Recommending APIs for API Related Questions in Stack Overflow

Jingxuan Zhang, *Student Member, IEEE*, He Jiang, Zhilei Ren, *Member, IEEE*, and Xin Chen

***Abstract*—API related questions are increasingly posted and discussed by developers in popular Question and Answer (Q&A) forums like Stack Overflow. However, their extremely long resolution time seriously delays the working schedules of developers. Despite researchers have investigated how to automatically resolve API related questions by recommending correct APIs for them, there is still much room for additional improvement. In this paper, we propose a novel approach named RASH towards recommending correct APIs for API related questions in Stack Overflow by leveraging both API specifications and historical resolved questions. Given a new API related question, RASH recommends APIs for it guided by two central observations. First, the more lexically similar the functional description in an API's specification is to the new question, the more likely that the API can resolve the new question. Second, the APIs that have resolved more historical similar questions can also help to resolve the new question. To verify the effectiveness of RASH, we construct and publish a corpus containing 1,234 API related questions with their correct APIs from Stack Overflow, and conduct extensive experiments over it. The experimental results show that RASH is relatively stable and robust to different quality of questions. In addition, RASH hits nearly 70% correct APIs and outperforms the state-of-the-art approach by 15.64% when recommending 15 APIs for each question.**

***Index Terms*—Application Programming Interfaces, Information Retrieval, Recommendation System, Stack Overflow**

I. Introduction

Software developers tend to reuse Application Programming Interfaces (APIs) in existing frameworks and libraries to facilitate their development process [1, 2, 3]. When they have no idea about what exact APIs to use or how to use specific APIs properly, they usually submit questions illustrating the API usage problems to seek professional help in Stack Overflow, a popular technical Question and Answer (Q&A) forum attracting over 50 million visitors each month [4, 5, 6, 7]. However, resolving these API related questions may take a very long time, since it is difficult to propose an *accepted answer*, which needs to be discussed continuously by developers [8]. For example, the average resolution time of API related questions in the constructed corpus is nearly 17 days, which is 3 days longer than that of other questions (see Section II.B). Such a long resolution time may heavily decrease the working efficiency and seriously delay the working schedules of developers [9, 10]. Furthermore, API related questions usually receive wide attentions from developers who may encounter the same or similar API usage problems. For instance, an API related question is viewed more than 4,600 times on average, which is twice as many as that of other questions (see Section II.B). Hence, automatically resolving API related questions could bring tremendous benefits for developers.

Recently, a new task named Question-to-API recommendation (Q2API) is issued [11]. When a new API related question is submitted to Stack Overflow, this task aims to automatically resolve it by recommending *correct* APIs, whose API specifications have non-trivial semantic overlap with the *accepted answer*. Therefore, by checking the recommended APIs and reading through their API specifications, the *submitter* can efficiently program with the correct APIs or easily think out the solution on his/her own, even before the *accepted answer* is posted [11]. In such a way, addressing this task could accelerate the resolution of API related questions and boost *submitters'* productivity.

In the literature, Ye *et al.* propose a seminal approach towards addressing the Q2API task based on the word embedding technique [11]. Given a new API related question, this approach aims to rank all the APIs in the same programming language (e.g., Java) as the new question and recommend the top ranked APIs for it. More specifically, this approach first achieves three features for each API by calculating similarities between this new question and the functional description in API specifications, including the cosine similarity and two word embedding based similarities. Then, a weighting scheme is employed to calculate the final score for each API by combining the three features, and the weight of each feature is achieved by a leaning-to-rank system. At last, all the APIs are ranked in a descending order based on their final scores, and the top ranked APIs are recommended. Evaluated over a non-publicly available corpus, this approach is superior to the simple method, which only uses the cosine similarity to rank APIs. However, the existing approach does not leverage the domain specific knowledge to address the

This paper is partially supported by the National Natural Science Foundation of China under Grant No. 61722202 and Grant No. 61370144.

J. Zhang, H. Jiang, Z. Ren, and X. Chen are with the School of Software, Dalian University of Technology, Dalian, China. E-mail: jingxuanzhang@mail.dlut.edu.cn, jianghe@dlut.edu.cn, zren@dlut.edu.cn, chenxin4391@mail.dlut.edu.cn.

Q2API task. Hence, there is still much room for improvement.

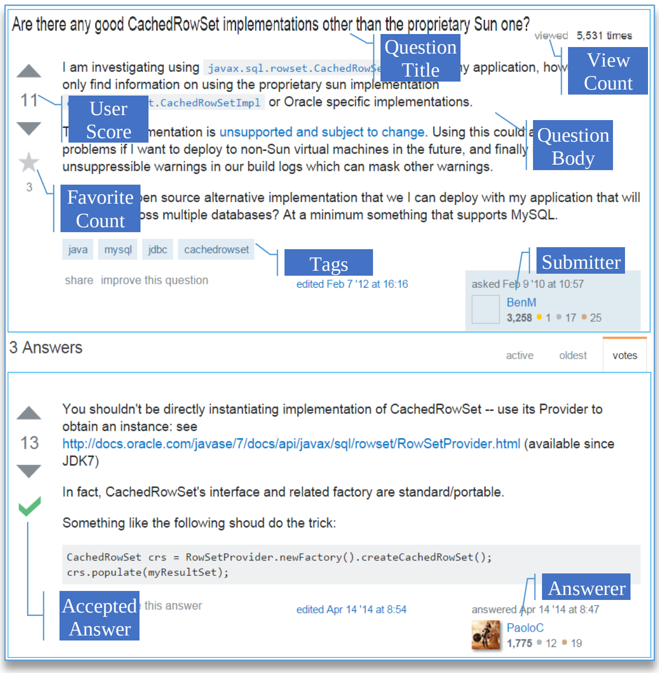
In this paper, we propose a novel approach of **R**ecommending **A**PIs for API related questions based on API **S**pecifications and **H**istorical resolved questions (**RASH**). In contrast to the existing approach, RASH fully leverages the domain specific knowledge in Stack Overflow to better address the Q2API task. By observing plentiful API related questions with their correct APIs in Stack Overflow, we find that if more overlapping words are contained in both the new API related question and the functional description in an API's specification, the API is highly likely to resolve the new question (see Section II.C). In addition, similar questions can be resolved by similar or the same correct APIs. Hence, we can leverage the correct APIs that have resolved historical similar questions to resolve the new API related question (see Section II.C). These important observations motivate us to consider and better leverage the valuable information in API specifications and historical resolved questions.

More specifically, RASH works as follows. Given a new API related question targeted towards a specific programming language (e.g., Java), RASH first achieves two correlation scores for each API in the same programming language by leveraging both API specifications and historical resolved questions. RASH obtains the first correlation score by calculating the cosine similarity between the new question and the functional description in each API's specification. Next, RASH ranks all the APIs based on their correlation scores and selects the top 500 APIs as candidate APIs, which are likely to be correct APIs. Meanwhile, RASH also achieves the second correlation score for each API by analyzing similar questions that have been resolved in history with their correct APIs. Then, after normalizing the two correlation scores, RASH calculates their arithmetic mean and treats it as the final score for each API. Finally, RASH employs an API ranking scheme based on candidate APIs with their final scores, and recommends the top ranked APIs to the *submitter* of the new API related question.

We collect and construct a corpus containing 1,234 API related questions with their correct APIs from Stack Overflow, and open it to the public [12]. We conduct extensive experiments over the corpus to evaluate the performance of RASH. From the experimental results we can see that, in terms of parameter selection, RASH achieves the best results when the number of candidate APIs is equal to 500. From the perspective of robustness, RASH performs similarly over high-quality questions and low-quality questions, which indicates that RASH is insensitive to different quality of API related questions. In terms of stability, the performance of RASH is steadily increasing when the number of API related questions is accumulated large enough, i.e., 200. In terms of effectiveness, RASH achieves the Hit@15 (Hit Rate when recommending 15 APIs) of 69.12% and outperforms the state-of-the-art approach by 15.64%.

In summary, this paper makes the following contributions:

- We propose a novel approach named RASH, which leverages the information in both API specifications and historical resolved questions, to better recommend correct APIs for API related questions.



Are there any good CachedRowSet implementations other than the proprietary Sun one?

viewed 5,531 times

I am investigating using `javax.sql.rowset.CachedRowSe` ...y application, how... only find information on using the proprietary sun implementation ...`t.CachedRowSetImpl` or Oracle specific implementations.

11

T...mentation is unsupported and subject to change. Using this could a... problems if I want to deploy to non-Sun virtual machines in the future, and finally unsuppressible warnings in our build logs which can mask other warnings.

3

...en source alternative implementation that we I can deploy with my application that will ...oss multiple databases? At a minimum something that supports MySQL.

java mysql jdbc cachedrowset

share improve this question

edited Feb 7 '12 at 16:16

asked Feb 9 '10 at 10:57
BenM
3,258 ● 1 ● 17 ● 25

3 Answers

active oldest votes

You shouldn't be directly instantiating implementation of CachedRowSet -- use its Provider to obtain an instance: see http://docs.oracle.com/javase/7/docs/api/javax/sql/rowset/RowSetProvider.html (available since JDK7)

13

In fact, CachedRowSet's interface and related factory are standard/portable.

Something like the following shoud do the trick:

```
CachedRowSet crs = RowSetProvider.newFactory().createCachedRowSet();
crs.populate(myResultSet);
```

this answer

edited Apr 14 '14 at 8:54

answered Apr 14 '14 at 8:47
PaoloC
1,775 ● 12 ● 19

Fig. 1. A Q&A pair example

- Experiments over the constructed corpus show that RASH outperforms the state-of-the-art approach by 15.64% in terms of Hit@15.
- We construct a corpus containing 1,234 API related questions with their correct APIs from Stack Overflow and open it to the public [11]. Other researchers can benefit from it for further research.

The remainder of the paper is organized as follows. In Section II, we first show the motivation of this study. We illustrate the framework of RASH with its main components in Section III. Then, we elaborate the experimental setup and experimental results in Section IV and Section V, respectively. Next, in Section VI and Section VII, we introduce the threat to validity and related work. At last, we make a conclusion and mention future work in Section VIII.

II. Motivation

In this section, we first present some preliminaries about how *submitters* post API related questions in Stack Overflow. Next, we demonstrate the importance of API related questions in Stack Overflow, which motivates us to propose an approach to address the Q2API task. Finally, we show our observations on API related questions, which motivate us to better leverage the information in API specifications and historical resolved questions.

A. *Preliminaries*

Fig. 1 shows a Q&A pair example[1] with several essential items, such as *question title*, *question body*, and *tags*. Generally, when a developer (*submitter*) encounters an API usage problem and wants to seek professional help from experienced developers, he/she needs to follow a series of guidelines to

[1] https://stackoverflow.com/questions/2228462/are-there-any-good-cachedrowset-implementations-other-than-the-proprietary-sun-o.

submit a new question in Stack Overflow. First, the *submitter* should summarize and refine the key point of the problem using one sentence, which is called *question title*. Then, the *submitter* should specify the details of the problem, which is called *question body*, in natural language with some code samples (if necessary). In addition, the *submitter* is required to mark this new question with some keywords, which are called *tags*, to categorize this new question. Other developers can answer the new question (*answer*), vote for the question or the answer based on its quality (*user score* of the question or the answer), and mark it as favorite (*favorite count*). Stack Overflow also automatically records the view times of the new question (*view count*). After verifying the posted *answers*, the *submitter* can select one of them as the solution and mark it as accepted (*accepted answer*). However, the *submitter* may have to wait an extremely long time until the *accepted answer* is posted, thus decreasing the working efficiency of the *submitter* [8].

As shown in Fig. 1, *BenM* asks a question on Feb. 9, 2010 to find a good API, which can implement *CachedRowSet* other than the proprietary *Sun* one. After an extremely long time until Apr. 14, 2014, *PaoloC* posts an answer which is accepted. We can see that it takes more than 4 years to resolve this question. In addition, there is a hyperlink to Java API specifications in the *accepted answer*. By parsing the hyperlink, we can find that the *javax.sql.rowset.RowSetProvider* API is the correct API to resolve this question. In addition, this Q&A pair has been viewed 5,531 times, which implies that abundant developers may encounter the same or similar API usage problems.

B. The Importance of API Related Questions

Table I
COMPARISON BETWEEN API RELATED QUESTIONS AND OTHER QUESTIONS

Question	Avg. Question Score	Avg. Answer Score	Avg. View Count	Avg. Favorite Count	Avg. Resolution Time (days)
API Related	4.26	5.26	4,609	0.83	16.97
Others	1.96	3.13	2,225	0.62	13.69

Developers tend to reuse APIs in existing libraries to help them program [13, 14]. Hence, they may encounter various API usage problems when programming with APIs. It is difficult for developers to learn APIs by themselves, and seeking help from experienced developers by asking or searching questions in Stack Overflow is a common practice [4, 5, 6]. In such a way, API related questions usually receive wide attentions from developers, thus making API related questions more important than other questions.

To demonstrate the importance of API related questions, we compare Java API related questions in the constructed corpus (see Section IV.B) against other Java tagged questions with several characteristics shown in Table I. Obviously, both API related questions and their answers achieve higher average *user scores* than other Java tagged questions and their answers. For example, the average *user score* of API related questions is 4.26. In contrast, it is only 1.96 for other Java tagged questions. The average *view count* for API related questions is more than 4.6 thousand, which is more than twice as many as that of other Java tagged questions. The average resolution time of API related questions is nearly 17 days and 3 days longer than that of other Java tagged questions.

In summary, API related questions achieve higher quality, attract more developers, and take longer time to be resolved. Hence, automatically resolving API related questions is significant to abundant developers.

C. Observations on API Related Questions

After observing plentiful API related questions, we have the following two observations, based on which we design our novel approach RASH.

(1) The more lexically similar the functional description in an API's specification is to the new API related question, the more likely that the API can resolve this new question.

(2) The APIs that have resolved similar questions in history can also be used to resolve the new API related question.

A new API related question in Stack Overflow

Question Title: How to get ISO **format** from **time** in **milliseconds** in Java?
Question Body: Is it simple way to get **yyyy-MM-dd HH:mm:ss**,SSS from **time** in **millisecond**? I've found some information how to do this from new **Date**() or **Calendar**.getInstance(), but couldn't find if it can be done from long (e.g. 1344855183166)
Tags: <Java> <**date**>

Functional description in SimpleDateFormat

SimpleDateFormat is a concrete class for **formatting** and parsing **dates** in a locale-sensitive manner. It allows for **formatting** (**date** -> text), parsing (text -> **date**), and normalization…**Date** and **time formats** are specified by **date** and **time** pattern strings…
If the **formatter's Calendar** is the Gregorian **calendar**…
Date or **Time** Component: **Millisecond**…
The following examples show how **date** and **time** patterns are interpreted in the U.S. locale:
yyyy-MM-dd'T'**HH:mm:ss**.SSSZ"…

Fig. 2. An API related question and the functional description of its correct API

We present an example to illustrate the first observation. Fig. 2 shows a simplified API related question[2] in Stack Overflow. Once the question is submitted, other developers try to resolve it by providing a correct API among thousands of possible APIs. After a series of discussions, the correct API is recommended, i.e., *java.text.SimpleDateFormat*, whose functional description in API specification[3] is also shown in Fig. 2. We can see that many overlapping words (in bold fonts) appear in both the question and the functional description of its correct API, such as *date*, *time*, *format*, and *calendar*, hence there is a good lexical match between them.

To better present the second observation, we list all the correct APIs in the constructed corpus (see Section IV.B) and count their frequencies to resolve API related questions. We

[2]https://stackoverflow.com/questions/11933137/how-to-get-iso-format-from-time-in-milliseconds-in-java.
[3] http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html.

rank the correct APIs by their frequencies and show the results in Fig. 3. The x-axis shows the correct API id and the y-axis shows the frequency to resolve questions. We can see that, there are totally 419 correct APIs for 1,234 API related questions in the corpus. On average, one correct API can resolve nearly 3 API related questions. The most frequent correct API is *java.lang.String*, which can resolve as many as 51 API related questions. In addition, 212 correct APIs (more than half correct APIs) can resolve no less than 2 API related questions.

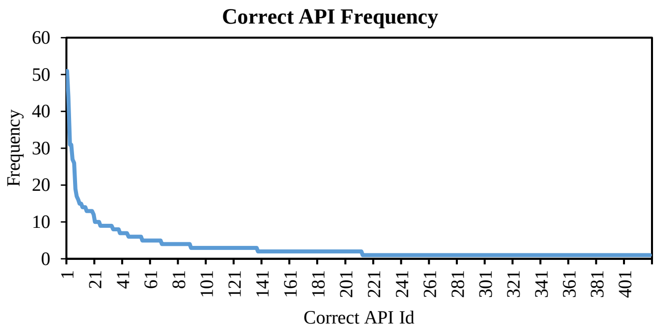


Fig. 3. The correct API frequency in the corpus

In conclusion, if there exists a good lexical match between the new API related question and the functional description in an API's specification, the API is highly likely to resolve this new question. In addition, correct APIs are overlapped for API related questions in Stack Overflow. Hence, the correct APIs of historical resolved questions can also be used to resolve this new API related question. These observations motivate us to consider both API specifications and historical resolved questions to better resolve the Q2API task.

III. Framework

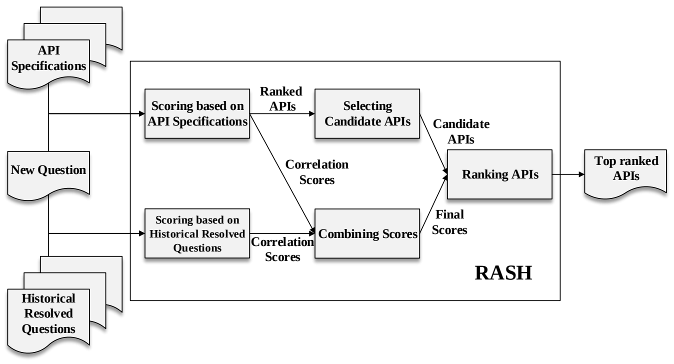


Fig. 4. The framework of RASH

In this section, we illustrate the framework of RASH shown in Fig. 4. The goal of RASH is to resolve API related questions in Stack Overflow by recommending correct APIs for them. RASH takes in the new API related question, API specifications, and historical resolved questions as input, and outputs top ranked APIs for the new question. Hence, the *submitter* of the new question can check the recommended APIs one by one until the correct APIs are found. In such a way, the correct APIs can be quickly located. Obviously, it could be ideal if the correct APIs are ranked as high as possible. RASH consists of five components, including Scoring based on API Specifications, Selecting Candidate APIs, Scoring based on Historical Resolved Questions, Combining Scores, and Ranking APIs. In the following part of this section, we take the API related question in Fig. 2 as an example to clearly illustrate how each component works.

A. *Scoring based on API Specifications*

This component aims to achieve a correlation score for each API in the same programming language (e.g., Java) as the new API related question based on API specifications. This component is designed by the rationality that, the more lexically similar the functional description in an API's specification is to the new API related question, the more likely that they describe the same or similar API usages, thus the more likely that the API can resolve this new question [11].

API specifications play an important role in explaining API usages, including functionalities, behaviors, concepts, and directives, etc., and developers highly expect to find their desired information in API specifications [15]. In this paper, we take Java API (version 7) as a case study and construct a corpus containing 1,234 Java API related questions with their correct Java APIs. Java API specifications are generated through Javadoc following a set of conventions with a uniform style and structure. They are organized as a series of HTML webpages, each of which introduces a specific Java API package or API type [15]. The same as [11], we introduce all the Java *interface* APIs, *class* APIs, *exception* APIs, and *error* APIs as the possible APIs to rank and recommend, and eventually achieve 3,871 Java APIs in total. As a result, it is challenging to recommend correct APIs within so many APIs for API related questions.

For each API, we achieve a correlation score by calculating the widely used cosine similarity between the new API related question and its functional description in API specification [16]. Before calculating the cosine similarity, both the new question and the functional description are transformed into vectors (known as Vector Space Model), where each dimension stands for a term and its corresponding value presents the term's weight. This process consists of a series of natural language processing steps, i.e., tokenization (including camel case splitting), stemming, and stop word removal [16]. Then, each term is given a weight measuring its importance. In this study, we employ the widely used Term Frequency (*TF*) × Inverse Document Frequency (*IDF*) to measure the weight for each term. Given a document (the new question or the functional description in this study), *TF* and *IDF* of a term in this document can be calculated as follows.

$$TF_t = \frac{T_t}{\sum_{i=1}^{n} T_i} \tag{1}$$

where t stands for a term, n stands for the number of distinct terms, and T_t corresponds to the frequency (occurrence number) of term t in the document.

$$IDF_t = log \frac{|D|}{|\{j: t \in d_j\}|} \quad (2)$$

where $|D|$ is the number of documents in total and $|\{j: t \in d_j\}|$ means the number of documents containing term *t*.

Based on *TF* and *IDF*, the weight of a term *t* can be calculated by the following formula.

$$Weight_t = TF_t \times IDF_t \quad (3)$$

In such a way, we can measure the importance of each term and transform both the new API related question and the functional description in each API's specification into vectors.

As shown in Fig. 1, a new API related question contains both *question title* and *question body*. We first transform *question title* and *question body* into two vectors separately using the above-mentioned method. Then, we combine the two vectors to form a final vector as the representation of the new question. Inspired from existing related studies [17, 18], we double the weights of terms in *question title* to strength their impact, since *question title* is a concise summary of the problem. Therefore, the final weights of the terms in the final vector of the new API related question can be calculated as follows.

$$W_t = 2 \times Weight_{t \in title} + 1 \times Weight_{t \in body} \quad (4)$$

After transforming the new API related question and the functional description in each API's specification into vectors, their cosine similarity is calculated by the following formula.

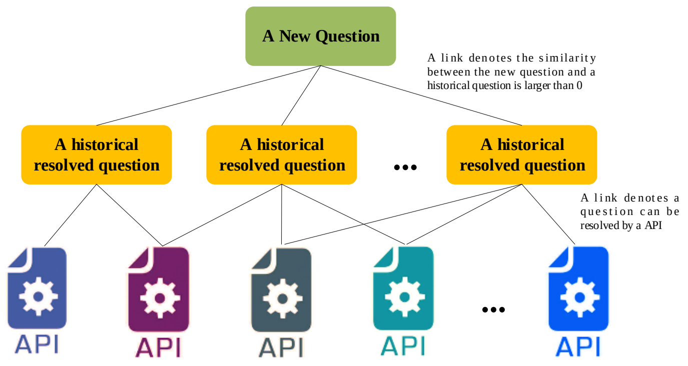
$$sim\_spe(Q, A) = cos(Q, A) = \frac{\sum_{i=1}^{n}(Q_i \times A_i)}{\sqrt{\sum_{i=1}^{n}(Q_i)^2}\sqrt{\sum_{i=1}^{n}(A_i)^2}} \quad (5)$$

where *Q* is the new question and *A* is an API. Q_i and A_i are the final weights of term *i* in *Q* and *A's* functional description.

Running Example. Given the API related question in Fig. 2, RASH achieves a correlation score for each API, ranging from 0.3708 to 0. The correlation scores of more than 1,600 APIs are 0, so there is no overlapping word in the new question and their functional description in API specifications. The correlation score of the correct API *java.text.SimpleDateFormat* is 0.3506, which is the second highest correction score in all the APIs.

B. *Selecting Candidate APIs*

This component aims to achieve candidate APIs after obtaining a correlation score for each API. Intuitively, the higher the correlation score of an API, the more likely the API can resolve the new API related question. Hence, we rank APIs based on their correlation scores and select top 500 APIs as candidate APIs (i.e., #candidate=500), which are highly likely to be the correct APIs to resolve the new API related question. Empirically, the bottom ranked APIs may introduce noises and impose a negative impact on the recommendation results. Hence, they are less likely to be the correct APIs and filtered out. In the Experimental Results section, we will validate whether selecting top 500 APIs as candidates is effective (see Section V.A).


A New Question
A link denotes the similarity between the new question and a historical question is larger than 0
A historical resolved question
A historical resolved question
•••
A historical resolved question
A link denotes a question can be resolved by a API
API
API
API
API
•••
API
Fig. 5. The new question and APIs linking graph

Running Example. Taking the API related question in Fig. 2 as an example, RASH obtains candidate APIs (i.e., top 500 APIs) based on their correlation scores in this component. The correct API *java.text.SimpleDateFormat* ranks the second highest, so it is also regarded as a candidate API. Those APIs whose functional description has no overlapping word with this question (i.e., achieving correlation scores equaling to 0) are filtered out. It implies that RASH can retain correct APIs and remove incorrect APIs as many as possible.

C. *Scoring based on Historical Resolved Questions*

This component aims to achieve another correlation score for each API based on historical resolved questions with their correct APIs. When a new API related question is submitted to Stack Overflow, we can examine and analyze historical similar questions with their correct APIs, since similar questions tend to be resolved by similar or the same APIs (see Section II.C). The correct APIs that have resolved similar questions in history can also help to resolve the new API related question. To the best of our knowledge, the information in historical resolved questions has not been used to address the Q2API task, and we first consider and fully leverage it in this paper.

We order all the questions based on their submission time. For the new API related question, we consider the linking information in the historical resolved API related questions, which have been submitted and resolved before the new question, with their correct APIs. As shown in Fig. 5, the first layer and the second layer present the new API related question and historical resolved questions, respectively, and the third layer shows all the APIs. We first calculate the cosine similarity between the new question and historical resolved questions using formula (5). If their cosine similarity is greater than 0, we link them together. Furthermore, for the historical resolved questions in the second layer, we link them to their correct APIs. In such a way, all the APIs can be indirectly linked to the new API related question through the historical resolved questions as middle agents. Hence, the correlation score for each API can be obtained by the following formula.

$$sim\_his(Q, A) = \sum_{his_q \in his\_sol(A)} (cos(Q, his_q)/m_q) \quad (6)$$

where *Q* is the new question and *A* is an API. $his\_sol(A)$ is the set of historical questions that can be resolved by *A*, and m_q means the number of correct APIs his_q has [16].

Table II
THE API RANKING SCHEME

```
Input: candidate APIs and the final scores of all APIs
1  obtain the final scores for all the candidate APIs
2  PriorSet = Φ
3  foreach (API A in candidate APIs)
4      if (A appears in the question title and tags)
5          add A to PriorSet
6  if (PriorSet is not null)
7      rank APIs in PriorSet based on their frequencies in the question
       title and tags. If two APIs have the same frequency, rank them
       based on their first emerging locations
8  if (the number of ranked APIs is less than 15)
9      rank the rest candidate APIs based their final scores
Output: a ranked API recommendation list containing 15 APIs
```

Running Example. The API related question in Fig. 2 is submitted on Aug. 13, 2012. Before the submission of this question, 302 API related questions have been resolved in the constructed corpus (see Section IV.B). The correct API *java.text.SimpleDateFormat* have resolved 9 API related questions in history. After analyzing these historical resolved questions, RASH achieves the second correlation score for the correct API of 2.2012, which is the highest in all the APIs.

D. Combining Scores

This component aims to achieve a final score for each API. After obtaining the two correlation scores for each API, we first normalize them into the range from 0 to 1 by divided by their maximal values. Then, we combine the two correlation scores into a final score by calculating their arithmetic mean as follows.

$$FinalSocre(Q,A) = \frac{\overline{sim\_spe(Q,A)} + \overline{sim\_his(Q,A)}}{2} \qquad (7)$$

where $\overline{X}$ means the normalized value of X, e.g., $\overline{sim\_spe(Q,A)}$ is the normalized $sim\_spe(Q,A)$.

In this study, the two correlation scores for each API are given the same weight due to two major reasons. First, treating the two correlation scores equally is a simple but efficient method [17]. Second, we find that the weights of the correlation scores have little effect on the final results of RASH in some preliminary experiments. In the Experimental Results section, we will validate the effectiveness of treating them equally (see Section V.B).

Running Example. For the API related question in Fig. 2, RASH achieves the final score for each API in this component. The two normalized correlation scores of the correct API *java.text.SimpleDateFormat* are 0.9455 and 1, respectively. Hence, the final score of the correct API is 0.9728, which is the highest in all the APIs.

E. Ranking APIs

This component aims to rank candidate APIs and recommend top 15 APIs by an API ranking scheme. Recommending top 15 results is a common practice in the recommendation systems within the software engineering domain, and many similar works also employ the same mechanism [11, 16]. The API ranking scheme is designed by the following observations. If a candidate API appears in the *question title* or *tags* of the new API related question, it is highly likely that the question is discussing the usage of the candidate API, since the *submitter* sometimes puts the APIs, which he/she does not know the proper usages, in the *question title* or *tags* to make the question easy to be found and resolved. Hence, such candidate APIs should be ranked the highest. In contrast, if there is no candidate API in the *question title* or *tags* of the new question, the candidate APIs achieving the larger final scores should be ranked higher.

Table II shows the API ranking scheme, which takes in candidate APIs as well as the final scores of all APIs, and outputs the top 15 ranked APIs. First, we obtain the final scores for all the candidate APIs, since we only consider candidate APIs and filter out the rests (line 1). Next, we define an API set named *PriorSet*, which is initialized as empty (line 2). The APIs in *PriorSet* are first-rank APIs, so they are ranked the highest in the recommendation list. Then, for each candidate API (line 3), we check whether it appears in the *question title* and *tags* of the new API related question (line 4). If true, this candidate API is added into the *PriorSet* (line 5). In such a way, after checking all the candidate APIs, we can obtain an API *PriorSet*. For the APIs in *PriorSet* (line 6), we rank them based on their frequencies in the *question title* and *tags* of the new API related question. If any two APIs have the same frequency, we rank them based on their first emerging locations, i.e., the APIs emerging in the front are ranked higher. In such a way, we can rank candidate APIs in *PriorSet* (line 7). Finally, if the number of ranked APIs is less than 15 (line 8), we rank the rest candidate APIs based on their final scores (line 9). In this manner, we can obtain a ranked API recommendation list containing 15 APIs and recommend them to the *submitter* of the new API related question.

Running Example. Still taking the API related question in Fig. 2 as an example, there is no API contained in the *question title* and *tags* of the question. As a result, the *PriorSet* is empty. Then, all the candidate APIs are ranked based on their final scores. The correct API *java.text.SimpleDateFormat* achieves the largest final score, so it is ranked the highest. Obviously, the correct API can be easily found by the *submitter* of the question, thus accelerating the resolution of this question.

IV. EXPERIMENTAL SETUP

In this section, we first describe the experiment settings. Next, we present how we collect and construct the corpus used in the experiments. Then, we illustrate the details of the baseline approach. Finally, we show the evaluation metrics used in this paper.

A. Experiment Settings

In this study, we conduct all the experiments on a Core i7 CPU computer with 8 GB memory running Windows 7. RASH is implemented in the Java Programming language compiled by JDK 7. In addition, we are to open all source code of RASH after this paper is published.

B. Data Collection

In the previous study [11], Ye *et al.* propose a seminal approach towards addressing the Q2API task, and evaluate their approach over a corpus of 604 API related questions with their correct APIs. However, this corpus is not publicly available. Hence, we construct a new corpus and open it to the public [12]. Similarly, we follow four steps described in [11] to construct the corpus.

(1) We download the Stack Overflow dump files published in September 2016[4], and combine the questions tagged with *Java* with their *accepted answers* to generate a series of Q&A pairs. Eventually, we obtain 990,923 Java tagged Q&A pairs in this step.
(2) For the Java tagged Q&A pairs, we only retain those Q&A pairs whose *accepted answers* have hyperlinks to the Java API specifications. In such a way, we can ensure that these questions are API related questions. After this step, we achieve 3,926 Q&A pairs.
(3) For the retained Q&A pairs, we further filter out those Q&A pairs whose *user score* of either the question or the *accepted answer* is lower than 0. This step can reduce low-quality Q&A pairs and false positive correct APIs as many as possible. After this step, 1,234 high-quality Q&A pairs can be obtained.
(4) For each question in the retained Q&A pair, we obtain the correct APIs by parsing the hyperlinks to the Java API specifications in the *accepted answer*. In such a way, the API specifications of the correct APIs have semantic overlap with the *accepted answers*, thus the correct APIs can resolve the API related questions. In this step, we eventually achieve 1,234 Java API related questions with their correct APIs.

It should be noted that Stack Overflow contains more API related questions in reality. To make it easy to obtain the correct APIs, conform to the definition of Q2API, and follow the same procedures as [11], we construct the corpus containing 1,234 API related questions, which is more than twice as large as the corpus in [11]. In the future, we plan to introduce more API related questions to verify RASH.

Table III
CHARACTERISTICS OF THE CORPUS

# Avg. distinct words in question title	4.60
# Avg. distinct words in question body	46.12
# Avg. tags each question has	3.07
# Avg. sentences in question body	5.12
Submission time of the first question	Sep. 9, 2008
Submission time of the last question	Aug. 31, 2016

The characteristics of the corpus are shown in Table III. On average, a *question title* includes 4.60 distinct words, while a *question body* contains 46.12 distinct words within 5.12 sentences. In addition, each question involves nearly 3 *tags*. The first question is submitted on Sep. 9, 2008 and the last question is submitted on Aug. 31, 2016.

C. The Baseline Approach

Ye *et al.* first issue the task of Q2API and present their attempts toward resolving this task using an IR technique. It is the state-of-the-art approach, so we employ it as the baseline approach for comparison [11]. This baseline approach uses the word embedding technique to calculate similarities and recommend APIs for API related questions. Word embedding is a technique to map words into vectors of real numbers. Based on word embedding, the similarity between two words can be calculated. Furthermore, the asymmetric document similarity can also be calculated.

More specifically, for each new API related question, the baseline approach calculates three features for each API based on API specifications, including the cosine similarity, the word embedding based similarity from the new question to API, and the word embedding based similarity from API to the new question. The cosine similarity is calculated between the new question and the functional description in each API's specification. In the word embedding based similarity, an asymmetric similarity is calculated between them after words are represented into vectors. Then, a weighted sum of the three features is calculated, and the weight of each feature is trained from a training set using a learning-to-rank system, which aims to optimize the rank so that the correct APIs are ranked in the top of the training set. Finally, all the APIs are ranked based on their weighted sums, and the top ranked APIs are recommended. They validate their approach over a non-publicly available corpus containing 604 Java API related questions with their correct APIs. The results show that the baseline approach is superior to the straightforward method, which only uses the simple cosine similarity to rank APIs.

D. Evaluation Metrics

To measure the effectiveness of different approaches from various aspects, inspired from [11, 16, 19], we employ four evaluation metrics in this study, including Hit Rate, Normalized Discounted Cumulative Gain (NDCG), Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR). Among them, Hit Rate and NDCG are often used to evaluate recommendation systems [20], and MAP as well as MRR are widely used in IR [11, 16]. Since we recommend top 15 APIs for each API related question, we calculate the four evaluation metrics from top 1 to top 15 to clearly and incrementally present the performance, which are denoted as Hit@K, NDCG@K, MAP@K, and MRR@K (K is the recommended number ranging from 1 to 15), respectively.

Hit Rate measures the percentage of questions that can be resolved by the recommended APIs [20]. Hit Rate is calculated by the number of questions whose correct APIs are exactly recommended divided by the number of all the questions, of which the formula is shown as follows.

$$Hit@K = \frac{\#\, questions\ can\ be\ resolved\ by\ the\ top\ K\ recommended\ APIs}{\#\, questions\ in\ total} \quad (8)$$

NDCG measures the quality of the rank by calculating the gain of each result according to its position [20]. As a normalized DCG, NDCG is calculated by divided by a special ideal DCG, which ranks all 1s higher than 0s. Therefore, NDCG can be calculated as follows.

[4] https://archive.org/details/stackexchange

$$NDCG@K = \frac{DCG@K}{ideal\ DCG@K} \qquad DCG@K = \sum_{i=1}^{K} \frac{2^{rel(i)}-1}{log_2(i+1)} \qquad (9)$$

where i is the rank. $rel(i)$ is a binary function to check whether the API in rank i is correct or not. For example, if the API in rank i is a correct API, $rel(i) = 1$. Otherwise, $rel(i) = 0$.

MAP measures the quality of the rank when a query (a new API related question in this paper) may have multiple correct answers (correct APIs in this paper) [11, 16]. MAP is the mean of all the average precisions of queries, and it can be calculated as follows.

$$MAP@K = \frac{1}{|Q|}\sum_{j=1}^{Q} \frac{\sum_{i=1}^{K}(P(i)\times rel(i))}{\#\ correct\ answers} \qquad P(i) = \frac{\#\ correct\ answers\ in\ top\ i}{i} \quad (10)$$

where j is a query, $|Q|$ is the number of queries, and $P(i)$ is the precision at a given cut-off rank i.

MRR is another widely used evaluation metric to measure the quality of the rank in IR [11, 16]. MRR is the average of the reciprocal ranks for all the queries. The reciprocal rank of a single query is the multiplicative inverse of the first correct answer. Hence, MRR can be calculated as follows.

$$MRR@K = \frac{1}{|Q|}\sum_{j=1}^{Q} \frac{1}{K\_Rank_i} \qquad (11)$$

where K_Rank_i means the rank position of the first correct answer in the top K recommended list for the i-th query.

V. Experimental Results

In this section, we investigate five Research Questions (RQs) to investigate the effectiveness of RASH.

A. *Investigation to RQ1*

RQ1: How does the number of candidate APIs influence the performance of RASH?

Motivation. In the Selecting Candidate APIs component of RASH, the number of candidate APIs (#candidate) is set to 500 by default, i.e., #candidate=500. To verify whether setting #candidate=500 is effective and close to the optimal value, we set up this RQ.

Approach. We adjust #candidate by setting it to several different values, including 100, 500, 1,000, and the number of all the APIs (i.e., 3,871). By comparing the results among {100, 500, and 1,000}, we can know which value is the best and close to the optimal value. In addition, regarding all the APIs as candidate APIs is equal to removing the component of Selecting Candidate APIs. By comparing the results between 500 and the number of all the APIs, we can know whether the component of Selecting Candidate APIs is effective.

Result. Fig. 6 shows the results of RASH when setting #candidate to different values in terms of Hit Rate and NDCG from top 1 to top 15, and Table IV presents the results of RASH in terms of MAP and MRR accordingly. A specific number in the figure and table means setting #candidate equaling to it, e.g., *100* means #candidate=100, and *all* means #candidate=the number of all the APIs.

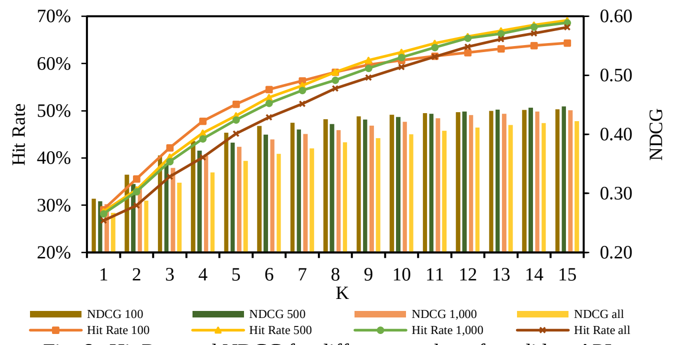


Fig. 6. Hit Rate and NDCG for different number of candidate APIs

Table IV
MAP and MRR for different number of candidate APIs

K	MAP				MRR			
	100	500	1,000	all	100	500	1,000	all
1	0.2753	**0.2769**	0.2725	0.2583	0.2853	**0.2869**	0.2820	0.2674
2	0.3014	**0.3020**	0.2976	0.2763	0.3096	**0.3100**	0.3055	0.2836
3	0.3239	**0.3244**	0.3182	0.2961	0.3325	**0.3329**	0.3266	0.3039
4	0.3363	**0.3376**	0.3309	0.3065	0.3445	**0.3457**	0.3387	0.3140
5	0.3448	**0.3456**	0.3395	0.3167	0.3521	**0.3530**	0.3467	0.3241
6	0.3507	**0.3521**	0.3455	0.3227	0.3580	**0.3595**	0.3526	0.3299
7	0.3550	**0.3559**	0.3495	0.3271	0.3622	**0.3631**	0.3564	0.3339
8	0.3580	**0.3596**	0.3525	0.3314	0.3651	**0.3666**	0.3592	0.3380
9	0.3605	**0.3621**	0.3552	0.3341	0.3678	**0.3694**	0.3620	0.3406
10	0.3624	**0.3639**	0.3576	0.3362	0.3696	**0.3711**	0.3642	0.3428
11	0.3642	**0.3657**	0.3595	0.3383	0.3713	**0.3728**	0.3662	0.3448
12	0.3657	**0.3670**	0.3611	0.3401	0.3727	**0.3740**	0.3678	0.3465
13	0.3666	**0.3679**	0.3618	0.3412	0.3737	**0.3749**	0.3685	0.3478
14	0.3673	**0.3688**	0.3629	0.3421	0.3744	**0.3758**	0.3696	0.3486
15	0.3680	**0.3694**	0.3635	0.3430	0.3751	**0.3765**	0.3702	0.3495

First, we try to compare the results when setting #candidate to {100, 500, and 1,000}. We can see from Fig. 6 and Table IV that, RASH achieves the best results on the whole when setting #candidate=500, especially when recommending 15 APIs. For example, RASH achieves Hit@15 and NDCG@15 of 69.12% and 0.4475. In contrast, when setting #candidate=100 and #candidate=1,000, RASH achieves 64.34% and 68.64% in terms of Hit@15 and 0.4427 and 0.4409 in terms of NDCG@15, respectively. As for MAP and MRR, RASH also achieves the best results when setting #candidate=500. Therefore, setting #candidate=500 is close to the optimal value.

Comparing the results between 500 and the number of all the APIs can show the effectiveness of the Selecting Candidate APIs component. As shown in Fig. 6 and Table IV, RASH achieves better results when setting #candidate=500 than that when setting #candidate=all. For example, when setting #candidate=500, RASH achieves Hit@15 and NDCG@15 of 69.12% and 0.4475, respectively. In contrast, when setting #candidate=all, RASH achieves Hit@15 of 67.67% and NDCG@15 of 0.4222, respectively. Additionally, in terms of MAP and MRR, RASH also achieves better results when setting #candidate=500. Therefore, the component of Selecting Candidate APIs is effective.

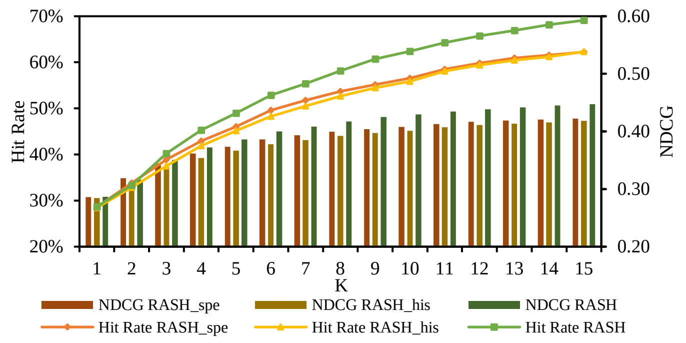


Fig. 7. Hit Rate and NDCG for RASH and its variants

Table V
MAP AND MRR FOR RASH AND ITS VARIANTS

K	MAP			MRR		
	RASH_spe	RASH_his	RASH	RASH_spe	RASH_his	RASH
1	0.2761	0.2749	**0.2769**	0.2861	0.2844	**0.2869**
2	**0.3041**	0.2986	0.3020	**0.3120**	0.3063	0.3100
3	0.3204	0.3140	**0.3244**	0.3290	0.3220	**0.3329**
4	0.3306	0.3252	**0.3376**	0.3391	0.3329	**0.3457**
5	0.3376	0.3324	**0.3456**	0.3453	0.3394	**0.3530**
6	0.3434	0.3377	**0.3521**	0.3512	0.3447	**0.3595**
7	0.3468	0.3410	**0.3559**	0.3544	0.3478	**0.3631**
8	0.3492	0.3436	**0.3596**	0.3568	0.3505	**0.3666**
9	0.3509	0.3456	**0.3621**	0.3584	0.3525	**0.3694**
10	0.3522	0.3470	**0.3639**	0.3598	0.3539	**0.3711**
11	0.3540	0.3491	**0.3657**	0.3616	0.3560	**0.3728**
12	0.3551	0.3502	**0.3670**	0.3626	0.3570	**0.3740**
13	0.3561	0.3510	**0.3679**	0.3635	0.3579	**0.3749**
14	0.3566	0.3515	**0.3688**	0.3640	0.3584	**0.3758**
15	0.3570	0.3523	**0.3694**	0.3644	0.3591	**0.3765**

The reason may be that, using a small parameter value (e.g., #candidate=100) will filter out some correct APIs. In contrast, using a large one (e.g., #candidate=1,000 or #candidate=all) will retain too many irrelevant APIs. Hence, choosing a suitable moderate value (e.g., #candidate=500) can retain correct APIs and filter out irrelevant APIs as many as possible.

Conclusion. RASH achieves the best results, when setting #candidate=500. The component of Selecting Candidate APIs is effective to retain correct APIs and reduce irrelevant APIs.

B. *Investigation to RQ2*

RQ2: Whether the combination of both the two correlation scores can achieve better results than any of them alone?

Motivation. RASH combines both the correlation scores from API specifications and historical resolved questions to rank APIs for new API related questions. To validate whether combining them can achieve better results than any of them alone, we set up this RQ.

Approach. We define two variants of RASH. The first variant named RASH_spe, which uses the correlation scores from API specifications to select candidate APIs, regards these correlation scores as the final scores for APIs, and applies the same API ranking scheme to rank and recommend candidate APIs. The second variant named RASH_his only considers the correlation scores from historical resolved questions in the same way. By comparing the results of RASH against its two variants, we can know whether combing the two correlation scores could achieve better results. In addition, by comparing the results between RASH_spe and RASH_his, we can acquire whether giving the two correlation scores from API specifications and historical resolved questions the same weight in formula (7) is effective (see Section III.D).

Results. Fig. 7 and Table V show the results of RASH and its two variants in terms of Hit Rate, NDCG, MAP, and MRR from top 1 to top 15. It is obvious that RASH achieve better results than RASH_spe and RASH_his, especially when the number of recommended APIs is increasing. Meanwhile, RASH_spe and RASH_his perform similarly in terms of all the evaluation metrics. When recommending only one API (i.e., K=1), RASH achieves similar results as RASH_spe and RASH_his. For example, the Hit@1 of RASH is 28.69%. While, RASH_spe and RASH_his achieve 28.61% and 28.44%, respectively. In terms of the other evaluation metrics, RASH also achieves the best results, but the disparity is trivial. When considering top 5 APIs, RASH still achieves better results than RASH_spe and RASH_his. For example, RASH achieves Hit@5 of 48.95% and improves RASH_spe and RASH_his by 2.92% and 3.81%, respectively. When recommending 10 APIs, RASH performs quite better than RASH_spe and RASH_his. In particular, when the recommended length is increased to 15, RASH achieves significantly better results than RASH_spe and RASH_his. For instance, RASH achieves Hit@15 of 69.12%. In contrast, RASH_spe and RASH_his only achieve 62.24% and 62.32%, respectively. In addition, as for NDCG@15, RASH reaches to 0.4475 and improves RASH_spe and RASH_his by 0.0249 and 0.0292, respectively. As for MAP@15 and MRR@15, RASH also outperforms its variants.

After demonstrating the effectiveness of combining the correlation scores from both API specifications and historical resolved questions, we would like to explore the underlying reasons. Correlation scores from API specifications detect the correct APIs for new API related questions from the lexical perspective. API specifications explain APIs' functionalities in the implementation domain, and API related questions in Stack Overflow describe the requirements in the problem domain. If they can match lexically, the APIs are highly likely to resolve the API related questions. In addition, we observe that the correct APIs are overlapped for similar questions. Therefore, we fully leverage the correct APIs that have resolved similar questions in history to resolve new API related questions. In such a way, the two correlation scores from API specifications and historical resolved questions complement and cooperate each other. Hence, RASH can achieve better results.

In addition, we can also find that RASH_spe and RASH_his achieve similar results in terms of all the evaluation metrics. It implies that the two correlation scores from API specifications and historical resolved questions make similar contributions to detect the correct APIs. Hence, it is reasonable to give the two correlation scores the same weight in formula (7), when calculating the final score for each API (see Section III.D).

Conclusion. By aggregating both the correlation scores from API specifications and historical resolved questions, RASH can better recommend correct APIs for API related questions.

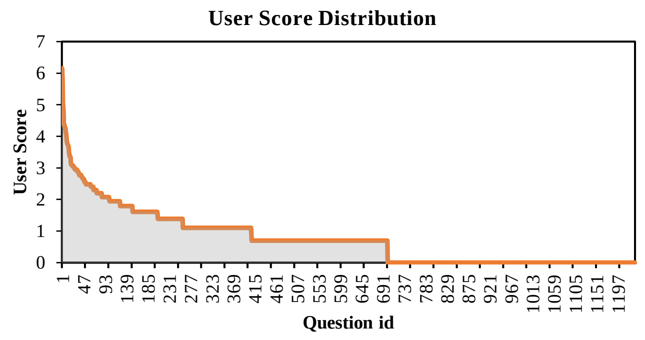


Fig. 8. User score of each question in the corpus. To better present the trend, we show the base-*e* logarithm of each user score.

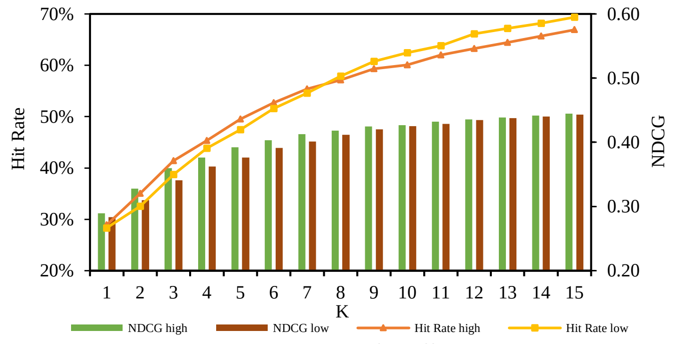


Fig. 9. Hit Rate and NDCG for different subsets.

Table VI
MAP AND MRR FOR DIFFERENT SUBSETS

K	MAP		MRR	
	high	low	high	low
1	**0.2800**	0.2730	**0.2892**	0.2833
2	**0.3117**	0.2963	**0.3199**	0.3045
3	**0.3321**	0.3169	**0.3411**	0.3251
4	**0.3425**	0.3299	**0.3509**	0.3378
5	**0.3515**	0.3377	**0.3592**	0.3450
6	**0.3566**	0.3443	**0.3645**	0.3519
7	**0.3615**	0.3486	**0.3684**	0.3562
8	**0.3633**	0.3529	**0.3705**	0.3603
9	**0.3658**	0.3562	**0.3730**	0.3635
10	**0.3666**	0.3580	**0.3737**	0.3652
11	**0.3683**	0.3592	**0.3755**	0.3664
12	**0.3693**	0.3610	**0.3765**	0.3684
13	**0.3703**	0.3619	**0.3775**	0.3692
14	**0.3712**	0.3625	**0.3783**	0.3699
15	**0.3720**	0.3633	**0.3792**	0.3707

C. *Investigation to RQ3*

RQ3: Is RASH sensitive to the quality of the questions?

Motivation. Due to different experience and expertise of *submitters*, the quality of API related questions may vary sharply [21]. Some questions can clearly describe the real problems without missing any important information. In contrast, the other questions may lack some critical details, making them hard to be resolved. To investigate how RASH performs over different quality of questions, we set up this RQ.

Approach. We split the constructed corpus into two subsets, i.e., high-quality subset and low-quality subset. Similar as [21], the quality of a question is judged by its *user score*. Inspired from [5, 8], we set up 2 as the threshold to split the corpus, thus the two generated subsets can retain similar characteristics with the corpus as much as possible. If the *user score* of a question is larger than 2, it is treated as a high-quality question and put into the high-quality subset. Otherwise, it is placed into the low-quality subset. We rank the API related questions in the constructed corpus based on their *user score*s, and find that it shows a long-tailed distribution as plotted in Fig. 8. About two-third questions (i.e., 826 questions) achieve *user scores* no more than 2, and they are allocated to the low-quality subset. The rest 408 questions achieving *user scores* larger than 2 are put into the high-quality subset. By applying RASH over the two subsets separately, we can obtain the comparison results. If RASH performs similarly over the two subsets, it indicates that RASH is insensitive and robust to the quality of the questions.

Results. Fig. 9 and Table VI show the result of RASH over the two subsets of questions with different quality in terms of Hit Rate, NDCG, MAP, and MRR. *high* and *low* present the results of RASH over the high-quality subset and the low-quality subset, respectively. We can see from the figure and table that, RASH performs similarly over the high-quality subset and low-quality subset. For example, RASH achieves Hit@15 of 66.91% over the high-quality subset and 69.37% over the low-quality subset. In terms of NDCG, RASH achieves NDCG@15 of 0.4449 over the high-quality subset and 0.4433 over the low-quality subset, in which the disparity is trivial. Similarly, RASH also achieves similar results over the two subsets in terms of MAP and MRR.

The reason why RASH is insensitive to different quality of questions may be that, RASH utilizes two correlation scores to rank APIs. A low-quality question may cause a correlation score fails to find the correct APIs. However, another correlation score can compensate for this deficiency to precisely detect the correct APIs. As a result, questions with different quality have little impact on the performance of RASH.

Conclusion. RASH performs similarly over high-quality questions and low-quality questions. RASH is insensitive and robust to the quality of questions.

D. *Investigation to RQ4*

RQ4: What is the impact of the question number on the performance of RASH?

Motivation. RASH leverages the information in historical resolved questions with their correct APIs to recommend APIs for new API related questions. More resolved questions with their correct APIs exist in history, more useful information can be leveraged by RASH. Hence, the number of questions may influence the performance of RASH. To investigate what is the impact of the question number on RASH's performance, we set up this RQ.

Approach. There are 1,234 API related questions in the constructed corpus, and we sort them sequentially based on their submission time. We verify RASH over the early N submitted questions, where N ranges from 1 to 1,234. The results of early N submitted questions are synthesized to form the final results of RASH. In such a way, we can know how RASH performs when the number of questions changes.

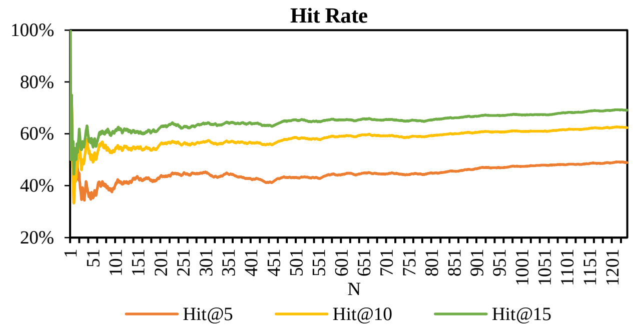


Fig. 10. Hit Rate for different number of questions

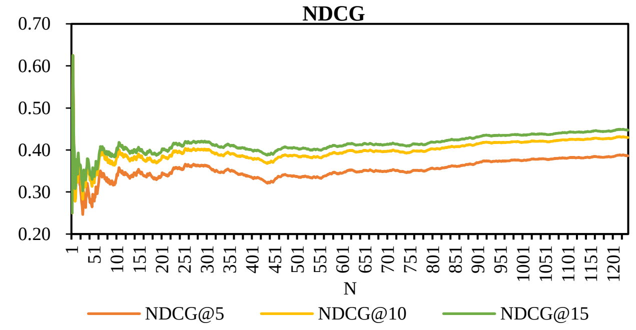


Fig. 11. NDCG for different number of questions

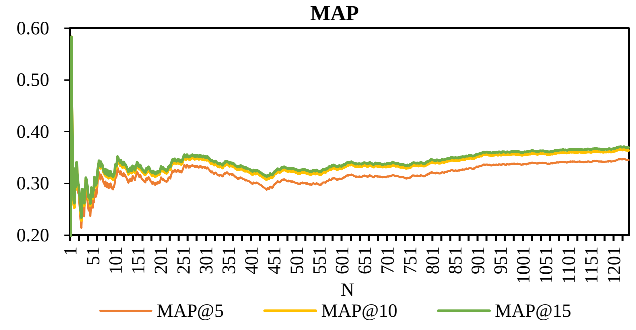


Fig. 12. MAP for different number of questions

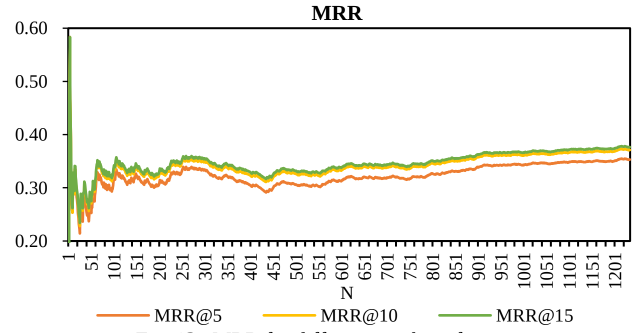


Fig. 13. MRR for different number of questions

Results. Fig. 10, 11, 12, and 13 show the results of RASH over different number of questions in terms of the four evaluation metrics. To clearly distinguish and show the results of each evaluation metric, we only present the results of top 5, top 10, and top 15.

We can see from the figures that all the evaluation metrics are unstable when RASH is applied over a small number of questions, i.e., less than about 200 questions. The values of these evaluation metrics raise in some specific number of questions, while fall in the others. For example, when the number of questions is only 10, RASH achieves Hit@15 of 50%. Then, it raises to 60% when the number of questions reaches to 20. However, the value of Hit@15 falls to 56.67% for 30 questions. The other evaluation metrics also show similar trends.

When the number of questions exceeds a specific value, i.e., 200, RASH behaves steadily and all the evaluation metrics show slightly upward trends along with the increasing of the question number. For example, along with the question number increasing from 200 to 1,234, RASH achieves Hit@15 from 62.5% to 69.12%. The curves of the other evaluation metrics also show similar trends along with the growth of the question number.

This phenomenon can be explained as follows. A small number of questions means that only limited number of historical resolved questions exist. In this situation, marginally less information in historical resolved questions with their correct APIs can be leveraged by RASH, which mainly relies on API specifications to detect correct APIs. Hence, RASH performs unstable. When the information in historical resolved questions is accumulated large enough, RASH learns from both API specifications and historical resolved questions, so RASH performs better.

Conclusion. The performance of RASH is steadily increasing, when the number of questions exceeds 200.

E. Investigation to RQ5

RQ5: How does RASH perform compared against the baseline approach?

Motivation. As we described, the baseline approach is the state-of-the-art approach to resolve the Q2API task. In this RQ, we try to investigate whether RASH is superior to the baseline approach.

Approach. Based on the procedures described in [11], we implement the baseline approach accordingly and verify it over the constructed corpus to achieve the results.

Results. Fig. 14 shows the results of RASH and the baseline approach in terms of Hit Rate and NDCG, and Table VII presents the results of MAP and MRR accordingly. From the figure and table we can see that, RASH achieves significantly better results than the baseline approach.

RASH achieves Hit@5 of 48.59% and NDCG@5 of 0.3862. However, the baseline only achieves 36.79% and 0.2679, respectively. In terms of MAP and MRR, RASH also outperforms the baseline approach by 0.1156 and 0.1183. When the length of the recommendation list improves to 10, RASH achieves Hit@10 of 62.40%. It indicates that more than 62% correct APIs can be recommended. In contrast, the baseline approach only achieves 47.08%. As for the other evaluation metrics, RASH also outperforms the baseline approach by about 0.12. When recommending 15 APIs for each API related question, RASH achieves Hit@15 of 69.12% and the baseline approach only achieves 53.48%. It implies that RASH recommends almost 70% correct APIs for API related questions, and outperforms the baseline approach by 15.64%. In addition, RASH achieves NDCG@15 of 0.4475 and the baseline approach only achieves 0.3173. In terms of MAP@15 and MRR@15, RASH also outperforms the baseline approach by 0.1206 and 0.1230, respectively.

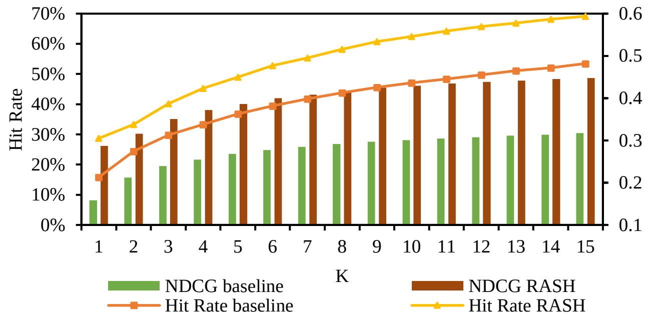


Fig. 14. Hit Rate and NDCG for different approaches

Table VII
MAP AND MRR FOR DIFFERENT APPROACHES

K	MAP			MRR		
	baseline	RASH	Improvement	baseline	RASH	Improvement
1	0.1536	0.2769	**+ 0.1233**	0.1580	0.2869	**+ 0.1288**
2	0.1960	0.3020	**+ 0.1060**	0.2010	0.3100	**+ 0.1090**
3	0.2143	0.3244	**+ 0.1101**	0.2188	0.3329	**+ 0.1141**
4	0.2230	0.3376	**+ 0.1146**	0.2277	0.3457	**+ 0.1180**
5	0.2300	0.3456	**+ 0.1156**	0.2347	0.3530	**+ 0.1183**
6	0.2342	0.3521	**+ 0.1179**	0.2391	0.3595	**+ 0.1203**
7	0.2375	0.3559	**+ 0.1184**	0.2424	0.3631	**+ 0.1207**
8	0.2399	0.3596	**+ 0.1197**	0.2449	0.3666	**+ 0.1217**
9	0.2420	0.3621	**+ 0.1201**	0.2470	0.3694	**+ 0.1224**
10	0.2435	0.3639	**+ 0.1205**	0.2484	0.3711	**+ 0.1227**
11	0.2448	0.3657	**+ 0.1209**	0.2495	0.3728	**+ 0.1232**
12	0.2461	0.3670	**+ 0.1209**	0.2507	0.3740	**+ 0.1233**
13	0.2472	0.3679	**+ 0.1207**	0.2518	0.3749	**+ 0.1232**
14	0.2479	0.3688	**+ 0.1209**	0.2525	0.3758	**+ 0.1233**
15	0.2488	0.3694	**+ 0.1206**	0.2534	0.3765	**+ 0.1230**

The reasons why RASH can achieve better results may be that, it fully leverages the information in historical resolved questions with their correct APIs to detect correct APIs. We have the observation that similar questions share similar or the same correct APIs (see Section II.C), based on which we design our novel approach RASH. In addition, RASH employs an important component, i.e., Selecting Candidate APIs, to accurately filter out incorrect APIs so as to further improve the results.

Conclusion. RASH significantly outperforms the state-of-the-art approach. RASH can better recommend correct APIs for API related questions in Stack Overflow.

VI. Threat to Validity

In this section, we introduce threats to validity, including threats to internal validity and threats to external validity.

A. Threats to Internal Validity

Threats to internal validity are the potential errors or biases in the experiments. RASH aims to recommend correct APIs for new API related questions in Stack Overflow based on API specifications and historical resolved questions. A threat of RASH is the quality of API specifications. API specifications are released accompanied with APIs to describe APIs' usages, and they are usually constructed in a standard process (e.g., Javadoc) by experienced developers [22, 23, 24]. Hence, the quality of API specifications can be guaranteed to a great extent. In addition, another threat of RASH is the parameter selection, i.e., the number of candidate APIs. It is hard to choose an optimal value for this parameter by experience, and the optimal value may be various in different corpora. In this paper, we set it to 500 by default and validate its effectiveness in RQ1. In the future, we plan to automatically configure the optimal value for this parameter in RASH.

B. Threats to External Validity

Threats to external validity are related to the generalization of RASH to other contexts and research settings. We verify RASH over a constructed corpus containing 1,234 Java API related questions in Stack Overflow, and the results show that RASH is robust and superior to the state-of-the-art approach. It is unknown how RASH performs over questions related to other APIs like C# and in other Q&A forums like *Quora*. In the future, we plan to extend the generalization of RASH by introducing more questions related to diverse APIs in other Q&A forums.

VII. Related Work

In this section, we briefly review and discuss two main related works, i.e., mining Stack Overflow and issues related to API usages.

A. Mining Stack Overflow

As a popular technical Q&A forum, Stack Overflow contains valuable information assembling crowd knowledge from millions of developers, and a lot of research tasks have been proposed to mine Stack Overflow in recent years. These research tasks can be roughly divided into two categories, i.e., analyzing Stack Overflow and utilizing Stack Overflow.

1) Analyzing Stack Overflow

Some empirical studies aim to analyze the information in Stack Overflow. Barua *et al.* explore what developers care about by studying all the posts in Stack Overflow, and use topic model to analyze the topics and trends [21]. Yang *et al.* study what security related questions developers ask by conducting a large-scale study on security related questions [25]. Similarly, Rosen and Shihab analyze mobile related questions in Stack Overflow [26]. Beyer and Pinzger find that the most commonly asked questions are "How" and "What" questions by analyzing Android related posts [27]. Bajaj *et al.* analyze web development related posts in Stack Overflow to uncover the challenges for web developers [28]. Linares-Vásquez *et al.* analyze how API changes trigger questions in Stack Overflow [29].

Our work belongs to the category of analyzing Stack Overflow. Different from these studies, we try to resolve API related questions by recommending correct APIs for them rather than empirically study them.

2) Utilizing Stack Overflow

The crowd knowledge in Stack Overflow can be leveraged to resolve other research tasks. Gao *et al.* fix recurring crash bugs by analyzing Q&A pairs in Stack Overflow [30]. Nie *et al.* employ Q&A pairs in Stack Overflow to expand the queries to improve the performance of code search [19]. Jiang *et al.* leverage API related Q&A pairs as features to better detect relevant tutorial fragments [31]. Treude and Robillard enrich API documentation with insight sentences extracted from Stack Overflow [32]. Wong *et al.* automatically generate code comments based on code segments with their descriptions in Stack Overflow [33].

Unlike the above-mentioned studies, we try to recommend correct APIs for API related questions in Stack Overflow, which could accelerate their resolution and save developers' time.

B. Issues Related to API Usages

APIs are hard to learn, and developers will encounter various usage issues when programming with APIs [34]. Robillard and DeLine find that the most severe problem for developers to learn APIs is inadequate API documentation and other learning resources [35]. Zhou and Walker conduct a retrospective analysis on API deprecation in open source libraries [36]. Robbes *et al.* study the react of developers to API deprecation in a Smalltalk ecosystem, and they find that developers sometimes do not consider API deprecation instructions [37]. Linares-Vásquez *et al.* find that change prone and bug prone APIs are threats to the success of mobile applications, and developers are suggested not to use change prone and bug prone APIs [38]. In addition, McDonnell *et al.* study the impact of unstable APIs to their client code, and they suggest that developers should avoid using unstable APIs [39].

Our work is different from these studies. In this paper, we try to resolve API related questions by recommending correct APIs for them.

VIII. Conclusion and Future Work

Developers usually encounter API related programming problems and ask them in Q&A forums like Stack Overflow. Hence, automatically answering API related questions is significant to developers. In this paper, we propose a novel approach named RASH towards resolving API related questions by recommending correct APIs for them. RASH combines and fully leverages the information in both API specifications and historical resolved questions to detect correct APIs for new API related questions. We conduct extensive experiments over a publicly available corpus. The experimental results show that RASH can hit nearly 70% correct APIs and outperform the state-of-the-art approach by 15.64% when recommending 15 APIs for each question. Hence, RASH is capable of better resolving API related questions to further boost developer productivity.

For the future work, we will improve RASH in the following directions. First, we plan to automatically configure the parameters in RASH. Second, we try to verify RASH over questions related to other commonly used APIs, e.g., C#. Third, a tool encapsulating RASH will be developed and distributed to help developers resolve API related questions.

References

[1] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of API usability," in *Proc. ESEM*, Baltimore, Maryland, US, 2013, pp. 5-14.

[2] T. Cho, H. Kim, and J. H. Yi, "Security assessment of code obfuscation based on dynamic monitoring in android things," *IEEE Access*, vol. 5, pp. 6361-6371, 2017.

[3] Y. Zhou, R. H. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proc. ICSE*, Buenos Aires, Argentina, 2017, pp. 27-37.

[4] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proc. ICSE*, Hyderabad, India, 2014, pp. 643-652.

[5] C. Parnin, C. Treude, L. Grammel, and M. A. Storey, "Crowd documentation: exploring the coverage and the dynamics of API discussions on stack overflow," Georgia Tech., Tech. Rep. GIT-CS-12-05, 2012.

[6] T. Mcdonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proc. ICSM*, Eindhoven, The Netherlands, 2013, pp. 70-79.

[7] H. W. Li, Z. C. Xing, X. Peng, and W. Y. Zhao, "What help do developers seek, when and how?" in *Proc. WCRE*, 2013, Germany, pp.142-151.

[8] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, "Answering questions about unanswered questions of stack overflow," in *Proc. MSR*, San Francisco, California, USA, 2013, pp. 97-100.

[9] X. Wang, J. Xu, M. Liu *et al.*, "An ontology-based approach for marine geochemical data interoperation," *IEEE Access*, vol. 5, pp. 13364-13371, 2017.

[10] S. Ercan, Q. Stokkink, and A. Bacchelli, "Predicting answering times on stack overflow," in *Proc. MSR*, Florence, Italy, 2015, pp. 442-445.

[11] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. ICSE*, Austin, Texas, USA, 2016, pp. 404-415.

[12] J. X. Zhang, DaLian, Liao Ning, China, 2017. [Online]. Available: http://oscar-lab.org/people/~jxzhang/Q2API/.

[13] S. Kubler, J. Robert, A. Hefnawy *et al.*, "Open IoT ecosystem for sporting event management," *IEEE Access*, vol. 5, pp. 7064-7079, 2017.

[14] H. Jiang, J. X. Zhang, Z. L. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for APIs," in *Proc. ICSE*, 2017, pp. 38-48.

[15] USA. (2015, Mar.). *Java™ Platform, Standard Edition 7 API Specification*. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/.

[16] J. Zhou, H. Y. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proc. ICSE*, Zürich, Switzerland, 2012, pp. 14-24.

[17] X. Y. Wang, L. Zhang, T. Xie, J. Anvik, and J. S. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ICSE*, Leipzig, Germany, 2008, pp. 461-470.

[18] C. N. Sun, D. Lo, X. Y. Wang, J. Jiang, and S. C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. ICSE*, Cape Town, South Africa, 2010, pp. 45-54.

[19] L. M. Nie, H. Jiang, Z. L. Ren, Z. Y. Sun, and X. C. Li, "Query expansion based on crowd knowledge for code search," *IEEE TranS. on Services Computing*, to be published. Doi: 10.1109/TSC.2016.2560165.

[20] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, "Dimensions and metrics for evaluating recommendation systems," in *Recommendation Systems in Software Engineering*, New York, USA: Springer, 2014, pp. 245-275.

[21] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in stack overflow," *EMSE*, vol. 19, no. 3, pp. 619-654, 2014.

[22] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proc. ICSE*, Zürich, Switzerland, 2012, pp. 47-57.

[23] L. Liu, C. F. Li, Y. M. Lei *et al.*, "A new fuzzy clustering method with neighborhood distance constraint for volcanic ash cloud," *IEEE Access*, vol. 4, pp. 7005-7013, 2016.

[24] M. A. Ashraf, H. Jamal, S. A. Khan *et al*, "A heterogeneous service-oriented deep packet inspection and analysis framework for traffic-aware network management and security systems," *IEEE Access*, vol. 4, pp. 5918-5936, 2016.

[25] X. L. Yang, D. Lo, X. Xia, Z. Y. Wan, and J. L. Sun, “What security questions do developers ask? A large-scale study of stack overflow posts,” *JCST*, vol. 31, no. 5, pp. 910-924, 2016.
[26] C. Rosen and E. Shihab, “What are mobile developers asking about? A large scale study using stack overflow,” *EMSE*, vol. 21, no. 3, pp. 1192-1223, 2016.
[27] S. Beyer and M. Pinzger, “A manual categorization of android app development issues on stack overflow,” in *Proc. ICSME*, Victoria, British Columbia, Canada, 2014, pp. 531-535.
[28] K. Bajaj, K. Pattabiraman, and A. Mesbah, “Mining questions asked by web developers,” in *Proc. MSR*, Hyderabad, India, 2014, pp.112-121.
[29] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, “How do API changes trigger stack overflow discussions? A study on the android SDK,” in *Proc. ICPC*, Hyderabad, India, 2014, pp. 83-94.
[30] Q. Gao, H. S. Zhang, J. Wang, Y. F. Xiong, L. Zhang, and H. Mei, “Fixing recurring crash bugs via analyzing Q&A sites,” in *Proc. ASE*, Nebraska, USA, 2015, pp. 307-318.
[31] H. Jiang, J. X. Zhang, X. C. Li, Z. L. Ren, and D. Lo, “A more accurate model for finding tutorial segments explaining APIs,” in *Proc. SANER*, Osaka, Japan, 2016, pp.157-167.
[32] C. Treude and M. P. Robillard, “Augmenting API documentation with insights from Stack Overflow,” in *Proc. ICSE*, Austin, USA, 2016, pp. 392-403.
[33] E. Wong, J. Q. Yang, and L. Tan, “AutoComment: Mining question and answer sites for automatic comment generation,” in *Proc. ASE*, Silicon Valley, USA, 2013, pp. 562-567.
[34] M. P. Robillard, “What makes APIs hard to learn? Answers from developers,” *IEEE Software*, vol. 26, no. 6, pp. 27-34, 2009, DOI. 10.1109/MS.2009.193.
[35] M. P. Robillard and R. DeLine, “A field study of API learning obstacles,” *EMSE*, vol. 16, no. 6, pp. 703-732, 2012, DOI. 10.1007/s10664-010-9150-8.
[36] J. Zhou and R. J. Walker, “API deprecation: A retrospective analysis and detection method for code examples on the web,” in *Proc. FSE*, Seattle, WA, USA, 2016, pp. 266-277.
[37] R. Robbes, M. Lungu, and D. Rothlisberger, “How do developers react to API deprecation: The case of a smalltalk ecosystem,” in *Proc. FSE*, Washington DC, USA, 2012, Article No. 56.
[38] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, “API change and fault proneness: A threat to the success of android apps,” in *Proc. ESEC/FSE*, Saint Petersburg, Russia, 2013, pp. 477-487.
[39] T. McDonnell, B. Ray, and M. Kim, “An empirical study of API stability and adoption in the android ecosystem,” in *Proc. ICSE*, Eindhoven, The Netherlands, 2013, pp, 70-79.