Empirical Software Engineering (2020) 25:4707–4758
https://doi.org/10.1007/s10664-020-09863-2

CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge

Rodrigo Fernandes Gomes da Silva[1] · Chanchal K. Roy[2] · Mohammad Masudur Rahman[2] · Kevin A. Schneider[2] · Klérisson Paixão[1] · Carlos Eduardo de Carvalho Dantas[1] · Marcelo de Almeida Maia[1]

Published online: 2 September 2020

Abstract

Developers often search for relevant code examples on the web for their programming tasks. Unfortunately, they face three major problems. First, they frequently need to read and analyse multiple results from the search engines to obtain a satisfactory solution. Second, the search is impaired due to a lexical gap between the query (task description) and the information associated with the solution (e.g., code example). Third, the retrieved solution may not be comprehensible, i.e., the code segment might miss a succinct explanation. To address these three problems, we propose CROKAGE (CrowdKnowledge Answer Generator), a tool that takes the description of a programming task (the query) as input and delivers a comprehensible solution for the task. Our solutions contain not only relevant code examples but also their succinct explanations written by human developers. The search for code examples is modeled as an Information Retrieval (IR) problem. We first leverage the crowd knowledge stored in Stack Overflow to retrieve the candidate answers against a programming task. For this, we use a fine-tuned IR technique, chosen after comparing 11 IR techniques in terms of performance. Then we use a multi-factor relevance mechanism to mitigate the lexical gap problem, and select the top quality answers related to the task. Finally, we perform natural language processing on the top quality answers and deliver the comprehensible solutions containing both code examples and code explanations unlike earlier studies. We evaluate and compare our approach against ten baselines, including the state-of-art. We show that CROKAGE outperforms the ten baselines in suggesting relevant solutions for 902 programming tasks (i.e., queries) of three popular programming languages: Java, Python and PHP. Furthermore, we use 24 programming tasks (queries) to evaluate our solutions with 29 developers and confirm that CROKAGE outperforms 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).

Communicated by: Tim Menzies

✉ Marcelo de Almeida Maia
marcelo.maia@ufu.br

Extended author information available on the last page of the article.

Keywords Mining crowd knowledge · Stack overflow · Word embedding · Code search

1 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 (Rahman and Roy 2018). Developers often choose an ad hoc query to describe their programming task, and then submit their query to a code search engine (e.g., Koders). Unfortunately, they face three major problems. First, they often need to spend considerable time and effort reading and digesting dozens of documents in order to synthesize a satisfactory solution for the task. Second, 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 appropriate keywords (e.g., relevant API classes) that could better explain the task. Third, the retrieved solution might always not be comprehensible. Either the retrieved code segment might miss a succinct explanation (Rahman and Roy 2018) or the textual solution might miss a required code segment (Xu et al. 2017).

Traditional Information Retrieval (IR) methods (e.g., Vector Space Model (Baeza-Yates et al. 1999)) 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. (2013a, b) recently propose word embedding technology (e.g., word2vec) that captures words' semantics and represents each word as a high-dimensional vector. Such vectors could be used to determine the semantic similarity between any two documents despite their lexical dissimilarity. This technique was later used by two recent studies: AnswerBot (Xu et al. 2017) and BIKER (Huang et al. 2018). They attempt to address the three aforementioned problems experienced 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 source code and explanations. However, BIKER is only able to provide the explanations from the official Java API documentations. Thus, their explanations are restricted to a limited set of APIs only.

In this paper, we propose a novel approach namely CROKAGE (**Cro**wd **K**nowledge **A**nswer **Ge**nerator) that takes a task description in natural language (the query) as input and then returns relevant, comprehensible programming solutions containing both code examples and succinct explanations. In particular, we address the limitations of the two earlier approaches (Xu et al. 2017; Huang et al. 2018). First, unlike AnswerBot (Xu et al. 2017) (i.e., provides only answer summary texts), we deliver both relevant code segments and their corresponding explanations. Second, while BIKER (Huang et al. 2018) returns only generic explanations extracted from the official Java API documentation, we provide succinct code explanations written by human developers. We extract code and explanations from Stack Overflow answers. They contain a wide range of development topics and APIs, and serve as a large repository of source code examples (code snippets + explanations) (Nasehi et al. 2012; Yang et al. 2017; Ciborowska et al. 2018; An et al. 2017; Ragkhitwetsagul et al. 2018). Thus, our explanations, different from BIKER, are not composed from the limited official API documentation, rather we harness the knowledge from 27.4 millions[1] Stack Overflow answers (Rahman et al. 2017).

[1] https://data.stackexchange.com/stackoverflow/query on July, 2019

In order to deliver comprehensible solutions for a programming task, CROKAGE first collects candidate Q&A pairs from Stack Overflow by employing an appropriate IR technique (e.g., BM25 (Robertson and Walker 1994)). Then, it employs a multi-factor relevance mechanism that overcomes the lexical gap problem with word embedding technology and finally returns the top quality answers related to the task. Furthermore, CROKAGE uses natural language processing and noise filtration to compose a succinct explanation for each of the suggested code examples.

To evaluate our approach, we first construct the ground truth for 115 programming tasks collected from three popular tutorial sites – KodeJava, JavaDB and Java2s (Rahman et al. 2016). We manually analyze 6,558 Java answers from Stack Overflow against these 115 programming tasks (or queries). Similarly, we manually analyze 201 PHP answers from Stack Overflow against 10 randomly selected tasks (queries) from the above list of 115 tasks. We also use a manually curated dataset provided by a previous work (Yin et al. 2018) containing questions (i.e., intent) and the Python code snippets that implement the intent. After constructing the ground truth for the three programming languages, we evaluate our approach in three different ways. First, we compare the performance of CROKAGE in code example suggestion against ten baselines, including the state-of-art, BIKER (Huang et al. 2018). We show that CROKAGE outperforms all baselines by a statistically significant margin in delivering relevant programming solutions (code segments + explanations) for programming tasks (i.e., queries). For Java language, CROKAGE achieves 81% Top-10 Accuracy, 49% precision, 22% recall, and a reciprocal rank of 0.55, which are 65%, 38%, 21%, and 44% higher respectively than those of BIKER. We also show that similar performances could be achieved for PHP and Python, which suggests CROKAGE's generalizability. Second, we investigate four weighted relevance factors and evaluate their individual contribution in CROKAGE's performance. We find that three factors (Text Frequency - Inverse Document Frequency, Method and Semantic) significantly influence CROKAGE's performance, which justifies their use in our approach. Third, we conduct a user study to investigate how our approach can benefit the software developers in their programming tasks. We ask 29 professional developers to evaluate our solutions for 24 programming tasks and compare with the state-of-the-art BIKER (Huang et al. 2018). Our findings suggest that solutions from CROKAGE are more effective than the ones of BIKER (Huang et al. 2018) in terms of relevance of the suggested code examples, benefit of the code explanations and the overall solution quality (code + explanation).

Novelty in Contribution This paper is a significantly extended version of our earlier work (Silva et al. 2019) in several aspects. First, we evaluate using more programming tasks. We extend our previous dataset containing 100 Java-only programming queries and 6,224 relevant answers to 1,805 programming queries and 11K answers from three programming languages: Java, Python and PHP. Second, we re-calibrate our parameters and settings. In our previous work, we employ three state-of-art API recommendation systems – BIKER (Huang et al. 2018), NLP2API (Rahman and Roy 2018) and RACK (Rahman et al. 2016) to collect the most relevant API classes for a given query (i.e., task description) and show five of their combinations to provide API classes for the queries. In this work, we perform extensive analyses and investigate how their all possible combinations can affect the performance of our approach. Third, we empirically evaluate 11 IR techniques and four relevance factors in retrieving relevant solutions for programming tasks, which justifies our choice for the IR technique and the relevance factors. Such an empirical justification was not present in our previous work. Fourth, we present the implementation of our approach

in form of a tool to assist developers with their programming issues. This tool called Stack Overflow's attention, and they have published a blog post[2] featuring CROKAGE. In the first week after the post, our tool received more than 12.7k queries. The number of queries have been boosted, probably because several technical blogs worldwide also reproduced the Stack Overflow post. Fifth, we present an usage analysis of the tool with 15,865 queries and 784 user ratings collected during five months of operation. Our findings show that the composition of the queries impose major challenges for the code search engines, since the queries are usually short and often misspelled. And although most of the developers (55.8%) are satisfied with our solutions, a manually classification of 349 queries with poor rating reveals that most of them (58.4%) have problems in their composition.

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

- A novel approach –CROKAGE– that suggests programming solutions containing both code snippets and explanations for programming tasks (i.e., queries) written in natural language texts by harnessing the crowd knowledge from Stack Overflow.
- An empirical evaluation of CROKAGE on the suggestion of relevant code examples for programming tasks and a comparison with the state-of-art study (Huang et al. 2018). Our approach outperforms ten baselines, including the state-of-art, in retrieving relevant and comprehensible programming solutions.
- A manually curated benchmark dataset composed of 11K answers (6,558 Java + 201 PHP + 4,691 Python) for 1,805 (115 Java + 10 PHP + 1,680 Python) programming tasks. The dataset was constructed by two professional developers spending 94 man hours.
- A replication package[3] containing CROKAGE's prototype, detailed results of our user study and our used dataset for replication or third party reuse.
- The implementation of CROKAGE in form of a tool[4] to support developers with their programming tasks, which can be invoked by a web browser or by any REST (Representational State Transfer) service.
- An usage analysis of the tool five months after its release regarding the composition of 15.865 queries and 784 user ratings.

The rest of this paper is structured as follows. Section 2 shows the motivation of our work and compare our approach with two state-of-art tools. Section 3 describes the technical details of CROKAGE. Section 4 presents the implementation of CROKAGE in form of a tool. Section 5 reports the experimental methods and the obtained results. Section 6 discusses the threats to validity of our experiments. Section 7 reviews the related works. Finally, Section 8 concludes the paper.

2 Motivating Examples

Let us consider a use-case scenario where a developer is looking for a solution to the programmin task: *"run .exe from text file"*. Figure 1 presents three solutions from three different approaches: AnswerBot (Xu et al. 2017), BIKER (Huang et al. 2018) and CROKAGE respectively. The solution proposed by AnswerBot (Fig. 1a) contains sentences

[2] http://tiny.cc/xh3kbz
[3] https://github.com/muldon/crokage-emse-replication-package
[4] http://isel.ufu.br:9000/

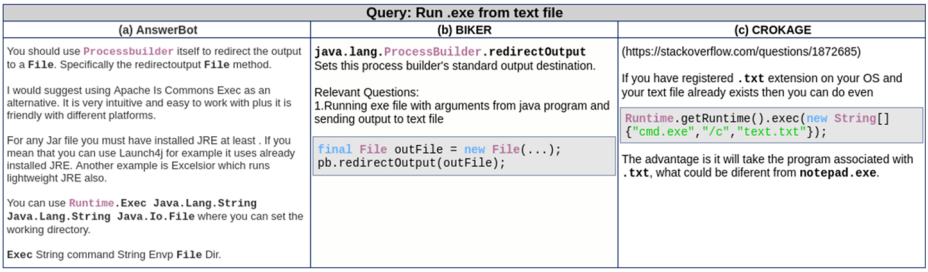


Fig. 1 Programming Solutions from **a** AnswerBot, **b** BIKER, and **c** CROKAGE

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 (Huang et al. 2018) provides both code example and corresponding explanation (Fig. 1b). 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 (Fig. 1c). Unlike AnswerBot (Xu et al. 2017), CROKAGE delivers a solution containing relevant code segment. Unlike BIKER (Huang et al. 2018) that 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 official documentation of the built-in Java APIs. Thus, our solution has a much more potential than the existing alternatives (i.e., AnswerBot (Xu et al. 2017) and BIKER (Huang et al. 2018)). Such a finding has also been confirmed by the user study (Section 5.6).

3 Proposed Approach

Figure 2 shows the schematic diagram of our proposed approach CROKAGE, which contains four stages. First, in Section 3.1 we prepare the corpus using 3.9M Q&A threads from Stack Overflow. Then, in Section 3.2 we construct several models (e.g., *fastText* model), maps and indices. Next, in Section 3.3 these models, maps and indices are employed to retrieve the relevant answers from the corpus against a programming task description. And finally, in Section 3.4 the top quality answers are used to compose and then suggest the programming solutions for the task. We discuss each of these stages as follows:

3.1 Corpus Preparation

In order to deliver appropriate solutions from Stack Overflow against a programming task (i.e., query), we need to construct the domain specific knowledge base (Fig. 2a). We collect

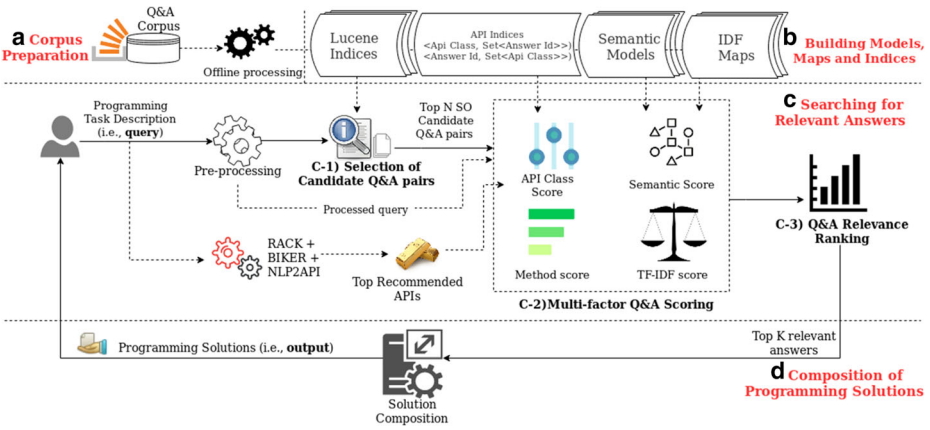


Fig. 2 Schematic diagram of CROKAGE. **a** Corpus Preparation, **b** Building Models, Maps and Indices, **c** Searching for Relevant Answers, and **d** Composition of Programming Solutions

a total of 10,248,824 questions and answers from Stack Overflow Q&A site[5] related to three programming languages, as shown in Table 1. We separate posts related to each programming language using tags. We employ regular expressions to analyse the "tags". For Java, we select all the questions that contain "java" but not "javascript". For Python and PHP, we select the questions that contain "python" and "php" respectively. Thus, we capture all the questions containing these tags and their derivatives such as "<java-8>", "<python-2.7>", and "<cakephp-3.x>". Posts with language intersections are also selected. The rate of intersection between languages ranges from 0.338% to 0.578%.

After selecting the questions, we also select all their answers for our study. We use *Jsoup* (Jsoup 2020) to parse these questions and answers and then separate texts and code. We identify the code using the `<code>` and `<pre>` tags. That is, any text found inside the tags are considered as code. The codes found outside the tags are considered as regular texts and incorporated to the texts. We then remove all punctuation symbols, stop words[6], small words (i.e., size lower than 2) and numbers from both codes and texts. We save these processed versions of Q&A threads alongside the original versions which are later used to compose the programming solutions (Section 3.4).

3.2 Building Models, Maps and Indices

Construction of Lucene Indices We first construct a Lucene index with all questions from Stack Overflow of a specific language (e.g., Java), along with their respective answers (Fig. 2b). We select question-answer pairs in such a way that answer and question has at least one upvote and each answer contains at least one code segment. We capture the pre-processed version of each pair as a document within a large document corpus. We then build

[5] https://archive.org/details/stackexchange - dump published in March 2019
[6] https://bit.ly/1Nt4eMh

Table 1 Stack overflow questions used for corpus preparation

Language	Type	#Posts	Total
Java	Questions	1,543,653	4,106,137
	Answers	2,562,484	
PHP	Questions	1,193,737	3,135,861
	Answers	1,942,124	
Python	Questions	1,212,706	3,006,826
	Answers	1,794,120	
		Total	10,248,824

an index with Lucene (Apache 2020) using all these documents. We perform this process for the three adopted languages, resulting in three indices which are later used to retrieve the most relevant answers against a query (i.e., task description).

Construction of fastText Models Task description (the query) from a developer might always not contain the required keywords such as relevant API references. Hence, the Lucene-based approach above might fail to retrieve the relevant answers due to the lexical gap issue. In order to overcome the lexical gap issue between the query and the answers from Stack Overflow, we employ a word embedding model namely *fastText* (Bojanowski et al. 2017). In this model, each word is represented as a high dimensional vector in such a way that similar words have similar vector representations. We employ *fastText* to build a *skip-gram* model and learn the vector representation of each word of our vocabulary. Please note that since we adopted three programming languages, we build three vocabularies, each containing the distinct words of all questions and answers of that language. For each language (i.e., Java, Python and PHP), we first combine the pre-processed contents of *title*, *body text* and *code segments* from all Q&A threads of the language into a file and then employ *fastText* to learn the vector representation of each word. 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 (Hill et al. 2012).

In order to learn the vector representations, we customize these parameters: vector size=100, epoch = 10, minimum size = 3 and maximal size = 8 whereas the other parameters remain default. That is, our model learns an embedding vector of size 100 for each word, loops 10 times over the data during the training phase and the subwords size (i.e., the size of substrings contained in a word) ranges from 3 to 8. The subwords size and vector size are important parameters of this model. We thus empirically test different values for the subwords size and find that the values 3 and 8 (i.e., minimum size = 3 and maximal size = 8) perform slightly higher (Section 5.2 describes the adopted metrics) than the default values (Facebook Inc 2020) (i.e., minimum size = 3 and maximal size = 6). We also investigate different values for the vector size and observe that as we gradually increase the value from 100 to 300, the model does not perform significantly higher. Chen et al. (2019) build a similar model to recommend likely analogical APIs for queries and test five different vector sizes ranging from 100 and 500. They find no significant differences in performance among the different settings. Thus, our choice of the embedding vector size is supported by Chen et al. Furthermore, operations involving larger vectors (e.g., cosine similarity) can be also expensive in terms of time and memory.

Once the models are built for each language (Fig. 2b), we use them as a dictionary for mapping between the words of the vocabulary and their respective vector representations.

Construction of IDF Maps Inverse Document Frequency (IDF) has been often used for determining the importance of a term within a corpus (Huang et al. 2018; Ye et al. 2016). 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 vocabularies contains a total of 4,210,516 distinct words for Java, 2,041,581 distinct words for PHP and 2,222,462 distinct words for Python. For each language, we calculate the IDF of each word and build an IDF Map (Fig. 2b) 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 between the query and the candidate solutions).

Construction of API Indices We build two API indices (Fig. 2b). The first index namely *postApis* maps each post (i.e., question or answer) from Stack Overflow to all APIs classes present in their code segments. The second index namely *apiAnswers* maps each API class to all corresponding answers from Stack Overflow that use that class. To build the *postApis* index, we first select all questions and answers containing code (i.e., containing tags `<pre><code>`), and extract code elements from them using *Jsoup* (Jsoup 2020). Then, we identify their API classes using appropriate regular expressions. The process to build the *apiAnswers* index is similar. However we only select answers and the map is inverted: 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 5), 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"*[7] 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 indices. We believe that such API classes could be less appropriate for our problem context.

Our *postApis* index contains 1,143,602 mappings of posts (i.e., questions and answers) to Api classes while our *apiAnswers* index contains 396,640 mappings of Api classes to Stack Overflow answers IDs. These two indices are used later (for the Java language only) in our multi-factor relevance mechanism to calculate the similarity between the classes pertinent to the query and the classes from to the candidate answers.

3.3 Searching for Relevant Answers

Once the models, indices and maps are built, we use them in searching for relevant answers for a given task description (i.e., query). Our search component works in three stages as shown in Fig. 2c. In order to search for relevant answers, CROKAGE first loads the models and indices (Section 3.2) and then pre-processes the task description with standard natural language pre-processing. Then, CROKAGE navigates through the three stages as follows.

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

3.3.1 Selection of Candidate Q&A Pairs

Given a pre-processed task description (i.e., query) T and the Lucene index constructed above for the language of interest (e.g., Java), CROKAGE uses BM25 (Robertson and Walker 1994) function to filter the top N relevant Q&A pairs (Fig. 2c-1) from the pre-loaded index as follows:

$$filterPairs(P, T) = \sum_{i=1}^{n} idf(q_i) * \frac{f(q_i, P) * (k+1)}{f(q_i, P) + k * \left(1 - b + b * \frac{|P|}{avgdl}\right)} \quad (1)$$

where $|P|$ is the length of the Q&A pair P in words, $f(q_i, P)$ is keyword q_i's term frequency in pair P and $avgdl$ is the average document length in the index. k and b are two parameters where k is a non-negative finite value that controls non-linear term frequency normalization (saturation), and b controls to what degree document length normalizes term frequency values, varying in [0, 1]. $idf(q_i)$ is the inverse document frequency of keyword q_i and computed as follows:

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

where $n(q_i)$ is the number of documents containing keyword q_i and D is the total number of documents in the index (or corpus). CROKAGE retrieves the top N Q&A pairs sorted by their relevance to the query, which are then used in the next stage. We investigate k, b and N (as shown in Section 5.4) and determine their optimal values across the three languages.

3.3.2 Multi-factor Q&A Scoring

In this stage, CROKAGE calculates scores for the candidate Q&A pairs using four similarity factors (Fig. 2c-2). It takes as input a pre-processed query, the top N candidate Q&A pairs from the previous stage, the *fastText* model, the IDF Map and the two API indices (i.e., *postApis* and *apiAnswers*). Then, for each Q&A pair, CROKAGE calculates four scores that represents the similarity between the input query and the pair. We choose the four similarity factors so that we can capture different kinds of similarities and mitigate the lexical gap problem between the task description (i.e., the query) and relevant pairs. We also consider factors in such a way they do not compromise CROKAGE's performance in terms of time to return the solution. Therefore, we choose factors that combined together do not exceed the threshold of one second per query to execute.

The first adopted factor is the lexical similarity, commonly adopted in IR (Campos et al. 2014; Lv et al. 2015; Zagalsky et al. 2012; Bajracharya et al. 2010), and herein represented by TF-IDF. Although we use BM25 (Robertson and Walker 1994) scoring function for selecting the candidate Q&A pairs, our empirical tests using the training set attest that for a small set of documents (e.g., 100 documents), the use of TF-IDF to score documents performs better than BM25 in terms of the adopted metrics (i.e., Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall). Therefore, after obtaining the candidate Q&A pairs using BM25 scoring function, we use TF-IDF to calculate the lexical score for these candidates.

The second factor is the semantic similarity. We use this factor to capture the semantics of the words, since the lexical similarity alone could miss relevant documents that do not

share common words with the task description. We leverage the semantics of words captured by *fastText* (Bojanowski et al. 2017) (Section 3.2) to calculate the similarity between two documents (e.g., the query and a candidate Q&A pair).

In complement to the aforementioned lexical and semantic similarity factors, we propose two other factors. Huang et al. (2018) suggest that if an API method occurs across multiple candidate answers for a given question, it could be relevant to the question. Thus, we adopt a method factor that considers the presence of common methods among the candidate Q&A pairs. Like API methods, the presence of API classes could also indicate important pairs. Exploratory studies with Stack Overflow questions and answers reveal that 65% of them are associated to programming APIs (Nasehi et al. 2012; Rahman and Roy 2017). We then create an API class factor to reward candidate Q&A pairs containing relevant API classes.

CROKAGE determines the scores of the four factors as follows:

Lexical Score Herein, we use TF-IDF to represent documents. 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:

$$TF - IDF(W) = TF(W) * log_{10}\left(\frac{N}{df_w}\right) \tag{3}$$

where N denotes the total number of documents in the corpus and df_w denotes the number of documents containing the word w. We determine the lexical similarity between the document representing the task description (i.e., query) and the document representing each Q&A pair using their cosine similarity as follows:

$$lexScore(P, T) = \frac{d_T \cdot d_P}{|d_T| \cdot |d_P|} = \frac{\sum_1^N tfidf_{ti,dq} \cdot tfidf_{ti,da}}{\sqrt{\sum_1^N tfidf_{ti,dq}^2} \cdot \sqrt{tfidf_{ti,da}^2}} \tag{4}$$

where d_T refers to the task description, d_P represents the Q&A pair and $tfidf_{ti,dk}$ is the term weight for each word of the document (task description or Q&A pair).

Semantic Score once the models are built (Section 3.2), CROKAGE transforms the preprocessed information from each Q&A pair (*body texts + title*) namely P and the task description (i.e., query) namely T into two bag of words. Then, CROKAGE computes the asymmetric relevance between P and T as follows:

$$asym(P \rightarrow T) = \frac{\sum_{w \in P} sim(w, T) * idf(w)}{\sum_{w \in P} idf(w)} \tag{5}$$

where $idf(w)$ is the correspondent IDF value of the word w, $sim(w, T)$ is the maximum value of $sim(w, w_T)$ for every word $w_T \in T$, and $sim(w, w_T)$ is the cosine similarity between w and w_T embedding vectors. The other asymmetric relevance namely $asym(T \rightarrow P)$ can be calculated by swapping P and T in (5). Thus, the final similarity between the task T and the Q&A pair P is the harmonic mean of the two asymmetric relevance scores as follows:

$$semScore(P, T) = \frac{2 * asym(P \rightarrow T) * asym(T \rightarrow P))}{asym(P \rightarrow T) + asym(T \rightarrow P)} \tag{6}$$

API Method Based Score herein, CROKAGE rewards the Q&A pairs whose answers contain the most frequent API methods among the candidate Q&A pairs. For this, CROKAGE

selects the methods used in each answer of a Q&A pair using appropriate regular expressions and identify the most frequent API method. Then, CROKAGE assigns each of the Q&A pair P an API method score as follows:

$$methodScore(P) = \frac{log_2(freq_m)}{x} \tag{7}$$

where $freq_m$ is the top method frequency and x controls the scale of the score. We test different values for x and find that $x = 10$ gives the best performance. If the answer of the Q&A does not contain any of the top API methods, the score is set to zero.

API Class Score CROKAGE leverages the association between the APIs relevant to a programming task (i.e., query) and the candidate Q&A pair. First, we obtain the relevant API classes for a programming task by employing three state-of-art API recommendation systems – BIKER (Huang et al. 2018), NLP2API (Rahman and Roy 2018) and RACK (Rahman et al. 2016). Our findings suggest that the combination of these tools provides the best results which justifies our choice of combining their API suggestions (Section 5.3). Second, we select the candidate Q&A pairs containing any API classes. For this, we use our *apiAnswers* index (Section 3.2) to select the answers IDs containing APIs and then identify their respective pairs among the candidate pairs. Third, we then combine the API classes from each pair (question + answers), and construct an API list namely *pairApis*. Fourth, for each recommended API class ($c \in C$) from the three tools (NLP2API + RACK + BIKER), we calculate the API class based score for each Q&A pair as follows:

$$apiScore(A) = \sum_{c \in C} \frac{1}{pos(c) + n} \tag{8}$$

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 *pairApis*. After careful investigation, we found the best value of n as 2. Our goal is to reward the Q&A pairs with more relevant classes and penalize the pairs with irrelevant classes. We calculate this score (i.e., *apiScore*) for all candidate Q&A pairs for the Java language only, since the three tools are limited to Java. For Python and PHP, we skip this factor.

3.3.3 Q&A Relevance Ranking

After calculating the four scores — *semScore, apiScore, lexScore* and *methodScore*, we normalize them and combine them in a final score (*factorsScore*) representing the relevance of each Q&A pair P to the task description (i.e., query) T as follows:

$$\begin{aligned} factorsScore(P, T) = {} & semScore \cdot semWeight \\ & + apiScore \cdot apiWeight \\ & + lexScore \cdot lexWeight \\ & + methodScore \cdot methodWeight \end{aligned} \tag{9}$$

here *semWeight*, *apiWeight*, *lexWeight*, and *methodWeight* are relative weights for each factor. We conduct controlled iterative experiments and determine a set of weights that return the highest Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall (Section 5.5). Once the final score (i.e., *factorsScore*) is calculated (Fig. 2c-3), we collect the Top-K Stack Overflow answers from the top scored Q&A pairs in order to compose the solution.

3.4 Composition of Programming Solutions

After collecting the Top-K recommended answers from the previous step, CROKAGE uses them to compose appropriate solutions for a desired task (i.e., query). In particular, CROKAGE discards the answers without any succinct explanations, and select a subset of n answers where $n <= K$. To select these answers, CROKAGE uses the following pseudocode in Algorithms 1 and 2.

Algorithm 1 Select answers with succinct code explanations.

```
1  answersSelection(query, recommendedAnswers, n)
2      relevantAnswers ← {}
3      for each answer in recommendedAnswers do
4          importantSentences ← filterImportantSentences(query, answer)
5          if (notEmpty(importantSentences)) then
6              relevantAnswers.add(answer)
7              if (relevantAnswers.size == n) then
8                  break
9              end if
10         end if
11     end for
12     return relevantAnswers
13 end
```

First, the algorithm iterates through the recommended answers (lines 3 to 11) and invokes the function *filterImportantSentences* passing the query and an answer as parameters (line 4). This function returns a list of important sentences, stored at *importantSentences* variable. If the answer contains at least one important sentence, this answer is added to a list of relevant answers (line 6). When the size of the list of relevant answers reaches n (lines 7 to 9), the algorithm breaks the loop and return the list in line 12. Let us consider for example the query *"Connect to a SQLite database"* and $K = 10$. Among the top 10 Stack Overflow answers retrieved by CROKAGE is the answer 4917877,[8] whose explanation is *"Try this:-"*, followed by a code snippet. The provided explanation is discarded by CROKAGE because the sentence is not important. CROKAGE checks the importance of a sentence according to two patterns proposed by Wong et al. (2013):

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

The two patterns above identify important sentences based on their POS structure, ensuring 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 (line 4) using the pseudocode shown in Algorithm 2.

[8] https://stackoverflow.com/questions/4917877

Algorithm 2 Pseudocode to filter important sentences from answers.

```
1  filterImportantSentences(query, answer)
2      pattern1 ← VP << (NP < /NN.?/) < /VB.?/
3      pattern2 ← NP! < PRP[<< VP|$VP]
4      importantSentences ← {}
5      processedBody ← preProcess(answer.getBody())
6      answerDocument ← st.coreDocument(processedBody)     /* Stanford
       lib */
7      st.annotate(answerDocument)                  /* POS annotation */
8      candidateSentences ← answerDocument.getSentences()
9      for each candidateSentence in candidateSentences do
10         parseTree ← candidateSentence.constituencyParse()
11         matcher1 ← pattern1.matcher(parseTree)
12         matcher2 ← pattern2.matcher(parseTree)
13         if ((matcher1) or (matcher2) or specialSentence(candidateSentence))
           then
14             importantSentences.add(candidateSentence)
15         end if
16     end for
17     return importantSentences
18 end
```

The algorithm receives two parameters: pre-processed query and recommended answer. CROKAGE first sets the patterns (lines 2 and 3) and initializes the list of important sentences (line 4). Then, it performs standard natural language pre-processing on the body texts of the answer (line 5). Next, the algorithm uses Stanford Part-Of-Speech Tagger (POS Tagger) to create a document out of these processed texts and assign a POS tag to each word of the sentence (lines 6 and 7). The algorithm then obtains the assigned candidate sentences from the document (line 8) and iterates over them (lines 9 to 16). For each candidate sentence, the algorithm builds the parse tree (line 10) and generates two pattern matchers to obtain the nodes that satisfy *pattern1* and *pattern2* (lines 11 and 12). If any of the two patterns are satisfied or the sentence contains a special condition (line 13 to 15), the algorithm add this sentence to the important sentences list (line 14). We consider a sentence to belong to a special condition if the sentence contains any number, camel case words, important words (i.e., "insert", "replace", "update"[9]) and shared words with the query. Then, the algorithm returns only the important sentences at line 17. If this list is not empty, it means that the answer contains important sentences, which are returned as the explanation for the code segment. In our aforementioned example, the provided explanation (i.e., *"Try this:-"*) is composed by only one sentence, which does not fit to the criteria of important sentence (i.e., is not a special sentence and does not match with any of the two patterns in Algorithm 2). Thus no candidate sentence is added to the list of important sentences (Algorithm 2 - line 14) and hence this answer is not added to the list of relevant answers (Algorithm 1 - line 6). In another example, CROKAGE retrieves the answer 7189370[10] for the query *"How do I iterate each characters of a string?"*. In this case, the explanation is composed by two sentences: *"Iterate over the characters of the String and while storing in a new array/string*

[9]the complete list of words is available at: https://bit.ly/2Hjv0tW
[10]https://stackoverflow.com/questions/7189370

you can append one space before appending each character" and *"Something like this :"*. Herein, the Algorithm 2 filters the first sentence and discards the second. Since at least one important sentence is returned to Algorithm 1, the answer is selected.

Our intuition is that the removal of unnecessary sentences may help developers with a more concise explanation. Thus, CROKAGE discards trivial or 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"*) and finally delivers a solution containing code and explanations. Figure 3 shows Top-3 results delivering comprehensible solution for the task: *"Convert Between a File Path and a URL"*.

4 CROKAGE Tool

We implement our approach in the form of a tool to assist developers with their daily programming issues. Herein, we describe the details of the tool. We divide this Section into

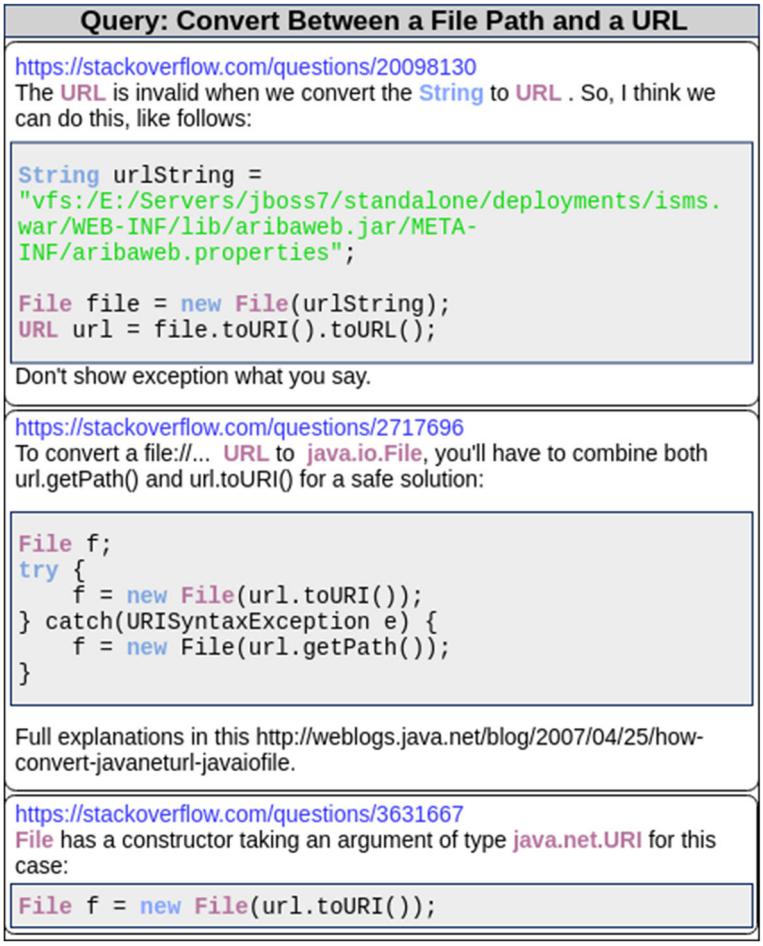


Fig. 3 Comprehensible solution generated by CROKAGE for the query: *"Convert Between a File Path and a URL"*

three subsections: Tool Presentation, Implementation Details and Usage Analysis, described as follows.

4.1 Tool Presentation

Figure 4 shows the tool front-end website, accessed by the URL http://isel.ufu.br:9000/. Herein, the user inputs the parameters and hit the button *Search* to search for the programming solutions. First, in **1)** the user inputs the query (i.e., the programming task description). Then, in **2)** she chooses the number of answers to be retrieved. We offer three options: one, five and ten. Next, in **3)** the user has the option to check if she wants only answers containing valid explanations. That is, the tool would filter only valid sentences, as showed in the Algorithm 2.

After the user hits the button *Search*, the tool collects the three parameters, performs the search (as described in Section 3.3) and then returns the solutions, as showed in Fig. 5. Two new areas are showed after the tool returns the results. In **4)** the tool collects the user feedback about the overall solution. And in **5)** the tool shows the answers sorted according to their relevance to the query (i.e., in descending order). The area has fixed height, but the user can scroll down and up to visualize the results.

4.2 Implementation Details

Figure 6 shows the tool architecture. We follow a REST (Representational State Transfer) architecture (Fielding and Taylor 2002), divided in three components: the front-end, the back-end and the database. The three components were built in such a way they can be hosted in different physical locations. As a pilot project, we hosted all of them in the same server equipped with Intel® Xeon® at 1.7 GHz on 86.4 GB RAM, 12 cores, and 64-bit Linux Ubuntu operating system.

The front-end is composed by a web server where the CROKAGE website is hosted. The website is an AngularJS application combined with the following technologies: HTML5, CSS (Cascading Style Sheets), Bootstrap, JQuery and JavaScript. The website is hosted in a Grunt server listening to calls at port 9000.

The back-end is composed by a Spring Boot application. This application contains a Tomcat Server listening to calls at port 8080. The application is a service containing a REST interface, a business layer and database layer. The REST interface is responsible for intercepting the HTTP calls from the front-end application, collect the parameters in JSON

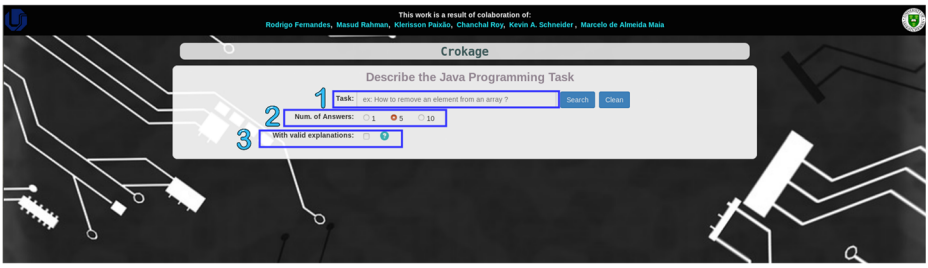


Fig. 4 CROKAGE implementation website: http://isel.ufu.br:9000/ . Search Parameters: 1) The field where the user inputs the query (i.e., *"how to insert an element array in a given position"*). 2) The number of answers to be retrieved (i.e., 5). 3) Whether only valid explanations should be filtered

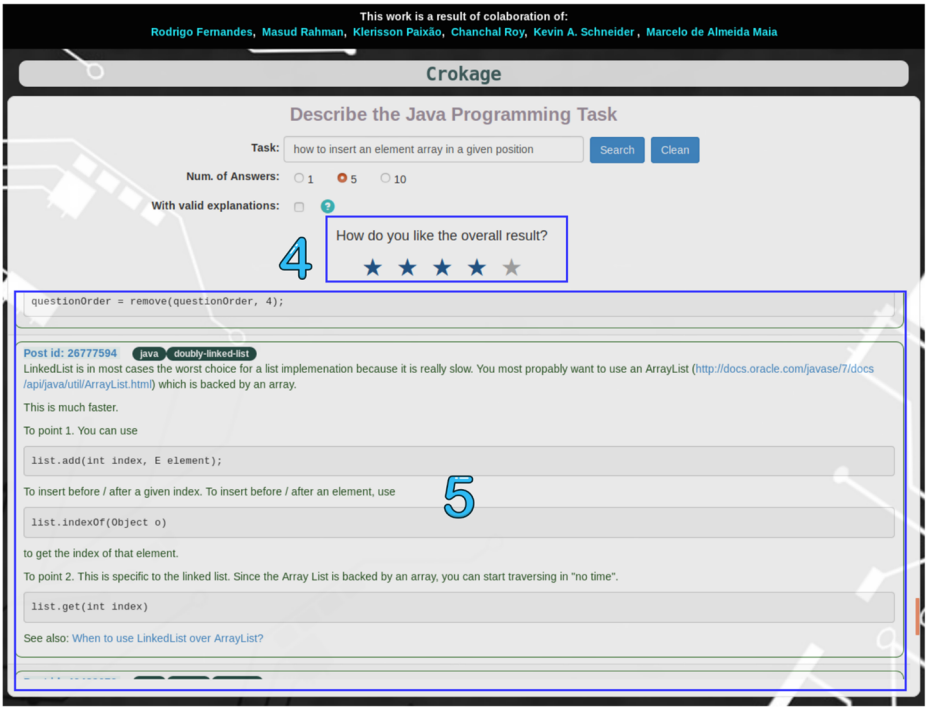


Fig. 5 Search Results: 4) The user feedback about the overall solution. 5) The top n answers related to the parameters, where n is the number of answers selected by the user

format, encapsulate them into Java objects and pass them to the business layer. Likewise, the interface is also responsible for returning the results processed by the business layer to the front-end in JSON format. The business layer contains the core of CROKAGE. Once it receives the parameters, it uses them to perform the search and return the results. The database layer is only responsible to communicate with the database and does not contain any business logic. The communication with the database is done via Java Database Connectivity (JDBC).

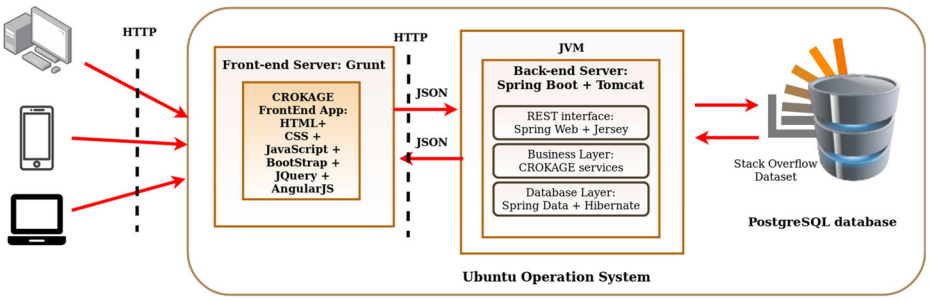


Fig. 6 CROKAGE REST architecture composed by a Front-end Server, a Back-end Server and a database

The database is PostgreSQL and contains the Stack Overflow dataset. It is programmed to listen to calls at port 80. The dataset was constructed by processing the Stack Overflow dump, as described in Section 3.1.

When the user first invokes the tool URL (i.e., *http://isel.ufu.br:9000/*) though a web browser, the front-end server (i.e., Grunt) returns the whole front-end application (i.e., the website resources) to the browser of the user device. Once CROKAGE is loaded in the user browser, all further functionalities invoke the back-end server through asynchronous HTTP calls at port 8080. Let us consider for example the scenario where the user wants to obtain the programming solutions to the query: *"how to insert an element array in a given position"*. The user fill the parameters and hit the button *Search*. At this point, the AngularJS engine makes a HTTP POST call to *http://isel.ufu.br:8080/crokage/query/getsolutions* passing the parameters in JSON format. Then, the back-end application listening at port 8080 is triggered. The REST interface then captures the parameters, converts them from JSON format to Java objects and pass them on to the business layer. Then, the business layer performs the search and returns the results in form of Java objects to to the REST layer, which in turn converts them again to JSON and returns them to the front-end application to be exhibited. CROKAGE does not consult the database at this point for performance reasons. Instead, CROKAGE builds a cache during the initialization of the back-end application containing the all Q&A pairs, the embeddings models, the maps and indexes as described in Section 3.2.

4.3 Usage Analysis

We released our tool on August, 2019. During the first week after the release, the tool registered an average of 1.8k searches per day. We analyze the queries asked by the community during five months after the release. In total, CROKAGE collects 15,865 queries during that interval.

4.3.1 Composition of the Queries

Since the developers can choose the desired number of answers in the result, many queries were duplicated, having different number of answers as parameter. We also found duplicates with different IP addresses, in different dates, probably indicating that the queries were made by different users. In total, we found 4,567 duplicates among the 15,865 queries (a duplication rate of 28.78%).

In order to better understand the composition of the queries, we first filter out those queries not related to Java, since CROKAGE only supports the Java language[11]. For this, two professional developers with more than ten years of experience in Java programming inspected the set of 11.298 queries and identified keywords that indicate a language other than Java (e.g., "python", "javascript", "kotlin"). After identifying the keywords not related to Java, an automatic process filtered out 871 queries containing these keywords. This process also filtered out 57 queries containing non-ASCII characters. After those two filters, 10,370 valid queries remained (i.e., non duplicated Java queries containing only ASCII characters), which corresponds to 65.36% of all queries inputted by developers (i.e., 15,865) in the analyzed period. Table 2 summarizes the overview of the queries and their totals.

[11] Despite this limitation was explicitly stated in the CROKAGE web Page (i.e., *http://isel.ufu.br:9000/*), a significant number of non Java queries were found

Table 2 Overview of queries submitted to CROKAGE during five months of usage

Total number of queries submitted by the developers	15,865
Number of duplicated queries	4,567
Number of non duplicated queries	11,298
Number of non Java queries among the non duplicated queries	871
Number of queries containing non ascii characters	57
Total of non duplicated Java queries containing only ascii characters	10,370

After filtering the queries, we analyze the remaining set of valid queries (i.e., 10,370) regarding four aspects: the number of tokens, the number of stop words, and the number of verbs and nouns in their POS composition.

Number of Tokens we classify the valid queries according to their number of tokens[12] as showed in Fig. 7. We observe that more than half of the queries, corresponding to 54.03%, have up to four tokens while only 18.14% of the queries have more than six tokens. Furthermore, only 2.89% have ten or more tokens. In total, we count 46,207 tokens in the 10,370 valid queries, which corresponds to an average of 4.4 tokens per query. These results suggest that developers tend to input short queries when looking for solutions to their programming tasks, as for example: *"Convert from Double to Float"*.

Number of Stop Words we also classify the queries according to the number of stop words[13] as showed in Fig. 8. We identify 6,557 queries containing stop words, which corresponds to 63.23% of the valid queries. In total, we count 16,294 stop words in the set of valid queries, which corresponds to an average of 1.57 stop words per query. We observe that the majority of the queries containing stop words (more than 80%) have from one to three stop words in their composition, as for example *"**How to** write **a** file driver?"*.

Number of Nouns and Verbs in order to understand the composition of the queries concerning the presence of verbs and nouns, we use Stanford Part-Of-Speech Tagger (POS Tagger) to assign a POS tag to all words of the queries. Then we classify the queries according the number of verbs and nouns as showed in Fig. 9. We observe that despite a significant amount of the queries, corresponding to 36.65%, does not have verbs, the majority of the queries containing verbs have only one verb (i.e., 53.58%). We also find that a small fraction of the queries, corresponding to 3.89%, have no nouns but the majority of the queries have one or two nouns. These results suggest that the developers usually describe their programming tasks using one verb representing an action followed by up to two nouns, as for example *"How to **remove duplicates** from an **array**?"*.

4.3.2 User Feedback

CROKAGE collects the feedback provided by the users in a 5-star rating (Section 4). That is, the higher the number of stars, the more the developer is satisfied with the solution. During the analyzed period (i.e., five months), CROKAGE collected 784 ratings from the

[12] CROKAGE search requires the query to have a minimum of one character and a maximum of 70 characters to run

[13] We adopt the list provided by Stanford: https://bit.ly/1Nt4eMh

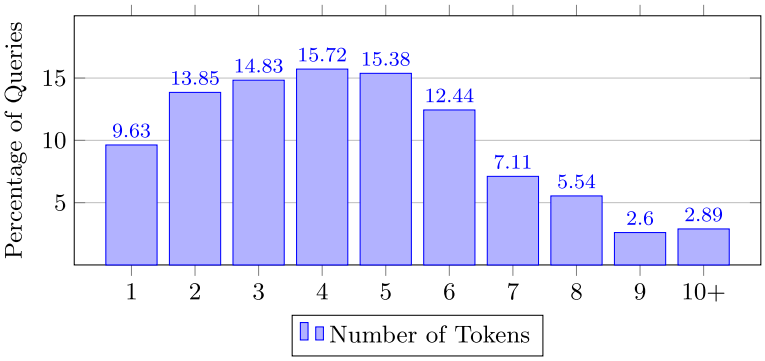


Fig. 7 Composition of developers' queries in terms of the number of tokens and their proportion (in percentage)

developers for the 15,865 provided solutions, which corresponds to a feedback rate of 4.9%. The ratings are showed in Fig. 10.

Overall, CROKAGE received more positive ratings than negatives: 329 5-star ratings, 106 4-star ratings, against 200 1-star ratings and 75 2-star ratings. The positive ratings, as showed in Fig. 10b, correspond to 55.86% while the negative ones correspond to 35.07%. In 9.43% of the cases (74), the developers found the solution in the middle (3-star).

We investigate whether any of the four analyzed aspects (i.e., number of tokens, number of stop words, and the number of verbs and nouns) of the queries have influence on the number of stars, but no correlation is found for any aspect. We then perform an open coding over the queries that received Likert values lower than 4 in order to investigate the reason of the developers' dissatisfaction. For this, two developers with more than ten years of experience in Java programming independently classify the 349 queries with 1-2-3-star into categories, i.e., labels. The labels are not initially established. Rather, they are found during the open coding, making the developers relabel the queries multiple times as new labels are discovered and others are abstracted. After labeling all queries, the conflicts are discussed and then solved. The results of our labeling are as showed in Table 3. In the end, nine labels/categories are found as described below:

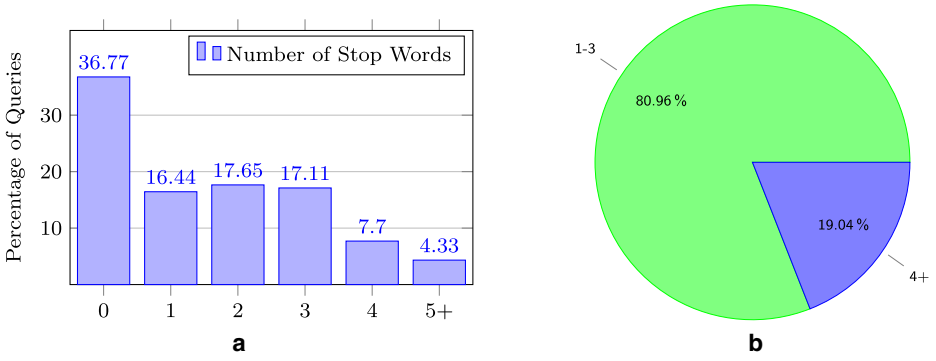


Fig. 8 **a** Composition of developers' queries in terms of the number of stop words and their proportion in percentage and **b** the proportion of stop words in the queries containing stop words. The queries are divided in two groups: 1-3 and 4+

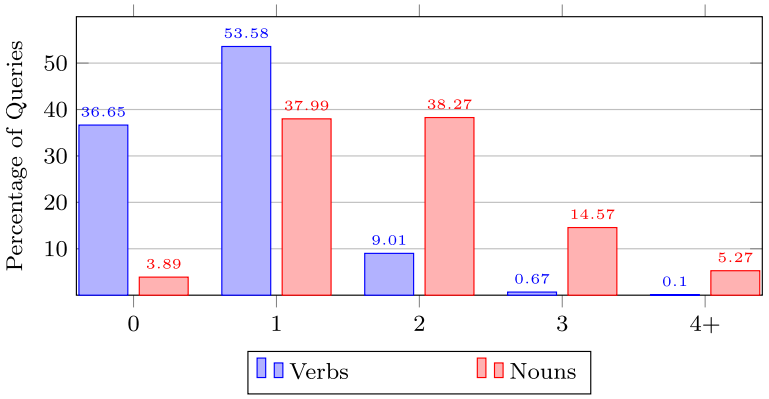


Fig. 9 Composition of developers' queries in terms of the number of verbs and nouns in their POS composition

- **Debug Corrective**: related to errors or problems found during the execution of an application, as for example *"502 bad gateway running a jsp on tomcat"*.
- **Not Applicable**: related to jokes such as *"why would woodchucks whittle wood when whacking water would wield wet?"*, or subjects not related to Java like *"how to edit minecraft minecart speed?"*.
- **Too Specific**: related to very specific situations, such as "loop for 6 times".
- **Environment**: related to environment issues, as for example *"How to get IntelliJ to run my code?"*.
- **Poorly Formulated Query**: queries containing misspelled words, as for example *"How to use jason"*.
- **Conceptual**: related to conceptual doubts on a particular topic, as for example *"How does hashmap implement entry"*.
- **Too Generic**: queries too generic, usually missing the query problem, as for example *"Hibernate"*.

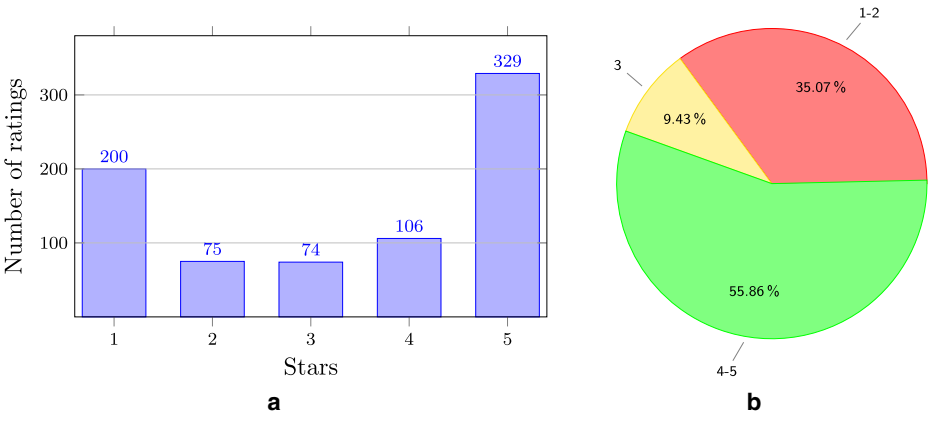


Fig. 10 **a** Number of ratings by number of stars in absolute numbers, and **b** the number of grouped ratings by group 1-2 (poor solution), 3 (related but incomplete) and 4-5 (good solution)

Table 3 Manually classification of 349 queries according into categories

Category	Num. Queries	% of the Total
No obvious reason	145	41.54
Other language	59	16.90
Too generic	46	13.18
Conceptual	35	10.02
Poorly formulated query	19	5.44
Environment	15	4.29
Too specific	15	4.29
Not applicable	10	2.86
Debug corrective	5	1.43

– **Other Language**: queries not related to Java,[14] as for example *"pandas drop duplicate indexes"*.
– **No Obvious Reason**: queries that do not fit into the above categories, as for example *"Reverse each string in a list"*.

4.3.3 Conclusions on the Tool Usage

Our findings show that 58.4% of the 349 manually classified queries, which are not classified as "No obvious reason", do not meet the requirements to be adherent to Crokage's purposes. For instance, 16.9% of the queries do not belong to Java language and 41.5% do not represent a *how-to* programming task or are not well written in such a way to convey a well defined problem that can be solved by code. Furthermore, the queries classified as "No obvious reason" do not necessarily have answers on Stack Overflow. For those cases, our tool also received poor ratings.

The observations made by the two evaluators after the ratings confirm the results obtained in Section 4.3.1: in general, during the code search, the developers tend to use short queries, containing stop words, and containing one or two nouns, but not always a verb. Furthermore, the words may be misspelled.

We conclude that the composition of the developers' queries impose major challenges for the code search engines that must handle the aforementioned restrictions. Our results on the classification of the 349 1-2-3-star queries show that on-the-fly improvements, such as, query reformulation, auto-completion, or query suggestion, are essential features, since problematic queries may lead to the search engine fail when trying to retrieve relevant results.

5 Experiments and Results

We evaluate our approach in multiple ways. We first evaluate the performance of 11 IR techniques to select candidate Q&A pairs for programming tasks (Section 5.4). Then, we evaluate the performance of CROKAGE in code example suggestion and compare with ten

[14] despite we use a semi-automatic process to filter out queries not related to Java, several still queries remained

baselines, including the state-of-art BIKER (Huang et al. 2018) (Section 5.5). We show evaluation for the baselines considering three programming languages and four relevance factors. Moreover, we explore different weights for each factor for each language and show which factors have more importance for each language. We contrast the recommended results against the ground truth (Section 5.1) and use four classical evaluation metrics to measure the performance (Section 5.2). We also show how we leveraged three state-of-art API recommendation systems – BIKER (Huang et al. 2018), NLP2API (Rahman and Roy 2018) and RACK (Rahman et al. 2016) to recommend API Classes for queries (i.e., task description), which are later used to compose one of the relevance factors (API Class factor).

Furthermore, we perform a user study (Section 5.6) 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 four research questions using our experiments:

RQ1: How different IR techniques perform in retrieving candidate Q&A Pairs for given programming tasks written in natural language?

RQ2: How does CROKAGE perform compared to other baselines, including the state-of-art BIKER, in retrieving relevant answers for given programming tasks?

RQ3: To what extent do the factors individually influence the ranking of candidate answers?

RQ4: Can CROKAGE provide more comprehensible solutions containing code and explanations for given queries (task descriptions) compared to those of the state-of-art, BIKER?

5.1 Ground Truth Generation

We construct a ground truth for three languages: Java, Python and PHP. We first select 115 Java programming tasks (i.e., queries) from three Java tutorial sites: (Java2s 2020), (BeginnersBook 2020) and KodeJava (Saryada 2020). 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 (Google Inc 2020), Bing (Microsoft Inc 2020) and Stack Overflow search (Stack Exchange Inc 2020). We pre-process each query by removing stop words, punctuation symbols, numbers and small words (length smaller than 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 [15]. 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 views.

For each of the 115 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 6,558 answers. Then, two professional developers manually evaluate the 6,558 answers by rating each answer in Likert scale from 1 to 5 according to the following criteria:

[15] we append to the query "site:stackoverflow.com"

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. If they are in doubt whether the code works, they manually run it. After the evaluation, 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 consider an answer as relevant if its average rating is equal or higher than 4. Using this criteria, we obtain 1,743 relevant answers namely *goldSet* for the 115 queries. 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 (Landis and Koch 1977).

We repeat the same process for PHP language. For this, we randomly select 10 generic purpose queries (e.g., "Append string to a text file") from the 115 Java queries. The process produces 201 answers which are manually analyzed by the two professional developers. We apply the same criteria used for Java to build the *goldSet* (i.e., select answers with mean Likert $>= 4$), resulting 153 relevant answers. We measure kappa before and after the agreement phase, obtaining 0.1261 and 0.4792 respectively (p-value < 0.05). These values indicate that the agreement improves from slight to moderate (Landis and Koch 1977).

These procedures have taken 94 man hours. The languages were chosen based on their popularity and the knowledge of the participants. Java and PHP fit to those restrictions. The number of questions were also restricted by the availability of the participants to evaluate the answers.

Differently from Java and PHP, we adopt an automatic process for Python language. We leverage a manually curated dataset previously constructed by a Yin et al. (2018) containing a set of triples composed of an intent (i.e., query), its corresponding Stack Overflow ID and a source code snippet that implements the intent. To construct our ground truth, we use a series of filters and heuristics over their triples to select relevant answers for each intent. For this, we first collect a total of 2.563 triples (i.e., the intent + the SO ID + the code snippet) of their dataset. Then, for each triple, we extract the method calls of its code snippet using appropriate regular expressions. We then use the ID information to retrieve the corresponding Stack Overflow thread (i.e., the question and its answers) and filter the answers containing upvotes and code segments (i.e., containing `<code>` and `<pre>` tags). We also assure that the code segment of each answer contain at least one method call. From the remaining answers, we use appropriate regular expressions to extract their method calls and select the answers containing at least M methods call in common with the intent. After manual investigation, we find that $M = 1$ is a reasonable value to filter relevant answers. This process results in 4,691 relevant answers (i.e., *goldSet*) for 1,680 queries (i.e., programming tasks).

After building the *goldSet* for all queries, for the three languages, we construct two sets for each language namely *training* and *testing*, each containing 50% of their queries

along with their *goldSets*. We use these two sets later to train and test our approach and the baselines (Section 5.5), as well as to compare different IR techniques (Section 5.4).

5.2 Performance Metrics

We choose four performance metrics commonly adopted by related literature (Xu et al. 2017; Rahman and Roy 2018; Rahman et al. 2016; Rahman and Roy 2017; Huang et al. 2018). Herein, an item is relevant if it is correct. Since in our work the items being evaluated are Stack Overflow answers, we consider an answer as relevant if it is contained in our *goldSet* (Section 5.1). The metrics are described as follows:

Top-K Accuracy (Hit@K) the percentage of search queries of which at least one recommended item (e.g., Stack Overflow answer) is relevant within the Top-K results. The metric can be defined as follows:

$$Hit@K(Q) = \frac{\sum_{q \in Q} isRelevant(q, K)}{|Q|} \tag{11}$$

where $isRelevant(q, K)$ returns 1 if there exists at least one relevant answer in the Top-K results, or 0 otherwise. Q refers to the set of queries (i.e., task descriptions) used in the experiment. The final result $Hit@K(Q)$ (in percentage) is in [0, 1].

Mean Reciprocal Rank (MRR@K) Reciprocal Rank is the multiplicative inverse of the rank of the first relevant answer recommended within the Top-K results. The Mean Reciprocal Rank (i.e., MRR) is the average of Reciprocal Ranks of a set of queries Q, defined as follows:

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank(q, K)} \tag{12}$$

where $rank(q, K)$ denotes the rank (i.e., position) of the first relevant answer from a ranked list of answers of size K and $|Q|$ denotes the number of queries. The division $\frac{1}{rank(q,K)}$ relies in the range of [0,1], where 1 means that a relevant answer was found at the first position of the list, while 0 means that no relevant answer was found within the list of size K. The MRR@K also relies in [0,1]. The higher its value is, the closest the recommend relevant answers are to the top positions.

Mean Average Precision (MAP@K) the mean of all *Average Precisions* for a set of queries Q. *Average Precision (AP@K)* averages the *Precision@K* of all relevant items (i.e., Stack Overflow answers) within the Top-K results for a search query. *Precision@K* is the precision of every relevant item in the ranked list. The metric MAP@K can be defined as follows:

$$MAP@K(Q) = \frac{\sum_{q \in Q} AP@K(q)}{|Q|} \tag{13}$$

where q denotes every single query and $AP@K(q)$ is the Average Precision of q, which is defined as:

$$AP@K(Q) = \frac{1}{T_{gt}} \sum_{i=1}^{k} P_k \cdot f_k \tag{14}$$

where T_{gt} denotes the total number of relevant answers in the gold set for a query. P_k refers to the precision at k^{th} result while f_k represents the relevance function of the k^{th} result in the ranked list of recommended answers, whose return is 1 if the result is relevant or 0 otherwise.

Mean Recall (MR@K) the average of all *Recall@K* for a set of queries Q. *Recall@K* is the percentage of relevant answers recommended within the Top-K results. MR@K can be defined as follows:

$$MR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|GS(q) \bigcap recommended(q, K)|}{|GS(q)|} \tag{15}$$

herein, GS(q) denotes all the relevant answers (i.e., goldSet) for the query q in the set of queries Q, while $recommended(q, K)$ refers to the Top-K recommended answers by the technique for the query q. The MR@K is also a percentage value in [0, 1] and the higher it is, the more the technique is able to recommend relevant answers within the Top-K results.

5.3 Obtaining API Classes for Natural Language Queries

In order to obtain the API classes for a given query (i.e., task description), we use three state-of-art API recommendation systems – BIKER (Huang et al. 2018), NLP2API (Rahman and Roy 2018) and RACK (Rahman et al. 2016). These tools mine the knowledge of Stack Overflow and provide a list of relevant API classes for programming task written in natural language. We investigate their combination to provide API classes in such a way to obtain the highest performance. For this, we test 308 queries using the dataset and the manually prepared ground truth of NLP2API (Rahman and Roy 2018). For each query, we use all the possible combination of the tools (a total of 15) considering their order to recommend the top-10 API Classes and extract the four metrics (i.e., Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision, and Mean Recall) by contrasting the recommendations with the ground truth. For example, for the combination BIKER + NLP2API + RACK we collect the first class from BIKER, the second from NLP2API and the third from RACK, disconsidering the duplications. We repeat the process until obtain the top-10 API classes for each query. The results of this process are showed in Table 4. We observe that BIKER alone has the worse performance, while the combination of the three tools has the best performance with similar values for the metrics. Thus, we choose the combination NLP2API + RACK + BIKER with 0.83 Top-K Accuracy, 0.53 Mean Reciprocal Rank, 0.46 Mean Average Precision, and 0.47 Mean Recall and then employ it to provide API Classes in CROKAGE (Fig. 2c-2) for Java language, since the tools are limited to Java.

5.4 Evaluation of IR Techniques to Select Candidate Q&A Pairs

RQ1: *How different IR techniques perform in retrieving candidate Q&A Pairs for given programming tasks written in natural language?*

Before selecting answers to compose the solution for a programming task, we need to reduce the search space for candidate Q&A pairs (Fig. 2c-1). After obtaining the candidate Q&A pairs, we extract their answers and contrast against our *goldSet*. CROKAGE considers the search for candidate Q&A pairs as an information retrieval problem. In order to obtain appropriate candidate Q&A pairs, we compare the performance of 11 IR techniques, including BM25 (Robertson and Walker 1994) (Section 3.3.1),[16] which are described as follows:

[16]herein we test the IR techniques using their default parameters. In the case of BM25, the default parameters are: k=1.2 and b=0.75

Table 4 Performance of the combination of three API extractors

Approach	Hit	MRR	MAP	MR
BIKER	0.52	0.37	0.35	0.23
NLP2API	0.73	0.49	0.44	0.38
RACK	0.74	0.47	0.42	0.40
BIKER + RACK	0.76	0.45	0.41	0.41
RACK + BIKER	0.75	0.48	0.43	0.41
BIKER + NLP2API	0.80	0.47	0.42	0.43
NLP2API + BIKER	0.79	0.52	0.46	0.42
RACK + NLP2API	0.80	0.50	0.44	0.46
BIKER + NLP2API + RACK	0.81	0.50	0.44	0.47
BIKER + RACK + NLP2API	0.82	0.50	0.44	0.47
NLP2API + RACK	0.81	0.52	0.45	0.46
RACK + BIKER + NLP2API	0.83	0.51	0.44	0.47
NLP2API + BIKER + RACK	0.82	0.53	0.46	0.47
RACK + NLP2API + BIKER	0.83	0.52	0.45	0.48
NLP2API + RACK + BIKER	0.83	0.53	0.46	0.47

Boolean the general boolean model defines a query Q as a Boolean expression as follows:

$$Q = (W_1 \ \vee \ W_2 \vee \ W_3 ...) \wedge ... \ \wedge (W_i \ \vee \ W_{i+1} \vee \ W_{i+2} \ ...) \tag{16}$$

and a set of documents D as $D = \{D_1, D_2 ... D_n\}$, where W_i is true for D_j when $t_i \in D_j$ and t_i is a term from the index of terms, which is composed by the terms of all documents. The model scores documents that satisfy Q in two stages: first it selects the set of documents S_k that satisfy W_j. That is, $S_k = \{D_i | W_j\}$. Then, the model selects the final set of documents F that satisfy Q by the formula:

$$F = (S_1 \ \cup \ S_2 \ \cup \ S_3 \ ...) \ \cap \ ... \ \cap \ (S_i \ \cup \ S_{i+1} \ \cup \ S_{i+2} \ ...) \tag{17}$$

We adopt the simplest form of the boolean model where the score is only based on whether the query terms match or not. That is, our implementation represents Q as $(W_1 \ \vee \ W_2 \vee \ W_3 ...)$. Consequently, the final set of documents F has the form $(S_1 \ \cup \ S_2 \ \cup \ S_3 \ ...)$. This model does not require the presence of the full text for selecting documents and it gives terms a score equal to their query boost. In our experiments, we used the default value of the boost, which is 1.

Classic this model is an extension of TF-IDF model. The scoring function is calculated as follows:

$$score(q, d) = \sum_{t \in q} (tf(t \in d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t, d)) \tag{18}$$

where $t.getBoost()$ is a relevance factor to boost a term t in the query q (we adopt the default value 1). $norm(t, d)$ is an boost factor that depends on the number of tokens of this field in the document. Despite shorter fields contribute more to the score, specifying a field during a search is optional. We adopted the default implementation containing only one

field. $tf(t \in d)$ represents the number of times a term t appears in the document d and is calculated as $\sqrt{freq(t)}$. The idf of a term t is calculated as follows:

$$idf(t) = 1 + log\left(\frac{docCount + 1}{docFreq + 1}\right) \tag{19}$$

where $docFreq$ stands for the number of documents in which the term t appears, while $docCount$ stands for the total number of documents in the search space.

Jelinek Mercer and Dirichlet Priors This model belongs to the family of smoothing methods (Zhai and Lafferty 2004) that considers the probability of words in a document. Smoothing methods basically discount the probability of words seen in a document and re-allocate extra probabilities in such a way that unseen words have non-zero probability. Jelinek Mercer model calculates the probability of a word w in a document d as follows:

$$p(w|d) = (1 - \lambda)\frac{c(w, d)}{|d|} + \lambda * p(w|C) \tag{20}$$

where $\frac{c(w,d)}{|d|}$ is the maximum likelihood estimate of a unigram language model.[17] That is, the count of words in the document $c(w, d)$, divided by the document length $|d|$. $p(w|C)$ stands for the probability of a word w based on the collection of documents C. Lambda (λ) is a smoothing parameter whose value is in [0, 1] and its optimal value depends on both the collection and the query. High values of λ tends to retrieve documents containing all query words, while low values are more disjunctive, consequently suitable for long queries. Considering that the size of our queries varies, we adopted the intermediate value $\lambda = 0.5$ in our experiments. Dirichlet Priors is similar to Jelinek Mercer Similarity. However, while in the Jelinek Mercer the language model is controlled by the fixed smoothing parameter λ, the Dirichlet's language model is controlled by a dynamic smoothing parameter μ. The smoothing normalizes score based on size of the document. That is, it applies less smoothing for larger documents since larger documents are more complete language model and require less adjustment. The model calculates the probability of a word w in a document d as follows:

$$p(w|d) = \frac{c(w; d) + \mu p(w|C)}{|d| + \mu} \tag{21}$$

where $|d|$ is the total count of words in the document d, $c(w; d)$ denotes the frequency of word w in d and $p(w|C)$ is the probability of a word w based on the collection of documents C, normalized by the smoothing parameter μ. We adopt the default value of μ, which is 2000.

Axiomatic initially proposed by Fang and Zhai (2005), the axiomatic approach seeks to formally define a set of desired properties (i.e., constraints) of a retrieval function to guide the search in order to find a function that satisfies all the properties. They modified existing retrieval functions to satisfy some constraints and derived six retrieval functions to calculate the score between a query Q and a document D as shown in Table 5. Herein, $|D|$ and $|Q|$ are the lengths of document D and the query Q and C_t^Q is the count of the term t in document D. TF and LN stands for term frequency and document length components respectively, while LW and EW represent the IDF component and γ represents the gamma component. We show how these components are calculated in Table 6 and describe them as follows: ln

[17] The simplest form of language model that disconsiders all conditioning context, and estimates each term independently

Table 5 Derived Retrieval Functions of Axiomatic Approach

Function	Formula
Axiom. F1EXP	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF(C_t^Q) * LN(\|D\|) * EW(t))$
Axiom. F1LOG	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF(C_t^Q) * LN(\|D\|) * LW(t))$
Axiom. F2EXP	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF\_LN(C_t^Q, \|D\|) * EW(t))$
Axiom. F2LOG	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF\_LN(C_t^Q, \|D\|) * LW(t))$
Axiom. F3EXP	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF(C_t^Q) * EW(t) - \gamma(\|D\|, \|Q\|)$
Axiom. F3LOG	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF(C_t^Q) * LW(t) - \gamma(\|D\|, \|Q\|)$

is the natural logarithm (base e), N is the total number of documents in the corpus, $df(t)$ is the number of documents containing term t and $avdl$ is the average document length in the corpus. k and s are two parameters where $0 \leqslant k \leqslant 1$ and $0 \leqslant s \leqslant 1$.

In order to answer RQ1, we first implement the 11 IR techniques using Lucene (Apache 2020). Then, we adapt CROKAGE run partially and only return the candidate Q&A pairs, interrupting its execution just after the selection of the candidate Q&A pairs by the IR technique (Fig. 2c-1). We run CROKAGE for each query of the training set (Section 5.1), considering each adopted language (i.e., Java, Python and PHP) and each IR technique. For each IR technique and language, we collect the top-10 recommended candidate Q&A pairs and contrast their answers against the *goldSet* of that language. We measure the performance using the four performance metrics (i.e., Top-K Accuracy, Mean Average Precision, Mean Recall, and Mean Reciprocal Rank), as showed in Table 7.

Answering RQ1: According to our findings, the models Boolean, Dirichlet priors, Axiomatic F3LOG, and Axiomatic F3EXP perform the worst, whereas the models Classic, Jelinek Mercer, Axiomatic F1EXP, BM25, and Axiomatic F1LOG perform the best. In particular, BM25 (Robertson and Walker 1994) achieves the top-3 Accuracy (i.e., Hit) and Mean Reciprocal Rank (i.e., MRR) in two out of the three languages (i.e., Java and Python) and the top-1 Accuracy for Java, justifying our choice as the adopted IR technique for CROKAGE.

Discussion Despite some IR techniques like Classic, Jelinek Mercer, Axiomatic F1EXP, BM25, and Axiomatic F1LOG show better performance compared to the others in general, we conclude that the choice of the technique to retrieve candidate answers for programming tasks must take into consideration the programming language. Compared with Axiomatic F1EXP for example, BM25 performs better for Python but worse for PHP.

Table 6 Axiomatic Formulas' Components

Component	Formula
TF	$TF(i) = 1 + ln(1 + ln(i))$
LW	$LW(t) = ln\frac{N+1}{df(t)}$
EW	$EW(t) = \left(\frac{N+1}{df(t)}\right)^k$
LN	$LN(i) = \frac{avdl+s}{avdl+i \cdot s}$
TF_LN	$TF\_LN(i,j) = \frac{i}{i+s+\frac{s \cdot j}{avdl}}$
γ	$\gamma(i,j) = \frac{(i-j) \cdot i \cdot j}{avdl}$

Table 7 Results of 11 Information Retrieval techniques in selecting candidate answers for programming tasks for three programming languages in terms of Hit@K, MRR@K, MAP@, and MR@K

TRM	Java				Python				PHP			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
Boolean	0.16	0.08	0.07	0.02	0.46	0.25	0.25	0.41	0.00	0.00	0.00	0.00
Dirichlet priors	0.26	0.10	0.09	0.05	0.36	0.20	0.20	0.29	0.60	0.11	0.10	0.05
Axiom. F3LOG	0.36	0.16	0.16	0.06	0.59	0.37	0.35	0.48	0.60	0.18	0.17	0.07
Axiom. F3EXP	0.36	0.18	0.16	0.07	0.57	0.36	0.34	0.47	0.80	0.20	0.20	0.09
Axiom. F2EXP	0.40	0.20	0.17	0.08	0.73	0.50	0.47	0.65	0.60	0.35	0.27	0.08
Axiom. F2LOG	0.41	0.20	0.17	0.08	0.73	0.49	0.47	0.64	0.60	0.35	0.25	0.08
Classic	0.45	0.15	0.14	0.08	0.77	0.51	0.49	0.66	0.60	0.60	0.43	0.13
Jelinek Mercer	0.43	0.21	0.16	0.07	0.77	0.53	0.50	0.67	0.60	0.11	0.10	0.05
Axiom. F1EXP	0.45	0.25	0.22	0.07	0.70	0.47	0.45	0.60	0.80	0.53	0.49	0.13
Axiom. F1LOG	0.48	0.24	0.21	0.08	0.71	0.47	0.45	0.61	0.80	0.63	0.54	0.13
BM25	0.50	0.23	0.19	0.08	0.76	0.51	0.49	0.65	0.60	0.34	0.26	0.13

Previous works in Software Engineering also leveraged the power of BM25 to reduce the search space for candidate Stack Overflow Q&A. Ahasanuzzaman et al. (2016) tune the two BM25 parameters (i.e., k and b) and find that $k = 0.05$ and $b = 0.03$ gives the best performance. However, a replication of their work (Silva et al. 2018) could not assert their findings. We then investigate a range of values for both parameters and their combination in such a way to recommend candidate Q&A pairs with the best performance. That is, for which the Top-K Accuracy (i.e., Hit@K), Mean Reciprocal Rank (i.e., MRR@K), Mean Average Precision (i.e., MAP@K) and Mean Recall (i.e., MR) are the highest respectively. For this, we load the training set (Section 5.1) of each programming language (i.e., Java, Python and PHP) and run CROKAGE multiple times to search for relevant answers for the queries of the training set (Section 3.3), varying the values k and b. We tested a reasonable range for both: k in [0.5, 1.5] and b in [0, 1], with the increment of 0.1. We collect the top-10 results of each run and contrast them against the *goldSet*. We find that the values of k and b with best performance not only differs from the ones of Ahasanuzzaman et al. (2016), but also varies according to the language, as shown in Table 8. These optimal values are then used for next stages of our experiments.

5.5 Experimental Results for the Retrieval of Relevant Answers

RQ2: *How does CROKAGE perform compared to other baselines, including the state-of-art BIKER, in retrieving relevant answers for given programming tasks?*

Table 8 Optimal values for BM25 parameters after tuning

Language	Optimal value for k	Optimal value of b
Java	1.2	0.9
PHP	1.3	0.6
Python	0.5	0.9

To answer RQ2, we calibrate the parameters (including the factors' weights) of CROKAGE using the training set (Section 5.1) of each programming language (i.e., Java, Python and PHP), choosing the ones for which the Top-K Accuracy (Hit@K), Mean Reciprocal Rank (MRR@K), Mean Average Precision (MAP@K) and Mean Recall (MR) are the highest, respectively. For this, we run CROKAGE multiple times to search for relevant answers for the queries of the training set (Section 3.3), varying the value of each parameter, and contrasting the results against the *goldSet*. Table 9 shows how we vary the parameters and the found optimal values for them. Note that we merge the columns for Java and PHP because the optimal combinations of parameters were the same, except for *topApiClasses* and *apiWeight*, which are not applicable for PHP. After discovering the optimal values (i.e., for which the Hit@K, MRR@K, MAP@K and MR@K are the highest) of the parameters for each language, we calibrate CROKAGE with them and construct ten other 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 in this baseline, we adapt CROKAGE to run partially and only return the candidate answers, interrupting its execution just after the selection of the candidate Q&A pairs by BM25 (Fig. 2c-1).

BM25 + Factors we build four baselines representing the relevance factors (Section 3.3.3, Fig. 2c-2): BM25 + API Class, BM25 + Method, BM25 + fastText, and BM25 + TF-IDF. For this, we preserve the weight associated to the baseline and set the other three weights (Formula (9)) to zero (e.g., to build *BM25 + Method* baseline we set all factors' weights to zero, except *methodWeight*).

Sent2Vec herein, we replace the corresponding embedding vectors generated by *fastText* (Bojanowski et al. 2017) in our semantic factor (*semScore* - Section 3.3.2) by the

Table 9 CROKAGE's parameters and their descriptions, ranges, variations and the optimal values for Hit@10, MRR@10, MAP@10 and MR@10 for Java, PHP and Python

Parameter	Description	Range	Step	Optimal Value for Java/PHP	Optimal Value for Python
topBm25	Top scored answers in BM25	[50,200]	10	100	60
topApiClasses	Number of top classes extracted from the three API Recommendation Systems combined	[5,30]	5	30*	–
apiWeight	Weight associated with the api score (apiScore)	[0,1]	0.25	0.25*	–
lexWeight	Weight associated with the lexical score (lexScore)	[0,1]	0.25	0.50	0.50
methodWeight	Weight associated with the method score (methodScore)	[0,1]	0.25	1.00	1.00
semWeight	Weight associated with the semantic relevance score (semScore)	[0,1]	0.25	1.00	1.00

*Applicable only for Java

embedding vectors generated by *Sent2Vec* (Pagliardini et al. 2017). We then build two baselines: BM25 + *Sent2Vec* and a version of CROKAGE with *Sent2vec* (instead of *fastText*) called CROKAGEs2vec. *Sent2vec* is an extension of *fastText* that represents sentences instead of words in a vector space. Differently from *fastText* that predicts from character sequences to target words, *Sent2vec* predicts from word sequences to target words. Furthemore, unlike *fastText* that misses word ordering, *Sent2vec* learns source embeddings for n-grams of words. Thus, it can distinguish two sentences that could have the same word embeddings representation but having completely different meanings such as "Convert Long to Integer" and "Convert Integer to Long". In order to generate the vectors for sentences, we first build *Sent2vec* models, one for each language (i.e., Java, Python and PHP). In this case, each sentence is represented by the concatenation of the pre-processed fields *title* of the question and *body text* of the answer of each Q&A pair[18]. We use the same customized parameters as used for constructing the *fastText* models (Section 3.2) and set the n-grams parameter to three. Then, for each language, we use its model to learn the vector representations for all sentences of the vocabulary and for all queries of ground truth.

After learning the *Sent2vec* vectors for all sentences, we calculate the semantic score between a Q&A pair P and a task description T (i.e., the query) using the cosine distance between their vectors as follows:

$$semScore(P,T) = \frac{Vec1 \cdot Vec2}{|Vec1| \times |Vec2|} = \frac{\sum_1^n Vec1_i \times Vec2_i}{\sqrt{\sum_1^n Vec1_i^2} \times \sqrt{\sum_1^n Vec2_i^2}} \quad (22)$$

where $VecK$ is the 100-dimensional vector representing P or T and $VecK_i$ is the element of the vector $VecK$ at position i.

BERT herein, we replace the corresponding embedding vectors generated by *fastText* (Bojanowski et al. 2017) in our semantic factor (*semScore* - Section 3.3.2) by the embedding vectors generated a technology developed by Google called Bidirectional Encoder Representations from Transformers (a.k.a. *BERT*) (Devlin et al. 2018). Similarly to *Sent2vec* above, we build two baselines: BM25 + *BERT* and CROKAGE with *BERT* (instead of *fastText*) namely CROKAGEbert. *BERT* also represents sentences of words in a vector space. However, the technology reads the entire sequence of words at once, unlike directional models which read the sequence of words sequentially (left-to-right or right-to-left). In order to generate the vectors for the sentences, we leverage a pre-trained model provided by *BERT's* research team.[19] The model was trained on BooksCorpus and Wikipedia for English words and has the embeddings size parameter (i.e., vector size) of 768. We use the pre-trained model to learn the vector representations for all sentences of our vocabulary (including the ground truth) for the three languages the same way as for *Sent2vec*. That is, for Q&As, the sentence is a composition of the *question title* + *body text* of the answer. The semantic score (i.e., *semScore*) between two given sentences is calculated by the cosine distance as in Formula (22).

Like for CROKAGE's parameters (i.e., Table 9), we calibrate the weights for *Sent2vec* and *BERT* obtaining the weights of each parameter as showed in Table 10. The *apiWeight* remains the same for Java (i.e., 0.25) while the *topBm25* parameter remains the same for all languages. We also keep the range (i.e., [0,1]) and variation (i.e., 0.25) during the training.

[18] Although the title of the Q&A pair alone could represent the query intent, our output is the answer, thus we concatenate the *title* and *body text* of the answer in order to match the query with the answers of the candidate pairs

[19] https://bit.ly/3gevm22

Table 10 Optimal weight values for lexWeight (lex), methodWeight (method) and semWeight (sem) for Sent2Vec and BERT for Java, PHP and Python

	Java			Python			PHP		
	lex	method	sem	lex	method	sem	lex	method	sem
Sent2Vec	0.75	1.00	0.50	0.25	0.50	0.75	0.75	1.00	0.50
BERT	0.25	1.00	0.50	0.50	0.75	0.25	0.25	0.75	1.00

After building all baselines, we run CROKAGE to search for relevant answers (Section 3.3) for each baseline against the queries of 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, K=5 and K=1. Tables 11, 12 and 13 show the metrics for all the baselines, including the state-of-art BIKER (Huang et al. 2018), for K=10, K=5, and K=1 respectively.

The results of Tables 11, 12 and 13 show that the baselines perform similarly compared to the others for the three values of K. But, the higher the value of K, the more answers are evaluated, and hence the more the metrics can reflect the strength of the baseline. Values too high (e.g., > 10) however, might not fit to our domain problem, since developers tend not browse too many answers when looking for a solution to their programming tasks. We believe that K=10 may be an adequate trade-off. Thus, in order to assess the differences among the baselines, we set K=10 for the further experiments.

Although CROKAGE performs better than all baselines in terms of the metrics, we investigate whether this difference is statistically significant. For this, we employ the non-parametric Wilcoxon signed-rank test (Wilcoxon 1945) on the paired metrics (K=10) and calculate the effect size with $r = Z/\sqrt{n}$ (Fritz et al. 2012). The results containing the comparison between CROKAGE and each baseline are shown in Table 14.

Table 11 Performance of CROKAGE and other baseline methods in terms of Hit@10, MRR@10, MAP@10 and MR@10 for Java, Python and PHP

	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
BIKER	0.16	0.11	0.11	0.01	–	–	–	–	–	–	–	–
BM25 + API Class	0.58	0.18	0.17	0.10	–	–	–	–	–	–	–	–
BM25 + BERT	0.54	0.25	0.22	0.08	0.37	0.14	0.14	0.19	0.06	0.12	0.13	0.03
BM25 + Sent2Vec	0.49	0.22	0.20	0.08	0.71	0.47	0.44	0.54	1.00	0.58	0.43	0.17
BM25	0.56	0.22	0.22	0.13	0.80	0.54	0.52	0.72	0.80	0.44	0.44	0.12
BM25 + Method	0.72	0.40	0.36	0.16	0.54	0.45	0.43	0.48	0.60	0.32	0.29	0.07
BM25 + FastText	0.67	0.39	0.34	0.13	0.72	0.45	0.41	0.56	1.00	0.55	0.48	0.16
BM25 + TF-IDF	0.63	0.34	0.32	0.16	0.81	0.49	0.47	0.71	1.00	0.61	0.52	0.21
CROKAGEbert	0.74	0.43	0.40	0.20	0.82	0.53	0.51	0.74	1.00	0.73	0.61	0.22
CROKAGEs2vec	0.77	0.44	0.42	0.20	0.83	0.58	0.55	0.73	1.00	0.90	0.72	**0.27**
CROKAGE	**0.81**	**0.55**	**0.49**	**0.22**	**0.83**	**0.59**	**0.56**	**0.74**	**1.00**	**0.90**	**0.80**	0.23

Table 12 Performance of CROKAGE and other baseline methods in terms of Hit@5, MRR@5, MAP@5 and MR@5 for Java, Python and PHP

	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
BIKER	0.16	0.11	0.11	0.01	–	–	–	–	–	–	–	–
BM25 + API Class	0.35	0.15	0.15	0.04	–	–	–	–	–	–	–	–
BM25 + BERT	0.44	0.24	0.22	0.06	0.21	0.12	0.12	0.09	0.40	0.09	0.09	0.02
BM25 + Sent2Vec	0.35	0.21	0.20	0.04	0.61	0.46	0.44	0.42	1.00	0.58	0.58	0.08
BM25	0.40	0.20	0.20	0.08	0.73	0.53	0.52	0.61	0.80	0.44	0.43	0.10
BM25 + Method	0.56	0.38	0.38	0.10	0.54	0.44	0.43	0.42	0.60	0.32	0.30	0.07
BM25 + FastText	0.61	0.38	0.36	0.09	0.62	0.44	0.42	0.42	0.80	0.52	0.47	0.11
BM25 + TF-IDF	0.51	0.32	0.31	0.10	0.61	0.44	0.43	0.43	1.00	0.61	0.53	0.15
CROKAGEbert	0.60	0.42	0.39	0.13	0.21	0.12	0.12	0.09	0.40	0.09	0.09	0.02
CROKAGEs2vec	0.63	0.43	0.42	0.13	0.75	0.57	0.55	0.61	1.00	0.90	0.76	0.19
CROKAGE	**0.63**	**0.53**	**0.50**	**0.14**	**0.76**	**0.58**	**0.57**	**0.62**	**1.00**	**0.90**	**0.81**	**0.21**

Answering RQ2: In terms of Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision, and Mean Recall for K=10, CROKAGE outperforms all baselines to retrieve relevant answers for given programming tasks written in natural language (i.e., query) for the three languages (i.e., Java, Python and PHP). In particular, CROKAGE outperforms its two variants CROKAGEbert and CROKAGEs2vec, whose semantic factors are represented by *BERT* and *Sent2Vec* respectively, instead of *fastText*. Statistical tests show the superiority of CROKAGE over all the baselines for Java and Python. However, the tests show no superiority for PHP, which requires more investigation. Compared to the state-of-art BIKER, the above mentioned metrics are 65%, 44%, 38%, and 21% higher respectively in absolute values.

Table 13 Performance of CROKAGE and other baseline methods in terms of Hit@1, MRR@1, MAP@1 and MR@1 for Java, Python and PHP

	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
BIKER	0.07	0.07	0.07	0.01	–	–	–	–	–	–	–	–
BM25 + API Class	0.00	0.00	0.00	0.00	–	–	–	–	–	–	–	–
BM25 + BERT	0.12	0.12	0.12	0.01	0.08	0.08	0.08	0.03	0.00	0.00	0.00	0.00
BM25 + Sent2Vec	0.12	0.12	0.12	0.01	0.37	0.37	0.37	0.17	0.40	0.40	0.40	0.02
BM25	0.09	0.09	0.09	0.01	0.41	0.41	0.41	0.24	0.20	0.20	0.20	0.01
BM25 + Method	0.25	0.25	0.25	0.02	0.38	0.38	0.38	0.19	0.20	0.20	0.20	0.01
BM25 + FastText	0.28	0.28	0.28	0.03	0.33	0.33	0.33	0.16	0.40	0.40	0.40	0.03
BM25 + TF-IDF	0.21	0.21	0.21	0.03	0.36	0.36	0.36	0.19	0.40	0.40	0.40	0.03
CROKAGEbert	0.32	0.32	0.32	0.04	0.41	0.41	0.41	0.21	0.60	0.60	0.60	0.04
CROKAGEs2vec	0.32	0.32	0.32	0.05	0.45	0.45	0.45	0.24	0.80	0.80	0.80	0.05
CROKAGE	**0.46**	**0.46**	**0.46**	**0.06**	**0.47**	**0.47**	**0.47**	**0.25**	**0.80**	**0.80**	**0.80**	**0.05**

Table 14 Effect size for statistical difference between the metrics of CROKAGE and the baselines for K=10

Baseline	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
BIKER	L	L	L	L	–	–	–	–	–	–	–	–
BM25 + API Class	M	L	L	L	–	–	–	–	–	–	–	–
BM25 + BERT	M	L	L	L	L	L	L	L	○	○	○	○
BM25 + Sent2Vec	L	L	L	L	S	M	M	M	○	○	○	○
BM25	M	L	L	M	S	S	S	S	○	○	○	○
BM25 + Method	○	S	M	○	M	M	S	M	○	○	○	○
BM25 + FastText	M	M	M	M	S	M	M	M	○	○	○	○
BM25 + TF-IDF	M	L	L	M	S	M	M	S	○	○	○	○
CROKAGEbert	○	M	M	○	○	S	S	○	○	○	○	○
CROKAGEs2vec	○	M	M	○	○	S	S	○	○	○	○	○

°= not statistically different

S= small effect size

M= medium effect size

L= large effect size

Discussion We analyze the findings for each language as follows:

- Java: CROKAGE is statistically superior than all baselines, including the state-of-art BIKER (Huang et al. 2018). Indeed, the Wilcoxon signed-rank test (Wilcoxon 1945) confirms the statistical superiority of CROKAGE over all baselines for the four metrics. The difference is the highest between CROKAGE and BIKER, with large effect size (Fritz et al. 2012) for all metrics. BM25 alone also performs much superior than BIKER in all metrics. The results suggest that Method, fastText (i.e., Semantic) and TF-IDF (i.e., Lexical) factors individually improve BM25 performance, where Method performs the best. On the other hand, in general, API Class factor, *Sent2Vec* and *BERT* worsen the results when associated with BM25.
- Python: CROKAGE is statistically superior than eight baselines. Contrary to the Java language, the baseline BM25 + BERT shows the worst performance. The difference in performance between CROKAGE and BM25 + BERT is 46%, 45%, 42% and 55% in Hit@K, MRR@K, MAP@K, and MR@K respectively. BM25 alone, CROKAGEbert and CROKAGEs2vec have the highest performances among the baselines and performs similarly, but are still outperformed by CROKAGE. Indeed, the difference between CROKAGE and all baselines is statistically confirmed. The effect size (Fritz et al. 2012) is similar between CROKAGE and BM25 + fastText and between CROKAGE and BM25 + Sent2Vec ranging from small to medium, but small for all metrics between CROKAGE and BM25. The lowest differences in the effect size are between CROKAGE and CROKAGEbert and between CROKAGE and CROKAGEs2vec. Concerning the factors, the results suggest that, for Python language, TF-IDF factor (i.e., BM25 + TF-IDF) is the one with the best performance among the considered factors. Furthermore, the union of the IR technique with the three factors (i.e., Method, fastText and TF-IDF) representing our approach CROKAGE performs better than the IR technique alone.

– PHP: although CROKAGE shows higher metrics compared to the other baselines, this superiority could not be confirmed by the Wilcoxon signed-rank test (Wilcoxon 1945). The difference in performance is the highest between CROKAGE and BM25 + BERT (i.e., variance of 94%, 78%, 67% and 20% in Hit@K, MRR@K, MAP@K and MR@K respectively) and the lowest between CROKAGE and CROKAGEs2vec (i.e., variance of 8% and 4% in MAP@K and MR@K respectively). The results suggest that, similarly to Python language, TF-IDF factor representing the lexical similarity, is the most important factor to retrieve relevant answers for given programming tasks among the considered factors.

In general, we observe that the union of the factors (i.e., CROKAGE) is statistically superior than any proposed factor alone and the versions of CROKAGE with *Sent2vec* or *BERT*. In particular, CROKAGE statistically outperforms the lexical-based factor (i.e., BM25 + TF-IDF), evidencing the efficiency of CROKAGE to fill the lexical gap between the task description (i.e., the query) and the recommended relevant answers.[20] Since in general the *fastText* performed better than *Sent2vec* and *BERT*, we leave the semantic factor of CROKAGE represented by the *fastText* embedding vectors.

Regarding the semantic factors (i.e., *fastText*, *Sent2vec* and *BERT*) in particular, our experiments show that, in general, *BERT* is outperformed by *Sent2vec* and *fastText*. It should be noted, however, that contrary to *Sent2vec* or *fastText*, whose models were trained using the SO dataset, we used a generic model for *BERT* which was trained on BooksCorpus and Wikipedia. Although the provided *BERT* pre-trained model uses much more vector positions (i.e., 768) to represent sentences than our *Sent2vec* or *fastText* models (i.e., 100), we believe that the non domain specific context used to train the *BERT* model could influence its performance. Thus, more investigation with *BERT* technology (Devlin et al. 2018) is warranted using models trained on the SO context to assess whether it can effectively outperform our approach. Since training *BERT* models poses more complexity compared to *fastText* or *Sent2vec*, we leave this investigation to a future work.

We attribute the satisfactory results of CROKAGE's semantic factor not only on the quality of the embeddings generated by *fastText*, but also to the power of the asymmetric relevance (Ye et al. 2016; Mihalcea et al. 2006) in harnessing these embeddings to capture the similarity between two given sentences. Such technique could not be used in *Sent2vec* (Pagliardini et al. 2017) or *BERT* (Devlin et al. 2018) technologies due to their nature in representing sentences instead of words.

RQ3: *To what extent do the factors individually influence the ranking of candidate answers?*

We investigate the individual influence of each of the four factors in the ranking of the relevant answers (Fig. 2c-3) in terms of Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, for K=10 (i.e., considering the top 10 recommendations). For this, we build four configurations of CROKAGE, each having the weight associated to one factor equals to zero (e.g., to build CROKAGE - TF-IDF we set the weight parameter *lexWeight* to zero in Formula (9)). Since the removal of one factor could influence the relative weight of the others, we re-calibrate the weights of the remaining factors in each configuration. Then, we run each configuration (i.e., CROKAGE - *factor*) against our test set queries to search for relevant answers (Section 3.3) and collect the top-10 answers from the recommended Q&A pairs. Like for RQ2, we contrast the recommended answers

[20] Our general conclusions are not statistically confirmed for PHP language, despite supported by the four adopted metrics.

Table 15 Performance of four configurations of CROKAGE (i.e., CROKAGE without factor) in terms of Hit@K, MRR@K, MAP@K, and MR@K, for K=10

Approach	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
CROKAGE - API Class	0.81	0.50	0.46	0.20	–	–	–	–	–	–	–	–
CROKAGE - Method	0.74	0.47	0.40	0.19	0.83	0.57	0.53	0.72	1.00	0.67	0.55	0.25
CROKAGE - fastText	0.75	0.44	0.40	0.20	0.80	0.53	0.51	0.73	1.00	0.90	0.76	0.24
CROKAGE - TF-IDF	0.72	0.44	0.40	0.17	0.72	0.46	0.42	0.57	1.00	0.70	0.60	0.17
CROKAGE	0.81	0.55	0.49	0.22	0.83	0.59	0.56	0.74	1.00	0.90	0.80	0.23

against the *goldSet*, and collect the four metrics (i.e., Hit@K, MRR@K, MAP@K, and MR@K, for K=10) as shown in Table 15. Furthermore, we employ the Wilcoxon signed-rank test (Wilcoxon 1945) on the paired metrics of each configuration of CROKAGE to verify whether the difference is statistically significant, as shown in Table 16.

Answering RQ3: In terms of influencing the ranking of candidate answers:
- API Class factor (for Java): shows to be irrelevant and its absence does not affect the performance significantly. Moreover, the cost associated to its use is high. Therefore, we consider this factor dispensable.
- Method, fastText (i.e., Semantic) and TF-IDF (i.e., Lexical): in general, their individual absence negatively influence the performance of CROKAGE. However, the importance of each factor varies according to the language: for Java, the three factors have similar importance[a], while for Python, TF-IDF is the most important factor. For PHP, although the metrics suggest slight importance for fastText and similar importance for Method and TF-IDF, more investigation is required using a larger test set.

[a]not statistically confirmed for TF-IDF, despite confirmed by the four adopted metrics

Table 16 Effect size for statistical significance between CROKAGE and the four configurations of CROKAGE in terms of Top-K Accuracy, Mean Average Precision, Mean Recall, and Mean Reciprocal Rank for K=10

Approach	Java				Python				Php			
	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR	Hit	MRR	MAP	MR
CROKAGE - API Class	○	○	○	○	–	–	–	–	–	–	–	–
CROKAGE - Method	○	S	M	○	○	S	S	S	○	○	○	○
CROKAGE - fastText	○	M	M	○	S	S	S	○	○	○	○	○
CROKAGE - TF-IDF	○	○	○	○	S	M	M	M	○	○	○	○

° = not statistically different

S= small effect size

M= medium effect size

L= large effect size

Discussion according to the results of the four configurations (Table 15) and the comparison with CROKAGE (Table 16), we complement and confirm the results for RQ2 as follows:

- Java: the API Class factor is the least important factor. The absence of any of the other three factors (i.e., Method, fastText or TF-IDF) negatively impact the performance similarly. The Wilcoxon signed-rank test (Wilcoxon 1945) could not confirm significant difference between the metrics of CROKAGE and CROKAGE - API Class configuration for any metric. The test however, confirms significant difference for MRR@K and MAP@K between CROKAGE and CROKAGE - Method and between CROKAGE and CROKAGE - fastText with effect size (Fritz et al. 2012) ranging from small to medium. The same test fails to confirm statistical difference between CROKAGE and CROKAGE - TF-IDF, despite the difference in the metrics.
- Python: The absence of any of the three factors (i.e., Method, fastText and TF-IDF) significantly worsen the performance with an effect size (Fritz et al. 2012) ranging from small to medium. The impact is similar between fastText and Method factors with a small effect size for three metrics, and higher for TF-IDF with a medium effect size for three metrics.
- PHP: The absence of Method or fastText (i.e., Semantic) factors have small influence on performance to rank candidate answers. The absence of fastText factor in particular worsen the performance in 4% in MAP@K but improves the MR@K in 1%. The absence of Method or TF-IDF also negatively influence the performance for MRR@K and MAP@K. No influence in performance is found for Hit@K for any baseline. Furthermore, the Wilcoxon signed-rank test (Wilcoxon 1945) could not confirm statistical difference between CROKAGE and each configuration, confirming the necessity of more investigation for PHP using a larger test set.

In general, we observe an intersection in the power of the factors in the ranking of candidate answers. Besides, the importance of each factor seems to vary according to the language. While in Java language the factors Method, fastText (representing the semantic similarity) and TF-IDF (representing the lexical similarity) has similar importance, Python language shows to be much more affected by the TF-IDF, rather than the other two factors. The prevalence of the lexical factor over the semantic factor observed in Python is also observed in PHP. Such findings suggest that the verbosity of each language may be correlated to the importance of the factors, specially the semantic factor (i.e., *fastText*). Thus, since Java is more verbose than the other two languages, the semantic factor (i.e., *fastText*) influences the performance more for Java than for Python or PHP. Other studies with more languages are necessary to confirm this correlation.

Furthermore, the re-calibration of weights for each configuration of CROKAGE (i.e., CROKAGE - factor) makes the other factors compensate the absence of the missing factor. This compensation occurs more efficiently in CROKAGE - API Class configuration. In this baseline, the union of the other three remaining factors performs statistically equal to CROKAGE. That is, the presence of the API Class factor shows to be irrelevant.

We attribute the unsatisfactory results involving the use of API Classes in ranking candidate answers to four main reasons. First, although we adopted the best combination of the three state-of-art API recommenders (Section 5.3), they still may recommend irrelevant APIs (i.e., the precision is 46%) and may miss around half of the relevant APIs (i.e., the recall is 47%). Second, we combined the recommended APIs in such a way to benefit the APIs recommended first with a smoothing factor of $pos + 2$, where pos is the position of

Table 17 Performance of BM25 + API Class and CROKAGE and their extended configurations (α) where the goldSet contains only answers with APIs

Approach	Hit	MRR	MAP	MR
BM25 + API Class	0.58	0.18	0.17	0.10
BM25 + API Class α	0.58	0.18	0.17	0.11
CROKAGE	0.81	0.55	0.49	0.22
CROKAGE α	0.81	0.55	0.49	0.23

the class within recommended API ranked list, considering only those present in the Q&A pair (Section 3.3.2). Although we tested several scoring functions, it could be possible that other unexplored functions would recommend better Q&A pairs containing important API Classes. Third, although the presence of a recommended API Class in the pair is an indicative of importance for the query problem, it is possible that this API class is being used for a purpose other than the query problem. Fourth, since not all relevant answers contain API classes in their solutions, the API Class factor fails to address these answers.

We investigate the proportion of API classes in our Java *goldSet* and find that out of the 1,743 manually evaluated relevant answers, 117 (6.7%) do not contain API classes. We then investigate whether the API Class factor has better performance when evaluated with a *goldSet* composed only with answers containing API classes. For this, we build two new baselines containing this new *goldSet*: BM25 + API Class α and CROKAGE α. We then run CROKAGE to search for relevant answers (Section 3.3) for each baseline, collect the four metrics and compare the results against their original versions, as shown in Table 17. Although we observe a gain of 1% in MR@K between the siblings versions, which is confirmed by the Wilcoxon signed-rank test (Wilcoxon 1945), we conclude that the use of the API Class factor does not significantly pay off regarding the ranking of candidate answers due to the cost associated in obtaining the API classes pertinent to a query. Even adopting state-of-art tools, the extraction of the API classes consumes considerable resources in terms of time and memory. For RACK, NLP2API and BIKER the consumption is around 874, 100 and 366 seconds and 8.1, 7.2 and 10.3 Gigabytes of RAM to process 57 programming tasks, respectively. Hence, we extract all API classes previously and build a cache to be used in our approach, since invoking the tools to recommend the API classes for CROKAGE would not be feasible, considering that our approach is meant to help developers with their programming tasks in real time.

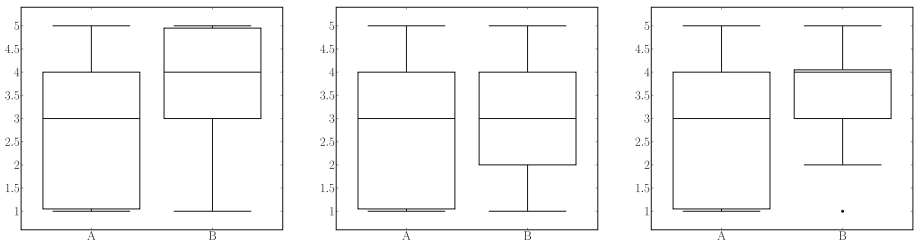
All the experiments were conducted over a server equipped with Intel® Xeon® at 1.70 GHz on 86.4 GB RAM, twelve cores, and 64-bit Linux Mint Cinnamon operating system. After loading the models, CROKAGE spends an average of 0.15, 0.09 and 0.12 seconds to return the candidate answers for each query (i.e., task description) for Java, Python and PHP respectively. Like CROKAGE, we design the other baselines for practical use.[21] Thus, we construct baselines in such a way that each query can be executed in less than one second.

5.6 Comparison with State-of-art Using Developer Study

RQ4: *How does CROKAGE perform to provide comprehensible solutions containing code and explanations for given queries (task descriptions) compared to 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

[21] except BIKER, whose behaviour we do not change



a Relevance of the Suggested Code Examples **b** Benefit of the Code Explanations **c** Overall Solution Quality

Fig. 11 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

to build the ground truth). For a fair comparison, we restrict CROKAGE to Java language, since BIKER is limited to Java. We augment each 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 24 among these questions and apply standard natural language pre-processing. We run BIKER (Huang et al. 2018) 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 evaluation. We asked participants (professional software developers) 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 5.1). We also asked the participants how many years of experience they have as Java programmers and as Professional Software 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 11 represents the participants' evaluation.

Answering RQ4: Our findings show that CROKAGE outperforms BIKER (Huang et al. 2018) for the three considered criteria: relevance of the suggested code examples (Fig. 11-(a)), benefit of the code explanations (Fig. 11-(b)) and the overall solution quality (Fig. 11-(c)). That is, developers prefer CROKAGE over BIKER to find solutions to programming tasks. Furthermore, the results suggest that developers prefer explanations provided by other users (i.e., from Stack Overflow) instead of generic explanations from official API documentations.

Discussion In general, participants reported that Tool B (CROKAGE) was considered better than Tool A (BIKER (Huang et al. 2018)). For the three considered criteria, two of them were reported by users as much superior: relevance of the suggested code examples (Fig. 11a) and the overall solution quality (Fig. 11c). Both approaches showed the same median value (i.e,. 3) for the benefit of the code explanations (Fig. 11b). However CROKAGE showed much less Likert 1 (i.e., 19 against 72) and much more Likert 5 (i.e., 52 against 20). To make sure whether CROKAGE outperforms BIKER in each criteria, we run Wilcoxon signed-rank test (Wilcoxon 1945) on their paired data. 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 $r = Z/\sqrt{n}$ (Fritz et al. 2012).

To get more insight on how users would perceive the potential benefits of CROKAGE, we also conducted a small scale qualitative study to understand how CROKAGE would compare to an approach where users try to find solutions in Stack Overflow using Google as the primary search engine. In this study, we randomly selected 10 questions out the 24 questions selected for the BIKER comparison. We invited seven individuals, and six of them accepted the invitation to analyze the pros and cons of the results returned from CROKAGE and Google. These individuals are former graduate students, except one, with experience on teaching, and an average of 8 years of experience in Java. They should compare the solutions of CROKAGE and the threads returned by Google under the following criteria: the quality of the ranking of the top-5 results, the user experience related to the navigation until the solution, the focus and directness of the solution, the adequacy of code examples and explanations. The answers were written as free text and then we proceeded with open coding (Corbin and Strauss 1990) to extract and quantify the findings. Table 18 synthesizes the results of the open coding and maps the perceptions of each respondent. We can observe a clear indication on the respondents' perception that CROKAGE solutions extracted from Stack Overflow and their respective presentation are simple and direct. Moreover, the organization of CROKAGE's solutions in a clean list has been shown to be adequate. Some quotes of users perception are: *"The list of Crokage's solution prevent the developer getting lost in other discussions present in Stack Overflow threads"*; *"Crokage solutions seems to be more direct because they tend to be more summarized"*. Interestingly, one respondent raised an interesting point: *"Crokage seems to be better suited for beginners because the returned solutions tend to be simpler and more direct, however, as a more experienced developer, I would prefer some threads returned by Google, which provide more insightful and deep solutions"*. Actually, this comment aligns with our intended goal for CROKAGE as a tool that provides solutions that can be easily grasped by everyone, specially those with lower experience on the technology they are trying to get knowledge on. Nonetheless, we also suggest that the parameterization of user profiles, especially considering their level of expertise, seems to be an important feature for a recommendation tool, since some developers may want more elaborated solutions, whereas others may want more direct solution. Despite the good performance of CROKAGE on the above criteria, there is a clear indication that Google has a slightly better performance on finding and ranking the adequate posts, indicating that there is still room for improving CROKAGE's search mechanism. Nonetheless, the users report that although CROKAGE had some inadequate ranking of solutions, these cases were rare considering the 10 proposed questions. These perceptions on the ranking results are consistent with our early experimental results.

Table 18 Qualitative user perception of CROKAGE compared to Google search on Stack Overflow

Approach	R1	R2	R3	R4	R5	R6	Total	
Crokage								
Pros								
	Easy/simple answers	•					•	2
	Quick access to solutions		•		•			2
	Organized solution		•	•	•	•		4
Cons								
	Rare ranking problems	•	•	•		•		4
	Very focused solutions						•	1
Google								
Pros								
	Precise ranking	•	•	•		•		4
	Insightful and deep solutions						•	1
Cons								
	More complex code examples	•					•	2
	More complex access to solutions		•	•	•	•		4

6 Threats to Validity

Threats to Internal Validity are related to the baseline methods, the user study, and the tool usage analysis. 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. However, implementation errors could still exist despite our careful examination. Thus we provide a replication package to the community so that other developers can replicate and check our results. 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 in such a way that each questionnaire has at least five participants with a minimum of six years of experience in Java programming. Besides, we only selected participants who showed interest in participating in the study. Finally, concerning the tool usage analysis, two threats can exist. One is related to implementation errors in the processing of the queries submitted by the community. We mitigate this threat by double checking the implementation of this processing and all the generated outcomes. The other threat is related to the manual classification of the queries, that could produce inaccurate results. In order to mitigate this threat, two professional developers with more than ten years of experience in Java programming independently classify the queries and then discuss and solve the disagreements.

Threats to External Validity relates to the quality of our ground truth and to the generalizability of our results. Concerning the ground truth, we mitigate the threat by covering a wide range of different programming tasks for three popular programming languages. For Java and PHP, we selected the programming tasks from three popular tutorial sites and constructed our *goldSets* by only selecting good quality answers (i.e., Likert equal or higher than 4), which were independently evaluated by two professional developers. For Python, we leveraged a manually curated dataset provided by a previous work (Yin et al. 2018).

We process their dataset to extract the programming tasks and construct the *goldSet* using a series of filters and heuristics to select relevant answers for the programming tasks.

Regarding the generalizability of our results, we identify two threats. The first one relates to our API factor. Although this factor could also be applied to PHP and Python, we could not find tool support for providing API classes for those languages where the input is a natural language query. Thus, we only apply this factor to Java. Nevertheless, we mitigate this threat by adopting multiple similarity factors for the three languages. The second threat relates to the number of queries used for PHP. Although the number of queries used in the tests may have not been sufficient enough to produce statistical difference, we consider that CROKAGE's results can be generalized. The results are obtained in a recent dataset of Stack Overflow composed of 4.1M Java, 3.1M PHP and 3M Python posts. Furthermore, the superiority of CROKAGE over the baselines are confirmed for three programming languages using four performance metrics: Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, which are widely adopted by related literature in software engineering (Xu et al. 2017; Rahman and Roy 2018; Rahman et al. 2016; Rahman and Roy 2017; Huang et al. 2018).

7 Related Work

7.1 Exploring Q&A Features to Mine Knowledge from Stack Overflow

Research over Stack Overflow spans several areas related to software development. Herein, we concentrate on works that explore Q&A features and use information retrieval techniques to mine knowledge from Stack Overflow.

Some works explore features from Stack Overflow Q&A pairs to detect duplicated questions (Zhang et al. 2015; Ahasanuzzaman et al. 2016; Zhang et al. 2017a, b; Hoogeveen et al. 2018; Silva et al. 2018; Diamantopoulos and Symeonidis 2015). (Zhang et al. 2015) build an approach called DupPredictor, which leverages features from Stack Overflow questions to compute a similarity score between a new question and all the others. They adopt as features the title, the body (i.e., description), the tag and a LDA topic of each question. The score between two questions is calculated as the sum of the scores of the four pair of features (i.e, title-title, body-body, tag-tag, and topic-topic), where each pair is associated with a weight. In the end, the duplicated question(s) would ideally be the one(s) with the highest similarity score(s). The other works follow the same principle of scoring and ranking questions to a new question. Ahasanuzzaman et al. (2016) combine the power of the state-of-art information retrieval technique BM25 (Robertson and Walker 1994) and a logistic regression classifier to score and rank the most similar questions to a new question. They claim to have achieved better results than DupPredictor to find duplicates, however a reproduction of both works (Silva et al. 2018) does not assert their findings. Zhang et al. (2017a) leverage features to detect duplicates through embeddings similarity using word2vec, topical Similarity and association rules represented by frequently co-occurred phrases pairs. According to their results, their approach called PCQADup outperforms both previous works (i.e., DupPredictor and Dupe). In another work, Zhang et al. (2017b) analyze the performance of a combination of features to detect the duplicated questions. They find that the combination of the relevance (e.g., provided by BM25), vector similarity (i.e., doc2vec) and association features (i.e., phrases that co-occurs frequently) gives the best performance, with more than 95% of recall rate. Hoogeveen et al. (2018) address

not only the duplicated questions, but also the misflagged ones. They combine traditional machine learning models and deep learning techniques on series of features composed by text and meta data of the questions. They find that meta data features are more effective to find duplicates than textual features. Diamantopoulos and Symeonidis (2015) propose a methodology to find similar questions on Stack Overflow using not only the main features of questions like title, description and tags, but also their code snippets. They calculate the similarity between two questions according to question's fields: for texts (i.e., titles of bodies) they use TF-IDF, for tags they calculate the *Jaccard index* and for snippets they calculate the similarity based on their Longest Common Subsequence (LCS).

Our work share several principles with these works: we also use features and techniques to score and rank posts from Stack Overflow according to their similarity with a target object. For these works, the target object is a question while for CROKAGE is a simple programming task description (i.e., query). They calculate the similarity between a target question and all other questions in order to obtain the candidate duplications. Similarly, we calculate the similarity between the task description and all other Q&A pairs in order to obtain candidate answers. However, our typical input queries carry less information than typical questions. The implication is that our problem turns out to be a bit more challenging because of limited information delivered to the information retrieval techniques.

Similarly to detect duplicated questions, existing studies (Xu et al. 2016; Fu and Menzies 2017; Xu et al. 2018) try to identity duplicated knowledge units on Stack Overflow, which are composed by a question and all their answers. Xu et al. (2016) look for semantically related posts using word embeddings and Deep Learning models, but instead of using human-engineered classifier features, they explore many different types of knowledge links created by users to train their model. They claim to have outperformed traditional methods using word representations. However, their results were questioned by Fu and Menzies (2017) who presented a simpler and faster method based on support-vector machine (SVM). Later, Xu et al. (2018) reproduce both works and recognize that tuned SVM is a better model for such task against a larger and diverse dataset. Like these works, our work also mine the knowledge from Stack Overflow, but instead of applying deep learning techniques, we employ traditional information retrieval techniques.

Other works leverage Stack Overflow Q&A features to recommend relevant Stack Overflow Q&A discussions (Ponzanelli et al. 2013b; Ponzanelli et al. 2014a; 2014b; Ponzanelli et al. 2013a; Ponzanelli et al. 2014b). Ponzanelli et al. (2013b) integrate pertinent Stack Overflow Q&A discussions with the programmer development environment. They leverage lexical features like TF-IDF to match user queries (i.e., task descriptions) with Q&A threads, allowing the user to import the code from the crowd to their IDE. They implement their approach called SEAHAWK in form of a Eclipse plugin (Ponzanelli et al. 2013a). Later on, they improve their work to consider the context of the IDE in the search for relevant Q&A threads (Ponzanelli et al. 2014a). They define eight features to calculate the relation between Stack Overflow discussions and the code in the IDE, ranging from textual features like TF-IDF to social features like question score, accepted score and user reputation. Their tool PROMPTER (Ponzanelli et al. 2014b), allows the developer to define a threshold of similarity between the context of IDE and a Q&A discussion and notifies the developer when this threshold is achieved. Our work is similar to the ones of Ponzanelli et al. regarding exploring different kinds of features to match relevant Q&A discussions on Stack Overflow. However, instead of retrieving pertinent discussions composed by questions and answers, we go further and synthesize solutions containing source code and explanations to programming tasks (i.e., queries).

7.2 Code Example Suggestion

There have been several studies (Nguyen et al. 2016; Bajracharya et al. 2010; Zagalsky et al. 2012; Campbell and Treude 2017; Wang et al. 2016; Gvero and Kuncak 2015; Raghothaman et al. 2016; Gu et al. 2016; Campos et al. 2016; De Souza et al. 2014; McMillan et al. 2011; Rahman et al. 2017; Gu et al. 2018; Huang et al. 2018) that return relevant code against natural language queries. McMillan et al. (2011) 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 (2017) develop a tool that assists users providing suggestions to the queries. Rahman et al. (2017) 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 maximum number of searches in a period of time).

Some works (Nguyen et al. 2016; Gu et al. 2016; Raghothaman et al. 2016) infer API usage sequences for a given task. T2API (Nguyen et al. 2016) learns API usages via graph-based language model. They use a statistical machine translation to associate descriptions and corresponding code. DeepAPI (Gu et al. 2016) composes the associations between the sequence of words in a query and APIs through deep learning. SWIM (Raghothaman et al. 2016) 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 distinguishes code aspects like method and class names. This concern has been also addressed by DeepCS (Gu et al. 2018). 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 (Campos et al. 2014; Lv et al. 2015; Zagalsky et al. 2012; Bajracharya et al. 2010) rely on lexical similarities to retrieve relevant code. Campos et al. (Campos et al. 2014) rank related code documents by applying a combination of Vector Space and Boolean models. The same idea is used by Lv et al. (2015). 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. (2012) propose a tool to retrieve source code based on keywords using TF-IDF to score code documents. Bajracharya et al. (2010) 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 (Huang et al. 2018) is the most related work to ours and we compare our work with theirs in multiple ways, as shown in Section 5.

7.3 Code Explanation Generation

Several early studies (Wong et al. 2013; Rahman et al. 2015; Xu et al. 2017; Chatterjee et al. 2017; Hu et al. 2018; Wong et al. 2015) propose automatic approaches to extract explanations to code. For this, they explore lexical properties usually in combination with strategies like clone detection (Wong et al. 2013, 2015), topics (like LDA) (Rahman et al. 2015), word embeddings (Xu et al. 2017), machine learning (Chatterjee et al. 2017) and deep learning (Hu et al. 2018). Wong et al. (2013) 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. (2015) 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. (2015) 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. (2017) 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. (2017) 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. (2018) 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 comprehensible survey by Wang et al. (2018) 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.

8 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 (i.e., queries). CROKAGE leverages the knowledge stored in Stack Overflow to generate solutions containing source code and explanations for programming tasks written in natural language. For this, CROKAGE first searches for relevant answers in Stack Overflow for a task and then, after obtaining top the quality answers, uses natural language processing on them to compose comprehensible solutions. We evaluate 11 IR techniques to search for candidate Stack Overflow answers for the queries in terms of performance and find that BM25 (Robertson and Walker 1994) performs among the best. Likewise, we evaluate the performance of each of our relevance factors individually. We find that the use of API classes (i.e., API Class factor) does not influence the performance significantly. However, the other three factors combined (TF-IDF, Method and fastText) perform well, filling lexical gap between the task description (i.e., the query) and the recommended relevant answers. Our findings show that

CROKAGE outperforms several other baselines to retrieve relevant answers for programming tasks, including the state-of-art. The effectiveness of CROKAGE to provide quality solutions for programming tasks is demonstrated by a user study. We show that developers prefer CROKAGE over the state-of-art tool to find solutions for programming tasks. Furthermore, the usage analysis of CROKAGE after five months of operation show that most of the developers are satisfied with the provided solutions.

This work has opened up important research directions, as follows:

- *Search Mechanism:* our current search mechanism relies on pre-determined weights associated to each factor to calculate the similarity between each Q&A pair and the query. A recent work of Van Nguyen et al. (2017) instead, dynamically combines lexical and semantic similarity to search for code examples based on a threshold. An investigation is warranted to evaluate whether this flexible weighting scheme could better determine the appropriate weights for each factor by analyzing the query at runtime, instead of using pre-determined values. The challenge is how to estimate the level of lexical and semantic similarity that should be associated to each factor for the query at runtime. However, this idea has potential to generate a better search mechanism and thus outperforms our current one.
- *Quality Analysis of Programming Tasks:* determining the best query among a set of queries representing a programming task is challenging. The developers can use different keywords to express their intent and some keywords may better represent the intent them others. For example, our approach is able to retrieve a relevant result for the query *"how to insert an element in an list in a specific index ?"* in the Top-1 position. However, the following query *"how to add an element in an list in a fixed position ?"* produces the first relevant result in the Top-6 position. A quality analysis of queries could show directions of how to reformulate those queries in such a way to obtain more relevant results. Query reformulation has been proposed before by previous previous works (Nie et al. 2016; Wang et al. 2014; Li et al. 2016; Rahman et al. 2017; Rahman and Roy 2018; Rahman et al. 2016). Our similarity factors (TF-IDF, Semantic and Method) could possibly be leveraged to determine the query quality and propose reformulations.

Acknowledgments 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). We also thank the Brazilian funding agencies, CAPES, CNPq and FAPEMIG for supporting this research. At last, but not least, we thank the participants that worked in the qualitative evaluation of this work.

References

Ahasanuzzaman M, Asaduzzaman M, Roy CK, Schneider KA (2016) Mining duplicate questions in Stack Overflow. In: Proceeding MSR, pp 402–412

An L, Mlouki O, Khomh F, Antoniol G (2017) Stack overflow: a code laundering platform? In: Proceeding SANER, pp 283–293

Apache (2020) Lucene, http://lucene.apache.org/

Baeza-Yates R, Ribeiro-Neto B, et al. (1999) Modern information retrieval, vol 463. ACM Press, New York

Bajracharya S, Ossher J, Lopes C (2010) Searching API usage examples in code repositories with Sourcerer API search. In: Workshop on search-driven development, pp 5–8

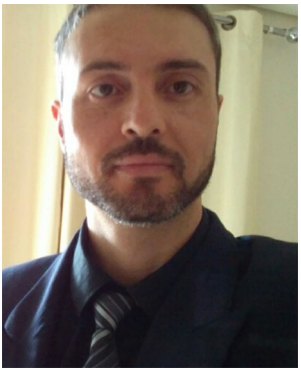
BeginnersBook (2020) BeginnersBook, http://beginnersbook.com

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. TACL 5:135–146

Campbell BA, Treude C (2017) NLP2code: Code snippet content assist via natural language tasks. In: Proceeding ICSME, pp 628–632
Campos EC, Souza LBLD, Maia MA (2014) Nuggets miner: assisting developers by harnessing the Stack Overflow crowd knowledge and the Github traceability. In: Proceeding CBSoft-Tool Session
Campos EC, de Souza LB, Maia MA (2016) Searching crowd knowledge to recommend solutions for API usage tasks. J Softw Evol Process 28(10):863–892
Chatterjee P, Gause B, Hedinger H, Pollock L (2017) Extracting code segments and their descriptions from research articles. In: Proceeding MSR, pp 91–101
Chen C, Xing Z, Liu Y, Ong KLX (2019) Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding, TSE
Ciborowska A, Kraft NA, Damevski K (2018) Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the web. In: Proceeding MSR, pp 94–97
Corbin J, Strauss A (1990) Basics of qualitative research: techniques and procedures for developing grounded theory sage publications
De Souza LBL, Campos EC, Maia MA (2014) Ranking crowd knowledge to assist software development. In: Proceeding Intl. Conf. on Program Comprehension, pp 72–82
Devlin J, Chang M-W, Lee K, Toutanova K (2018) Bert: pre-training of deep bidirectional transformers for language understanding, arXiv:1810.04805
Diamantopoulos T, Symeonidis AL (2015) Employing source code information to improve question-answering in Stack Overflow. In: Proceeding MSR, pp 454–457
Facebook Inc (2020) Word representations in fastText, https://fasttext.cc/docs/en/unsupervised-tutorial.html
Fang H, Zhai C (2005) An exploration of axiomatic approaches to information retrieval. In: Proceeding SIGIR ACM, pp 480–487
Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Trans Int Technol (TOIT) 2(2):115–150
Fritz C, Peter E, Richler J (2012) Effect size estimates: current use, calculations, and interpretation. JEPG 141(1):2–18
Fu W, Menzies T (2017) Easy over hard: A case study on deep learning. In: Proceeding ESEC/FSE, pp 49–60
Google Inc (2020) Google search engine, http://google.com
Gu X, Zhang H, Zhang D, Kim S (2016) Deep API learning. In: Proceeding FSE, pp 631–642
Gu X, Zhang H, Kim S (2018) Deep code search. In: Proceeding ICSE, pp 933–944
Gvero T, Kuncak V (2015) Interactive synthesis using free-form queries. In: Proceeding ICSE, pp 689–692
Hill E, Rao S, Kak A (2012) On the use of stemming for concern location and bug localization in Java. In: Proceeding SCAM, pp 184–193
Hoogeveen D, Bennett A, Li Y, Verspoor KM, Baldwin T (2018) Detecting misflagged duplicate questions in community question-answering archives. In: Proceeding ICWSM, pp 112–120
Hu X, Li G, Xia X, Lo D, Jin Z (2018) Deep code comment generation. In: Proceeding ICPC, pp 200–210
Huang Q, Xia X, Xing Z, Lo D, Wang X (2018) API method recommendation without worrying about the task-API knowledge gap. In: Proceeding ASE, pp 293–304
Java2s (2020) Java2s, http://java2s.com
Jsoup (2020) Java HTML parser, http://jsoup.org
Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174
Li Z, Wang T, Zhang Y, Zhan Y, Yin G (2016) Query reformulation by leveraging crowd wisdom for scenario-based software search. In: Proceedings of the 8th asia-pacific symposium on internetware ACM, pp 36–44
Lv F, Zhang H, Lou J-G, Wang S, Zhang D, Zhao J (2015) Codehow: effective code search based on API understanding and extended boolean model (e). In: Proceeding ASE, pp 260–270
McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usage. In: Proceeding ICSE, pp 111–120
Microsoft Inc (2020) Bing search engine, http://bing.com
Mihalcea R, Corley C, Strapparava C, et al. (2006) Corpus-based and knowledge-based measures of text semantic similarity. In: Aaai 6(2006):775–780
Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013a) Distributed representations of words and phrases and their compositionality. In: Proceeding NIPS, pp 3111–3119
Mikolov T, Chen K, Corrado G, Dean J (2013b) Efficient estimation of word representations in vector space, arXiv:1301.3781
Nasehi SM, Sillito J, Maurer F, Burns C (2012) What makes a good code example?: A study of programming q&a in stackoverflow. In: Procedding ICSM IEEE, pp 25–34

Nguyen T, Rigby PC, Nguyen AT, Karanfil M, Nguyen TN (2016) T2API: synthesizing API code usage templates from English texts with statistical translation. In: Proceeding FSE, pp 1013–1017
Nie L, Jiang H, Ren Z, Sun Z, Li X (2016) Query expansion based on crowd knowledge for code search. IEEE Trans Serv Comput 9(5):771–783
Pagliardini M, Gupta P, Jaggi M (2017) Unsupervised learning of sentence embeddings using compositional n-gram features, arXiv:1703.02507
Ponzanelli L, Bacchelli A, Lanza M (2013a) Seahawk: Stack Overflow in the IDE. In: International conference on software engineering (ICSE), pp 1295–1298
Ponzanelli L, Bacchelli A, Lanza M (2013b) Leveraging crowd knowledge for software comprehension and development. In: Proceeding CSMR, pp 57–66
Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (2014a) Mining Stack Overflow to turn the IDE into a self-confident programming prompter. In: Proceeding MSR, pp 102–111
Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (2014b) Prompter: A self-confident recommender system. In: Proceeding ICSME. IEEE, pp 577–580
Raghothaman M, Wei Y, Hamadi Y (2016) SWIM: Synthesizing what I mean-code search and idiomatic snippet synthesis. In: Proceeding ICSE, pp 357–367
Ragkhitwetsagul C, Krinke J, Paixao M, Bianco G, Oliveto R (2018) Toxic code snippets on Stack Overflow, arXiv:1806.07659
Rahman MM, Roy CK (2017) STRICT: Information retrieval based search term identification for concept location. In: Proceeding SANER, pp 79–90
Rahman MM, Roy CK (2018) Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In: Proceedings ICSME, pp 473–484
Rahman MM, Roy CK, Keivanloo I (2015) Recommending insightful comments for source code using crowdsourced knowledge. In: Proceeding SCAM, pp 81–90
Rahman MM, Roy CK, Lo D (2016) RACK: Automatic API recommendation using crowdsourced knowledge. In: Proceeding SANER, pp 349–359
Rahman MM, Roy CK, Lo D (2017) Rack: Code search in the IDE using crowdsourced knowledge. In: Proceeding ICSE, pp 51–54
Robertson SE, Walker S (1994) Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In: Proceeding ACM SIGIR, pp 232–241
Saryada W (2020) Kodejava, http://kodejava.org
Silva RFG, Paixao KVR, Maia MA (2018) Duplicate question detection in Stack overflow: a reproducibility study. In: Proceeding SANER, pp 572–581
Silva RF, Roy CK, Rahman MM, Schneider KA, Paixao K, de Almeida Maia M (2019) Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In: Proceedings of the 27th international conference on program comprehension, IEEE Press, pp 358–368
Stack Exchange Inc (2020) Stack Overflow search engine, http://stackoverflow.com
Van Nguyen T, Nguyen AT, Phan HD, Nguyen TD, Nguyen TN (2017) Combining word2vec with revised vector space model for better code retrieval. In: Proceeding ICSE IEEE Press, pp 183–185
Wang Y, Feng Y, Martins R, Kaushik A, Dillig I, Reiss SP (2016) Hunter: next-generation code reuse for Java. In: Proceeding FSE, pp 1028–1032
Wang S, Lo D, Jiang L (2014) Active code search: incorporating user feedback to improve code search relevance. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, pp 677–682
Wang X, Peng Y, Zhang B (2018) Comment generation for source code:, State of the art, challenges and opportunities, arXiv:1802.02971
Wilcoxon F (1945) Individual comparisons by ranking methods. Biomet Bull 1(6):80–83
Wong E, Yang J, Tan L (2013) Autocomment: mining question and answer sites for automatic comment generation. In: Proceeding ASE, pp 562–567
Wong E, Liu T, Tan L (2015) Clocom: mining existing source code for automatic comment generation. In: Proceeding SANER, pp 380–389
Xu B, Ye D, Xing Z, Xia X, Chen G, Li S (2016) Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: Proceeding ASE, pp 51–62
Xu B, Xing Z, Xia X, Lo D (2017) Answerbot: automated generation of answer summary to developers technical questions. In: Proceedings ASE, pp 706–716
Xu B, Shirani A, Lo D, Alipour MA (2018) Prediction of relatedness in stack overflow: deep learning vs. svm: a reproducibility study. In: Proceeding ESEM ACM, p 21
Yang D, Martins P, Saini V, Lopes C (2017) Stack overflow in github: any snippets there? In: Proceeding MSR, pp 280–290

Ye X, Shen H, Ma X, Bunescu R, Liu C (2016) From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceeding ICSE, pp 404–415
Yin P, Deng B, Chen E, Vasilescu B, Neubig G (2018) Learning to mine aligned code and natural language pairs from stack overflow. In: Proceeding MSR, ser MSR ACM, pp 476–486
Zagalsky A, Barzilay O, Yehudai A (2012) Example overflow: using social media for code recommendation. In: Proceeding RSSE, pp 38–42
Zhai C, Lafferty J (2004) A study of smoothing methods for language models applied to information retrieval. TOIS 22(2):179–214
Zhang Y, Lo D, Xia X, Sun J-L (2015) Multi-factor duplicate question detection in Stack Overflow. JCST 30(5):981–997
Zhang WE, Sheng QZ, Lau JH, Abebe E (2017a) Detecting duplicate posts in programming qa communities via latent semantics and association rules. In: Proceeding WWW, pp 1221–1229
Zhang WE, Sheng QZ, Shu Y, Nguyen VK (2017b) Feature analysis for duplicate detection in programming qa communities. In: Proceeding ADMA. Springer, New York, pp 623–638



Rodrigo Fernandes Gomes da Silva is a Software Developer at the Federal University of Uberlândia (UFU) since 2009. He received an M.Sc. degree in Software Engineering at UFU in 2014 and a Ph.D. degree also at UFU in 2020. His main research interests include mining software repositories, crowd knowledge and information retrieval. Before becoming a researcher, he worked as developer in software houses and an international company. Currently he develops software projects using cutting edge programming languages and technologies in order to bring innovation to his University.

Chanchal K. Roy is Professor of Software Engineering at the University of Saskatchewan, Canada. As the co-author of the widely used NICAD code clone detection system, he has published more than 170 refereed publications, with many of them in premier software engineering conferences and journals that have been cited more than 6,000 times. Dr. Roy works in the broad area of software engineering, with particular emphasis on software clone detection and management, software evolution and maintenance, and recommender systems in software engineering. He received the most influential paper awards both at SANER 2018 and at ICPC 2018, the Outstanding Young Computer Science Researcher Award by CS-Can/Info-Can in 2018 and the New Researcher Award of University of Saskatchewan in 2019. He extensively serves in conference organizations and conference program committees.

Mohammad Masudur Rahman is a tenure-track Assistant Professor in the Faculty of Computer Science, Dalhousie University. He received his Ph.D. in Computer Science/Software Engineering from the University of Saskatchewan. He also completed his postdoc from the Polytechnique Montreal. Masud is interested in software debugging, code search and code reviews, and he develops intelligent, automated, and cost-effective software solutions to support the developers in these activities. He uses a blend of Software Engineering, Information Retrieval, Machine/Deep Learning, Mining Software Repositories, Natural Language Processing and Big Data Analytics in his research. To date, his works were published at several major venues of Software Engineering (e.g., ICSE, ESEC/FSE, EMSE, ASE, ICSME). Dr. Rahman has been awarded several prestigious awards such as Governor General's Gold Medal, U of S Doctoral Thesis Award, Keith Geddes Award, and Chancellor Gold Medal for his research excellence and outstanding academics. He has been serving as a reviewer for several top Software Engineering journals (e.g., TSE, TOSEM, EMSE, JSS).

Dr. Kevin A. Schneider Professor, Computer Science, University of Saskatchewan (USask) is Director of the Software Research Lab, USask. He is an elected member of the International Federation for Information Processing working group 2.7/13.4 on user interface engineering and has served on numerous program committees and grant selection committees. Previously he was President & CEO of Legasys Corp., a software research and development company and he continues to promote software entrepreneurship and work closely with industry. He has published over 100 papers and supervised/co-supervised dozens of graduate students. His expertise is in advanced software analytics, human computer interaction, visualization, software evolution, software renovation, programming languages, and data-intensive discovery.

Klérisson Paixão received a PhD degree in Computer Science from the Federal University of Uberlandia. He is currently a Staff Software Engineer at TQI and ocassionaly guest lecture on Data Science and Big Data specialization courses at the Pontifical Catholic University of Minas Gerais - Brazil.

Carlos Eduardo de Carvalho Dantas is Assistant Professor at Federal Institute of Triangulo Mineiro (IFTM) since 2014. He received the master's degree (2017) in Computer Science from the Federal University of Uberlândia, Brazil. He is currently a PHD student under the supervision of Professor Marcelo Maia at Federal University of Uberlândia, Brazil. His research interests include Information Retrieval (IR) ranking models such like deep neural networks.

Marcelo de Almeida Maia is Professor at the Faculty of Computing of the Federal University of Uberlândia (UFU). He received his M.Sc. (1994) and Ph.D. (1999) degrees in Computer Science from the Federal University of Minas Gerais (UFMG). He has been awarded with a Research Productivity Grant from CNPq-Brazil (2018), and has published more than 90 referred papers. His interest area is Software Engineering and Programming Languages. Currently, he leads the Intelligent Software Engineering Lab at UFU, and his research interests includes software repository mining, software analytics, recommendation systems for software development, and program comprehension.

Affiliations

Rodrigo Fernandes Gomes da Silva[1] · Chanchal K. Roy[2] · Mohammad Masudur Rahman[2] · Kevin A. Schneider[2] · Klérisson Paixão[1] · Carlos Eduardo de Carvalho Dantas[1] · Marcelo de Almeida Maia[1]

Rodrigo Fernandes Gomes da Silva
rodrigofernandes@ufu.br

Chanchal K. Roy
chanchal.roy@usask.ca

Mohammad Masudur Rahman
masud.rahman@usask.ca

Kevin A. Schneider
kevin.schneider@usask.ca

Klérisson Paixão
klerisson@ufu.br

Carlos Eduardo de Carvalho Dantas
carlos.dantas@ufu.br

[1] Federal University of Uberlândia, Uberlândia, (MG), Brazil

[2] University of Saskatchewan, Department of Computer Science, 110 Science Place, S7N 5C9, Saskatoon, SK, Canada