2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

Recommending Comprehensive Solutions for Programming Tasks by Mining Crowd Knowledge

Rodrigo F. G. Silva*, Chanchal K. Roy†, Mohammad Masudur Rahman†,
Kevin A. Schneider†, Klerisson Paixao*, Marcelo de Almeida Maia*
*Federal University of Uberlândia, Uberlândia (MG), Brazil
{rodrigofernandes, klerisson, marcelo.maia}@ufu.br
†University of Saskatchewan, Saskatoon, Canada
{chanchal.roy, masud.rahman, kevin.schneider}@usask.ca

***Abstract*—Developers often search for relevant code examples on the web for their programming tasks. Unfortunately, they face two major problems. First, the search is impaired due to a lexical gap between their query (task description) and the information associated with the solution. Second, the retrieved solution may not be comprehensive, i.e., the code segment might miss a succinct explanation. These problems make the developers browse dozens of documents in order to synthesize an appropriate solution. To address these two problems, we propose CROKAGE (Crowd Knowledge Answer Generator), a tool that takes the description of a programming task (the query) and provides a comprehensive solution for the task. Our solutions contain not only relevant code examples but also their succinct explanations. Our proposed approach expands the task description with relevant API classes from Stack Overflow Q&A threads and then mitigates the lexical gap problems. Furthermore, we perform natural language processing on the top quality answers and then return such programming solutions containing code examples and code explanations unlike earlier studies. We evaluate our approach using 97 programming queries, of which 50% was used for training and 50% was used for testing, and show that it outperforms six baselines including the state-of-art by a statistically significant margin. Furthermore, our evaluation with 29 developers using 24 tasks (queries) confirms the superiority of CROKAGE over the state-of-art tool in terms of relevance of the suggested code examples, benefit of the code explanations and the overall solution quality (code + explanation).**

***Index Terms*—Mining Crowd Knowledge, Stack Overflow, Word Embedding**

I. Introduction

Software developers often search for relevant code examples on the web to implement their programming tasks. Although there exist several Internet-scale code search engines (e.g., Koders, Krugle, GitHub), finding code examples on the web is still a major challenge. Developers generally choose a few important keywords to describe their programming task, and then submit their query to a code search engine (e.g., Koders). Unfortunately, they face two major problems. First, the search is impaired due to a lexical gap between the task description (the query) and the information pertinent to the solution. Their query often does not contain the API references required for the task. Second, the retrieved solution might always not be comprehensive. The retrieved code segment might miss a succinct explanation [1] or the textual solution might miss a required code segment [2]. These problems make the developers browse dozens of search results in order to synthesize an appropriate solution for their task.

Traditional Information Retrieval (IR)-based code search engines generally do not work well with natural language queries due to a lexical mismatch between the keywords of a query and the available code examples on the web. Mikolov et al. [3] recently employ word embedding technology that captures words' semantics, represents each word using a high-dimensional vector and then estimates the semantic similarity between any two documents despite their lexical dissimilarity. This technique was later used by two recent studies: AnswerBot [2] and BIKER [4]. They also attempt to address the two aforementioned problems faced by developers. However, these studies are limited in several aspects. AnswerBot's answers do not contain any source code examples. Thus, they are not sufficient enough for implementing a *how-to* programming task. On the other hand, BIKER is able to provide answers containing both explanations and source code. However, BIKER is only able to provide the explanations from official Java SE API documentations. Thus, their explanations are restricted to a limited set of APIs only.

In this paper, we propose an approach namely CROKAGE (**Cro**wd **K**nowledge **A**nswer **Ge**nerator) that takes a task description in natural language (the query) and then returns relevant, comprehensive programming solutions containing both code examples and succinct explanations. In particular, we address the limitations of the two earlier approaches [2, 4]. First, unlike AnswerBot [2] (i.e., provides only answer summary texts), we deliver both relevant code segments and their corresponding explanations. Second, we provide succinct code explanations written by human developers whereas BIKER [4] returns only generic explanations extracted from official API documentations. CROKAGE first employs word embeddings [3] to overcome the lexical gap between the query and the code example and then expands the task description (the query) with relevant API classes from Stack Overflow. Then it constructs a multi-factor retrieval mechanism to fetch from Stack Overflow the most relevant code examples to the target programming task. Furthermore, CROKAGE uses natural language processing to compose a succinct explanation for each of the suggested code examples.

We evaluate our approach in two different ways. First, we compare our performance in code example suggestion with six baselines including BIKER [4], TF-IDF [5] and BM25 [6]. For this, we use four different metrics and show that our

978-1-7281-1519-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPC.2019.00054

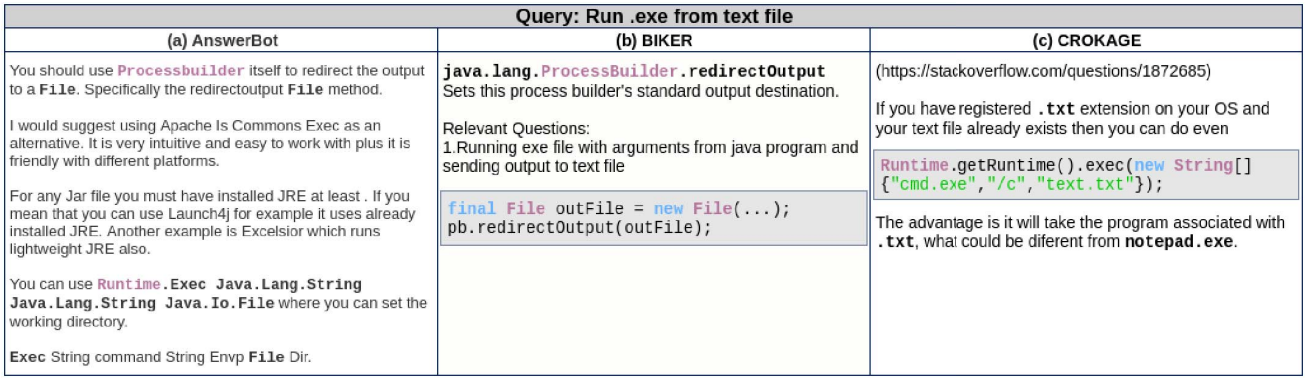


Fig. 1. Programming Solutions from (a) AnswerBot, (b) BIKER, and (c) CROKAGE

approach outperforms all baselines by a statistically significant margin. We construct our ground truth by manually analyzing 6,224 answers from Stack Overflow against 100 questions (or queries) collected from three popular tutorial sites (KodeJava, JavaDB, Java2s). We select 97 questions containing API classes in their answers, of which 50% was used for training and 50% was used for testing. Our experiments show that CROKAGE provides relevant programming solutions (code segments + explanations) with 79% Top-10 Accuracy, 40% precision, 19% recall, and a reciprocal rank of 0.46 which are 64%, 30%, 18%, and 36% higher respectively than those of the state-of-art, BIKER [4]. Second, we conduct a user study with 29 developers using 24 programming tasks. Our findings suggest that solutions from CROKAGE are more effective than the ones of BIKER [4] in terms of relevance of the suggested code examples, benefit of the code explanations and the overall solution quality (code + explanation).

Thus, the main contributions of this paper are as follows:

- A novel approach that suggests programming solutions containing both code and explanations against tasks written in natural language texts by harnessing the crowd knowledge stored in Stack Overflow.
- We perform an empirical evaluation on the suggestion of relevant code examples using 97 programming tasks containing API classes in their answers and a comparison with the state-of-the-art study [4]. Our approach achieves significant improvements over six baselines and outperforms the state-of-art in retrieving relevant and comprehensive programming solutions.
- A ground truth and benchmark dataset of 6,224 answers against 100 Java tutorial questions constructed by two professional developers after spending 87 man hours.
- A replication package[1] containing CROKAGE's prototype, detailed results of our user study and our used dataset for replication or third party reuse.

The rest of this paper is structured as follows. Section II shows the motivation of our work and compare our approach with two state-of-art tools. Section III describes the technical details of CROKAGE. Section IV reports the experimental methods and the obtained results. Section V discusses the threats to validity of our experiments. Section VI reviews the related works. Finally, Section VII concludes the paper.

[1]https://github.com/muldon/CROKAGE-replication-package

II. Motivating Examples

Let us consider a use-case scenario where a developer is looking for a solution to the query: *"run .exe from text file"*. Figure 1 presents three solutions from three different approaches: AnswerBot [2], BIKER [4] and CROKAGE respectively. The solution proposed by AnswerBot (Fig. 1-(a)) contains sentences describing the use of several API classes as well as opinions from Stack Overflow users. Despite describing solutions using relevant APIs (e.g., `Processbuilder`, `Runtime`), no actual code is provided. That is, the solution is half-baked and thus might not help the developer properly.

On the contrary, BIKER [4] provides both code example and corresponding explanation (i.e., Fig. 1-(b)). Unfortunately we notice two major problems. First, the suggested code does not completely match with the intent of the query. Second, the explanation is limited to only official Java API documentation and thus might fail to explain the functionalities of other external API classes or methods.

Finally, our approach CROKAGE provides a solution containing (1) a code segment using `Runtime` API and (2) an associated prose explaining the code (i.e., Fig. 1-(c)). Unlike AnswerBot [2], CROKAGE delivers a solution containing relevant code segment. Unlike BIKER [4] (i.e., generates explanation from official API documentation only) CROKAGE delivers a solution containing code segment which is carefully explained and curated by Stack Overflow users. It also should be noted that unlike BIKER, our explanations are much more generic, informative and not restricted to standard Java APIs. Thus, our solution has a much more potential than the existing alternatives (i.e., AnswerBot [2] and BIKER [4]) as also confirmed by the user study (Section IV-D).

III. Proposed Approach

Figure 2 shows the schematic diagram of our proposed approach CROKAGE. It has four different stages. We first prepare the corpus using Q&A threads from Stack Overflow (Section III-A), and then construct several models (e.g., *FastText* model) and indices (Section III-B). Then these models and indices are employed to retrieve the relevant answers from

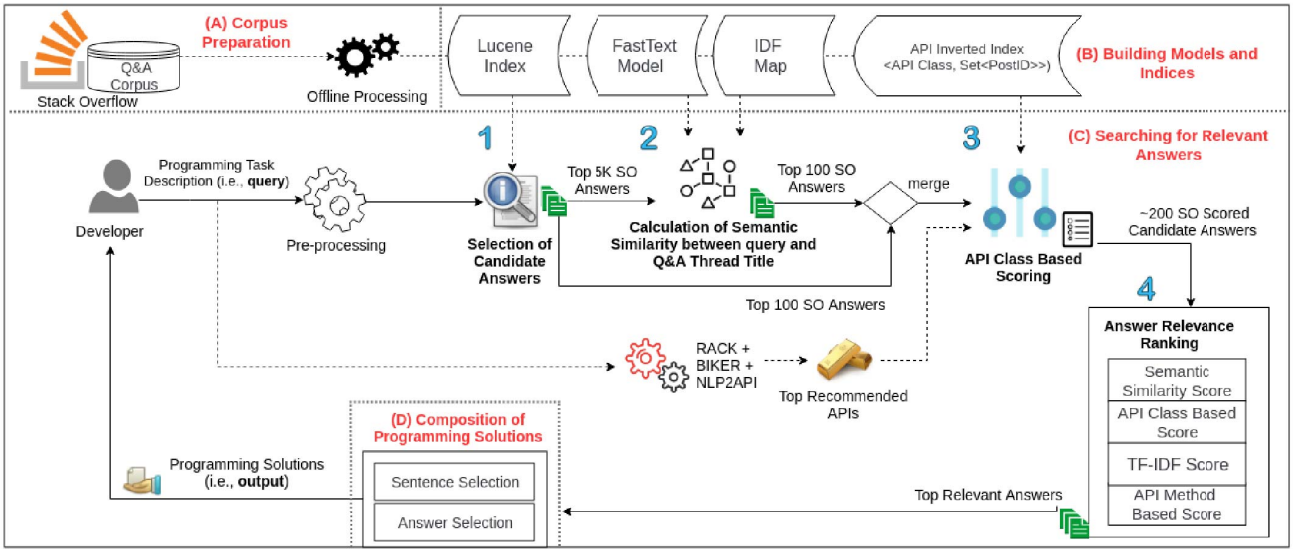


Fig. 2. Schematic diagram of CROKAGE - (A) Corpus Preparation, (B) Building Models and Indices, (C) Searching for Relevant Answers, and (D) Composition of Programming Solutions

the corpus against a programming task description (Section III-C). Finally, the top quality answers are used to compose and suggest the programming solutions for the task (Section III-D). We discuss each of these stages as follows:

A. Corpus Preparation

In order to deliver appropriate solutions from Stack Overflow against a given task description (i.e., query), we need to construct domain specific knowledge base. We collect a total of 3,889,303 questions and answers related to "Java" from Stack Overflow Q&A site[2]. We use *Jsoup* [7] to parse these questions and answers and then separate texts and code using `<code>` and `<pre>` tags. We remove all punctuation symbols, stop words[3], small words (i.e., size lower than two) and numbers from them. We save these processed versions of Q&A threads alongside the original versions which are later used to compose programming solutions (Section III-D).

B. Building Models and Indices

Construction of Lucene Index: We first construct a Lucene index by considering all Java related questions and answers from Stack Overflow Fig. 2-(B)). We make sure that each answer from each Q&A thread contains one or more code segments. Similar to Delfim et al. [8], we capture the pre-processed version of each question-answer pair as a document within a large document corpus. We then build an index with Lucene [9] using all these documents. This index is later used to retrieve the most relevant answers against each query (i.e., task description).

Construction of FastText Model: Task description (the query) from a developer might always not contain the required API references. Hence, the Lucene-based approach above might fail to retrieve the relevant answers due to lexical gap issue. In order to overcome the lexical gap issue between the query and answers from Stack Overflow, we employ a word embedding model namely *FastText* [10]. In this model, each word is represented as a high dimensional vector and similar words have similar vector representations. We learn the vector representation of each word of our vocabulary using *FastText*. We first combine the pre-processed contents of *title, body texts* and *code segments* from each of the Q&A threads into a file and then employ *FastText* to learn the vector representation of each word. We customize these parameters: vector size=100, epoch = 10, minimum size = 2 and maximal size = 5 whereas the other parameters remain default. Please note that we do not perform stemming on the texts since *FastText* is able to look for subwords. Besides, the impact of stemming over source code is found to be controversial [11]. Once the model is built (i.e., Fig. 2-(B)), we use it as a dictionary for mapping between the words of our vocabulary and their respective vector representations.

Construction of IDF Map: Inverse Document Frequency (IDF) has been often used for determining the importance of a term within a corpus [4, 12]. IDF represents the inverse of the number of documents containing the word. That is, more frequent words across the corpus carry less important information than infrequent words. Our vocabulary contains a total of 1,118,667 distinct words. We calculate the IDF of each word and build an IDF Map (i.e., Fig. 2-(B)) that points each word to its corresponding IDF value. The IDF value of each word is later used as a weight during the calculation of embedding similarity (a.k.a., semantic similarity).

Construction of API Inverted Index: We also build an API inverted index that maps API classes to their corresponding answers from Stack Overflow (i.e., Fig. 2-(B)). To build this index, we first select all the answers containing code (i.e., containing tags `<pre><code>`), and extract code elements from them using *Jsoup* [7]. Then we identify the API classes from

[2]https://archive.org/details/stackexchange - dump published in June 2018
[3]https://bit.ly/1Nt4eMh

them using appropriate regular expressions. We then build an inverted index where each API class is associated with the IDs of all answers containing that class. We notice that many classes have a low frequency (i.e., lower than five), which originally come from dummy examples submitted by users to explain a very specific scenario. (e.g., *"You can use @Qualifier on the* `OptionalBean` *member"*[4] where `OptionalBean` is a class created by the user to illustrate a scenario). Thus we discard the API classes with low frequencies from our API inverted index. We believe that such API classes could be less appropriate for our problem contexts.

C. Searching for Relevant Answers

Once the models and indices are built, we use them in searching for relevant answers for a given query. Our search component works in four stages as shown in Figure 2. In order to search for relevant answers, CROKAGE first loads the models and indices (Section III-B) and then pre-processes the task description (i.e., query) with standard natural language pre-processing. Then CROKAGE navigates through the four stages as follows.

1) Selection of Candidate Answers: Given a pre-processed query and our Lucene index constructed above, CROKAGE uses BM25 [6] function to determine the lexical relevance of each answer from the pre-loaded index as follows:

$$lexicalSim(A,Q) = \sum_{i=1}^{n} idf(q_i) * \frac{f(q_i,A)*(k+1)}{f(q_i,A)+k*(1-b+b*\frac{|A|}{avgdl})} \quad (1)$$

where $|A|$ is the length of the answer A in words, $f(q_i, A)$ is keyword q_i's term frequency in answer A and $avgdl$ is the average document length in the index. k and b are two parameters where k controls non-linear term frequency normalization (saturation), and b controls to what degree document length normalizes term frequency values. CROKAGE uses the same values as used by previous works in Software Engineering [13, 14] with best performance. $idf(q_i)$ is the inverse document frequency of keyword q_i and computed as follows:

$$idf(q_i) = log\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \quad (2)$$

where $n(q_i)$ is the number of documents containing keyword q_i and N is the total number of documents in the index (or corpus). CROKAGE retrieves the top answers sorted by their relevance to the query and save them into two sets: *smallSet* and *bigSet*. We add the top 100 answers into the *smallSet* and the top 5K answers into the *bigSet*. Both sets are used in the next stages.

2) Calculation of Semantic Similarity between Query and Q&A Thread Title: This stage takes as input a pre-processed query, the *bigSet* from the previous stage, the *FastText* model and the IDF Map. CROKAGE first transforms the pre-processed information from each Q&A thread (*body texts* + *title*) and the query into two bag of words namely A and Q respectively. Then CROKAGE computes the asymmetric relevance between A and Q as follows:

$$asym(A \rightarrow Q) = \frac{\sum_{w \in A} sim(w,Q) * idf(w)}{\sum_{w \in A} idf(w)} \quad (3)$$

where $idf(w)$ is the correspondent IDF value of the word w, $sim(w, Q)$ is the maximum value of $sim(w, w_Q)$ for every word $w_Q \in Q$, and $sim(w, w_Q)$ is the cosine similarity between w and w_Q embedding vectors. The other asymmetric relevance namely $asym(Q \rightarrow A)$ can be calculated by swapping A and Q in Equation 3. Thus, the final similarity between the query Q and the answer A is the harmonic mean of the two asymmetric relevance scores as follows:

$$semScore(A,Q) = \frac{2 * asym(A \rightarrow Q) * asym(Q \rightarrow A))}{asym(A \rightarrow Q) + asym(Q \rightarrow A)} \quad (4)$$

CROKAGE computes semantic relevance between the query and the top 5K answers from the *bigSet* and then selects the top 100 relevant answers which are used for the later stages.

3) API Class Based Scoring: Although the answers from the two previous stages are lexically and semantically relevant to a given query, they might still contain noise. Hence, relevant answers need to be promoted over the noise. One possible way is rewarding the answers based on to their APIs. First, we employ three state-of-art API recommendation systems – BIKER [4], NLP2API [1] and RACK [15] and then collect the most relevant API classes for a given query (i.e., task description). Our findings suggest that the combination of three tools provides the best results which justifies our choice for combining their API suggestions (Section IV-C). We then combine the top 100 answers from the *smallSet* and another top 100 answers from the above stage removing duplicate answers. From each answer, we extract the API classes using appropriate regular expressions and store them in a set called *allApis*, after removing the duplicate classes. Next, for each recommended API class ($c \in C$) from the ranked list (RACK+NLP2API+BIKER), we calculate the API class based score for each answer as follows:

$$apiScore(A) = \sum_{c \in C} \frac{1}{pos(c) + n} \quad (5)$$

where n is a smoothing factor and pos is the position (starting with zero) of the class c within recommended API ranked list that is also found in *allApis*. After careful investigation, we found the best value of n as two. Our goal is to reward answers with more relevant classes and penalize the answers from Stack Overflow with irrelevant classes.

4) Answer Relevance Ranking: The previous stages deliver a set of around 200 answers from Stack Overflow along with their lexical, semantic and API relevance with a given query (i.e., task description). However, we consider two additional relevance factors – TF-IDF Score and API Method Based Score (Fig. 2-(C-4)) as follows:

TF-IDF Score: Although we use BM25, a lexical similarity method for selecting the candidate answers, we employ

[4]https://stackoverflow.com/questions/9416541

another lexical similarity method namely TF-IDF (Fig. 2-(C-4)) to determine their relevance against a given query. TF-IDF stands for term frequency (TF) times inverse document frequency (IDF). It can be calculated for each word of a document (query or answer) as follows:

$$TF - IDF(W) = TF(W) * log_{10}(\frac{N}{df_w}) \tag{6}$$

We determine lexical similarity between the document representing the query and the document representing each answer using their cosine similarity as follows:

$$\begin{aligned} tfidfScore(A, Q) =& \frac{d_Q \cdot d_A}{|d_Q| \cdot |d_A|} \\ =& \frac{\sum_1^N tfidf_{ti,dq} \cdot tfidf_{ti,da}}{\sqrt{\sum_1^N tfidf_{ti,dq}^2} \cdot \sqrt{tfidf_{ti,da}^2}} \end{aligned} \tag{7}$$

where d_Q refers to the query, d_A represents the answer and $tfidf_{ti,dk}$ is the term weight for each word of the document (query or answer).

API Method Based Score: Huang et al. [4] suggest that if an API method occurs across multiple candidate answers for a given query, it is more likely relevant to the query. CROKAGE rewards the answers containing the most relevant API methods among the candidate answers. For this, CROKAGE selects the methods used in each answer using appropriate regular expressions and identify the most frequent API method. Then CROKAGE assigns each of the answers an API method based score (Fig. 2-(C-4)) as follows:

$$methodScore(A) = \frac{log_2(freq_m)}{10} \tag{8}$$

where $freq_m$ is the top method frequency. If the answer does not contain the top method, the score is set to zero.

After calculating the four scores — *semScore, apiScore, tfidfScore* and *methodScore*, we normalize them and combine them in a final score (*factorsScore*) representing the relevance of each answer A to the query Q:

$$\begin{aligned} factorsScore(Q, A) = & \, semScore \cdot semWeight \\ & + apiScore \cdot apiWeight \\ & + tfidfScore \cdot tfidfWeight \\ & + methodScore \cdot methodWeight \end{aligned} \tag{9}$$

where *semWeight*, *apiWeight*, *tfidfWeight*, and *methodWeight* are relative weights for each factor. We conduct controlled iterative experiments and employ a set of weights that return the best Top-K Accuracy and MRR (Section IV-C). Once the final score is calculated, we collect the Top-K Stack Overflow answers for the solution composition.

D. Composition of Programming Solutions

After collecting the most relevant answers for a given query (i.e., task description), CROKAGE uses them to compose appropriate solutions to the desired task, discarding answers without important sentences. CROKAGE adopts two patterns, previously used by Wong et al. [16], to identify important sentences based on their POS structure:

$$\begin{aligned} & VP << (NP < /NN.?/) < /VB.?/ \\ & NP! < PRP[<< VP | \$VP] \end{aligned} \tag{10}$$

These patterns ensure that each sentence has a verb which is associated with a subject or an object. The first pattern guarantees that a verb phrase is followed by a noun phrase while the second pattern guarantees that a noun phrase is followed by a verb phrase. They also ensure that a verb phrase is not a personal pronoun. CROKAGE filters important sentences using pseudocode as shown in Algorithm 1.

Algorithm 1: Pseudocode to filter important sentences from answers

```
1  filterSentences(query, answer, pattern1, pattern2)
2      removedSentences
3      stfCoreNLP            /* Stanford Core lib */
4      selectedSentences ← answer.getBody()
5      procBody ← preProcess(selectedSentences)
6      sentences ← stfCoreNLP.getSentences(procBody)
7      for each sentence in sentences do
8          parseTree ← sentence.constituencyParse()
9          matcher1 ← pattern1.matcher(parseTree)
10         matcher2 ← pattern2.matcher(parseTree)
11         if (not(matcher1) & not(matcher2)) then
12             if (not(specialSentence(sentence)) then
13                 selectedSentences ←
                     selectedSentences.replace(sentence, ")
14             end if
15         end if
16     end for
17     return selectedSentences
18 end
```

The algorithm receives four parameters: the pre-processed query, one recommended answer and the patterns. CROKAGE first performs standard natural language pre-processing on the body texts of the answer (line 5). Next, the algorithm annotates these processed texts using Stanford Part-Of-Speech Tagger (POS Tagger) [17] where each word of the sentence is assigned a POS tag (line 6). The algorithm then iterates over the sentences (lines 7 to 16). For each sentence, it builds the parse tree (line 8) and generates two pattern matchers to obtain the nodes that satisfy *pattern1* and *pattern2* (lines 9 and 10). If none of the two patterns are satisfied (line 11 to 15), the algorithm checks whether the sentence belongs to any special conditions (line 12). This special condition is composed of 4 heuristics. That is, the sentence does not contain: numbers, camel case words, important words (i.e., "insert", "replace", "update")[5] and shared words with the query. If the sentence does not satisfy to any of these conditions, the algorithm removes the sentence from the selected sentences list (line 13). After iterating over all sentences, the algorithm returns only the valid sentences (*selectedSentences* at line 17). If this list is not empty, it means that the answer contains valid sentences, which are returned as the explanation of associated code segment.

Our intuition is that removing unnecessary sentences may help developers with a more concise explanation. CROKAGE

[5]the complete list of words is available at: https://bit.ly/2UxYgzc

Fig. 3. Comprehensive solution generated by CROKAGE for the query: *"Convert Between a File Path and a URL"*

is able to remove irrelevant sentences from the answers (e.g., *"Try this:", "You could do it like this:", "It will work for sure", "It seems the easiest to me"* or *"Yes, like doing this"*). The output is a solution containing code and explanations. Figure 3 shows Top-3 solutions comprising a comprehensive solution for the task (i.e., query): *"Convert Between a File Path and a URL"*.

IV. Experiments and Results

We evaluate our approach in different dimensions. First we evaluate the search for relevant answers (Section IV-C) using 97 queries of our manually evaluated ground truth, of which 50% was used for training and 50% was used for testing. For the evaluation, we use four classical evaluation metrics and contrast the results with the six considered baselines, including the state-of-art BIKER [4]. Second, we perform a user study (Section IV-D) with 29 developers using 24 queries to evaluate CROKAGE and BIKER in terms of relevance of the suggested code examples, benefit of the code explanations and the overall solution quality (code + explanation). In particular, we answer to three research questions using our experiments as follows:

RQ1: Can CROKAGE outperform existing baseline methods in retrieving relevant solutions for given programming tasks written as natural language queries?

RQ2: To what extent do the factors individually influence the ranking of candidate answers?

RQ3: Can CROKAGE provide more comprehensive solutions containing code and explanations for given queries (task descriptions) compared to those of the state-of-art, BIKER?

A. Ground Truth Generation

To build our ground truth, we first select 100 programming tasks (i.e., queries) from three Java tutorial sites: Java2s [18], BeginnersBook [19] and KodeJava [20]. We select these queries in such a way so that they cover different API tasks and use these queries as input to three search engines: Google [21], Bing [22] and Stack Overflow search [23]. We pre-process each query by removing stop words, punctuation symbols, numbers and small words (length smaller then 2). For Google and Bing, we augment the query with the word *"java"* (if the query does not contain it) and collect only such results that are from Stack Overflow. For Stack Overflow search, we filter results using the tag *"java"*. We collect the first 10 results from Google and the first 20 from Bing. We observe lower efficiency of Stack Overflow search mechanism regarding the relevance of results when compared to Google and Bing. We thus establish a more rigorous criteria to fetch results from Stack Overflow search by setting a threshold of a minimum of 100 visualizations.

For each of the 100 queries, we merge results from the three search engines and remove duplicates. Results from these engines point to Stack Overflow threads. Each Stack Overflow thread is composed of a question and its answers. Since we are interested about the relevant answers to our query, we iterate over the questions, discard the question with no answers and select only answers with at least 1 upvote and containing source code. This automatic process results into 6,224 answers for 100 queries. Then, two professional developers manually evaluate the 6,224 answers by rating each answer in Likert scale from 1 to 5 according to the following criteria:

1 = Unrelated: the answer is not related to the query.

2 = Weakly related: the answer does not address the query problem objectively.

3 = Related: the answer needs considerable amount of changes in the source code to address the query problem, or is too long, or is too complex.

4 = Understandable: the answer addresses the query problem after feasible amount of changes in the source code.

5 = Straightforward: the answer addresses the query problem after few or no changes in the source code.

Two professional developers first evaluate the answers independently. After the evaluations, the average rating is calculated. If two ratings differ more than 1 Likert and at least one of them is higher than 3, this answer is marked to be re-evaluated by both in an agreement phase. The two professional developers then discuss these conflits. If after discussing, two ratings still differ in more than 1 Likert, this answer is discarded. We then re-calculate the average rating for the marked answers. We measure kappa before and after the agreement phase and we obtain the following values respectively: 0.3149 and 0.5063 (p-value < 0.05). That is, our agreement improves from fair to moderate[24].

TABLE I
CROKAGE PARAMETERS AND THEIR DESCRIPTIONS, RANGES, VARIATIONS AND THE HIGHEST VALUES FOR HIT@10 AND MRR@10.

Parameter	Description	Range	Variation	Best Value
BM25Limit1	Top scored answers in BM25 to be used in semantic similarity relevance mechanism	[1k,10k]	100	5k
BM25Limit2	Top scored answers in BM25 to be merged with the output of the semantic similarity relevance mechanism	[10,200]	10	100
topAsymRelevanceNum	Top scored answers by the semantic relevance mechanism	[10,200]	10	100
numberOfAPIClasses	Number of classes extracted from the three API Recommendation Systems combined	[5,30]	5	20
semWeight	Weight associated with the semantic relevance score (semScore)	[0,1]	0.25	1.00
apiWeight	Weight associated with the api score (apiScore)	[0,1]	0.25	0.25
tfidfWeight	Weight associated with TF-IDF score (tfidfScore)	[0,1]	0.25	0.50
methodWeight	Weight associated with method score (methodScore)	[0,1]	0.25	0.75

We consider an answer as relevant if its average Likert is equal or higher than 4. Since our approach relies on solutions containing API classes, we discard queries whose relevant answers do not contain API classes in their answers. Following this criteria, we discard three queries out of the 100, resulting in 97 queries along with their 1588 relevant answers.

B. Performance Metrics

We choose four performance metrics commonly adopted by related literature [1, 2, 4, 15, 25]. The four metrics are described as follows:

Top-K Accuracy (Hit@K): the percentage of search queries of which at least one recommended answer is relevant within the top-K results.

Mean Reciprocal Rank (MRR@K): the multiplicative inverse of the rank of the first relevant answer recommended within the top-K results.

Mean Average Precision (MAP@K): the average of all *Precision@K* for a set of queries. *Precision@K* is the precision of the all relevant answers within the first top-K recommnedations for every query.

Mean Recall (MR@K): the average of all *Recall@K* for a set of queries. *Recall@K* is the percentage of relevant answers recommended within the top-K results.

C. Experimental Results for the Retrieval of Relevant Answers

RQ1: *Can CROKAGE outperform existing baseline methods in retrieving relevant solutions for given programming tasks written as natural language queries?*

We load the queries from our ground truth (i.e., 97 queries after discarding three), as well as their relevant answers namely *goldSet* (i.e., average Likert equals or higher than 4). We split the queries into two sets namely training and test containing 49 and 48 queries respectively, along with their *goldSets*. We use the training set and its *goldSet* to calibrate the weights for each parameter of CROKAGE for which the Hit@K and MRR@K are the highest respectively. Table I shows how we varied the parameters and the best values found for them. After discovering the best values for the parameters, we calibrate the weights and use the test set and its *goldSet* to test CROKAGE and all the other baselines. To test CROKAGE, we run CROKAGE to search for relevant answers for each query of the test set (Section III-C). We extend CROKAGE to generate the baselines as follows:

BIKER: BIKER extracts snippets from Stack Overflow answers to compose solutions. We extend the tool to show the answers IDs of which the snippets are extracted without altering its behaviour.

BM25: we set CROKAGE to return only the top scored answers contained in the *smallSet* (i.e., top *lexicalSim*, Section III-C1, Fig. 2-(C-1)).

Semantic Relevance, API Class Relevance, TF-IDF Relevance and API Method Relevance: we build four baselines representing CROKAGE relevance factors (Section III-C4, Fig. 2-(C-4). For this, we preserve the weight associated to the baseline and set the other three weights (Formula 9) to zero (e.g., to build *Semantic Relevance* baseline we set all factors' weights to zero, except *semWeight*).

After building all baselines, we run CROKAGE to search for relevant answers (Section III-C) for each baseline against our test set. To evaluate each baseline, we compare their recommended answers against the *goldSet* and collect the metrics Hit@K, MRR@K, MAP@K, and MR@K, for K=10 (i.e., we consider the top 10 recommendations). Table II shows the metrics for all the baselines, including the state-of-art BIKER [4]. All the experiments were conducted over a server equipped with Intel® Xeon® at 3.1 GHz on 32 GB RAM, four cores, and 64-bit Linux Mint Cinnamon operating system. The total time to run the approach is around 188 seconds, of which 121 seconds are spend to load all models and indices and 67 seconds are spend to process the test set. That is, after loading the models, our approach takes less than 1.5 seconds to process each query.

The non-parametric Wilcoxon signed-rank test on paired data showed significant difference between CROKAGE and BIKER for all considered metrics (i.e., p-values < 0.05), with large effect size calculated with $r = Z/\sqrt{n}$ [26] ranging from 0.61 to 0.82. Compared to BM25 [6], this variation is lower: 17%, 14%, 8% and 17% for Hit@K, MAP@K, MR@K and MRR@K respectively (in absolute values), with a small-medium effect size (i.e., ranging from 0.25 to 0.35).

In terms of Top-K Accuracy, Mean Average Precision, Mean Recall, and Mean Reciprocal Rank for K=10, CROKAGE shows the highest values compared to all baselines, whereas BIKER [4] shows the lowest. For these metrics, CROKAGE significantly outperforms BIKER by 64%, 30%, 18%, and 36% respectively (in absolute values) to retrieve relevant answers for given programming tasks written in natural language (i.e., query).

TABLE II
PERFORMANCE OF CROKAGE AND OTHER BASELINE METHODS IN TERMS OF HIT@K, MRR@K, MAP@K, AND MR@K, FOR K=10

Approach	Hit	MRR	MAP	MR
BIKER	0.15	0.10	0.10	0.01
API Class Relevance	0.38	0.11	0.11	0.06
API Method Relevance	0.46	0.17	0.15	0.06
Semantic Relevance	0.50	0.25	0.22	0.07
TF-IDF	0.58	0.25	0.23	0.11
BM25	0.62	0.29	0.26	0.11
CROKAGE	0.79	0.46	0.40	0.19

RQ2: *To what extent do the factors individually influence the ranking of candidate answers?*

CROKAGE obtains around 200 Stack Overflow candidate answers (i.e., duplicates are removed) to rank in the last stage of the search for relevant answers (Section III-C4). We investigate the individual influence of each of the four factors on candidate answers ranking in terms of Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, for K=10 (i.e., considering top 10 recommendations). For this, we extend CROKAGE by setting the weight associated to each factor to zero (Formula 9), while keeping the others as follows:

CROKsemWeight0: we set *semWeight* to zero, representing the semantic relevance influence.

CROKapiWeight0: we set *apiWeight* to zero, representing the API Class relevance influence.

CROKtfidfWeight0: we set *tfidfWeight* to zero, representing TF-IDF relevance influence.

CROKmethodWeight0: we set *methodWeight* to zero, representing API Method relevance influence.

We run each version against our ground truth queries to search for relevant answers (Section III-C) and collect the metrics as shown in Table III.

Our findings suggest that, in the presence of the other factors associated with their calibrated weights, TF-IDF shows the highest influence on the ranking of candidate answers. Compared to CROKAGE, the version with TF-IDF weight (tfidfWeight) set to zero shows values for metrics 8%, 15%, 13%, and 8% lower for Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision, and Mean Recall respectively. This difference is also significant for API Method (i.e., 14% in Top-K Accuracy). Semantic and API Class relevance instead, show small influence on the ranking of candidate answers (i.e., 8% and 4% in Mean Reciprocal Rank respectively).

In order to collect the API classes for a given query (i.e., task description), we use three state-of-art API recommendation systems – BIKER [4], NLP2API [1] and RACK [15]. We investigate their combination to provide API classes in such a way to obtain the highest Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, respectively. We tested 308 queries using the dataset and ground truth of NLP2API in terms of the metrics. In general, the combination of the three tools performs better than two tools combined or each tool taken isolated for different values

TABLE III
PERFORMANCE OF FOUR EXTENDED VERSIONS OF CROKAGE IN TERMS OF HIT@K, MRR@K, MAP@K, AND MR@K, FOR K=10

Version	Hit	MRR	MAP	MR
CROKtfidfWeight0	0.71	0.31	0.27	0.11
CROKmethodWeight0	0.65	0.37	0.31	0.14
CROKsemWeight0	0.79	0.38	0.36	0.19
CROKapiWeight0	0.83	0.42	0.37	0.18

of K (e.g., 1, 5, 10). We employ the combination of RACK + BIKER + NLP2API to provide API classes to a given query considering the order of the recommendations. We tested different numbers of API classes (Table I) and found that collecting 20 classes from the three approaches combined gives the best performance. We show the metrics for each tool and for their combination with the highest performance for K=10 (i.e., top 10 recommendations) in Table IV.

TABLE IV
PERFORMANCE OF THREE API RECOMMENDATION SYSTEMS AND TWO DIFFERENT COMBINATIONS IN NLP2API DATASET IN TERMS OF HIT@K, MRR@K, MAP@K, AND MR@K, FOR K=10

Approach	Hit	MRR	MAP	MR
RACK (RA)	0.74	0.47	0.42	0.40
BIKER (BI)	0.52	0.37	0.35	0.23
NLP2API (NL)	0.73	0.49	0.44	0.38
RA+NL	0.80	0.50	0.44	0.46
RA+BI+NL	0.83	0.51	0.44	0.47

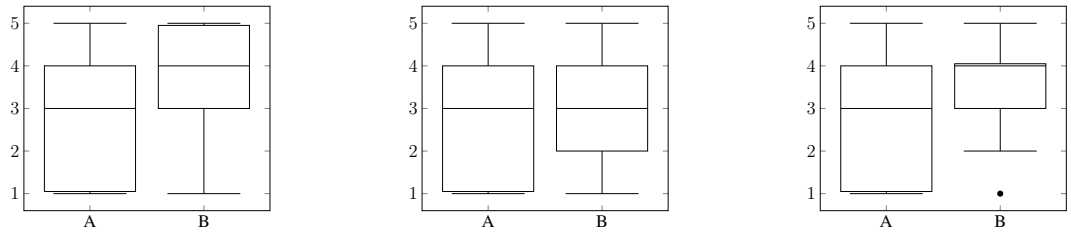
D. Comparison with State-of-the-art using Developer Study

RQ3: *Can CROKAGE provide more comprehensive solutions containing code and explanations for given queries (task descriptions) compared to those of the state-of-art, BIKER?*

To answer this question, we first choose 50 most popular questions from the same three tutorial sites used to generate our ground truth (discarding questions already used to build the ground truth). We augment the question with the filter "*site:stackoverflow.com*" and use Google to measure the popularity of each one. For this, we check the number of Google's results returned for them. We randomly select 30 among these questions and apply standard natural language pre-processing. We assume that developers would use CROKAGE like a search engine. Thus, we restrict the queries to not contain too many words (i.e., maximum of seven and a minimum of two tokens). This process results in 24 queries. We run BIKER [4] and CROKAGE to generate the solutions for these queries. We focus in providing solutions with high precision. Thus, we set both tools to use only the top 1 recommended answer to build solutions. We then ask developers to evaluate the tools in terms of three aspects:

1) The relevance of the suggested code examples
2) The benefit of the code explanations
3) The overall solution quality (code + explanation)

We built three questionnaires, each containing eight different questions and their solutions for both tools (identified as Tool A and Tool B). We also provided instructions for the evaluations. We asked participants to provide a value from 1 to 5 for each aspect in each tool considering the same criteria used to evaluate our ground truth (Section IV-A). We also asked the participants how many years of experience they have as Java programmers and as Professional Software



(a) Relevance of the Suggested Code Examples

(b) Benefit of the Code Explanations

(c) Overall Solution Quality

Fig. 4. Box plots of Relevance of the Suggested Code Examples (a), Benefit of the Code Explanations (b), and the Overall Solution Quality (c) performance (Likert scale) for tools A (BIKER) and B (CROKAGE). Lower and upper box boundaries 25th and 75th percentiles, respectively, line inside box median. Lower and upper error lines 10th and 90th percentiles, respectively. Filled circle data falls outside 10th and 90th percentiles.

developers. The averages are 7.92 and 8.78 years respectively. In total, 29 participants answered our study and each query was answered by at least nine participants. We assured that each questionnaire contains at least five participants with a minimum of six years of experience in Java programming.

Figure 4 represents the participants evaluations. In general, participants reported that Tool B (CROKAGE) was considered better than Tool A (BIKER [4]). For the three considered criteria, two of them were reported by users as much superior: relevance of the suggested code examples (Fig. 4-(a)) and the overall solution quality (Fig. 4-(c)). Both approaches showed the same median value (i.e,. 3) for the benefit of the code explanations (Fig. 4-(b)). However CROKAGE showed much less Likerts 1 (i.e., 19 against 72) and much more Likerts 5 (i.e., 52 against 20). These results suggest that developers prefer explanations provided by other users instead of generic explanations extracted from official API documentations.

Our findings show that CROKAGE outperforms BIKER[4] for the three considered criteria: relevance of the suggested code examples (Fig. 4-(a)), benefit of the code explanations (Fig. 4-(b)) and the overall solution quality (Fig. 4-(c)).

To make sure that the performance of CROKAGE and BIKER are statistically different, we run Wilcoxon signed-rank [27] for each adopted criteria. We find that the evaluation of CROKAGE is statistically better than the one of BIKER for the three criteria with a confidence level of 95% (p-value < 0.05) and a medium effect size ranging from 0.40 to 0.42, calculated with $r = Z/\sqrt{n}$ [26].

V. Threats to validity

Threats to internal validity are related to the baseline methods and the user study. One of the baselines is a state-of-art tool and we extend it to produce the solutions with supplementary information (i.e., answers IDs) without altering its behavior. We double checked the implementation of all baselines to assure they do not contain implementation errors. For the user study, the experience of each participant in Java programming and their effort in manual evaluation could affect the accuracy of the results. We mitigate this threat by organizing questionnaires where each questionnaire has at least five participants with a minimum of six years of experience in Java programming. We only selected participants who showed interest in participating in our research.

Threats to external validity relates to the quality of our ground truth and to the generalizability of our results. Concerning the ground truth, to mitigate the threat two professional developers independently evaluated each answer and then discussed the disagreements. Furthermore, we only selected good quality answers for the *goldSet* (i.e., Likert equal or higher than 4). We tried to cover a wide range of different API tasks by selecting programming tasks from three popular tutorial sites. CROKAGE only considers Java-tagged questions and their answers due to the background knowledge of the evaluators. However, we consider this as a limitation instead of a threat, since our dataset contains 3.8 M of Java posts. CROKAGE could be easily adapted to support recommendations to other languages as long as it can extract information from Stack Overflow.

Threats to construct validity relates to suitability of our evaluation metrics. We choose four performance metrics: Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, which are commonly adopted by related literature in software engineering[1, 2, 4, 15, 25].

VI. Related Work

A. Code Example Suggestion

There have been several studies [4, 28]–[39] that return relevant code against natural language queries. McMillan et al. [37] propose a search engine that combines PageRank with Keyword matching to retrieve relevant functions. Our work differs from theirs on the granularity of the suggested code, since we do not restrict our search to functions. Campbell and Treude [31] develop a tool that assists users providing suggestions to the queries. Rahman et al. [38] instead, propose a tool to reformulate the queries before applying the search by using associations between keywords and APIs. Both tools however rely on third-part search engines. While the first relies on Google search API to retrieve relevant code, the second uses GitHub code search API. This dependency constrains their tools to the limitations of the third-part APIs (e.g., the number of searches in a period of time).

Some works [28, 34, 35] infer API usage sequences for a given task. T2API [28] learns API usages via graph-based

language model. They use a statistical machine translation to associate descriptions and corresponding code. DeepAPI [35] composes the associations between the sequence of words in a query and APIs through deep learning. SWIM [34] uses statistical word alignment to relate query words with API elements. Our work instead, exploits more than just API sequences. While these tools could return the same API sequences for two queries with different purposes, our work distinguish code aspects like method and class names. This concern has been also addressed by DeepCS [39]. Their tool jointly embeds natural language descriptions and code examples into a high-dimensional vector space in such a way that the description and their accompanying code examples have similar vector representations. They use such representations to calculate the similarity between the query and the code. Our tool is similar, but instead of using deep learning, we rely on information retrieval techniques.

Several tools [29, 30, 36, 40]–[42] rely on lexical similarities to retrieve relevant code. Campos et al. [36] rank related code documents by applying a combination of Vector Space and Boolean models. The same idea is used by Lv et al. [40]. Their tool however, extends the Boolean model to integrate the benefits of both models. Like our tool, Lv et al.'s tool also enriches the search with API names related to the input query. Zagalsky et al. [30] propose a tool to retrieve source code based on keywords using TF-IDF to score code documents. Bajracharya et al. [29] mine relevant API elements through shared concepts between the query and suggested words from open source systems. These mentioned approaches however, miss relevant documents if the query and the documents do not share common words. Our tool addresses this weakness by harnessing embeddings to capture words' semantics. That is, our tool is able to find documents that share semantically similar words with the query, despite having lexical dissimilarity. Furthermore, our tool can distinguish the order of the words, another limitation of their approaches.

Our work, differently from the mentioned tools, not only retrieves code but also provides explanations. BIKER [4] is the most related work to ours and we compare our work with theirs in multiple ways, as shown in Section IV.

B. Code Explanation Generation

Several early studies [2, 16, 43]–[48] propose automatic approaches to extract explanations to code. For this, they explore lexical properties usually in combination with strategies like clone detection [16, 48], topics (like LDA) [43], word embeddings [2], machine learning [46] and deep learning [47]. Wong et al. [16] propose a series of heuristics to match the code with natural language. They select the best descriptions for a code and use natural language processing to filter relevant sentences to compose the descriptions. Our work harness two patterns they develop to select relevant sentences. Similarly, in another study, Wong et al. [48] synthesize comments from similar code snippets. They try to address the limitation of their previous work by using GitHub instead of Stack Overflow to extract the comments, since comments in Q&A websites are not often written in full sentences. Rahman et al. [43] use heuristics to extract comments from Stack Overflow. Their approach combines the heuristics to rank the top most relevant comments for a source code. Chatterjee et al. [46] develop a technique to extract descriptions associated with code segments from articles. Differently from Q&A websites, the code in articles is not delineated by markers. They also convert documents (e.g., pdf and images) to text and learn the associations between text and code using machine learning. Xu et al. [2] employ word embeddings to handle the lexical gap between between natural language queries and Stack Overflow question titles. They use the answers from relevant questions to produce summaries. Despite they generate diverse summaries to the queries, their summaries do not contain source code. Hu et al. [47] propose an approach to generate comments for java methods through neural networks. But instead of relying on words to learn associations between code and descriptions, they use Abstract Syntax Trees to represent methods. This strategy showed efficiency to lean the associations even when methods and identifiers in the code are poorly named.

We refer the reader to the survey by Wang et al. [49] to more information about works in the context of comment generation for source code. Our work is closely related to these works in the sense that we also capture explanations for source code. We leverage natural language processing and explore lexical properties by considering the context surrounding the code.

VII. Conclusion and Future Work

In this work, we propose CROKAGE, a tool to help developers with the daily problem of seeking relevant code examples on the web for programming tasks. CROKAGE leverages API knowledge stored in Stack Overflow to generate solutions containing source code and explanations for tasks written in natural language. For this, we first employ lexical similarity combined with word embeddings to select candidate answers from Stack Overflow to a programming task (i.e., query). Then, we assign on each answer an API class score according to three state-of-art API recommendation systems. Next, we combine four weighted factors to rank candidate answers. And lastly, we use natural language processing on the top quality answers to compose the solutions. Our findings show that CROKAGE outperforms several other baselines to retrieve relevant answers for programming tasks, including a state-of-the-art one. We also perform a user study that demonstrate the effectiveness of CROKAGE to provide quality solutions regarding code examples and explanations. In the future, we will implement CROKAGE in form of an Eclipse plugin to enable developers to obtain our solutions right from the IDE.

Acknowledgement

We thank the authors of BIKER for sharing their tool. This research is supported in-part by a Canada First Research Excellence Fund (CFREF) grant coordinated by the Global Institute for Food Security (GIFS).

REFERENCES

[1] M. M. Rahman and C. K. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *Proc. ICSME*, 2018, pp. 473–484.

[2] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developersź technical questions," in *Proc. ASE*, 2017, pp. 706–716.

[3] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. NIPS*, 2013, pp. 3111–3119.

[4] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *Proc. ASE*, 2018, pp. 293–304.

[5] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *IP&M*, vol. 24, no. 5, pp. 513–523, 1988.

[6] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *Proc. ACM SIGIR*, 1994, pp. 232–241.

[7] Jsoup, "Java html parser," http://jsoup.org.

[8] F. M. Delfim, K. V. R. Paixao, D. Cassou, and M. A. de Almeida Maia, "Redocumenting apis with crowd knowledge: a coverage analysis based on question types," *JBCS*, vol. 22, no. 1, p. 9, 2016.

[9] Apache, "Lucene," http://lucene.apache.org/.

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *TACL*, vol. 5, pp. 135–146, 2017.

[11] E. Hill, S. Rao, and A. Kak, "On the use of stemming for concern location and bug localization in java," in *Proc. SCAM*, 2012, pp. 184–193.

[12] 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*, 2016, pp. 404–415.

[13] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions in stack overflow," in *Proc. MSR*, 2016, pp. 402–412.

[14] R. F. G. Silva, K. V. R. Paixao, and M. A. Maia, "Duplicate question detection in stack overflow: A reproducibility study," in *Proc. SANER*, 2018, pp. 572–581.

[15] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *Proc. SANER*, 2016, pp. 349–359.

[16] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proc. ASE*, 2013, pp. 562–567.

[17] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proc. ACL*, 2014, pp. 55–60.

[18] Java2s, "Java2s," http://java2s.com.

[19] BeginnersBook, "BeginnersBook," http://beginnersbook.com.

[20] W. Saryada, "Kodejava," http://kodejava.org.

[21] Google Inc., "Google search engine," http://google.com.

[22] Microsoft Inc., "Bing search engine," http://bing.com.

[23] Stack Exchange Inc., "Stack Overflow search engine," http://stackoverflow.com.

[24] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[25] M. M. Rahman and C. K. Roy, "STRICT: Information retrieval based search term identification for concept location," in *Proc. SANER*, 2017, pp. 79–90.

[31] B. A. Campbell and C. Treude, "Nlp2code: Code snippet content assist via natural language tasks," in *Proc. ICSME*, 2017, pp. 628–632.

[26] C. Fritz, E. Peter, and J. Richler, "Effect size estimates: current use, calculations, and interpretation." *JEPG*, vol. 141, no. 1, p. 218, 2012.

[27] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[28] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2API: synthesizing API code usage templates from english texts with statistical translation," in *Proc. FSE*, 2016, pp. 1013–1017.

[29] S. Bajracharya, J. Ossher, and C. Lopes, "Searching API usage examples in code repositories with sourcerer API search," in *Workshop on Search-driven Development*, 2010, pp. 5–8.

[30] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using social media for code recommendation," in *Proc. RSSE*, 2012, pp. 38–42.

[32] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss, "Hunter: next-generation code reuse for java," in *Proc. FSE*, 2016, pp. 1028–1032.

[33] T. Gvero and V. Kuncak, "Interactive synthesis using free-form queries," in *Proc. ICSE*, 2015, pp. 689–692.

[34] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing what I mean-code search and idiomatic snippet synthesis," in *Proc. ICSE*, 2016, pp. 357–367.

[35] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. FSE*, 2016, pp. 631–642.

[36] E. C. Campos, L. B. L. D. Souza, and M. A. Maia, "Nuggets miner: Assisting developers by harnessing the Stack Overflow crowd knowledge and the github traceability," in *Proc. CBSoft-Tool Session*, 2014.

[37] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proc. ICSE*, 2011, pp. 111–120.

[38] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Code search in the IDE using crowdsourced knowledge," in *Proc. ICSE*, 2017, pp. 51–54.

[39] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. ICSE*, 2018, pp. 933–944.

[40] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on API understanding and extended boolean model (e)," in *Proc. ASE*, 2015, pp. 260–270.

[41] G. Santos, K. V. R. Paixao, N. Anquetil, A. Etien, M. A. Maia, and S. Ducasse, "Recommending source code locations for system specific transformations," in *Proc. SANER*, 2017, pp. 160–170.

[42] G. Dotzler, M. Kamp, P. Kreutzer, and M. Philippsen, "More accurate recommendations for method-level changes," in *Proc. ESEC/FSE*, 2017, pp. 798–808.

[43] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *Proc. SCAM*, 2015, pp. 81–90.

[44] L. B. L. de Souza, E. C. Campos, and M. A. Maia, "Ranking crowd knowledge to assist software development," in *Proc. ICPC*, 2014, pp. 72–82.

[45] E. C. Campos, L. B. de Souza, and M. A. Maia, "Searching crowd knowledge to recommend solutions for api usage tasks," *JSEP*, vol. 28, no. 10, pp. 863–892, 2016.

[46] P. Chatterjee, B. Gause, H. Hedinger, and L. Pollock, "Extracting code segments and their descriptions from research articles," in *Proc. MSR*, 2017, pp. 91–101.

[47] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. ICPC*, 2018, pp. 200–210.

[48] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *Proc. SANER*, 2015, pp. 380–389.

[49] X. Wang, Y. Peng, and B. Zhang, "Comment generation for source code: State of the art, challenges and opportunities," *arXiv:1802.02971*, 2018.