9-2018

API method recommendation without worrying about the task-API knowledge gap

Qiao HUANG

Xin XIA

Zhenchang XING

David LO
Singapore Management University, davidlo@smu.edu.sg

Xinyu WANG

Citation

HUANG, Qiao; XIA, Xin; XING, Zhenchang; LO, David; and WANG, Xinyu. API method recommendation without worrying about the task-API knowledge gap. (2018). *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), Montpellier, France, 2018 September 3-7*. 293-304.
Available at: https://ink.library.smu.edu.sg/sis_research/4297

API Method Recommendation without Worrying about the Task-API Knowledge Gap

Qiao Huang
Zhejiang University
China
tkdsheep@zju.edu.cn

Xin Xia
Monash University
Australia
xin.xia@monash.edu

Zhenchang Xing
Australian National University
Australia
zhenchang.xing@anu.edu.au

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Xinyu Wang
Zhejiang University
China
wangxinyu@zju.edu.cn

ABSTRACT

Developers often need to search for appropriate APIs for their programming tasks. Although most libraries have API reference documentation, it is not easy to find appropriate APIs due to the lexical gap and knowledge gap between the natural language description of the programming task and the API description in API documentation. Here, the lexical gap refers to the fact that the same semantic meaning can be expressed by different words, and the knowledge gap refers to the fact that API documentation mainly describes API functionality and structure but lacks other types of information like concepts and purposes, which are usually the key information in the task description. In this paper, we propose an API recommendation approach named BIKER (Bi-Information source based KnowledgE Recommendation) to tackle these two gaps. To bridge the lexical gap, BIKER uses word embedding technique to calculate the similarity score between two text descriptions. Inspired by our survey findings that developers incorporate Stack Overflow posts and API documentation for bridging the knowledge gap, BIKER leverages Stack Overflow posts to extract candidate APIs for a program task, and ranks candidate APIs by considering the query's similarity with both Stack Overflow posts and API documentation. It also summarizes supplementary information (e.g., API description, code examples in Stack Overflow posts) for each API to help developers select the APIs that are most relevant to their tasks. Our evaluation with 413 API-related questions confirms the effectiveness of BIKER for both class- and method-level API recommendation, compared with state-of-the-art baselines. Our user study with 28 Java developers further demonstrates the practicality of BIKER for API search.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238191

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques*;

KEYWORDS

API Recommendation, API Documentation, Stack Overflow, Word Embedding

ACM Reference Format:
Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238191

1 INTRODUCTION

Application Programming Interfaces (APIs) in software libraries (e.g., Java SDK) play an important role in modern software development. With the help of APIs, developers can complete their programming tasks more efficiently. However, it is not easy to be familiar with all APIs in a large library. Thus, developers often need to check the API documentation to learn how to use an unfamiliar API for a programming task, and the prerequisite is that they already know which API to use but are just unfamiliar with the API. This situation can be referred to as "known unknowns".

However, a more practical scenario is that developers only have the requirement of a programming task, while they do not even know which API is worth learning (i.e., "unknown unknowns"). A possible solution is to use the natural language description of the programming task as a query, and use Information Retrieval (IR) approaches to obtain some candidate APIs whose documentation is similar to the query. However, this solution may not work well due to the lexical gap between the query and the API documentation. For example, given the query "*How to initialize all values in an array to false?*", the description of the most appropriate Java API method *Arrays.fill* is "*Assigns the specified boolean value to each element of the specified array of booleans.*", which does not contain any important keywords like *initialize* or *false* in the query.

Recently, a neural network-based approach called word embedding [27] has been proposed to capture the semantic meaning of different words. It represents each word by a low dimensional vector, and semantically similar words (e.g., *initialize* and *assign*, *false*

and *boolean*) would be close in the vector space. Ye et al. [49] leveraged word embedding to bridge the lexical gap between the query of programming task and Java API documentation. However, by replicating their study, we observe two major problems, as listed below.

The first problem is that they investigated API recommendation at class-level only. Given the above query example, their approach recommends only the *Arrays* class and developers still have to check about 50 methods to locate *Arrays.fill* if they check the methods one by one in the default order in the *Arrays* documentation. While their approach can be applied for method-level recommendation, its effectiveness is unknown.

The second problem is that even if their approach could bridge the lexical gap, it is still difficult to find the relevant API whose description does not share *semantically similar* words with the query. For example, given the query "*How to check whether a class exists?*", the most relevant Java API method recommended by Ye et al.'s approach is *org.omg.CORBA.Object._is_a*, whose description is "*Checks whether this object is an instance of a class that implements the given interface.*", and the similarity score between this description and the query is 0.669, since the two sentences have semantically similar words (e.g., *class* and *object*) or exactly the same words. However, the truly relevant API for the query is *java.lang.Class.forName*, whose description is "*Returns the Class object associated with the class with the given string name.*"; its similarity score with the query is only 0.377, since its description does not contain words similar to '*check*', '*whether*' or '*exists*'. However, *forName* can be used to "check whether a class exists". We call such mismatches between a task description and the API documentation as *task-API knowledge gap*, and our observation is also consistent with previous studies [23, 28, 31, 39], which pointed out that API documentation mainly describes API functionality and structure, but lacks other types of information (e.g., concepts or purposes).

To bridge this task-API knowledge gap, we conduct a survey with developers from two IT companies to understand how developers search for APIs to resolve programming tasks and the developers' expectations on automatic API recommendation techniques. From 47 responses, we find that when developers search APIs, a typical information seeking process is to browse a number of Stack Overflow (SO) questions and pick out the APIs that seem to be useful according to the discussions. Thus, SO is often exploited as a bridge between the programming task and the needed API(s). This is feasible because SO discussions are task centric and can complement API documentation with the missing concepts and purposes [39]. However, the decision on which API(s) to use is often not purely based on the SO discussions, and developers may further check API documentation to confirm the relevance of API(s). Furthermore, in the known unknowns setting, information like API description and code examples is crucial for determining which API(s) to use.

Inspired by this information seeking process, we propose an automatic approach named BIKER (Bi-Information source based KnowledgE Recommendation) which leverages both SO posts and API documentation to recommend APIs for a programming task. To bridge the knowledge gap, BIKER retrieves the top-k questions from SO that are semantically similar with the query. Since these questions and the query share similar purposes, the APIs mentioned in the questions are also likely to resolve the programming task in the query. In this way, we can greatly narrow down the search space of candidate APIs. To rank the relevance of a candidate API to the query, we consider the query's similarity with both the SO posts in which the candidate API is mentioned and the candidate API's official description. In this way, we can balance the API information from both the API designer and user perspectives. To bridge the lexical gap between semantically similar texts that are expressed by different words, we follow Ye et al. [49] to use word embedding techniques to calculate the similarity score. In addition to recommending APIs, BIKER also summarizes supplementary information like official API description and code snippets in SO posts to help developers better understand why these APIs are recommended so that they can select the right API(s) more easily.

To evaluate BIKER, we manually selected 413 questions from SO that are seeking APIs to resolve programming tasks and labelled the ground-truth APIs for these questions based on their accepted answers. For class-level recommendation, we enrich our dataset with the dataset published by RACK [34] which contains 150 questions and corresponding class-level APIs. Note that RACK only supports class-level recommendation. For class-level recommendation, BIKER achieves a mean reciprocal rank (MRR) and mean average precision (MAP) of 0.692 and 0.659 respectively, and this outperforms Ye et al.'s approach and the two state-of-the-art API recommendation approaches RACK [34] and DeepAPI [21] by at least 42% in MRR and 57% in MAP. For method-level recommendation, BIKER achieves an MRR and MAP of 0.573 and 0.521, and this outperforms Ye et al.'s approach and DeepAPI [21] by 205% in MRR and 241% in MAP. Our evaluation also confirms the importance of SO information in API recommendation and the usefulness of incorporating SO information and API documentation. Finally, we conduct a user study in which 28 Java developers are divided into four groups using different tools to answer 10 API-method-related questions randomly sampled from the 413 questions. On average, compared with the other three groups (i.e., web search only, using DeepAPI and using BIKER with only API recommendation but no supplementary information), the group using the full version of BIKER can improve *answer correctness* by at least 11% and save *answering time* by at least 28%.

The main contributions of this paper are:

(1) We conduct a survey of developers' API search behavior and expectations, which suggests the necessity of incorporating SO posts and API documentation for effective API search.
(2) Inspired by our survey results, we propose BIKER to recommend API methods by exploiting SO posts to bridge task-API knowledge gap, and by incorporating the information from both SO posts and API documentation for measuring API relevance and assisting developers in selecting recommended APIs.
(3) Both our quantitative evaluation and user study show that BIKER can help developers find the correct APIs for Java programming tasks more efficiently and accurately, compared with state-of-the-art baselines.
(4) We release the source code of BIKER and the dataset of our evaluation and user study[1] to help other researchers replicate and extend our study.

[1]The replication package can be downloaded at: https://github.com/tkdsheep/BIKER-ASE2018

Paper Organization. The remainder of the paper is organized as follows. We present the survey to investigate how developers search for APIs and their expectations of an effective API recommendation tool in Section 2. We describe the technical details of BIKER in Section 3. We present our experimental setup and results in Section 4 and Section 5, respectively. We present the results of our user study in Section 6. We discuss threats to validity in Section 7. We present related work in Section 8. We conclude the paper and mention future work in Section 9.

2 DEVELOPERS' EXPECTATIONS ON API RECOMMENDATION

To gain insights into how developers search for APIs to resolve programming tasks and the developers' expectations on automatic API recommendation techniques, we conducted a survey with 130 Java developers from two IT companies (both are outsourcing companies with more than 2,000 employees) and received 47 replies. Our survey includes the following questions: 1) Do you often need to search for appropriate APIs for your programming tasks? 2) What tools and/or resources do you usually use to search APIs? And why do you prefer these tools and/or resources? 3) Do you feel searching APIs on the Internet is a time-consuming task? 4) Which granularity of API recommendation (class or method or no preference) do you prefer? 5) What feature(s) do you expect an API recommendation tool to support?

According to the responses, we have the following findings:

- 87% of the respondents agreed or strongly agreed that they often need to search for appropriate APIs to resolve different programming tasks during development.
- 94% of the respondents chose search engines (e.g. Google) to perform general search, because search engines can return information from various sources like SO, Java API documentation and technical blogs. 74% of the respondents chose to focus search on Q&A website (e.g., SO), because they can find similar questions whose answers often contain relevant APIs to use. 45% of the respondents chose to directly read Java API documentation, when they have some candidate API classes in mind and they want to further check the documentation to decide which API method to use.
- 76% of the respondents agreed or strongly agreed that it is time-consuming to find appropriate APIs by searching and browsing resources on the Internet.
- 63% of the respondents preferred that the tool should recommend APIs at method-level. 19% preferred class-level and 18% had no preference.
- 85% of the respondents expect the tool to directly recommend relevant APIs for a programming task described in natural language. 90% of these respondents suggested that the tool should provide additional information to explain why it recommends certain APIs and how to use them.

The survey responses suggest that apart from API documentation, SO is also an important resource for developers to search APIs. By interviewing with several respondents, we find that a typical API search process they adopt is to first browse several relevant SO questions and pick out the APIs that seem to be useful in the discussions. The interviewed developers suggest that SO discussions are usually centered on some programming tasks, which makes it easier for them to narrow down some candidate APIs that may support their tasks. They also suggest that if they still cannot decide which API is the right choice, they will further check the APIs' documentation or code examples.

This API search process inspires us to design BIKER that exploits SO posts to bridge task-API knowledge gap and incorporates the information from both SO questions and API documentation to measure the relevance of an API to the programming task description. As suggested by developers, BIKER also summarizes supplementary API information for each recommended API to help developers better understand what an API can do and select the right API(s) for their tasks more easily.

3 APPROACH

Fig. 1 shows the overall framework of BIKER, which consists of three main components: building domain-specific language models for similarity calculation (Section 3.1), searching for relevant APIs based on SO posts and API documentation (Section 3.2), and summarizing API supplementary information (Section 3.3). Since BIKER recommends APIs at method level by default, we also introduce how to adapt BIKER for class-level recommendation in Section 3.4.

3.1 Building Language Models for Similarity Calculation

To measure a query's similarity to a SO post or an API description, we need to build domain-specific language models. We first build a text corpus by extracting the text content from SO posts in HTML pages. We remove long code snippets enclosed in HTML tag $\langle pre \rangle$, but keep short code fragments in $\langle code \rangle$ in natural language sentences. We use NLTK package [10] to tokenize the sentence. Note that if one is interested in a particular language or library's APIs, he may use a subset of SO post tagged with that library (e.g., Java). Using the SO corpus, we train a word embedding model using word2vec [27]. Word embedding model provides the basic model to measure word similarity. Then we build the word IDF (inverse document frequency) vocabulary. A word's IDF represents the inverse of the number of SO posts that contain the word. We reduce each word in the corpus to its root form (aka. stemming) using the NLTK package [10]. Thus, the words with the same root form will have the same IDF value. The more posts in which a word appears, the less likely the word carries important semantic information, and thus its IDF is lower. We use IDF as a weight on top of word embedding similarity. Finally, the words in API documentation would directly use this word embedding model and IDF vocabulary, since the text volume of SO posts is much larger than API documentation.

3.2 Searching for Relevant APIs

Our API search component has three steps: retrieving similar SO questions to the query, detecting API entities in the SO posts, and calculating the query's similarity with SO posts and API descriptions for ranking the relevance of candidate APIs to the query.

3.2.1 Retrieving Similar Questions. Given a query describing a programming task, the first step is to retrieve the top-k similar questions from SO. BIKER first transforms the text of a question's title and the query into two bags of words, denoted as T and Q, respectively. Then an asymmetric similarity score from T to Q

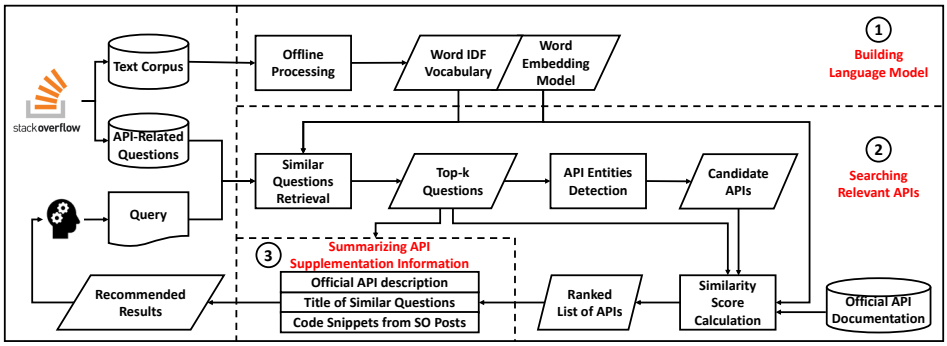

Figure 1: Overall framework of BIKER

is computed as a normalized, IDF-weighted sum of similarities between words in T and all words in Q:

$$sim(T \rightarrow Q) = \frac{\sum_{w \in T} sim(w, Q) * idf(w)}{\sum_{w \in T} idf(w)} \quad (1)$$

where $sim(w, Q)$ is the maximum value of $sim(w, w^{'})$ for each word $w^{'} \in Q$, and $sim(w, w^{'})$ is the cosine similarity of the word embedding vectors of w and $w^{'}$. The asymmetric similarity score $sim(Q \rightarrow T)$ is computed analogously, by swapping T and Q in Equation 1. Intuitively, a word with lower IDF value would contribute less to the similarity score. Finally, the similarity score between T and Q is computed as the *harmonic mean* of the two asymmetric scores:

$$sim(T, Q) = \frac{2 * sim(T \rightarrow Q) * sim(Q \rightarrow T)}{sim(T \rightarrow Q) + sim(Q \rightarrow T)} \quad (2)$$

The retrieved top-k similar questions will be used to detect candidate APIs for recommendation. In this paper, BIKER only retrieves the top-50 similar questions, since retrieving too many questions may introduce noise to the recommendation process.

3.2.2 Detecting API Entities. After retrieving the top-k similar questions, BIKER uses several heuristic rules to extract API entities from each question's answers. These APIs are considered as candidate APIs for recommendation. If an API is not mentioned in any of the top-k similar questions, it is less likely to be the right API for the query. Thus, we do not consider all APIs of a language or library for recommendation. In this way, a lot of irrelevant APIs would be filtered out.

To detect API entities, we first manually checked a large number of API-related questions. We observe that an important API mentioned by developers is often highlighted with the HTML tag ⟨*code*⟩ or referenced by a hyperlink to the API's corresponding documentation page. Thus, BIKER detects API entities using the following two heuristics:

- BIKER checks every hyperlink in each answer and uses regular expressions to identify the hyperlink to a library's official API documentation site, for example, https://docs.oracle.com for Java API documentation. Then it uses regular expressions to detect the full name of the corresponding API method from the hyperlink and mark this method as a candidate API. For example, given the hyperlink https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#forName(java.lang.String), it extracts the API method *java.lang.Class.forName*.
- BIKER first builds a dictionary that stores the names of all APIs of a language or library crawled from the language or library's official documentation site. Then it checks the plain text contained in every HTML tag ⟨*code*⟩ in each answer. If the text fully matches any API method in the dictionary, it is marked as a candidate API. Note that in most cases, developers would omit the package name of an API. For example, *java.lang.Class.forName* is usually written as *Class.forName*. Thus, our dictionary only stores the partially-qualified name of an API for string matching.

3.2.3 Calculating Similarity Score for Ranking Candidate APIs. After obtaining a list of candidate APIs from the top-k similar questions, BIKER calculates the similarity score between each candidate API and the query. Given an API and a query Q, their similarity score is a combination of two scores, namely *SimSO* and *SimDoc*. Specifically, *SimSO* measures the similarity between the query and the question title T of a top-k similar question in which the API is mentioned, and *SimDoc* measures the similarity between the query and the API's description in official API documentation.

Suppose that among all the top-k similar questions, the API is mentioned in n questions, then *SimSO* is computed as:

$$SimSO(API, Q) = min(1, \frac{\sum_{i=1}^{n} sim(T_i, Q)}{n} \times \log_2 n) \quad (3)$$

where $sim(T_i, Q)$ represents the similarity score between the query and the title of the i-th question that mentions the API, and $sim(T_i, Q)$ is calculated based on Equation 2. *SimSO* considers two aspects. First, the score should be related to the similarity between each question and the query. Thus, it calculates the average of the similarity score between each question's title and the query. Second, if the API is mentioned in multiple questions, it is more likely to be the right API for the query. Thus, the score is further boosted based on the number of questions. We add a logarithm transformation $\log_2 n$ to control the scale of boosting. For example, the score would be boosted by 20% if the API is detected in 4 questions. We also restrict that the boosted score should not exceed 1.

The *SimDoc* is also calculated based on Equation 2 given the query Q and the API description D. Finally, the similarity score between the query and the API is the harmonic mean of the corresponding *SimSO* and *SimDoc*.

3.3 Summarizing API Supplementary Information

After obtaining the ranked list of candidate APIs, BIKER summarizes supplementary information for each API in the list. We do

Table 1: An example of API summary

Query: *run linux commands in java code*
API1: *java.lang.Runtime.exec*
JavaDoc: Executes the specified string command in a separate process
Similar Questions
1. Run cmd commands through java
2. use cmd commands in java program
3. Unable to execute Unix command through Java code
Code Snippets
/**********code snippet 1 **********/
Process p = Runtime.getRuntime().exec(command);
/**********code snippet 2 **********/
Runtime.exec(-whatever cmd command you need to execute-)
/**********code snippet 3 **********/
String command1 = "mv $FileName /bgw/feeds/ibs/incoming/";
Runtime.getRuntime().exec(command1);

this because our survey responses and interviews with developers suggest that developers usually need to check more information about API description and API usage examples to decide which API should be chosen for their tasks. Thus, the supplementary information summarized by BIKER considers three aspects, as listed below:

- **Official API description:** It presents the API designer's official description of an API so that API users can quickly check the API's functionality.
- **Title of similar questions:** Based on the top-k similar questions, it extracts the title of all the questions whose answers mention the API. Then it ranks these questions by their titles' similarity scores with the query in descending order and present these questions titles (with hyperlinks to the corresponding webpage). To reduce information overloading, it only presents the top-3 questions. Thus, developers can compare question titles with their tasks.
- **Code Snippets:** Based on the top-k similar questions, it checks each question's answers and extracts the code snippets containing the API. Specifically, given an API (e.g., *Math.round*), a code snippet is extracted if it satisfies both the following conditions: 1) The number of lines of code is no more than five; 2) The API's class name (i.e., Math) and method name (i.e., round) are both contained in the code snippet. The extracted code snippets are ranked by their corresponding questions' similarity scores with the query in descending order. To reduce information overloading, it presents only the top-3 code snippets. Thus, developers can check these code snippets to understand how to use the API.

To better illustrate the outcome of this summarization step, Table 1 presents an example of the summary results for the top-1 recommended API "*java.lang.Runtime.exec*", given the query "*run linux commands in java code*".

3.4 Adapting BIKER for Class-Level Recommendation

By default, BIKER recommends APIs at method-level. However, it can be easily adapted to support class-level recommendation. First, we need to revise the heuristic rules for detecting API entities. Specifically, we change the regular expressions so that it only extracts the API's class name (with full path of its package) from the hyperlink to an API documentation page. We also change the dictionary to store all APIs' class names for string matching. Second, we need to change the way of calculating *SimDoc* in the step of similarity score calculation. Although an API class has its own description like an API method, we do not use it since we observe that the description of an API class is rather long in most cases and it usually does not contain much useful information for specific task requirements. Thus, BIKER calculates the similarity score between the query and the description of each method in the class, and chooses the maximum score as the result of *SimDoc* for this API class.

4 EXPERIMENTAL SETUP

In this section, we describe the experimental setup that we follow to evaluate BIKER. The experimental environment is a laptop equipped with Intel(R) Core(TM) i7-6700HQ CPU and 16GB RAM, running Ubuntu 16.04 LTS (64-bit).

4.1 Data Collection and Tool Implementation

4.1.1 SO Text Corpus. We downloaded the official data dump [2] of SO (published in: Dec 9th, 2017). As our current tool focuses on Java API, we extracted 1,347,908 questions that are tagged with "java". Based on these questions and their answers, we built a text corpus using the plain text in each post to train the word embedding model and build the IDF vocabulary. We used *Gensim* [35] (a python package which implements word2vec [27]) to train the word embedding model.

4.1.2 SO Question Base. To create the knowledge base of API-related questions for similar questions retrieval, we selected only the questions satisfying the following criteria: 1) the question should have positive score; and 2) at least 1 answer to the question contains API entities and the answer's score should be positive. Note that the API entities mentioned in a post were automatically detected by the heuristics described in Section 3.2.2. In this way, we collected 125,847 questions as the knowledge base of API-related SO questions.

4.1.3 Experimental Queries and Ground-Truth APIs. To create experimental queries for the evaluation of BIKER, we followed Ye et al. [49] to select a small number of API-related questions satisfying the following criteria: 1) the score of the question itself should be at least 5. Ye et al. set this threshold to 20 but this leaves only 604 candidate questions which is too few; 2) the question's accepted answer should contain API entities and the answer's score should be positive.

In this way, we collected 3,395 questions in total. Among these questions, we randomly selected 1,000 questions. We manually checked each selected question's title to remove the questions that do not aim to search APIs for programming tasks. We examine only the question titles because we assume that developers would use BIKER like a search engine, and thus BIKER is not likely to receive a query with too many words. The first author and another PhD student independently labelled the questions to be removed. Typical examples of questions being removed are shown below:

- The question seeks for comparison of multiple APIs (e.g., *Difference between HashSet and HashMap?*).
- The question seeks for the theories or algorithms behind an API (e.g., *why HashMap Values are not cast in List?*)
- The question's title contains the word like 'this', 'that' or 'it', which makes its purpose unclear (e.g., *how to parse this string in java?*).
- The question describes an error or a bug (e.g., *IP Address not obtained in java*).

We use Fleiss Kappa [18] to measure the agreement between the two labelers. The Kappa value is 0.85, which indicates almost perfect agreement. After completing the manual labeling process, the two labelers and another post-doc discussed together their disagreements to reach a common decision. In this way, we collected 469 questions for further inspection.

By default, for each question, all the API entities in the accepted answer are considered as relevant APIs to resolve the question. However, some of the API entities may not be truly helpful and some truly helpful APIs may not be detected by our heuristic rules. Thus, the first author and the same PhD student manually checked each question's title, body and its accepted answer to fix this issue. The overall kappa value is 0.78, which indicates a substantial agreement, and the two labelers also discussed their disagreements with the same postdoc to reach a common decision.

Specifically, a small number of questions were removed since they cannot be easily resolved by Java APIs. For example, in the question "*How can I set the System Time in Java?*", the accepted answer clearly stated that Java does not have an API to do this. For most questions, we mainly relied on each question's accepted answer to decide the ground truth APIs. However, since both the two labelers have at least 3 years of Java development experience, if the question is asking a common programming task, we also checked the other answers to add other APIs that are also helpful but not mentioned in the accepted answer. For example, for the question "*How to round a number to n decimal places in Java*", the accepted answer only mentioned *DecimalFormat.setRoundingMode*, but the other two APIs (i.e., *Math.round* and *BigDecimal.setScale*) mentioned in other answers are also helpful.

After this manual labeling process, we got 413 questions along with their ground truth APIs as the testing dataset for the evaluation of BIKER. We use the title of these 413 questions as the query for API search. Note that these 413 questions and their duplicate questions were excluded from the SO question base.

4.1.4 Java API Dictionary and API Description. We downloaded the Java SE 8 API documentation [1] and parsed the html file of each API class to extract all API methods, along with their descriptions. For simplicity, Java interfaces were also treated as Java classes. In total, we extracted 4,216 classes and 31,736 methods and built a Java API dictionary with the name of these API classes and methods.

4.2 Baseline Approaches

We compare the performance of BIKER with two baseline methods, as listed below:

Baseline 1 (RACK): Rahman et al. [34] proposed RACK, which constructs a keyword-API mapping database where the keywords are extracted from SO questions and the mapped APIs are collected from corresponding accepted answers. Based on this database, RACK recommends a ranked list of API classes for a given natural language query. Note that we only compare BIKER with RACK at class-level, since RACK does not support recommendation at method-level. Although RACK also leverages SO to bridge the knowledge gap, it does not consider API documentation and its technique is different from BIKER.

Baseline 2 (DeepAPI): Gu et al. [21] proposed DeepAPI, which adapts a Recurrent Neural Network (RNN) Encoder-Decoder model. DeepAPI encodes a word sequence (user query) into a fixed-length context vector, and generates an API-method sequence based on the context vector. For example, given the query "*open a url*", its first recommended result is "*URL.new→URL.openConnection*". DeepAPI's technique is different from BIKER and their knowledge base is a large corpus of annotated API sequences extracted from code repositories.

Note that we do not choose Ye et al.'s approach [49] as our baseline, since it can be considered as part of BIKER. If BIKER uses only Java API documentation, then BIKER is reduced to be the same as Ye et al.'s approach. We also have a research question (RQ2 in Section 5.2) to discuss the effectiveness of BIKER when using Java API documentation only.

4.3 Evaluation Metrics

We evaluate BIKER and other baselines using MRR and MAP, which are classical evaluation metrics for information retrieval [25]. MRR measures how far we need to check in the recommended list to find the first correct answer, while MAP considers the ranks of all correct answers. MRR and MAP are also widely used in previous software engineering studies [24, 34, 37, 40, 45–48, 50]. In addition, we run the Wilcoxon signed-rank test [41] with Bonferroni correction [6] to check if the differences between the performance of BIKER and the baselines are statistically significant. We consider that one approach performs significantly better than the other one at the confidence level of 95% if the corresponding Wilcoxon signed-rank test result (i.e., p-value) is less than 0.05. We also use the Cliff's delta (δ) [15] to quantify the amount of difference between two approaches. The amount of difference is considered negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), moderate ($0.33 \leq |\delta| < 0.474$), or large ($|\delta| \geq 0.474$), respectively.

5 EXPERIMENT RESULTS

5.1 RQ1: How effective is BIKER? How much improvement can it achieve over the baseline methods?

Motivation. BIKER aims to automatically recommend appropriate APIs for programming tasks described in natural language queries. Thus, for the approach to be useful, we need to see how accurate it is in API recommendation and how it compares with existing API recommendation methods.

Approach. To answer this research question, we compare BIKER with the two baselines (i.e., RACK and DeepAPI) using our testing dataset including 413 queries and ground-truth APIs. Since RACK's authors have published an executable tool [4] for replication, we directly use this tool to compare with BIKER. For DeepAPI, the authors have deployed an online demo tool [3], which receives a user query and presents the recommendation results on the webpage. Thus, to compare with DeepAPI, we wrote a web-crawler to automatically send all queries in the testing dataset one by one and retrieve the recommendation results through HTTP requests. We also carefully checked the JavaScript code behind the webpage to make sure that we did the same text preprocessing for each query. Since DeepAPI recommends API sequence, we consider an API sequence is correct if any one of the APIs in the sequence is the ground truth API. This makes the fair comparison with DeepAPI. Finally, RACK's authors also published their testing dataset, which

Table 2: Performance of BIKER and the baseline methods for class-level recommendation

Appraoch	Class-Level Recommendation			
	Our Dataset		RACK's Dataset	
	MRR	MAP	MRR	MAP
BIKER	0.692	0.659	0.428	0.271
RACK	0.296	0.266	0.302	0.171
DeepAPI	0.462	0.420	0.276	0.149
Improve. RACK	134%	148%	42%	58%
	p<0.001	p<0.001	p<0.001	p<0.001
	$\lvert\delta\rvert$=0.57	$\lvert\delta\rvert$=0.59	$\lvert\delta\rvert$=0.12	$\lvert\delta\rvert$=0.17
Improve. DeepAPI	50%	57%	55%	82%
	p<0.001	p<0.001	p<0.001	p<0.001
	$\lvert\delta\rvert$=0.33	$\lvert\delta\rvert$=0.35	$\lvert\delta\rvert$=0.28	$\lvert\delta\rvert$=0.30

Table 3: Performance of BIKER and DeepAPI for method-level recommendation

Appraoch	Method-Level Recommendation (Our Dataset)	
	MRR	MAP
BIKER	0.573	0.521
DeepAPI	0.188	0.153
Improve.	205% (p<0.001, $\lvert\delta\rvert$=0.57)	241% (p< 0.001, $\lvert\delta\rvert$=0.59)

contains 150 code search queries randomly chosen from several Java tutorial sites. Thus, we also evaluate all approaches using this dataset, which only supports class-level evaluation.

Results. Table 2 presents the performance of BIKER and the two baselines for class-level recommendation. The results show that BIKER significantly outperforms RACK and DeepAPI in terms of MRR and MAP for both datasets, with an improvement of at least 42% in MRR and at least 57% in MAP. We also note that the MRR and MAP achieved by BIKER for RACK's dataset are relatively lower than those achieved for our dataset. By manually checking RACK's dataset, we find that about 19% of its questions include ground-truth APIs from third-party packages (e.g., MongoDB, Apache Commons, etc.) or Java EE, which is beyond the knowledge based of our current tool (i.e., we only consider APIs from Java SE). Except for the MRR and MAP comparison with RACK on our dataset and for the MAP comparison with DeepAPI on our dataset, the amount of difference between the compared methods for other comparisons is either small or negligible.

Table 3 presents the performance of BIKER and DeepAPI for method-level recommendation using our dataset. RACK and RACK's dataset are not used since RACK only supports class-level recommendation. The MRR and MAP achieved by BIKER is 0.573 and 0.521, respectively, which significantly outperforms DeepAPI by 205% in MRR and 241% in MAP. The amount of difference between the two approaches are large for both MRR and MAP.

To sum up, BIKER significantly outperforms the two state-of-the-art baseline methods for both class- and method-level API recommendation. The advantage of BIKER is more evident for method-level API recommendation.

5.2 RQ2: How effective is BIKER when using the two different information sources individually?

Motivation. BIKER leverages both SO posts and Java API documentation to calculate the similarity score between an API and the query. However, BIKER can still work if we only use one of the two information sources individually. Thus, we would like to investigate whether the combination of the two information sources results in better or poorer performance.

Table 4: Performance of BIKER for our dataset when using one or both information sources

Info Source	Class-Level		Method-Level	
	MRR	MAP	MRR	MAP
Stack Overflow	0.559	0.529	0.524	0.476
Java Documentation	0.287	0.265	0.097	0.079
Both	**0.692**	**0.659**	**0.573**	**0.521**
Improve. SO	24%	25%	9%	9%
Improve. JavaDoc	141%	149%	491%	559%

Approach. To answer this research question, we evaluate the performance of BIKER when using either SO posts or Java API documentation for calculating the query-API similarity score, and compare that performance with the performance of BIKER using both information sources. When using only SO, the candidate APIs are extracted from top-k similar questions, and the similarity score of each candidate API with the query is calculated based on only SO questions (i.e., *SimSO*). When using only Java API documentation, the list of candidate APIs is the list of all API methods (or classes) in Java API documentation, and the similarity score of each candidate API with the query is calculated based on Java API documentation (i.e., *SimDoc*). Note that the only-Java-API-documentation setting is essentially Ye et al.'s approach [49].

Results. Table 4 presents the performance of BIKER when using each information source individually. In general, when combining both information sources together, BIKER performs better than using each information source individually. Comparing the improvement ratio over SO or Java documentation, we can see the importance of SO information in BIKER. Using only SO information, the performance is only 24% worse in MRR and 25% worse in MAP than using both information sources for class-level recommendation, and only 9% worse in both MRR and MAP for method-level recommendation. However, using only Java documentation, the performance becomes significantly worse than using two information sources. But using Java documentation as an additional information source can further improve the recommendation performance than using only SO information.

5.3 RQ3: How efficient is BIKER for practical use?

Motivation. During the model building process, BIKER needs to train word embedding model and build IDF vocabulary using the corpus extracted from more than one million SO questions. This would require substantial computational time, especially for the word embedding model. Another time-consuming process is to transform the title of all the 125,847 questions in question knowledge base and the description of all the 31,736 API methods into *matrix representation* based on each word's embedding vector and IDF value, so that we can compute the similarity score between the query and the documents efficiently. During the recommending process, given a query, BIKER needs to calculate the similarity between this query and each question in the question base, which could also be time-consuming. If BIKER cannot run with a reasonable runtime performance, developers may not be willing to use it in practice.

Approach. To answer this research question, we record model training time and query processing time of BIKER and the two baselines using our testing dataset for class-level API recommendation. The time cost for BIKER and DeepAPI do not change under method-level recommendation.

Table 5: Time cost for model training and query processing of BIKER and the baseline methods

Approach	Model Training Time	Query Processing Time
BIKER	36 minutes	2.8s / query
DeepAPI	240 hours	2.6s / query
RACK	unknown	12.8s / query

Results. Table 5 presents the model training time and the average query processing time of BIKER and the two baseline methods. As reported by DeepAPI's authors [21], their approach takes 240 hours of model training, since their approach is based on RNN (i.e., a deep neural network), which is computationally expensive during training [20]. The training time cost of RACK is unknown since it is not reported by the authors and it is not easy to replicate the training process without RACK's source code. BIKER takes 36 minutes to train, which is also relatively slow, and almost the whole time cost is due to training word embedding model. The word embedding model only needs to be trained once and it does not need to be updated frequently since the text corpus is already very large (i.e., extracted from 1.3 million questions). If we use pre-trained word embedding model, we just need about 10 seconds to transform text into matrix representation.

For the average query processing time, RACK is slowest (12.8 seconds) to process each query, while DeepAPI is the fastest (2.6 seconds). BIKER (2.8 seconds) is slightly slower than DeepAPI. The major computation cost of BIKER for query processing is due to the step of similar questions retrieval, where we need to compare the query with the titles of about 120 thousand questions. To improve the time efficiency, we can reduce the size of questions to be compared with some heuristic rules (e.g., only comparing with the question whose score is larger than k) or accelerate similarity score computation by GPU [17].

6 USER STUDY

In this section, we conduct a user study to investigate how developers interact with BIKER and whether it can help developers find correct APIs more efficiently and accurately.

6.1 Study Design

6.1.1 Experimental Queries and Ground-Truth APIs. To conduct our user study, we randomly selected 10 questions from our testing dataset, as shown in Table 6. The last column shows the ground-truth answers, which refer to the APIs extracted from each question's accepted answer. Three questions (i.e., Q1, Q3 and Q10) require multiple APIs (i.e., an API sequence) to complete the programming task.

6.1.2 Participants. We recruited 28 participants from both university and IT companies. 16 of them (2 postdocs, 9 PhDs and 5 graduate students) are from the first author's university, and 12 of them are from two IT companies. All of them have Java developing experience in either commercial or open source projects, and the years of their developing experience vary from 1 year to 5 years, with an average of 2.9 years.

6.1.3 Experimental Groups. Next, we divided the participants uniformly based on years of development experience into four groups, with the following settings: 1) **WSO:** Find appropriate API methods by searching and browsing resources on the Internet (i.e., Web Search Only); 2) **DeepAPI:** Use DeepAPI's online tool; 3) **BIKER-Simple:** Use a simplified version of BIKER, which only recommends the name of APIs; 4) **BIKER-Full:** Use the fully-featured version of BIKER.

RACK is not evaluated since it does not support method-level recommendation and it runs much slower than DeepAPI and BIKER. The DeepAPI, BIKER-Simple and BIKER-Full groups are also allowed to search any resources on the Internet if the participants deem the information provided by the tool is not enough to answer the questions. Since the 10 questions were extracted from SO, to be fair across different techniques, we instructed the participants to ignore the 10 questions on SO when searching the Web.

6.1.4 Procedure. We deployed a simple website with 10 pages, each corresponding to one question. When a participant clicked the webpage of a question, a timer in the background would collect how much time he/she spent until submitting the answer. Participants were encouraged to complete each question without interruption and they would explicitly inform us if there was interruption.

6.2 Results Analysis

We analyze two metrics with the user study results, as shown below:

- **Correctness:** This metric evaluates whether a participant can find the correct APIs. For the question that only needs one API method, correctness is 1 if the participant submitted the correct API, otherwise 0. For the question that needs an API sequence, correctness is the proportion of the correct APIs submitted by the participant among all APIs in the correct API sequence. Some questions can also be resolved using other APIs different from the ground-truth APIs. For example, *BigDecimal.setScale* or *Math.round* are also the correct answers for Q10. Thus, we manually check each participants' answers to make sure the correctness is also 1 if they submitted the correct but not ground-truth APIs.
- **Completion time:** This metric evaluates how fast a participant can answer the question. One problem is that in some cases, the recorded completion time may not reflect the true effort needed to answer the question. For example, for the DeepAPI group, while 6 participants needed at least 30 seconds to answer Q9, there is 1 participant who only spent 12 seconds. By consulting with this participant, we found that he is a senior Java developer and he can directly answer this question without any tool's support. On the other hand, we recorded more than 20 minutes completion time for a single question for a few participants. They explained that they were interrupted by urgent tasks or bad network condition. To avoid the effect of outliers, for each question, we report the median value of the time spent for each group.

Table 7 presents the results of user study. In general, participants in *BIKER-Full* group performed as well as or better than the other three groups for every question in terms of correctness, and they were the fastest to solve six out of the ten questions. On average, the full version of BIKER can improve correctness by at least 11% and save the time cost by at least 28%. We also note that the correctness of different groups vary a lot for several questions (e.g., Q3 and Q7). By manually checking the participants' answers, we have the following two findings:

First, although BIKER does not recommend API sequences, participants can find the necessary sequence by themselves with the help of code snippets provided by BIKER. For example, in Q3, all participants in *BIKER-Simple* group only chose the first recommended API

Table 6: Ten questions and their standard answers for user study

PID	StackOverflow ID	Query	Answers
Q1	15788453	Resolving ip-address of a hostname?	InetAddress.getByName→InetAddress.getHostAddress
Q2	29259201	How to make a list thread-safe for serialization?	Collections.synchronizedList
Q3	11284938	Remove trailing zeros from double?	BigDecimal.stripTrailingZeroes→BigDecimal.toPlainString
Q4	33773708	How to check whether a class exists?	Class.forName
Q5	10383688	Is there any way to find os name using java?	System.getProperty
Q6	19486077	Java Fastest way to read through text file with 2 million lines?	BufferedReader.readLine
Q7	4584541	Check if a class is subclass of another class in Java?	Class.isAssignableFrom
Q8	5505927	How to generate a random permutation in Java?	Collections.shuffle
Q9	10078867	How to initialize all the elements of an array to any specific value in java?	Arrays.fill
Q10	153724	How to round a number to n decimal places in Java?	DecimalFormat.setRoundingMode→DecimalFormat.format

Table 7: Results of user study

Metrics	Group	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Average
Correctness	WSO	**0.79**	0.79	0.86	**1.0**	0.71	0.71	0.57	0.79	0.71	**1.0**	0.79
	DeepAPI	**0.79**	0.86	0.64	0.86	0.86	**1.0**	**1.0**	0.86	0.86	**1.0**	0.87
	BIKER-Simple	0.64	0.86	0.50	**1.0**	0.71	**1.0**	**1.0**	0.86	**1.0**	**1.0**	0.86
	BIKER-Full	**0.79**	**1.0**	**0.93**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**0.97**
Completion Time	WSO	132s	74s	91s	76s	57s	97s	146s	**33s**	53s	82s	84s
	DeepAPI	104s	93s	72s	87s	49s	**44s**	41s	73s	68s	21s	65s
	BIKER-Simple	113s	52s	**43s**	72s	53s	86s	61s	59s	45s	**19s**	60s
	BIKER-Full	**81s**	**28s**	65s	**42s**	**44s**	51s	**32s**	35s	**29s**	26s	**43s**

"*BigDecimal.stripTrailingZeroes*" as their answers, possibly because the API's name seems to be the right choice and the key phrase in its documentation (i.e., *with any trailing zeros removed*) also seems to meet the task requirement. However, as shown in SO post, this API would transform a number like 600.0 into scientific notation. To fix this issue, developers need to call *BigDecimal.toPlainString* after stripping the trailing zeros. In *BIKER-Full* group, six out of the seven participants chose both the two APIs as their answers, since the code snippets with *stripTrailingZeroes* has clearly showed that *toPlainString* should be called before printing. Such phenomenon also appeared in the answers for Q1.

Second, in some cases, participants can find the correct APIs more easily or with more confidence if they have tool support. For example, both DeepAPI and BIKER recommended *Class.isAssignableFrom* as the top-1 or top-2 answer for Q7, which may help participants narrow down the search space. On the other hand, three out of the seven participants in *WSO* group submitted *Class.isInstance* or *instanceof* (not an API but Java operator), which are both incorrect. Actually, many developers are confused about the difference [5] between *Class.isInstance* and *Class.isAssignableFrom*. Thus, it is not surprising that these participants submitted *Class.isInstance*, which is also "relevant" to the question, but cannot directly solve the task.

To sum up, BIKER can help developers find appropriate APIs more efficiently and accurately. This can be attributed to its capability of effectively narrowing down candidate APIs and providing supplementary information for understanding and selecting recommended APIs.

6.3 Participants' Comments

We encouraged the participants in *BIKER-Full* group to write their comments and suggestions for BIKER after the experiment. For the participants in the other three groups, we also showed them the results recommended by the full version of BIKER after they finished their tasks, and invited them to provide comments and feedbacks too. Among all the 28 participants, 13 participants provided some comments and suggestions. Based on these comments, we summarized several major aspects of BIKER that are liked or disliked by participants, as shown below:

- **Positive Opinions**
  - *"Given the Javadoc and code snippets, **I can easily decide whether this API is useful**. I don't need to Google for more information in most cases, this **saves me a lot of time**."*
  - *"Since the tool recommended 5 APIs, there must be some APIs not helpful to solve the question. However, I especially appreciate the fact that some of these unrelated APIs also **inspired me a lot**. For example, in Q2, it also recommended the API for unmodifiable list and map, which would be useful if the **scenario or requirement is broadened**."*
  - *"The **code snippets** is **very useful**. It gives me **more confidence** to make the final choice and shows me **how to use the API**."*
- **Negative Opinions**
  - *"Although I can easily judge which API is correct with the information (like Javadoc) provided, **sometimes I still don't know how to use it**. Yes, your tool can provide code snippets for most APIs, but some APIs are not and sometimes they are just the exact APIs I want to further check! Is this a bug? For example, in Q6, you recommended BufferedReader.readLine as the first result, but **no code snippet provided** and the **javadoc is also too simple**..."*
  - *"**The layout is not ideal**. Sometimes it just looks like a mess, especially when every recommended API has multiple code snippets with many lines."*
  - *"Sometimes the API name is already enough for me to judge. Why don't you **make additional information folded up** and let me to decide read it or not by myself?"*

From these comments, we can see that participants can benefit from the supplementary information provided for each API. However, sometimes BIKER may fail to extract code snippets for some APIs, because we only scanned the top-k similar questions. We could improve this component by building a mapping database which stores the API and its code snippets extracted from more questions. Finally, as pointed out by the participants, we need to carefully design the layout or the way we present the supplementary information to make the useful information more usable, which is an important aspect of user experience to be improved.

7 THREATS TO VALIDITY

Threats to internal validity relates to the errors in the implementation of BIKER and the baseline methods. We have double checked our code to make sure that the questions in testing dataset are not included in the question base. For the baseline methods, we directly used their published tools. Thus, there is little threat to the approach implementation. The degree of participants' carefulness and effort spent in our user study may also affects the validity of our user study results. To reduce this threat, we recruited participants who express interests in our research and made the average years of development experience in each group as uniform as possible.

Threats to external validity relates to the quality of our dataset and generalizability of our results. To ensure the quality of our dataset, we had two labelers to label the data and we relied on the accepted answer to label the ground-truth APIs. Although our

dataset contains only 413 questions, most of these questions have a large number of view count. Among these 413 questions, about 70% of the questions in our dataset have their view count ranked within top-5% and 45% of the questions are ranked within top-1% among the 1.3 million java-tagged questions on SO. This indicates that if BIKER can solve these questions, it can benefit a large number of developers. We also used the dataset published by RACK to demonstrate the effectiveness of BIKER. Another threat is that BIKER only supports Java API recommendation. But this is an implementation limitation, rather than a methodological threat. It would not be difficult to adapt BIKER to support API recommendation for other programming languages, as long as we can obtain related SO questions and API documentation.

Threats to construct validity relates to the suitability of our evaluation measures. We use MRR and MAP, which are classical evaluation measures for information retrieval [25] and are also widely used in previous studies in software engineering [24, 34, 37, 40, 50].

8 RELATED WORK

API Recommendation: In addition to RACK and DeepAPI, there are other approaches for API recommendation. McMillan et al. [26] proposed *Portfolio* to find relevant functions for a code search query from a large archive of C/C++ source code. Chan et al. [13] further improved Portfolio by employing graph search approach. Raghothaman et al. [33] proposed *SWIM*, a tool that learns common API usage patterns from open-source code repositories and synthesize idiomatic code describing the use of these APIs. In general, these methods do not leverage information from Q&A websites like SO or do not incorporate information from SO and API documentation. We do not choose them as baselines since they have been reported as less optimal than RACK or DeepAPI. On the other hand, a number of previous studies (e.g., [7, 12, 14, 22, 29, 51] have proposed different approaches to recommend code snippets for a programming task described in natural language. We did not compare BIKER with these approaches since we focus more on the recommendation of a specific API, which is different from the granularity of code snippet recommendation.

Empirical Studies on Developers' Behaviors: In this paper, we conducted a survey to investigate developers' API search behaviors and expectations. A number of previous studies also focused on developers' behaviors and some of their findings are relevant to ours [8, 11, 16, 36, 43, 44]. For example, in a study involving twenty developers, Duala-Ekoko and Robillard [16] identified different types of questions that are commonly asked by developers when working with unfamiliar APIs and they analyzed the cause of the difficulties when answering questions about the use of APIs. Sadowski et al. [36] investigated how developers search for code through a case study at Google. They found that developers search for code very frequently and generally seek answers to questions about how to use an API. Brandt et al. [11] observed that developers mostly leverage online resources for just-in-time learning of new skills, and to clarify or remind themselves of existing knowledge. Our survey serves as a complement to these studies, since we focus on developers' API search behaviors and we reveal the information seeking process when developers perform API search.

Mining API Usages: Many studies focused on mining API usages to help developers learn how to use an API. Moreno et al. [28] proposed *MUSE* for mining and ranking actual code examples that show how to use a specific method. *MUSE* combines static slicing with clone detection, and uses heuristics to select and rank the code examples in terms of reusability, understandability, and popularity. Petrosyan et al. [31] proposed an approach to discover tutorial sections that explain a given API type. Treude et al. [39] proposed an approach to automatically augment API documentation with usage insights extracted from SO. Jiang et al. [23] proposed *FRAPT*, an unsupervised approach for discovering relevant tutorial fragments for APIs. Nguyen et al. [30] proposed *API2VEC* which uses word embedding to infer the semantic relations between APIs. Our work is a complement to these studies, since they assume that developers already know the name of an API for further investigation. These approaches could be integrated in BIKER to improve the quality of the supplementary information for the recommended APIs.

Mining Developer Forums: Researchers leveraged the rich resources in developer forums to build tools for software engineering. Barua et al. [9] used topic model to discover main topics discussed in SO, as well as their relationships and trends over time. Treude et al.'s study [38] on how programmers ask and answer questions on the web found that Q&A websites are particularly effective at code reviews and conceptual questions. Gao et al. [19] proposed an approach to automatically fix recurring crash bugs by retrieving a list of Q&A pages to generate edit scripts. Wong et al. [42] proposed an approach to automatically generate code comments by mining comments extracted from Q&A sites. Ponzanelli et al. [32] proposed *Prompter* to automatically generate queries based on code context, and retrieve pertinent discussions from SO. Our work also leverages developer discussions in SO, but we focus on recommending APIs for programming tasks.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose BIKER to automatically recommend relevant APIs for a programming task described in natural language. Inspired by the information seeking process of developers, we leverage both Stack Overflow posts and API documentation to improve the effectiveness of BIKER, and summarize supplementary information for each recommended API to help developers better understand the API usage and determine their relevance to the query task. The evaluation with both our dataset and RACK's dataset confirms the effectiveness of BIKER. Our user study demonstrates that BIKER can help developers find the appropriate APIs more efficiently and accurately in practice. In the future, we will develop an automatic tool (e.g., a plugin in a web browser or IDE) to enable developers to use BIKER to search APIs for programming tasks. We will further improve the performance of BIKER and the interaction design of our tool as suggested by the participants in user study. Finally, we will extend BIKER to support more programming languages.

ACKNOWLEDGMENTS

We would like to thank Rahman et al. and Gu et al. for sharing their tools and dataset. We also appreciate the reviewers for their insightful comments to help us improve this paper. Xin Xia and Xinyu Wang are the corresponding authors. This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904) and NSFC Program (No. 61602403).

REFERENCES

[1] 2017. Java SE 8 API documentation downloading site. http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html.

[2] 2017. Stack Overflow Data Dump. https://archive.org/download/stackexchange.

[3] 2018. DeepAPI's online demo. http://www.cse.ust.hk/~xguaa/deepapi/tooldemo.html.

[4] 2018. RACK's dataset and tool demo. http://homepage.usask.ca/~masud.rahman/rack/.

[5] 2018. Stack Overflow question: Class.isInstance vs Class.isAssignableFrom. https://stackoverflow.com/questions/3949260/java-class-isinstance-vs-class-isassignablefrom.

[6] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[7] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[8] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Inference of development activities from interaction with uninstrumented applications. *Empirical Software Engineering* 23, 3 (2018), 1313–1351.

[9] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering* 19, 3 (2014), 619–654.

[10] Steven Bird and Edward Loper. 2004. NLTK: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 31.

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

[12] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 628–632.

[13] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 10.

[14] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 385–400.

[15] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.

[16] Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 266–276.

[17] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 133–137.

[18] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[19] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 307–318.

[20] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

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

[22] Tihomir Gvero and Viktor Kuncak. 2015. Interactive synthesis using free-form queries. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 689–692.

[23] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 38–48.

[24] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 476–481.

[25] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

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

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[28] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 880–890.

[29] Anh Tuan Nguyen, Peter C Rigby, Thanh Van Nguyen, Mark Karanfil, and Tien N Nguyen. 2017. Statistical translation of English texts to API code templates. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 331–333.

[30] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 438–449.

[31] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. 2015. Discovering information explaining API types using text classification. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 869–879.

[32] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 102–111.

[33] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean-Code Search and Idiomatic Snippet Synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 357–367.

[34] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 349–359.

[35] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[36] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[37] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 345–355.

[38] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?: Nier track. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 804–807.

[39] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 392–403.

[40] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 262–273.

[41] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[42] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 562–567.

[43] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[44] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* (2017).

[45] Xin Xia and David Lo. 2017. An effective change recommendation approach for supplementary bug fixes. *Automated Software Engineering* 24, 2 (2017), 455–498.

[46] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: automated generation of answer summary to developersź technical questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 706–716.

[47] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, Qingye Wang, and Shanping Li. 2016. Domain-specific cross-language relevant question retrieval. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 413–424.

[48] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 127–137.

[49] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*. ACM, 404–415.

[50] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543.

[51] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 956–961.