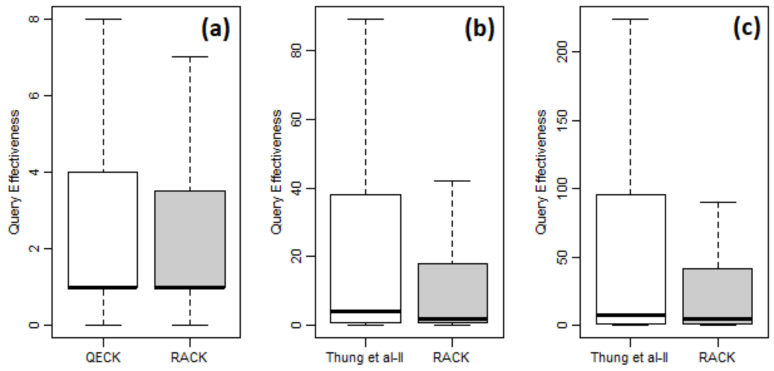


Fig. 23: Comparison of QE distribution with the state-of-the-art using (a) 4K-Corpus, (b) 256K-Corpus, and (c) 769K-Corpus

the Top-96 positions when dealing with extra-large corpus (i.e., 769K-Corpus). On the contrary, RACK returns such results within Top-5 and Top-42 positions which are 38% and 57% higher respectively. Similar findings can be observed with the remaining two corpora. All these findings above clearly demonstrate of superiority of our technique in query reformulation over the state-of-the-art.

Reformulated queries of RACK retrieve relevant code segments with **19%** higher accuracy and **13%** higher reciprocal rank than the state-of-the-art. Furthermore, RACK offers **48%** net improvement in the quality of baseline queries, which is **87%** higher than the state-of-the-art counterpart.

Table 15: **Comparison with Popular Web/Code Search Engines**

Technique	**Hit@10**	**MAP@10**	**MRR@10**	**NDCG@10**
Google	100.00%	68.56%	0.82	0.46
RACK$_{Google}$	100.00%	**83.71%**	**0.92**	**0.67**
Stack Overflow	91.43%	59.54%	0.67	0.43
RACK$_{SO}$	91.43%	**75.27%**	**0.82**	**0.62**
GitHub	89.71%	55.27%	0.58	0.47
RACK$_{GitHub}$	**90.29%**	**68.59%**	**0.74**	**0.59**

Emboldened= Comparatively higher than counterpart

4.13 Answering $\mathbf{RQ_{11}}$: How does RACK perform compared to the popular web search engines and code search engines?

Existing studies [50, 59, 64, 80] report that software developers frequently use general-purpose web search engines (e.g., Google) for code search. Hence, these search engines are natural candidates for comparison with our technique. We thus compare our approach with three popular web and code search engines– *Google*, *Stack Overflow native search* and *GitHub code search.* Unfortunately, we faced several challenges during our comparison with these commercial search engines. First, results from these search engines frequently change due to their dynamic indexing. This makes it hard to develop a reliable or stable oracle from their results. In fact, we found that Top-30 Google results collected for the same query in two different dates (i.e., two weeks apart) matched only 55%. Second, Google search API [2] was used for our experiments given that GUI based Google search is not a practical idea for 175 x 2 = 250 queries. However, this paid search API imposes certain restrictions on the number of API calls to be made. That is, results for 175 baseline queries and their reformulated queries could not be collected all at the same time. Given the changing nature of the underlying corpus, comparison between the results of baseline and reformulated queries could thus not be fair. Third, these commercial search engines are mostly designed for natural language queries. They also impose certain restrictions on the query length and query type. Hence, they might either produce poor results or totally fail to produce any results for our reformulated queries which mostly contain structured keywords (e.g., multiple API classes). Thus, we found a head-to-head comparison with these commercial search engines infeasible. Despite the above challenges, we still compare with them, and investigate whether our reformulated queries can improve their search results significantly or not through a post-processing step of their results.

Collection of Search Results and Construction of Oracle: We collect Top-30 results for each query from each search engine for oracle construction. We make use of *Custom Search API*[9] by Google and *native API endpoints* by Stack Overflow[10] and GitHub[11], and collect the search results. Given the large volume of search results (i.e., 175 x 30 = 5,250), it is impractical to manually analyze them all. Hence, we used a semi-automated approach in constructing the oracle for these web/code search engines. In particular, we extract the code segments from each of the result pages using appropriate tools (e.g., Jsoup[12]). In the case

[9] https://developers.google.com/custom-search
[10] https://api.stackexchange.com
[11] https://developer.github.com/v3
[12] https://jsoup.org/

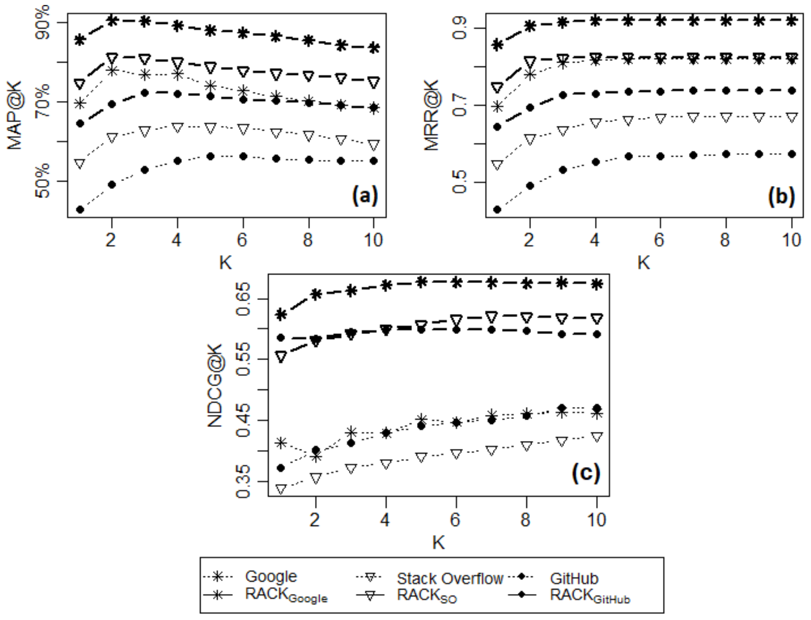


Fig. 24: Comparison of RACK with popular web/code search engines

of GitHub search results, we use *JavaParser*[13] to extract the method bodies as code segments. Then we determine their similarity against the original ground truth code that was extracted from tutorial sites in Section 4.1. For this, we use four code similarity algorithms – *Cosine similarity* [57], *Dice similarity* [30], *Jaccard similarity* [67] and *Longest Common Subsequence (LCS)* [63]. These algorithms are frequently used as the baseline for various code clone detection techniques [63, 67]. We collect four normalized code similarity scores from each result, average them, and then extract the Top-10 results containing the most relevant code segments. We then manually analyse a few of these results (and their code segments), and attempt to tweak them with various similarity score thresholds. Unfortunately, score thresholds were not sufficient enough to construct oracle for all the queries. We thus use these Top-10 results as the oracle for our web/code search engines.

Comparison between Initial Search Results and Re-ranked Results using the Reformulated Queries: Once a search engine returns results for natural language (NL) queries, we re-rank them with the corresponding reformulated queries provided by RACK. We first detect the presence of code segments in their contents, and then collect Top-10 documents based on their relevance to our reformulated queries (i.e., NL keywords + relevant API classes). We compare both the initial and re-ranked results with the oracle constructed above.

From Table 15, we see that our re-ranking approach improves upon the initial results returned by each of the web and code search engines. The improvements are observed especially in terms of precision, reciprocal rank and NDCG. For example, Google achieves 69% precision with a reciprocal rank of 0.82 and an NDCG of 0.46. However, our approach, RACK$_{Google}$ achieves 84% precision with a reciprocal rank of 0.92 and an NCCG of 0.67, which are 22%, 12% and 46% higher respectively.

[13] https://github.com/javaparser

That is, although Google performs high as a *general-purpose* web search engine, it might not be always precise for code search. Similar observation is shared by a recent survey [59] that reports that developers need more query reformulations during code search using the web search engines. GitHub native search achieves 55% precision, a reciprocal rank of 0.58 and an NDCG of 0.47. On the contrary, our approach, RACK$_{GitHub}$, delivers 69% precision with a reciprocal rank of 0.74 and an NDCG of 0.62, which are 24%, 28% and 26% higher respectively. Such findings demonstrate the potential of our reformulated queries. Fig. 24 further contrasts between our approach and the contemporary web/code search engines for Top-1 to Top-10 results. While Google is the best performer among the three search engines, our re-ranking using RACK outperforms Google with a significant margin in terms of precision (i.e., WSR, *p-value*$<$0.05, Δ=0.90 (large)), reciprocal rank (i.e., WSR, *p-value*$<$0.05, Δ=0.90 (large)) and NDCG (i.e., WSR, *p-value*$<$0.05, Δ=1.00 (large)). Thus, all the findings above suggest the high potential of our reformulated queries for improving the code search performed either with web or code search engines. The status quo of Internet-scale code search is far from ideal [59], and our reformulated queries could benefit the traditional practices.

Developers face difficulties in the code search while using contemporary web or code search engines (e.g., Google). Our technique can *significantly* improve their result ranks with the help of our reformulated queries that contain relevant API classes. In particular, RACK can improve upon the *precision* of Google in the code search by **22%**, which is promising.

5 Threats to Validity

We identify a few threats to the validity of our findings. While we attempt to mitigate most of them using appropriate measures, the remaining ones should be addressed in future work. Our identified threats and their mitigation details are discussed as follows:

5.1 Threats to Internal Validity

They relate to experimental errors and biases [83]. We develop a *gold set* for each query by analysing the code examples and the discussions from tutorial sites which might involve some subjectivity. However, each of the examples is a working solution to the corresponding task (i.e., NL-query), and they are frequently consulted. Thus, the gold set development using sample code from the tutorial sites is probably a more objective evaluation approach than human judgements of API relevance or code relevance that introduce more subjective bias [22]. According to the exploratory findings (Section 2.4), our technique might be effective only for the recommendation of popular and frequently used API classes. Since fully qualified names are mostly missing in Stack Overflow texts, third-party APIs similar to Java API classes could also have been mistakenly considered despite the fact that questions and answers selected for the study were tagged with `<java>`.

We use a dataset of 175 queries and a popular code search engine–*Apache Lucene* [30]–for determining their retrieval performance across three corpora of varying sizes. For the sake of simplicity, only one code segment was considered as relevant for each query. However, in practice, there could be multiple code segments in the corpus that are relevant to a given query. In this work, we trade such perfection with transparency and objectivity in our evaluation and validation.

During code or web search, developers generally choose the most appropriate keywords when a list of auto-generated suggestions are provided. We re-enact such behaviour of the developers by choosing only goldset API classes from within the suggested list, and use them for query reformulation. Such choice might have favoured the code retrieval performance of our queries. However, the same approach was carefully followed for all the existing techniques under study [51, 72, 84]. Thus, they received the same treatment in the performance evaluation as ours. Furthermore, the validation results (i.e., RQ_{10}) clearly report the superiority of our suggested queries over their counterparts from the existing techniques. Our investigation using the three contemporary web/code search engines also has drawn a similar conclusion for RACK (i.e., RQ_{11}).

5.2 Threats to External Validity

They relate to the generalizablity of a technique. So far, we experimented using API classes from only standard Java libraries. However, since our technique mainly exploits co-occurrence between keywords and APIs, the technique can be easily adapted for API recommendation in other programming domains. Since popularity of a programming language or change proneness of an API [43] has a significant role in triggering discussions at Stack Overflow which are mined by us, RACK could be effective for popular languages (e.g., Java, C#) but comparatively less effective for non-popular or less used languages (e.g., Erlang).

5.3 Threats to Construct Validity

Construct validity relates to suitability of evaluation metrics. Our work is aligned to both recommendation system and information retrieval domains. We use Hit@K and Reciprocal Rank which are widely used for evaluating recommendation systems [69, 72]. The remaining two metrics are well known in information retrieval, and are also frequently used by studies [22, 46, 72] relevant to our work. This confirms no or little threat to construct validity.

5.4 Threats to Statistical Conclusion Validity

Conclusion validity concerns the relationship between treatment and outcome [43]. We answer 11 research questions in this work, and collect our data from publicly available, popular programming Q & A and tutorial sites. In order to answer these questions, we use non-parametric tests for statistical significance (e.g., Mann-Whitney Wilcoxon, Wilcoxon Signed Rank), effect size analysis (e.g., Cliff's delta) and confidence interval analysis. We apply these tests to our experiments opportunistically and report the detailed test results (e.g., p-values, Cliff's Δ). Thus, threats to the statistical conclusion validity might be mitigated.

6 Related Work

Our work is aligned with three research topics–(1) API/API usage recommendation, (2) query reformulation for code search, and (3) crowdsourced knowledge mining. In this section, we discuss existing studies from the literature of each of these research topics, and compare or contrast our work with them.

6.1 API Recommendation

Existing studies on API recommendation accept one or more natural language queries, and recommend relevant API classes and methods by analysing code surfing behaviour of the developers and API invocation chains [46], API dependency graphs [22], feature request history or API documentations [72], and library usage patterns [71]. McMillan et al. [46] first propose *Portfolio* that recommends relevant API methods for a code search query by employing natural language processing, indexing and graph-based algorithms (e.g., PageRank [17]). Chan et al. [22] improve upon *Portfolio*, and return a connected sub-graph containing the most relevant APIs by employing further sophisticated graph-mining and textual similarity techniques. Gvero and Kuncak [28] accept a free-form NL-query, and return a list of relevant method signatures by employing natural language processing and statistical language modelling on the source code. A few studies offer NL interfaces for searching relevant program elements from the project source [40] or relevant artefacts from the project management repository [42]. Thung et al. [72] recommend relevant API methods to assist the implementation of an incoming feature request by analysing request history and textual similarity between API details and the request texts. In short, each of these relevant studies above analyse lexical similarity between a query and the signature or documentation of the API for finding out candidate APIs. Such approaches might not be always effective and might face vocabulary mismatch issues given that choice of query keywords could be highly subjective [25]. On the other hand, we exploit three co-occurrence heuristics that are derived from crowdsourced knowledge, and they are found to be more effective in the selection of candidate API classes. Co-occurrence heuristics overcome the vocabulary mismatch issues [25, 29], and provide a generic, both language and project independent solution. Besides, we exploit the expertise of a large crowd of technical users stored in Stack Overflow for API recommendation which none of the earlier relevant studies did. Zhang et al. [84] determine semantic distance between NL keywords and API classes using a neural network model (CBOW), and suggest relevant API classes for a generic NL query intended for code search. They collect their API classes from the OSS projects whereas ours are collected from Stack Overflow, the largest programming Q & A site on the web. Their work is closely related to ours. We compare with two variants of Thung et al. and Zhang et al., and readers are referred to Sections 4.10, 4.12 for the detailed comparison. Since Thung et al. outperform Chan et al. as reported [72], we compared with Thung et al. for our validation.

6.2 API Usage Pattern Recommendation

Thummalapenta and Xie [70] propose *ParseWeb* that takes in a *source object type* and a *destination object type*, and returns a sequence of method invocations that serve as a solution that yields the destination object from the source object. Xie and Pei [81] take a query that describes the method or class of an API, and recommends a frequent sequence of method invocations for the API by analysing hundreds of open source projects. Warr and Robillard [78] recommend a set of API methods that are relevant to a target method by analysing the structural dependencies between the two sets. Each of these techniques is relevant to our work since they recommend API methods. However, they operate on structured queries rather than natural language queries, and thus comparing ours with theirs is not feasible. Of course, we introduced three heuristics and exploited crowd knowledge for API recommendation, which were not considered by any of these existing techniques. This makes our contribution significantly different from all of them.

6.3 Query Reformulation for Code Search

There have been a number of studies on query reformulation that target either project-specific code search (e.g., concept/feature location [26, 29, 30, 33, 34, 39, 54, 55, 82], bug localization [23, 66]) or general-purpose code search [28, 41, 51]. Gay et al. [26] first propose "relevance feedback" based model for query reformulation in the context of concept location. Once the initial query retrieves search results, a developer is expected to mark them as either relevant or irrelevant. Then their model analyses these marked source documents, and expands the initial query using *Rocchio expansion* [61]. Although developer feedbacks on document relevance are effective, collecting them is time consuming and sometimes infeasible as well. Therefore, latter studies came up with a less efficient but feasible alternative–*pseudo relevance feedback*– for query reformulation where they consider only Top-K search results (retrieved by the initial query) as the relevant ones. Then they apply term weighting [38, 55, 61], term context analysis [33, 34, 66, 82], query quality analysis [29, 30], and machine learning [30] to reformulate a given query for concept/feature location. Our work falls into the category of general purpose code search. Relevance feedback models were also adopted in this case for query reformulation. Wang et al. [76] incorporate developer feedback in the code search, and improve result ranking. Nie et al. [51] employ Stack Overflow as the provider of relevance feedback on the initial query, and then reformulate it using Rocchio expansion. Although we do not apply relevance feedback for query reformulation, the work of Nie et al. is not only closely related to ours but also relatively more recent. Another closely related recent work by Zhang et al. [84] leverages semantic distance between NL keywords and API classes, and then expands the NL queries using semantically relevant API classes for code search. We thus compare our technique with three techniques above [51, 72, 84], and the detailed comparison can be found in RQ_{10}. Li et al. [41] develop a lexical database by using software-specific tags from Stack Overflow questions, and reformulate a given query using synonymy substitution. However, their approach searches for relevant software projects rather than source code segments. Campbell and Treude [18] mine titles from Stack Overflow questions, and suggest automatic expansion to the initial query in the form of auto-completion. However, this approach also relies on textual similarity between initial query and the expanded query, and thus, is subject to the vocabulary mismatch issues. On the contrary, we overcome such issues using three co-occurrence based heuristics. Besides, their approach is constrained by a fixed set of predefined queries from Stack Overflow questions, and thus, might not help much in the formulation of custom queries. RACK does not impose such restrictions on query formulation.

6.4 Crowdsourced Knowledge Mining

Existing studies [41, 51, 52, 54, 65, 83] leverage crowd generated knowledge to support several search related activities performed by the developers. Yuan et al. [83] first used programming questions and answers from Stack Overflow to identify semantically similar software specific word pairs. They first construct context of each word by collecting co-occurred words from Stack Overflow questions, answers and tags. Then they determine the semantic similarity between a pair of NL words based on the overlap between their corresponding contexts. Such word pairs might help in addressing the vocabulary mismatch issues with web search. However, they might not help much with code search given that source code and regular texts often hold different semantics for the same word [14, 82]. Wong et al. [79] mine developer's descriptions of the code snippets from Stack Overflow answers, and suggest them as comments for similar code segments. Rigby and Robillard [60] mine posts from Stack Overflow, and extract salient program elements using regular expressions and machine learning. Along the same line with the earlier studies, we mine Stack Overflow questions and answers to reformulate a given natural language query for code search. While our work is related to earlier studies [41, 51], it is also significantly different in many ways. First, we suggest relevant API classes for a NL-query by considering keyword-API co-occurrences whereas Nie et al. suggest mostly natural language terms as query expansions by employing pseudo-relevance feedback. Li et al. [41] reformulate queries using crowd wisdom from Stack Overflow for searching open source projects whereas our queries are targeted for more granular software artefacts, e.g., source code snippets. Furthermore, we suggest relevant API classes in contrast with synonymous NL tags by Li et al., which are more appropriate and effective for code search [14]. Another contemporary work [65] uses all program artifacts indiscriminately from Stack Overflow posts for expanding code search queries which could be noisy. On the contrary, we leverage co-occurrences between NL keywords (in the question title) and API classes (in the accepted answer) as a proxy to their relevance, and choose appropriate API classes only for our query reformulation.

Our work in this article also significantly extends our earlier work [9] in various aspects. We improve earlier heuristics by extensively calibrating their weights and thresholds, and introduce a novel heuristic– Keyword Pair API Co-occurrence– that performs better than the earlier ones. We conduct experiments with a relatively larger dataset containing 175 distinct queries, and further evaluate them in terms of relevant code retrieval performance which was missing in the earlier work. We not only compare with several state-of-the-art studies but also demonstrate RACK's potential for application in the context of traditional web/code search practices. Furthermore, we extend our earlier analysis and answer 11 research questions as opposed to seven questions answered by the earlier work.

7 Conclusion & Future Work

To summarize, we propose a novel query reformulation technique–RACK–that suggests a list of relevant API classes for a natural language query for code search. It employs three novel heuristics, and collects the the relevant API classes by exploiting crowdsourced knowledge stored in Stack Overflow questions and answers. Experiments using 175 code search queries from three Java tutorial sites show that RACK recommends relevant APIs with 83% Hit@10, 46% precision and 54% recall which are highly promising. Reformulated queries based on our recommended APIs significantly improve the baseline queries in terms of code retrieval performance. Comparison with the state-of-the-art techniques shows that our technique outperforms them not only in relevant API suggestion but also in query reformulation for code search by a significant margin. Furthermore, our technique is generic, project independent, and it exploits invaluable crowd generated knowledge for relevant API suggestion. Our work in this article has opened up the following future research directions:

– *Determining Relative API Salience:* Each programming task requires one or more API classes where some classes (e.g., `MimeMessage`) are more important than others (e.g., `Properties`) for the task (e.g., *"How do I send an HTML email?"*). However, based on our experience from this study, such relative importance is task-sensitive and sometimes even subjective. Given that code search queries are short and provide very little contexts about the task, determining the relative API salience is even more challenging. While we attempt to address this issue using three novel heuristics derived from crowd generated knowledge, further work is warranted (1) to better understand the issue, and (2) to return more effective ranking for the suggested API elements.
– *Query Quality Analysis:* Given multiple natural language queries for the same programming task, determining the best one without executing them is a challenging task. Identification of the best query could help the developers avoid numerous trials and errors or even performance regression. Information retrieval and Concept/feature location communities have long strived to address this challenge using several query quality/difficulty metrics and machine learning [20, 21, 29, 30]. Since we leverage keyword-API associations in this work for relevant API suggestion, such associations could possibly be leveraged for query quality estimation as well.

Acknowledgement

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

References

1. Theoretical CDF. URL `http://stats.stackexchange.com/questions/132652`.
2. Google custom search engine. URL `https://developers.google.com/custom-search`.
3. Stack Exchange Data Explorer. URL `http://data.stackexchange.com/stackoverflow`.
4. Java2s: Java Tutorials, . URL `http://java2s.com`.
5. JavaDB: Java Code Examples, . URL `http://www.javadb.com`.
6. Jsoup: Java HTML Parser. URL `http://jsoup.org`.
7. KodeJava: Java Examples. URL `http://kodejava.org`.
8. Apache Lucene Core. URL `https://lucene.apache.org/core`.
9. Rack website. URL `http://homepage.usask.ca/~masud.rahman/rack`.
10. Reflections Library. URL `https://code.google.com/p/reflections`.
11. Stopword List. URL `https://code.google.com/p/stop-words`.
12. A. Bacchelli, M. Lanza, and R. Robbes. Linking e-Mails and Source Code Artifacts. In *Proc. ICSE*, pages 375–384, 2010.
13. S. K. Bajracharya and C. V. Lopes. Analyzing and Mining a Code Search Engine Usage Log. *Empirical Softw. Engg.*, 17(4-5):424–466, 2012.
14. S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *EMSE*, 17(4-5):424–466, 2012.
15. P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.
16. J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.
17. S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.
18. B. A. Campbell and C. Treude. Nlp2code: Code snippet content assist via natural language tasks. In *Proc. ICSME*, pages 628–632, 2017.
19. G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based Traceability Recovery via Noun-Based Indexing of Software Artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.
20. D Carmel and E Yom-Tov. *Estimating the Query Difficulty for Information Retrieval.* Morgan & Claypool, 2010.
21. D Carmel, E Yom-Tov, A Darlow, and D Pelleg. What Makes a Query Difficult? In *Proc. SIGIR*, pages 390–397, 2006.
22. W. Chan, H. Cheng, and D. Lo. Searching Connected API Subgraph via Text Phrases. In *Proc. FSE*, pages 10:1–10:11, 2012.
23. O. Chaparro, J. M. Florez, and A Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proc. ICSME*, page to appear, 2017.
24. B. Dagenais and M.P. Robillard. Recovering Traceability Links between an API and its Learning Resources. In *Proc. ICSE*, pages 47–57, 2012.
25. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11): 964–971, 1987.
26. G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.

27. J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification: Java SE 7 Edition. 2012.
28. T. Gvero and V. Kuncak. Interactive synthesis using free-form queries. In *Proc. ICSE*, pages 689–692, 2015.
29. S. Haiduc and A. Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, June 2011.
30. S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.
31. Z. Harris. Mathematical Structures in Language Contents. 1968.
32. V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proc. ESEC/FSE*, pages 763–773, 2017.
33. E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.
34. M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.
35. K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.
36. I. Keivanloo and J. Rilling. Internet-scale java source code data set, 2011. URL `http://aseg.cs.concordia.ca/codesearch/#IJaDataSet`.
37. I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proc. ICSE*, pages 664–675, 2014.
38. K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.
39. K. Kevic and T. Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.
40. M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proc. ASE*, pages 376–379, 2011.
41. Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proc. Internetware*, pages 36–44, 2016.
42. J. Lin, Y. Liu, J. Guo, J. Cleland-Huang, W. Goss, W. Liu, S. Lohar, N. Monaikul, and A. Rasin. Tiqi: A natural language interface for querying software project data. In *Proc. ASE*, pages 973–977, 2017.
43. M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proc. ICPC*, pages 83–94, 2014.
44. C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010. URL `http://www.ics.uci.edu/~lopes/datasets/`.
45. L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design Lessons from the Fastest Q & A Site in the West. In *Proc. CHI*, pages 2857–2866, 2011.
46. C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proc. ICSE*, pages 111–120, 2011.
47. R. Mihalcea and P. Tarau. Textrank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.

48. L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proc. ICSME*, pages 151–160, 2014.
49. L Moreno, G Bavota, S Haiduc, M Di Penta, R Oliveto, B Russo, and A Marcus. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*, pages 567–578, 2015.
50. K. Nakasai, M. Tsunoda, and H. Hata. Web search behaviors for software development. In *Proc. CHASE*, pages 125–128, 2016.
51. L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *TSC*, 9(5):771–783, 2016.
52. L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proc. MSR*, pages 102–111, 2014.
53. M. M. Rahman and C. K. Roy. On the use of context in recommending exception handling code examples. In *Proc. SCAM*, pages 285–294, 2014.
54. M. M. Rahman and C. K. Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*, pages 220–225, 2016.
55. M. M. Rahman and C. K. Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.
56. M. M. Rahman and C. K. Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proc. ICSME*, page 12, 2018.
57. M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.
58. M. M. Rahman, C. K. Roy, and D. Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.
59. M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K T. Stolee, and Baishakhi Ray. Evaluating how developers use general-purpose web-search for code retrieval. In *Proc. MSR*, page 10, 2018.
60. P.C. Rigby and M.P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.
61. J.J. Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc.
62. J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.
63. C K Roy and J R Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. ICPC*, pages 172–181, 2008.
64. C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *Proc. ESEC/FSE*, pages 191–201, 2015.
65. R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *EMSE*, 2018.
66. B. Sisman and A. C. Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.

67. J. Svajlenko and C. K. Roy. Fast, scalable and user-guided clone detection. In *Proc. ICSE-C*, pages 352–353, 2018.
68. J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proc. ICSME*, pages 476–480, 2014.
69. P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.
70. S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.
71. F. Thung, D. Lo, and J. Lawall. Automated Library Recommendation. In *Proc. WCRE*, pages 182–191, 2013.
72. F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proc. ASE*, pages 290–300, 2013.
73. K. Toutanova and C. D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proc. EMNLP*, pages 63–70, 2000.
74. C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining Source Code Descriptions from Developers Discussions. In *Proc. ICPC*, pages 106–109, 2014.
75. S Wang and D Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proc. ICPC*, pages 53–63, 2014.
76. S. Wang, D. Lo, and L. Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proc. ASE*, pages 677–682, 2014.
77. Y. Wang, L. Wang, Y. Li, D. He, and T. Liu. A theoretical analysis of NDCG type ranking measures. In *Proc. COLT*, pages 25–54, 2013.
78. F. W. Warr and M. P. Robillard. Suade: Topology-Based Searches for Software Investigation. In *Proc. ICSE*, pages 780–783, 2007.
79. E. Wong, J. Yang, and L. Tan. AutoComment: Mining Question and Answer sites for Automatic Comment Generation. In *Proc. ASE*, pages 562–567, 2013.
80. Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *EMSE*, 22 (6):3149–3185, 2017.
81. T. Xie and J. Pei. MAPO: Mining Api Usages from Open Source Repositories. In *Proc. MSR*, pages 54–57, 2006.
82. J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.
83. T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.
84. F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related api class-names. *TSE*, page to appear, 2017.