CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge

Rodrigo Fernandes Gomes da Silva

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia - MG
2019

Rodrigo Fernandes Gomes da Silva

CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia - MG
2019

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S586 Silva, Rodrigo Fernandes Gomes da, 1982-
2020 CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge [recurso eletrônico] / Rodrigo Fernandes Gomes da Silva. - 2020.

Orientador: Marcelo de Almeida Maia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2020.454
Inclui bibliografia.

1. Computação. I. Maia, Marcelo de Almeida,1969-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 17/2020, PPGCO				
Data:	29 de maio de 2020	Hora de início:	13:45	Hora de encerramento:	17:05
Matrícula do Discente:	11613CCP004				
Nome do Discente:	Rodrigo Fernandes Gomes da Silva				
Título do Trabalho:	CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Marcelo Keese Albertini - FACOM/UFU, Fabiano Azevedo Dorça - FACOM/UFU, Otávio Augusto Lazzarini Lemos - UNIFESP, Gustavo Henrique Lima Pinto - UFPA e Marcelo de Almeida Maia - FACOM/UFU orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Gustavo Henrique Lima Pinto - Belém-PA; Otávio Augusto Lazzarini Lemos - São José dos Campos-SP; Marcelo Keese Albertini - Uberlândia-MG, Fabiano Azevedo Dorça - Uberlândia-MG, Marcelo de Almeida Maia - Uberlândia-MG. O discente participou da cidade de Uberlândia-MG.

Ressalta-se que a defesa foi transmitida ao vivo através do Youtube.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 01/06/2020, às 16:41, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Fabiano Azevedo Dorça**, **Professor(a) do Magistério Superior**, em 02/06/2020, às 10:49, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo Keese Albertini**, **Professor(a) do Magistério Superior**, em 02/06/2020, às 11:30, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Gustavo Henrique Lima Pinto**, **Usuário Externo**, em 03/06/2020, às 13:24, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Otávio Augusto Lazzarini Lemos**, **Usuário Externo**, em 04/06/2020, às 10:48, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2065168** e o código CRC **C721FC5A**.

Referência: Processo nº 23117.033418/2020-43 SEI nº 2065168

Este trabalho é dedicado aos meus pais Wanderley e Romilda que, me forneceram os alicerces da vida para chegar até aqui.

Agradecimentos

Agradeço...

A Deus, por minha vida e pela minha saúde.

Aos meus pais Wanderley e Romilda pelo amor incondicional. Sem o apoio deles desde meus primeiros momentos de vida eu não teria traçado essa trajetória e não teria chegado até aqui.

Ao meu orientador Marcelo Maia, por me guiar pelo caminho certo durante essa jornada, gastando comigo incontáveis horas de instruções e esclarecimentos.

Aos meus colegas de laboratório Carlos Eduardo e Klerisson, pelo apoio e colaboração nos trabalhos.

Aos meus ex-chefes de trabalho que me permitiram dedicar ao doutorado, e sem cujas permissões eu não teria conseguido: professores Ernando Reis, Valder Steffen, Odorico Coelho e Darizon Alves.

Ao meu atual chefe, professor Helder Eterno, que me autorizou 2 anos de afastamento integral para dedicação exclusiva ao doutorado, me permitindo inclusive ir estudar no exterior. Minha gratidão será Eterna.

Ao meu novo colega de laboratório do Canadá: Masud Rahman, que atuou como meu co-orientador durante meus trabalhos na Universidade de Saskatchewan e cujo papel foi fundamental para o sucesso de minhas publicações.

Ao professor Chanchal Roy que me possibilitou 1 ano de pesquisa e aprendizado na Universidade de Saskatchewan.

E por fim à Universidade Federal de Uberlândia, que me formou, me deu um mestrado, um trabalho e agora está me dando um doutorado. Me sinto em eterna dívida com essa instituição que considero minha minha segunda casa.

"Existem muitas hipóteses em ciência que estão erradas. Isso é perfeitamente aceitável, eles são a abertura para achar as que estão certas."

(Carl Sagan)

Resumo

Desenvolvedores frequentemente buscam por exemplos de código na internet para suas tarefas de programação. Infelizmente, eles enfrentam três grandes problemas. Primeiro, eles normalmente precisam ler e analisar diversos resultados das ferramentas de busca até obterem uma solução satisfatória. Segundo, a busca é prejudicada devido a uma lacuna léxica entre a consulta (descrição da tarefa) e a informação associada à solução (ex. exemplo de código). Terceiro, a solução recuperada pode não ser compreensível, como por exemplo, apenas um trecho de código sem uma explicação sucinta. Para tratar esses três problemas, propomos CROKAGE (Gerador de Respostas de Conhecimento da Multidão), uma ferramenta que recebe a descrição de uma tarefa de programação em linguagem natural e fornece uma solução compreensível para a tarefa, isto é, uma solução que contém não somente exemplos de código relevantes, mas também suas explicações sucintas escritas por desenvolvedores. Primeiramente, o conhecimento da multidão presente no Stack Overflow é utilizado para recuperar respostas candidatas para a tarefa de programação. Em seguida, um mecanismo de relevância composto de múltiplos fatores é usado para mitigar o problema da lacuna léxica e selecionar as respostas com melhor qualidade relacionadas à tarefa. Finalmente, ocorre um processamento de linguagem natural nas respostas de melhor qualidade para entregar as soluções de programação contendo os exemplos de código acompanhados de suas explicações, ao contrário dos estudos anteriores. CROKAGE foi avaliada e comparada com sete outras abordagens-base, incluindo o estado da arte. CROKAGE superou as sete outras no fornecimento de soluções relevantes para 902 tarefas de programação. Além disso, um estudo com 24 tarefas de programação e 29 desenvolvedores, confirmando a superioridade de CROKAGE sobre o estado da arte em termos de relevância dos exemplos de código sugeridos, benefício das explicações do código e da qualidade geral da solução (código + explicação).

Palavras-chave: Mineração de conhecimento da multidão. Stack overflow. *Word embedding*. Busca de código.

Rodrigo Fernandes Gomes da Silva

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia - MG
2019

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 comprehensive, i.e., the code segment might miss a succinct explanation. To address these three problems, we propose CROKAGE (Crowd Knowledge Answer Generator), a tool that takes the description of a programming task (the query) and delivers a comprehensive 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. 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 comprehensive solutions containing both code examples and code explanations unlike earlier studies. We evaluate and compare our approach against seven baselines, including the state-of-art. We show that CROKAGE outperforms the seven baselines in suggesting relevant solutions for 902 programming tasks (i.e., queries). Furthermore, a user study with 24 queries and 29 developers confirms the superiority of CROKAGE over the state-of-art tool in terms of relevance of the suggested code examples, benefit of the code explanations and the overall solution quality (code + explanation).

Keywords: Mining crowd knowledge. Stack overflow. Word embedding. Code search.

List of Figures

List of Tables

Acronyms list

API *Application Programming Interface*

CROKAGE *Crowd Knowledge Answer Generator*

Hit@K *Top-K Accuracy*

IR *Information Retrieval*

IDE *Integrated Development Environment*

LDA *Latent Dirichlet Allocation*

LR *Logistic Regression*

MAP@K *Mean Average Precision*

MR@K *Mean Recall*

MRR@K *Mean Reciprocal Rank*

Q&A *Question and Answer*

REST *Representational State Transfer*

SO *Stack Overflow*

TF-IDF *Text Frequency - Inverse Document Frequency*

WE *Word Embeddings*

Contents

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

Rodrigo Fernandes Gomes da Silva

Chapter 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 e 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 comprehensive. Either the retrieved code segment might miss a succinct explanation [Rahman e 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, Ribeiro-Neto 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. [Mikolov et al. 2013, Mikolov et al. 2013] 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. We further motivate these

limitations in Section 1.1.

In this work, 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) and then returns relevant, comprehensive programming solutions containing both code examples and succinct explanations. In particular, we address the limitations of the two earlier approaches [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, Kraft e Damevski 2018, An et al. 2017, Ragkhitwetsagul et al. 2018]. Thus, our explanations, different from BIKER, are not limited to the official API documentation, rather they cover as many APIs as covered by 27.4 millions[1] Stack Overflow answers [Rahman, Roy e Lo 2017].

Before we started on this domain, we first investigated about detecting duplicated questions on Stack Overflow. The replication [Silva, Paixao e Maia 2018] of two previous work, *DupPredictor* [Zhang et al. 2015] and *Dupe* [Ahasanuzzaman et al. 2016], gave us insights about the use of IR techniques to match an input query (i.e., the programming task) against the information associated with the solutions in the context of Stack Overflow. After analyzing the unsatisfactory results of these replications, we notice that two concerns should be addressed in order to effectively apply IR techniques to select candidate answers from Stack Overflow. First, an appropriate IR technique must be used for such a task. The very basic IR techniques used by *DupPredictor* were not sufficient enough to effectively match the query (i.e., a question title) against the questions' contents. This weakness is addressed by *Dupe*, that after using a more sophisticated IR technique, considerably improves the results. Second, relying solely on lexical similarity would not be enough to fill the lexical gap between the query and the relevant answers. Although *Dupe* employs a machine learning technique on top of a fine tuned IR technique, the approach trains a classifier using solely lexical features. Our replications show that, despite showing higher duplicate detection than *DupPredictor*, *Dupe* still falls short to detect the duplication when the master question (i.e., the duplicated question asked first) does not share common words with its duplicate.

In our approach, we address both aforementioned concerns. In order to deliver comprehensive solutions for a programming task, CROKAGE first employs an appropriate IR technique to select candidate Q&A pairs from Stack Overflow (e.g., BM25 [Robertson e Walker 1994]). Then, it employs a multi-factor relevance mechanism that overcomes the

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

lexical gap problem with word embedding technology, and finally returns the top quality answers related to the programming 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, Roy e Lo 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 1,840 questions (i.e., intent) and 4,691 answers containing the 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 seven 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).

We implement 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. 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

[2] http://tiny.cc/xh3kbz

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.

1.1 Motivation

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. 1-(a)) contains sentences describing the use of several API classes as well as opinions from Stack Overflow users. Despite describing solutions using relevant APIs (e.g., `Processbuilder, Runtime`), no actual code is provided. That is, the solution is half-baked and thus might not help the developer properly.

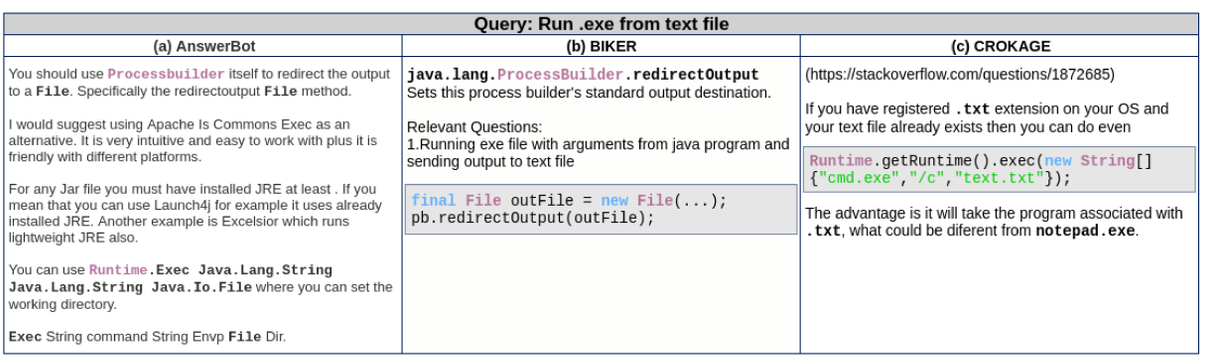


Figure 1 – Programming Solutions from (a) AnswerBot, (b) BIKER, and (c) CROKAGE

Alternatively, BIKER [Huang et al. 2018] provides both code example and corresponding explanation (Fig. 1-(b)). Unfortunately we notice two major problems. First, the suggested code does not completely match with the intent of the query. Second, the explanation is limited to only official Java API documentation and thus might fail to explain the functionalities of other external API classes or methods.

Finally, our approach CROKAGE provides a solution containing (1) a code segment using `Runtime` API and (2) an associated prose explaining the code (Fig. 1-(c)).

Unlike AnswerBot [Xu et al. 2017], CROKAGE delivers a solution containing relevant code segment. Unlike BIKER [Huang et al. 2018] (i.e., generates explanation from official API documentation only) CROKAGE delivers a solution containing code segment which is carefully explained and curated by Stack Overflow users. It also should be noted that unlike BIKER, our explanations are much more generic, informative and not restricted to standard Java APIs. Thus, this example has shown that our proposal, in principle, has a

much more potential than the existing alternatives (i.e., AnswerBot [Xu et al. 2017] and BIKER [Huang et al. 2018]).

1.2 Objectives and Contributions

In this work, we propose an approach that takes a programming task description and delivers solutions containing code examples with explanations. The main contributions are as follows:

- ❏ A novel approach –CROKAGE– that returns relevant and comprehensive programming solutions containing both code examples and succinct explanations against a programming task by harnessing the crowd knowledge from Stack Overflow.
- ❏ An empirical evaluation on the suggestion of relevant code examples for programming tasks and a comparison against seven baselines, including the state-of-art [Huang et al. 2018]. Our approach outperforms the seven baselines and the improvements are found for three programming languages.
- ❏ An empirical evaluation of 11 IR techniques in terms of performance to provide relevant solutions to programming tasks for three programming languages.
- ❏ An empirical evaluation of four relevance factors in terms of performance to provide relevant solutions to programming tasks for three programming languages.
- ❏ A user study with 29 developers to evaluate the real benefit of our approach and a comparison with the state-of-art study [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).
- ❏ 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.

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

- ❑ An usage analysis of the tool five months after its release regarding the composition of 15.865 queries and 784 user ratings.

1.3 Hypothesis

In this section, we present four hypotheses along with their research questions that will be investigated in the next chapters:

H1 Different IR techniques perform differently in retrieving candidate Q&A pairs for given programming tasks written in natural language.

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

Argumentation: In order to propose solutions to programming tasks, we first need to find good quality Q&A candidate pairs, of which we extract information from to compose the solutions. We hypothesize that different IR techniques will have different performances for such task. Thus, it is possible to rank the IR techniques in terms of their performances and choose an IR technique with the best performance for our approach.

H2 CROKAGE can outperform seven other baselines, including the state-of-art, in retrieving relevant answers for given programming tasks.

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

Argumentation: In order to propose solutions to programming tasks, we extract answers from Q&A candidate pairs and contrast against our ground truth. It is possible to compare the performance of different techniques (or baselines) by measuring the quality of their outcomes (i.e., recommended answers). We hypothesize that CROKAGE can recommend better answers than other baselines, including the state-of-art by a statistical significant margin.

H3 The four different factors adopted by CROKAGE have different influence in CROKAGE's performance.

RQ3 To what extent do the factors individually influence CROKAGE's performance?

Argumentation: We explore four weighted relevance factors and evaluate their individual contribution in CROKAGE's performance. We hypothesize that the four factors will influence the performance of CROKAGE differently and weights shall be associated to them in order to adjust their contribution in the ranking of candidate answers used to compose the solutions.

H4 CROKAGE can outperform the state-of-art BIKER in providing comprehensive solutions containing code and explanations for given programming tasks.

RQ4 How effective is CROKAGE compared to the state-of-art BIKER in providing comprehensive solutions containing code and explanations for given programming tasks ?

Argumentation: We hypothesize that CROKAGE can provide more comprehensive solutions containing code and explanations for given queries (task descriptions) compared to those of the state-of-art, BIKER [Huang et al. 2018]. For this, we ask professional developers to evaluate solutions for programming tasks and compare their evaluations for both approaches.

[Thesis Statement] The crowd knowledge stored in Q&A websites, such as Stack Overflow, can be harnessed to assist developers in obtaining solutions for their daily programming tasks in the form of code examples containing explanations. CROKAGE can provide such solutions to developers with higher quality than other baselines. In particular, CROKAGE outperforms seven other baselines, including the state-of-art, in retrieving relevant answers for given programming tasks by a statistical significant margin. Furthermore, 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).

1.4 Publications

As a result of this work, two papers were published. The first paper is a preliminary work where we employed information retrieving techniques to detect duplicated questions in Stack Overflow. It was published at SANER'2018:

- Silva, Rodrigo FG, Klérisson Paixão, and Marcelo de Almeida Maia. *Duplicate question detection in stack overflow: A reproducibility study.* 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018.

The second paper is directly related to our thesis. Herein, use information retrieval techniques to search for code examples containing explanations to programming tasks. It was published at ICPC'2019 :

- Silva, Rodrigo FG, et al. *Recommending comprehensive solutions for programming tasks by mining crowd knowledge..* Proceedings of the 27th International Conference on Program Comprehension (ICPC). IEEE, 2019.

Furthermore, another paper is under review. It is an extended version of our ICPC paper [Silva et al. 2019] that has been submitted to Empirical Software Engineering (EMSE).

1.5 Thesis Outline

This work is structured as follows:

- Chapter 2 provides the fundamental concepts related to our approach. We start presenting the basic concepts about Information Retrieval. Next, we present the models we used in our work. Finally, we show the four performance metrics that we have used in our measurements.

- Chapter 3 presents CROKAGE, a tool that takes the description of a programming task (the query) and delivers a comprehensive solution for the task. We first show how we used three state-of-art API recommendation systems to provide API classes to programming tasks. Next, we present the corpus preparation. Then, we present the building of the models, maps and indices used in our approach. Next, we show the architecture of CROKAGE and the details of each of its stages. Then, we present the composition of the programming solutions.

- Chapter 3 presents CROKAGE, a tool that takes the description of a programming task (the query) and delivers a comprehensive solution for the task. We first show how we used three state-of-art API recommendation systems to provide API classes to programming tasks. Next, we present the corpus preparation. Then, we present the building of the models, maps and indices used in

- Chapter 4 presents the experiments. First, we show how we constructed our ground truth in order to assess all baselines. Then we evaluate and compare 11 IR techniques in retrieving candidate Q&A Pairs for given programming tasks. Next we compare CROKAGE against seven other baselines, including the state-of-art. We also show a comparison between CROKAGE and the state-of-art through a developer study. And finally, we present the threats to the validity of this study.

- Chapter 5 we present the implementation of CROKAGE in form of a tool to assist developers with their daily programming issues. We also show an usage analysis containing the composition of the programming tasks inputted by the developers in our tool. Since our tool also collects the user feedback, we use these data to investigate the reasons of the developers' dissatisfaction on 349 proposed solutions. Finally, we show conclusions on the tool usage and future opportunities for improvements.

- Chapter 6 presents the related work, separated in three main aspects: exploring Q&A features to mine knowledge from Stack Overflow, code example suggestion and code explanation generation.
- Chapter 7 concludes the work, summarizing the activities we developed and highlighting the main achievements and future research directions.

Chapter 2

Information Retrieval Background

Information Retrieval (IR) has been used in Software Engineering to address many of the challenges faced by developers [Binkley e Lawrie 2010]. One major challenge is retrieving source code associated to information written in natural language from software repositories like Stack Overflow[1] and GitHub[2]. Like traditional web search engines, many previous work in software engineering [Ponzanelli et al. 2014, Xu et al. 2016, Rahman, Roy e Keivanloo 2015, Huang et al. 2018, Silva et al. 2019, Rahman, Roy e Lo 2016, Zamanirad et al. 2017, Bajracharya, Ossher e Lopes 2010, Campbell e Treude 2017] take as input queries written in natural language to harness the crowd knowledge from software repositories. Likewise, our approach takes as input a natural language query and use IR techniques to match the information associated to the code to retrieve relevant documents, or in another words, Q&A pairs from Stack Overflow.

The goal of the IR techniques in our work is to narrow down the search space for the Q&A candidate pairs that may contain relevant solutions to our query (i.e., programming task). We provide in this work an extensive evaluation of several IR techniques in our context problem. Therefore, in this chapter, we present fundamental concepts related to IR, which are the basis of our thesis proposal. We begin by introducing the basic concepts of IR in Section 2.1. Next, in Section 2.2 we describe the models we used in our approach. Then, in Section 2.3, we describe the four performance metrics we used in our measurements, which are widely adopted by related literature in software engineering.

2.1 Basic Concepts

The concept of IR can be understood by the process of finding resources (i.e., documents) that are relevant to a information need from within a collection of unstructured resources (or documents) [Manning, Raghavan e Schütze 2010]. In most of IR systems, a document can be perceived as a standard unit of retrieval [Büttcher, Clarke e Cormack 2016], and

[1] https://stackoverflow.com/
[2] https://github.com/

can have different granularities. For example, in the case of a text document, the document may be represented by a paragraph, a page or several pages. IR can also mean the search for information in a document, regardless the format of the document (e.g,. texts, images, sounds, videos, etc). The search (a.k.a., query) is mainly composed by keywords, unlike SQL queries in a database system which have complex structures [Han, Pei e Kamber 2011]. The result of a search is usually a set of documents that meet the search criteria. This set of documents may or may not be ranked, depending on the type of the IR model adopted in the search. As we describe further in this chapter, there are three types of IR models: Boolean, Vector Space, and Probabilistic. While the Vector Space and the Probabilistic models produce ranked documents, the Boolean model only threats the presence or absence of the terms in the query and no ranking is provided. Classical examples of IR applications are web search engines like Google [Google Inc.] and Bing [Microsoft Inc.], which are becoming the dominant form of access to information nowadays, and are widely used to retrieve documents related to a search.

In an IR system, the keywords of a query are called *index terms* [Baeza-Yates, Ribeiro-Neto et al. 1999]. In a document, each index term is a word or group of words. The set of all distinct index terms in the document or in the collection of documents constitute the vocabulary. In a very simple representation of a document or collection of documents, a vocabulary is built and the frequency of each word is taken into account, but not their order. This kind of representation, known as *bag-of-words*, is one of the most common in Software Engineering and it is commonly used to extract features from text documents.

Since multiple terms can occur in multiple documents, a document matrix is usually used to map terms and documents. In large collections of documents, however, it might be interesting to reduce the complexity of the document representation. This is usually accomplished by three strategies. One of them is the reduction of the set of keywords in the vocabulary like words that occur frequently in the language and carry little value. These words are also known as *stopwords*. Example of *stopwords* are: "the", "which", "on", "a". Two other strategies are *Stemming* and *Lemmatization*. They address the different inflections that the words can have. Take for example the word "like". This word can appear in a text in different forms: "liking", "liked", "likes". Another example is the word "be", that can appear as "are", "am" or "is". *Stemming* and *Lemmatization* help us to achieve the root forms of inflected words. However, they differ in the word they produce and in the approach they use to produce the root form. While *Stemming* is the process of reducing inflection in words to their root forms to the same stem, regardless if the stem belongs to the language, *Lemmatization* reduces the inflected words in such a way that the root word (the lemma) belongs to the language.

The above definitions provide the support for the understanding of the three classic types of IR models we adopt in our work, which are the Boolean, the Vector and the Probabilistic. We describe all the adopted models as follows.

2.2 Adopted Models

In this section, we describe the models used in our approach. We divide the section according to the model types as follows: boolean model, vector space models, probabilistic models, and axiomatic models.

2.2.1 Boolean Model

The Boolean model is a simple model based on algebra. It considers the presence or the absent of the index terms in the document. That is, in the document matrix, the frequencies of each term is a binary value, indicating that the term is present or not in each document. Three connectives link the terms in a query: "and", "not" and "or". More formally, the model defines a query Q as a Boolean expression as follows:

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

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:

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

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.

2.2.2 Vector Space Models

Differently from the Boolean model that only fetches complete matches, the Vector Space Model can fetch documents that partially match the query. This is possible due to the way the model computes the degree of similarity between the query and each document in the collection. The model assigns weights (non-binary) to the index terms in the queries and in the documents, which are used to compute the similarity. Thus, the model considers documents that match the query terms, regardless whether the match is complete or partial. The result is a list of documents ranked in decreasing order according to the similarity with the query.

A common way of measuring the similarity between two documents (the query is also represented by a document) in Software Engineering is measuring their cosine distance. The high popularity of cosine similarity calculation may be explained by its low complexity, mostly for sparse vectors where zero dimensions are predominant. Herein, each document is represented by a vector in a high-dimensional vector space and the distance between two documents is measured by the cosine of the angle between them [Baeza-Yates, Ribeiro-Neto et al. 1999]. Two vectors can be oriented in a interval of [0,1], meaning that if they have the same orientation, their cosine similarity is 1. Since only positive values make sense for similarity, vectors opposed have 0 similarity to each other. For non-zero vectors, the cosine similarity can be calculated by using the Euclidean distance [Danielsson 1980] between them as follows:

$$A \cdot B = \|A\| \, \|B\| \, cos\theta \tag{3}$$

where $cos\theta$ is calculate as follows:

$$cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} * \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4}$$

where $cos\theta$ is represented by a dot product and magnitude. A_i and B_i are the components of vectors A and B, denoting the representation of the respective documents. The cosine similarity sorts the documents based on their degree of similarity to the query, which may be considered the main document.

In general, the Vector Space Model, despite the simplicity, is resilient with general collections [Baeza-Yates, Ribeiro-Neto et al. 1999]. We describe next the three Vector Space Models we adopted in our approach: TF-IDF, Classic and Word Embeddings. These models may differ in the term weighting, in the similarity function, or even in the representation of terms.

2.2.2.1 TF-IDF

TF-IDF stands for term frequency (TF) times inverse document frequency (IDF). It is a weight used in IR and denotes how important a word is to a document in a collection of documents (i.e., a corpus). If a word occurs multiple times in a document, that is, the word has high term frequency (TF), it means that this word is more relevant than other words that appear fewer times in the document. However, if this word occurs multiple times in a document but also along many documents in the collection of documents, this word may not be so meaningful. That is, this word has high document frequency. The IDF component of TF-IDF stands for the inverse document frequency and decrease the weight of words (i.e., terms) with high frequency in the collection of documents and, at the same time, increases the weight of terms that occur rarely.

The calculation of the TF of a term t in a document d is done by counting the number of times the term t occurs in document d. Although other forms can be used to calculate TF, the raw count of t in d denoted by $f_{t,d}$, is one of the most used in Software Engineering and it is the one we adopted in our studies.

Likewise, the calculation of IDF has several variants. We adopt the default with no smoothing in our studies, which is calculated as follows:

$$idf(t, D) = log_{10} \frac{N}{|\{d \in D : t \in d\}|} = log_{10} \frac{N}{n_t} \tag{5}$$

where N denotes the total number of documents in the corpus and n_t is the number of documents where the term t appears. Thus, the tf-idf weight is calculated as the multiplication of TF and IDF values as follows:

$$tfIdf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{6}$$

and its value ranges between [0,1]. The *tf-idf* can be used to calculate the lexical similarity between two documents D_1 and D_2. For this, the similarity is determined using their cosine similarity based on the *tf-idf* term weight for each word of the document as follows:

$$lexScore(D_1, D_2) = \frac{d_{D_2} \cdot d_{D_1}}{|d_{D_2}| \cdot |d_{D_1}|} = \frac{\sum_1^N tfidf_{ti,dq} \cdot tfidf_{ti,da}}{\sqrt{\sum_1^N tfidf_{ti,dq}^2} \cdot \sqrt{tfidf_{ti,da}^2}} \tag{7}$$

where d_{D_1} and d_{D_2} represents the bag-of-words of each document and $tfidf_{ti,dk}$ is the term weight for each word of each document.

We use TF-IDF as one of the weighting schemes in our work. Herein, our corpus is composed by Stack Overflow words. Furthermore, the query and every post is a document within a vector space model.

2.2.2.2 Classic Model

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

$$score(q, d) = \sum_{t \in q} (tf(t \in d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t, d)) \tag{8}$$

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:

$$idf(t) = 1 + log(\frac{docCount + 1}{docFreq + 1}) \tag{9}$$

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.

2.2.2.3 Word Embeddings

Word Embeddings (WE) is a technology developed by Mikolov et al. [Mikolov et al. 2013, Mikolov et al. 2013] that represents each word using a high-dimensional vector in such a way that similar words have similar vector representations in the vector space. This technology can capture words' semantics, meaning that two words can have similar meanings according to their position in the vector space. That is, their meaning are considered similar if their vector representations are similar.

WE model takes as input a training corpus. The model uses this corpus to derive the word vectors for each word. WE implementations usually provide two models for computing word representations: skipgram and cbow (i.e., continuous-bag-of-words). While the skipgram model learns how to predict a target word according to a nearby word, the cbow model predicts the target word according to its surrounding context. The context is contained in a fixed size window and is represented as a bag-of-words around the target word. In our studies we use the skipgram model to compute the word representations since it has been showed before [Efstathiou e Spinellis 2019] that this model is more efficient than the cbow. Besides, compared to cbow, skipgram models works better with subword information [Bojanowski et al. 2017], which handles the variants of words like *list*, *listing* and *listed*.

A well known implementation of WE is *Word2vec*, developed by Mikolov et al. [Mikolov et al. 2013, Mikolov et al. 2013] in 2013. As an extension of *Word2vec*, *fastText* [Bojanowski et al. 2017] was proposed in 2016 by Facebook. *fastText* is a library for learning of word representations and sentence classification. Differently from *Word2vec* that feeds individual words into a neural network, *fastText* breaks them into n-grams (subwords). For example, the tri-grams of the word "horse" is "hor", "ors" and "rse". Thus, the WE vector for "horse" will be the sum of these three n-grams. After *fastText* trains the neural network, a WE for all n-grams will be produced for the dataset used for training. This neural network now is capable of represent rare words, since their n-grams may appear in other words. In our work, we adopt two implementations of WE to build neural networks containing all the words of Stack Overflow Q&A. The first one is *fastText*. We choose *fastText* rather than *Word2vec* because *fastText* outperforms *Word2vec* [Efstathiou e Spinellis 2019]. Furthermore, unlike *Word2vec*, we found the flexibility provided in handling subwords more appropriate to our domain problem. The second one is *Sent2vec*, 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".

WE have been used recently [Xu et al. 2017, Huang et al. 2018] to calculate the similarity between two documents. Differently from the traditional approaches that rely on lexical similarity, this calculus aims to calculate the semantic similarity. In this approach, Ye et al. [Ye et al. 2016] adapt the text-to-text similarity measure introduced by Mihalcea et al. [Mihalcea et al. 2006] to compute the an asymmetric similarity ($asym$) between two bag-of-words D_1 and D_2 representing two documents as follows:

$$asym(D_1 \rightarrow D_2) = \frac{\sum_{w \in D_1} sim(w, D_2) * idf(w)}{\sum_{w \in D_1} idf(w)} \tag{10}$$

where $idf(w)$ is the correspondent IDF value of the word w, $sim(w, D_2)$ is the maximum value of $sim(w, w_{D_2})$ for every word $w_{D_2} \in D_2$, and $sim(w, w_{D_2})$ is the cosine similarity between w and w_{D_2} embedding vectors. Herein, the embedding vectors denote the word embedding representations. The other asymmetric relevance namely $asym(D_2 \rightarrow D_1)$ can be calculated by swapping D_1 and D_2 in Equation 10. Thus, the semantic similarity ($semScore$) between the document D_1 and the document D_2 is the harmonic mean of the two asymmetric relevance scores as follows:

$$semScore(D_1, D_2) = \frac{2 * asym(D_1 \rightarrow D_2) * asym(D_2 \rightarrow D_1))}{asym(D_1 \rightarrow D_2) + asym(D_2 \rightarrow D_1)} \tag{11}$$

2.2.3 Probabilistic Models

Initially devised by Robertson and Jones [Robertson e Jones 1976], the probabilistic model proposes a formalism to rank documents according to their relevance to a given query. The relevance of a document is based on the probability of this document to be found for a given query. Thus, given a input query, the IR model retrieves documents that are considered relevant for the query based on properties. The model assumes that there exists a subset of documents among all documents which the user prefers for a given query. Such documents are considered relevant to the query, while the others are considered not relevant. The model then, tries to maximize the overall probability of relevance to the user by providing the relevant documents. In this model, a query q is a subset of the index terms. A document d_j can be represented by a vector of binary weights assigning the presence or absence of each index term as follows:

$$\vec{d_j} = (w_{1,j}, w_{2,j}, w_{3,j}..., w_{t,j}) \tag{12}$$

where $w_{i,j} = 1$ if the term k_i appears in document d_j and $w_{i,j} = 0$ otherwise. Let us assume that R is a set of documents considered relevant by the user for the query q and $\bar{R}$ is the complement of R (i.e., the set of non-relevant documents). The probability that

the document d_j is relevant to the query q is given by $P(R|\vec{d_j}, q)$, where $\vec{d_j}$ denotes the representation of d_j. Therefore, the probability that the document d_j is not relevant to the query q is given by $P(\bar{R}|\vec{d_j}, q)$. The similarity between the document d_j and the query q is determined as follows:

$$sim(d_j, q) = \frac{P(R|\vec{d_j}, q)}{P(\bar{R}|\vec{d_j}, q)} \tag{13}$$

In the raw form of this model, the index terms are not weighted and the terms are assumed to be mutually independent. Moreover, in the first run the model lacks accuracy since the properties are not known at the query time. We adopt two probabilistic models in our work, which are extension of the basic model: BM25 [Robertson e Walker 1994] and Jelinek Mercer [Jelinek 1980], described as follows.

2.2.3.1 BM25

BM25 [Robertson e Walker 1994] is a direct extension of the classic probabilistic model. This model improves the classic model by implementing two important principles missing in the classic probability model: term frequency and document length normalization. BM25 was created as a combination of BM11 and BM15 ranking functions. The model determines the similarity *sim* between a document D and a given query Q as follows:

$$sim(D, Q) = \sum_{i=1}^{n} idf(q_i) * \frac{f(q_i, D) * (k + 1)}{f(q_i, D) + k * (1 - b + b * \frac{|D|}{avgdl})} \tag{14}$$

where $|D|$ is the length of D in words, $f(q_i, D)$ is keyword q_i's term frequency in document D 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:

$$idf(q_i) = log\frac{D - n(q_i) + 0.5}{n(q_i) + 0.5} \tag{15}$$

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).

2.2.3.2 Jelinek Mercer and Dirichlet Priors

This model belongs to the family of smoothing methods [Zhai e 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 [Jelinek 1980] calculates the probability of a word w in a document d as follows:

$$p(w|d) = (1 - \lambda)\frac{c(w,d)}{|d|} + \lambda * p(w|C) \tag{16}$$

where $\frac{c(w,d)}{|d|}$ is the maximum likelihood estimate of a unigram language model[3]. 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:

$$p(w|d) = \frac{c(w;d) + \mu p(w|C)}{|d| + \mu} \tag{17}$$

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 (2000).

2.2.4 Axiomatic Models

Initially proposed by Fang and Zhai [Fang e 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 1. 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 2 and describe them as follows: ln is the natural logarithm (base e), N is the total

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

Table 1 – 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	$S(Q,D) = \sum_{t \in Q \cap D}(C_t^Q * TF(C_t^Q) * LW(t) - \gamma(\|D), \|Q\|)$

Table 2 – Axiomatic Formulas' Components

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

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 $0 \leqslant s \leqslant 1$.

2.3 Performance Metrics

Performance metrics are an objective and quantitative way of comparing performance of different models. We choose four performance metrics commonly adopted by related literature [Xu et al. 2017, Rahman e Roy 2018, Rahman, Roy e Lo 2016, Rahman e 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 4.1). 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 the defined as follows:

$$Hit@K(Q) = \frac{\sum_{q \in Q} isRelevant(q, K)}{|Q|} \tag{18}$$

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:

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank(q, K)} \tag{19}$$

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:

$$MAP@K(Q) = \frac{\sum_{q \in Q} AP@K(q)}{|Q|} \tag{20}$$

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

$$AP@K(Q) = \frac{1}{T_{gt}} \sum_{i=1}^{k} P_k \cdot f_k \tag{21}$$

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:

$$MR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|GS(q) \bigcap recommended(q, K)|}{|GS(q)|} \tag{22}$$

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.

Chapter 3

CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge

Developers spend a considerable time of their daily activities searching for code examples for their programming tasks [Brandt et al. 2009]. Although traditional search code engines such as GitHub, Koders and Krugle provide access to an abundant amount of source code, they are still limited to assist the developers with code examples for their programming tasks. These search engines usually rely on lexical similarity to match an input query (i.e., the programming task) against the content they host. This limitation impairs the search for code examples since the developer needs to carefully design the query, using appropriate words or API classes names that contains exact matches with the content provided by the search engines [Rahman, Roy e Lo 2017]. Furthermore, the retrieved code is not always comprehensive and usually misses a text explanation. Both problems make the developers browse dozens of search results in order to synthesize an appropriate solution for the task.

In order to fill the lexical gap between the query keywords and the information associated to the solution, two previous work called AnswerBot [Xu et al. 2017] and BIKER [Huang et al. 2018] harness word embeddings, a technology developed by Mikolov et al. [Mikolov et al. 2013, Mikolov et al. 2013]. This technology is able to bridge this lexical gap by representing words using a high-dimensional vector in such a way that words with similar meanings are close to each other in a vector space model. Although this technology seemed to be promising, the solutions provided by AnswerBot and BIKER are not sufficient enough to address the lexical gap problem between the query and the solutions. Both works are still limited in several aspects. AnswerBot's answers, for example, do not contain source code examples, only explanations. BIKER, on the other hand, can provide answers containing both code examples and explanations. Nevertheless, their answers are limited to official Java SE documentations.

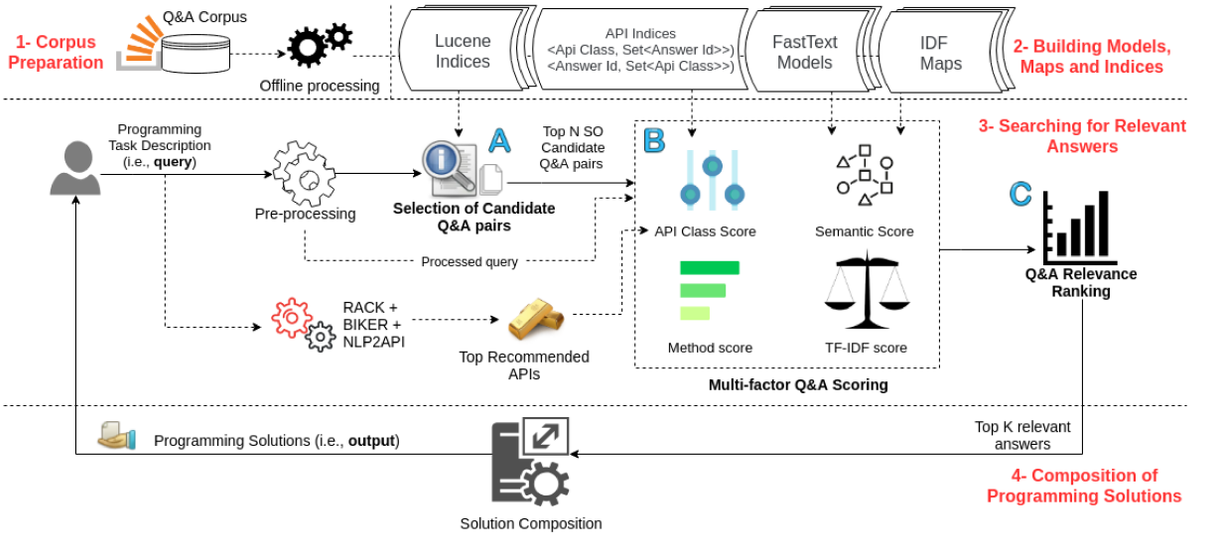


Figure 2 – Schematic diagram of CROKAGE. 1 - Corpus Preparation, 2 - Building Models, Maps and Indices, 3 - Searching for Relevant Answers, and 4 - Composition of Programming Solutions

In this work, we propose CROKAGE, an approach that takes as input a programming task, written in natural language, and provides comprehensive programming solutions containing both code examples and succinct explanations. CROKAGE addresses the limitations of AnswerBot and BIKER. Unlike AnswerBot (i.e., provides only answer summary texts), CROKAGE delivers both relevant code segments and their corresponding explanations. And unlike BIKER [Huang et al. 2018], which returns only generic explanations extracted from the official Java API documentation, CROKAGE provides succinct code explanations written by human developers. CROKAGE harnesses the knowledge stored in Stack Overflow, which is nowadays the largest community for developers where they share their experiences, doubts, issues or exchange ideas about programming tasks.

Stack Overflow can also be considered one of the most important repositories of knowledge about Software Engineering. It contains a wide range of programming topics and serve as a large repository of source code where developers copy code from [Yang et al. 2017, Ciborowska, Kraft e Damevski 2018] or post code from other platforms such as GitHub [An et al. 2017, Ragkhitwetsagul et al. 2018]. Since CROKAGE extracts the solutions from Stack Overflow answers, its knowledge base is as wide as the one containing in Stack Overflow answers. That is, the knowledge contained in 27.4 millions[1] answers.

Figure 2 shows the schematic diagram of our proposed approach CROKAGE, which is composed by four stages. We describe each stage in this chapter as follows. First, in Section 3.1 we show how we leverage three state-of-art API recommendation systems – BIKER [Huang et al. 2018], NLP2API [Rahman e Roy 2018] and RACK [Rahman, Roy e Lo 2016] to recommend API Classes for queries (i.e., task description), which are later

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

used to compose one of the relevance factors (API Class factor). Next, we describe in Section 3.2 the first stage: the corpus preparation using Q&A threads from Stack Overflow. Then, in Section 3.3, we describe the second stage, where we construct several models (e.g., *fastText* model), maps and indices. The third stage is described in Section 3.4, where these models, maps and indices are employed to retrieve relevant answers from the corpus against a programming task description. And finally, the fourth and final stage is describe in Section 3.5, where the top quality answers are used to compose and suggest the programming solutions for the task.

3.1 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 e Roy 2018] and RACK [Rahman, Roy e Lo 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 e 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 - Section 2.3)) 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 3. We observe that BIKER alone has the worst 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. 2-(3-B)) for Java language, since the tools are limited to Java.

3.2 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. 2-(1)).

Table 3 – 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

Like many previous academic papers [2] that harness the crowd knowledge stored in Stack Overflow to address Software Engineering problems, our work uses the Stack Overflow corpus to extract solutions to programming tasks. For this, we first download the Stack Overflow dump[3]. Although the dump contains several tables representing the corpus, we only use in our work the table *Posts*. Thus, we restore the table into a local database and collect a total of 10,248,824 questions and answers related to three programming languages, as shown in Table 4. Then, we employ natural language processing over these posts. We use *Jsoup* [Jsoup] 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[4], 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 programming solutions (Section 3.5).

3.3 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. 2-(2)). 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

[2] https://meta.stackexchange.com/questions/134495/academic-papers-using-stack-exchange-data
[3] https://archive.org/details/stackexchange - dump published in March 2019
[4] https://bit.ly/1Nt4eMh

Table 4 – Number of Stack Overflow Questions and Answers by Language

Language	Type	Number of 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

capture the pre-processed version of each pair as a document within a large document corpus. We then build an index with Lucene [Apache] 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 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 *skipgram* 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, Rao e Kak 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 2.3 describes the adopted metrics) than the default values [Mikolov et al.] (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. [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, choice of 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. 2-(2)), 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. 2-(2)) 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. 2-(2)). 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]. 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"*[5] 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

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

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.

3.4 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 Figure 2-(3). In order to search for relevant answers, CROKAGE first loads the models and indices (Section 3.3) and then pre-processes the task description with standard natural language pre-processing. Then, CROKAGE navigates through the three stages as follows.

3.4.1 Selection of Candidate Q&A pairs

Given a pre-processed task description (i.e., query) Q and the Lucene index constructed above for the language of interest (e.g., Java), CROKAGE uses BM25 [Robertson e Walker 1994] function to filter the top N relevant Q&A pairs (Fig. 2-(3-A)) from the pre-loaded index as $sim(P, Q)$ (Formula 14, Section 2.2.3.1). Herein, P denotes each Q&A pair from the index. 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 4.2) and determine their optimal values across the three languages.

3.4.2 Multi-factor Q&A Scoring

In this stage, CROKAGE calculates scores for the candidate Q&A pairs using four similarity factors (Fig. 2-(3-B)). 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, Souza e Maia 2014, Lv et al. 2015, Zagalsky, Barzilay e Yehudai 2012, Bajracharya, Ossher e Lopes 2010], and herein represented by TF-IDF. Although we use BM25 [Robertson e 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 3.3) 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. [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 e 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 between a programming task (i.e., the query) Q and a Q&A pair P as follows:

Lexical Score: Herein, we use TF-IDF to represent documents. TF-IDF stands for term frequency (TF) times inverse document frequency (IDF). CROKAGE computes the lexical similarity between P and Q as $lexScore(P, Q)$ (Formula 7, Section 2.2.2.1).

Semantic Score: CROKAGE computes the asymmetric relevance between P and Q as $semScore(P, Q)$ (Formula 11, Section 2.2.2.3).

API Method Based Score: 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 based score as follows:

$$methodScore(P) = \frac{log_2(freq_m)}{x} \tag{23}$$

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: herein, we leverage the association between the APIs relevant to a programming task (i.e., query) and the candidate Q&A pair in our approach. 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 e Roy 2018] and RACK [Rahman, Roy e Lo 2016]. Our findings suggest that the combination of these tools provides the best results which justifies our choice of combining their API suggestions (Section 3.1). Second, we select the candidate Q&A pairs containing any API classes. For this, we use our *apiAnswers* index (Section 3.3) 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:

$$apiScore(A) = \sum_{c \in C} \frac{1}{pos(c) + n} \tag{24}$$

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.

3.4.3 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) Q as follows:

$$\begin{aligned} factorsScore(P, Q) = {} & lexScore \cdot lexWeight \\ & + semScore \cdot semWeight \\ & + apiScore \cdot apiWeight \\ & + methodScore \cdot methodWeight \end{aligned} \tag{25}$$

where *semWeight*, *apiWeight*, *lexWeight*, and *methodWeight* are relative weights associated to each factor. We conduct controlled iterative experiments and employ a set of weights that return the highest Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall respectively (Section 4.3). Once the final score (i.e., *factorsScore*) is calculated (Fig. 2-(3-C)), we collect the Top-K Stack Overflow answers from the top scored pairs to compose the solution.

3.5 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 the Algorithms of Figures 3 and 4.

```
1  answersSelection(query, recommendedAnswers, n)
2      relevantAnswers
3      while answer ← 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 while
12     return relevantAnswers
13 end
```

Figure 3 – Select answers with succinct code explanations

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[6], 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. [Wong, Yang e Tan 2013]:

$$\begin{aligned} &VP << (NP < /NN.?/) < /VB.?/ \\ &NP! < PRP[<< VP|\$VP] \end{aligned} \tag{26}$$

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

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

```
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
```

Figure 4 – Pseudocode to filter important sentences from answers

a verb phrase is not a personal pronoun. CROKAGE filters important sentences using the pseudocode shown in the Algorithm of Figure 4.

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 adds 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")[7] 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 of Figure 4). Thus no candidate sentence is added to the list of important sentences (Algorithm of Figure 4 - line 14) and hence this answer is not added to the list of relevant

[7] the complete list of words is available at: https://bit.ly/2Hjv0tW

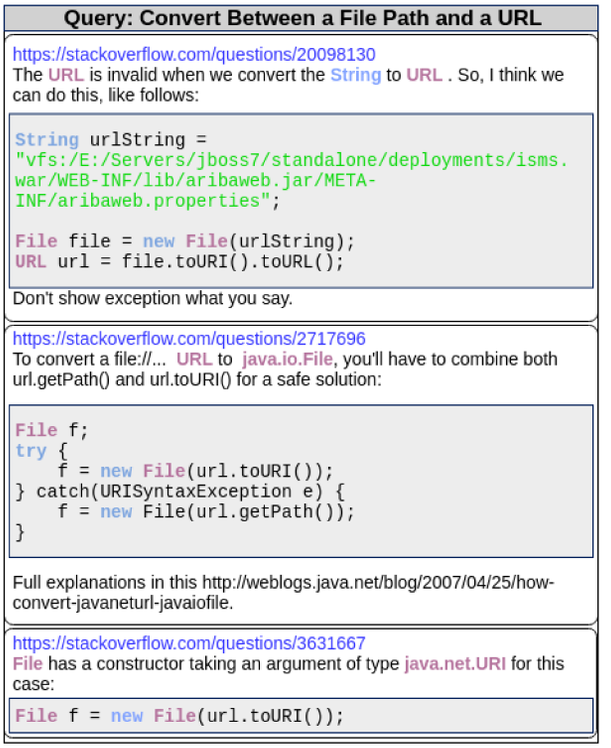


Figure 5 – Comprehensive solution generated by CROKAGE for the query: *"Convert Between a File Path and a URL"*

answers (Algorithm of Figure 3 - line 6). In another example, CROKAGE retrieves the answer 7189370[8] 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 you can append one space before appending each character"* and *"Something like this :"*. Herein, the Algorithm of Figure 4 filters the first sentence and discards the second. Since at least one important sentence is returned to Algorithm of Figure 3, the answer is selected.

Our intuition is that the removal of unnecessary sentences may help developers with a more concise explanation. CROKAGE also discards trivial, 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"*). Thus, it finally delivers a solution containing code and explanations. Figure 5 shows Top-3 results delivering comprehensive solution for the task: *"Convert Between a File Path and a URL"*.

[8] https://stackoverflow.com/questions/7189370

Chapter 4

Experimental Results

In this chapter, we show evaluations of our approach under multiple aspects. For this, we first need to construct a ground truth, which is showed in Section 4.1. Then, in Section 4.2 we evaluate the performance of 11 IR techniques to select candidate Q&A pairs for programming tasks, thus justifying our choice of the IR technique used in our approach. Next, in Section 4.3 we evaluate the performance of CROKAGE in code example suggestion and compare with seven baselines, including the state-of-art BIKER [Huang et al. 2018]. We show evaluations 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. For the evaluations, we contrast the recommended results against the ground truth and use four classical evaluation metrics to measure the performance (Section 2.3). In Section 4.4 we show a user study with 29 developers and 24 queries. In this study, CROKAGE and BIKER are evaluated 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, in retrieving relevant answers for given programming tasks?

RQ3: To what extent do the factors individually influence CROKAGE's performance?

RQ4: How effective is CROKAGE compared to the state-of-art BIKER in providing comprehensive solutions containing code and explanations for given programming tasks ?

Next, in Section 4.5 we discuss about the experimental process. Then, in Section 4.6, we present the concluding remarks of the Chapter and the summary of the obtained results. Finally, in Section 4.7 we describe the threats to the validity of the experiments carried out in the Chapter.

4.1 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 [Java2s], BeginnersBook [BeginnersBook] and KodeJava [Saryada]. 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.], Bing [Microsoft Inc.] and Stack Overflow search [Stack Exchange Inc.]. 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[1]. For Stack Overflow search, we filter results using the tag *"java"*. We collect the first 10 results from Google and the first 20 from Bing. We observe lower efficiency of Stack Overflow search mechanism regarding the relevance of results when compared to Google and Bing. We thus establish a more rigorous criteria to fetch results from Stack Overflow search by setting a threshold of a minimum of 100 visualizations.

For each of the 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:

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

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

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

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

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

Two professional developers first evaluate the answers independently. After the evaluations, the average rating is calculated. If two ratings differ more than 1 Likert and at least one of them is higher than 3, this answer is marked to be re-evaluated by both in an agreement phase. The two professional developers then discuss these conflits. If after discussing, two ratings still differ in more than 1 Likert, this answer is discarded. We then re-calculate the average rating for the marked answers. We consider an answer as

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

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 e Koch 1977].

We repeat the same process for PHP language. For this, we randomly select 10 queries from the 115 Java queries in such a way they do not refer to a specific API (e.g., "Append string to a text file"). For these queries, we collect 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 e Koch 1977].

For Python language, we leverage a manually curated dataset previously constructed by a Yin et al. [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 compare different IR techniques (Section 4.2) as well as to train and test our approach and the baselines (Section 4.3).

4.2 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. 2-(3-A)). After obtaining the candidate Q&A pairs, we extract their answers and contrast against our *goldSet*. CROKAGE threats the search for candidate Q&A pairs as an information retrieval problem. In order to answer RQ1, we fist need to obtain appropriate candidate Q&A pairs. For this, we use Lucene [Apache] to implement 11 IR techniques (described in Chapter 2, Section 2.2), including BM25 [Robertson e Walker 1994] (Section 2.2.3.1)[2]. 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. 2-(3-A)). We run CROKAGE for each query of the training set (Section 4.1), 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 show in Table 5.

Table 5 – 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

Answering RQ1: According to our findings, the models Boolean, Dirichlet priors, Axiomatic F3LOG, and Axiomatic F3EXP perform the worst, while the models Classic, Jelinek Mercer, Axiomatic F1EXP, BM25, and Axiomatic F1LOG perform the best. In particular, BM25 [Robertson e 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,

[2] here we use the default parameters: k=1.2, b=0.75

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.

Previous work in Software Engineering also leveraged the power of BM25 to reduce the search space for candidate Stack Overflow Q&A. Ahasanuzzaman et al.[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, Paixao e Maia 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 4.1) 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 (Chapter 3, Section 3.4), 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.[Ahasanuzzaman et al. 2016], but also varies according to the language, as showed in Table 6. These optimal values are then used for next stages of our experiments.

Table 6 – 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

4.3 Experimental Results for the Retrieval of Relevant Answers

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

To answer RQ2, we calibrate the parameters (including the factors' weights) of CROKAGE using the training set (Section 4.1) 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 (Chapter 3 - Section 3.4), varying the value

of each parameter, and contrasting the results against the *goldSet*. Table 7 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 seven other baselines as follows:

Table 7 – 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	Variation	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 TF-IDF score (lexScore)	[0,1]	0.25	0.50	0.50
methodWeight	Weight associated with 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

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. 2-(3-A)).

BM25 + factors: we build four baselines representing the relevance factors (Chapter 3, Section 3.4.3, Fig. 2-(3-B)): BM25 + API Class, BM25 + Method, BM25 + Semantic, and BM25 + TF-IDF. For this, we preserve the weight associated to the baseline and set the other three weights (Chapter 3, Formula 25) to zero (e.g., to build *BM25 + Semantic* baseline we set all factors' weights to zero, except *semWeight*).

Sent2Vec: in this baseline, the semantic factor (*semScore* - Section 3.4.2) has the corresponding embedding vectors generated by *fastText* [Bojanowski et al. 2017] replaced by the embedding vectors generated by *Sent2Vec* [Pagliardini, Gupta e Jaggi 2017]. (Section 2.2.2.3). 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[3]. We use the same customized parameters as used for constructing the *fastText* models (Section 3.3) 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.

Like for CROKAGE's parameters (i.e., Table 7), we calibrate the weights for *Sent2vec* obtaining the following weights for *lexWeight, methodWeight* and *semWeight* respectively: 0.75, 1.0 and 0.5 for Java/PHP and 0.25, 0.50 and 0.75 for Python. 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.

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 in Formula 4 (Chapter 2). Herein, A and B are the two 100-dimensional vectors representing P and T.

After building all baselines, we run CROKAGE to search for relevant answers (Chapter 3, Section 3.4) 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 8, 9 and 10 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.

Table 8 – 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 + 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	0.59	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
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

The results of Tables 8, 9 and 10 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 is a good trade-off. Thus, in order to assess the differences among the baselines, we set K=10 for the further experiments.

[3] 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

Table 9 – 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 + 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
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

Table 10 – 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 + 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
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

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 and calculate the effect size with $r = Z/\sqrt{n}$ [Fritz, Peter e Richler 2012]. The results containing the comparison between CROKAGE and each baseline are showed in Table 11.

Table 11 – 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 + 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	∘	∘	∘	∘

∘ = not statistically different
S = small effect size
M = medium effect size
L = large effect size

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). 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 metrics are 65%, 44%, 38%, and 21% higher respectively in absolute values.

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, Peter e Richler 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 and Sent2Vec worsen the results when associated with BM25.

- ❏ Python: CROKAGE is statistically superior than five baselines. Contrary to the Java language, the Method factor alone shows the worst performance. The difference in performance between CROKAGE and BM25 + Method is 24%, 14%, 13% and 26% in Hit@K, MRR@K, MAP@K, and MR@K respectively. BM25 alone, on the other hand, performs the best and slightly lower than CROKAGE, while fastText and Sent2Vec factors (i.e., BM25 + fastText and BM25 + Sent2Vec) performs similarly. Indeed, the difference between CROKAGE and all baselines is statistically confirmed. The effect size [Fritz, Peter e Richler 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 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 + Method (i.e., variance of 40%, 58%, 51% and 16% in Hit@K, MRR@K, MAP@K and MR@K respectively) and the lowest between CROKAGE and BM25 + TF-IDF (i.e., variance of 29%, 28% and 2% in MRR@K, 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 task 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. 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.[4] Since in general the *fastText* performed better than *Sent2vec*, we leave the semantic factor of CROKAGE represented by the *fastText* embedding vectors.

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 [5]. Thus, we construct baselines in such a way that each query can be executed in less than one second.

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. 2-(3-C)) 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 25 - Chapter 3). 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 (Chapter 3, Section 3.4) and collect the top-10 answers from the recommended Q&A pairs. Like for RQ2, we contrast the recommended answers 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 12. 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 13.

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

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

Table 12 – 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

Table 13 – 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

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

Discussion: according to the results of the four configurations (Table 12) and the comparison with CROKAGE (Table 13), 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, Peter e

Richler 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, Peter e Richler 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) have 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 3.1), 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

Table 14 – Performance of BM25 + API Class and CROKAGE and their extended configurations (α) where the goldSet contains only answers with APIs

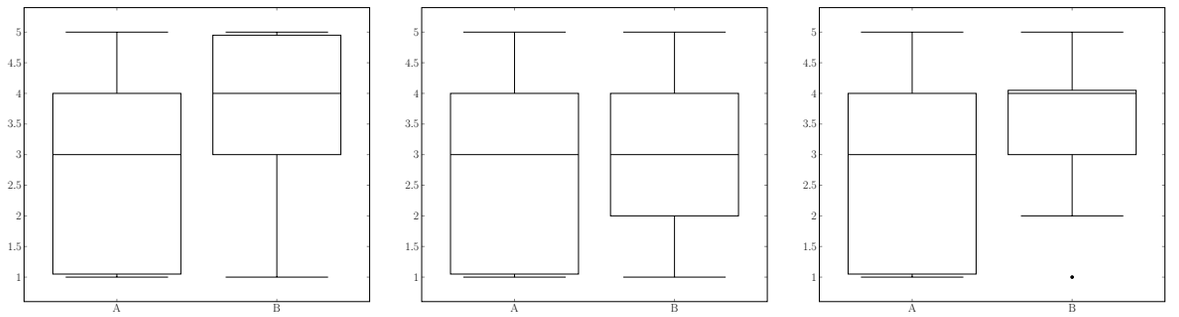
Approach	**Hit**	**MRR**	**MAP**	**MR**
BM25 + API Class	0.58	0.18	0.17	0.10
BM25 + API Class α	0.58	0.18	0.17	0.11
CROKAGE	0.81	0.55	0.49	0.22
CROKAGE α	0.81	0.55	0.49	0.23

position of the class within recommended API ranked list, considering only those present in the Q&A pair (Section 3.4.2). 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 3.4) for each baseline, collect the four metrics and compare the results against their original versions, as shown in Table 14. 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.

4.4 Comparison with State-of-art using a Developer Study

RQ4: *How does CROKAGE perform to provide comprehensive solutions containing code and explanations for given queries (task descriptions) compared to the state-of-art,*



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

Figure 6 – 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 25^{th} and 75^{th} percentiles, respectively, line inside box median. Lower and upper error lines 10^{th} and 90^{th} percentiles, respectively. Filled circle data falls outside 10^{th} and 90^{th} percentiles.

BIKER?

To answer this question, we first choose 50 most popular questions from the same three tutorial sites used to generate our ground truth (discarding questions already used to build the ground truth). 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 30 among these questions and apply standard natural language pre-processing. We assume that developers would use CROKAGE like a search engine. Thus, we restrict the queries to not contain too many words (i.e., maximum of 7 and a minimum of 2 tokens). This process results in 24 queries. 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 evaluations. We asked participants to provide a value from 1 to 5 for each aspect in each tool considering the same criteria used to evaluate our ground truth (Section 4.1). 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 6 represents the participants evaluations.

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. 6-(a)), benefit of the code explanations (Fig. 6-(b)) and the overall solution quality (Fig. 6-(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. 6-(a)) and the overall solution quality (Fig. 6-(c)). Both approaches showed the same median value (i.e,. 3) for the benefit of the code explanations (Fig. 6-(b)). However CROKAGE showed much less Likerts 1 (i.e., 19 against 72) and much more Likerts 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, Peter e Richler 2012].

4.5 Discussion on the Experimental Process

In this Chapter we perform several experiments in order to answer to our four research questions. To measure the performance of the techniques employed in the experiments, we needed a ground truth. Thus, we start the Chapter by adopting a well defined protocol to construct a ground truth for three programming languages in Section 4.1.

In order to select the programming languages, we considered two major factors: (1) popularity and (2) availability of ground truth. First, we select the languages that are within the Top-10 popular languages according to recent Stack Overflow survey[6]. Second, we select the languages for which the ground truth has already been constructed by the existing literature [Yin et al. 2018]. Python nicely fits with both of these criteria. We could not find reliable ground-truth for the other two languages, therefore we had to

[6] https://insights.stackoverflow.com/survey/2019

construct it ourselves. In particular, we choose Java and PHP considering their high popularity and the available expertise of our human evaluators.

After constructing the ground truth, we needed a strategy to search for the solutions for given programming tasks. Given the knowledge stored in Stack Overflow, we model CROKAGE as an Information Retrieval problem. That is, the tool should be able to harness the knowledge contained in Stack Overflow to search and return domain specific solutions for a given natural language query. Moreover, the returned solutions need to be relevant for the given query. We then adopt four well know metrics in software engineering [Xu et al. 2017, Rahman e Roy 2018, Rahman, Roy e Lo 2016, Rahman e Roy 2017, Huang et al. 2018, Silva et al. 2019] to measure the degree of relevance of the returned solutions (Chapter 2 - Section 2.3). Furthermore, since we design CROKAGE for practical use, the tool should be able to return the solutions in a reasonable time. Thus, we establish a threshold of one second for the tool to return the solution for a given query (i.e., programming task).

In order to meet these requirements, we then investigate in Section 4.2 the performance of a series of IR techniques alone. Our goal is to use the IR technique with the best performance to reduce the search space for solutions, since Stack Overflow contains millions of documents in its corpus. In the end, we find that BM25 performs among the best IR techniques and adopt it to reduce our search space in CROKAGE (RQ1).

After discovering the appropriate IR technique for our domain problem, we then, in Section 4.3, construct all baselines, including CROKAGE. We also reproduce the state-of-art BIKER [Huang et al. 2018]. We construct the baselines (except BIKER, whose behaviour we do not change) as the combination of BM25 and each considered factor alone, so that the baselines can return solutions in less than one second per query, as previously defined. We run all baselines to search for relevant answers against the queries of our test set and find that CROKAGE outperforms the other baselines by a significant margin, spending much less than one second per query for the three considered languages (RQ2).

Next, we investigate the influence of each factor of CROKAGE in the ranking of candidate answers (RQ3). For this, we re-calibrate the weights of the remaining factors in each configuration of CROKAGE before running each configuration. We also ensured that they are all executed in less than one second.

So far, we contrasted the recommended answers against the ground truth in order to calculate the four metrics (i.e., Top-K Accuracy, Mean Average Precision, Mean Recall, and Mean Reciprocal Rank) and thus measure the performance of each IR technique (RQ1), baseline (RQ2) or configuration of CROKAGE (RQ3). However, in order to confirm the effectiveness of CROKAGE in practical use, we perform an user study in Section 4.4 with 29 professional developers who evaluate both CROKAGE and the state-of-art BIKER [Huang et al. 2018] under three aspects. Unlike the four metrics adopted in RQ1, RQ2 and RQ3, the tools were evaluated under Likert scales and the results could effectively be used to

show the significant difference between the two tools under the three aspects (RQ4).

This Chapter presents the experiments that show effectiveness of CROKAGE in multiple ways. In particular, we show the comparison of CROKAGE against BIKER [Huang et al. 2018] under multiple aspects.

4.6 Concluding Remarks

In this Chapter we evaluated our approach in multiple aspects. First, we evaluate 11 IR techniques to select candidate Q&A pairs for programming tasks. We find a significant variation in the performance of the 11 tested IR techniques for such a task. In particular, BM25 [Robertson e Walker 1994] achieves the top-3 Accuracy (i.e., Hit) and Mean Reciprocal Rank (i.e., MRR) in two out of the three analyzed languages (i.e., Java and Python) and the top-1 Accuracy for Java, justifying our choice as the adopted IR technique for CROKAGE (RQ1). Second, we compare CROKAGE to other baselines, including the state-of-art, in retrieving relevant answers for given programming tasks. We find that CROKAGE outperforms all baselines for such a task (RQ2). Compared to the state-of-art BIKER [Huang et al. 2018], the superiority of CROKAGE is of 65%, 38%, 21%, and 44% for Hit@K, MAP@K, MR@K, and MRR@K respectively, for K=10. Third, we investigate the influence of each factor of CROKAGE in the ranking of candidate answers. Our findings (RQ3) show that the importance of the API Class factor is irrelevant and its absence does not affect the performance significantly, but the other three factors do matter and their influence vary according to the language. Fourth, we compared CROKAGE against BIKER in a user study (RQ4). Our findings suggest that CROKAGE outperforms BIKER for the three considered criteria: relevance of the suggested code examples, benefit of the code explanations and the overall solution quality.

Such results show the potential of CROKAGE as tool to assist developers with their daily programming issues.

4.7 Threats to validity

Threats to internal validity are related to the baseline methods and the user study. One of the baselines is a state-of-art tool and we extend it to produce the solutions with supplementary information (i.e., answers IDs) without altering its behavior. We double checked the implementation of all baselines to assure they do not contain implementation errors. 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.

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.

Threats to construct validity relates to suitability of our evaluation metrics. We choose four performance metrics: Top-K Accuracy, Mean Reciprocal Rank, Mean Average Precision and Mean Recall, which are widely adopted by related literature in software engineering [Xu et al. 2017, Rahman e Roy 2018, Rahman, Roy e Lo 2016, Rahman e Roy 2017, Huang et al. 2018, Silva et al. 2019].

Chapter 5

CROKAGE Application

We implement CROKAGE in form of a tool to assist developers with their daily programming issues and provide this service free of charge to the community. Our goal is to help the developers to find the desired solutions for their programming tasks in a practical way. As a pilot project, we develop a website so that the developers can input their programming tasks and search for solutions containing code and explanations in such a way they can easily reuse the provided code.

We programmed CROKAGE to log the data inputted by the developers. Our intuition is that the information provided by them during the searches can help us to understand what the developers are looking for, and how they are looking for, so that we can improve the tool in the future. Thus, we set CROKAGE to log the queries and the location of the developers. We use this data later to analyse the composition of the queries in a series of aspects. CROKAGE also collects the feedback provided by the developers on the returned solutions. We use this feedback to understand the reasons of the dissatisfactions.

Herein, we describe the details of the tool. First, in Section 5.1 we present the tool website and how it works. Next, in Section 5.2, we describe CROKAGE's architecture. Then, in Section 5.3, we show an analysis of the queries inputted by the developers during five months after its release in August, 2019.

5.1 Tool Presentation

Figure 7 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 in showed in the algorithm of Figure 4, Section 3.5.

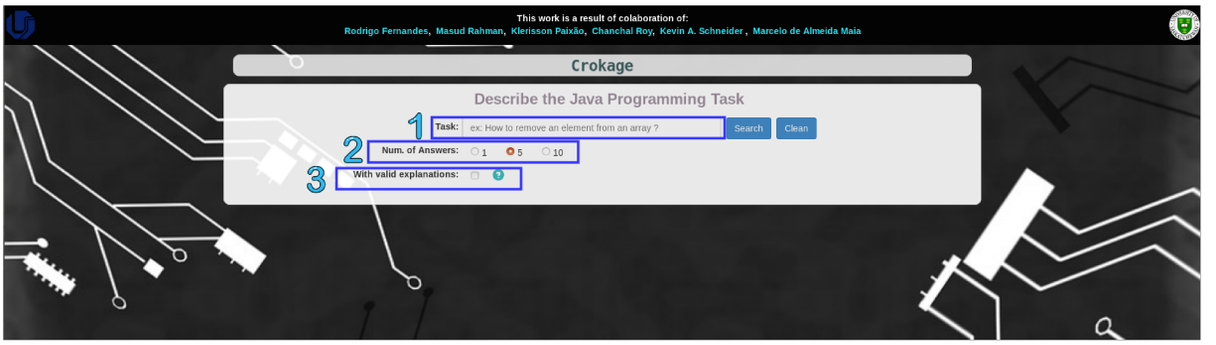


Figure 7 – CROKAGE implementation website: http://isel.ufu.br:9000/. Search Parameters: 1) The field where the user inputs the query (i.e., how to insert an element array in a given position). 2) The number of answers to be retrieved (i.e., 5). 3) Whether only valid explanations should be filtered.

After the user hits the button *Search*, the tool collects the three parameters, performs the search (as described in Section 3.4) and then returns the solutions, as showed in Figure 8. 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 user can scroll down and up to visualize the results.

5.2 Implementation Details

Figure 9 shows the tool architecture. We follow a REST (Representational State Transfer) architecture [Fielding e 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. We provide the replication package of our tool on GitHub[1]. 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 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

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

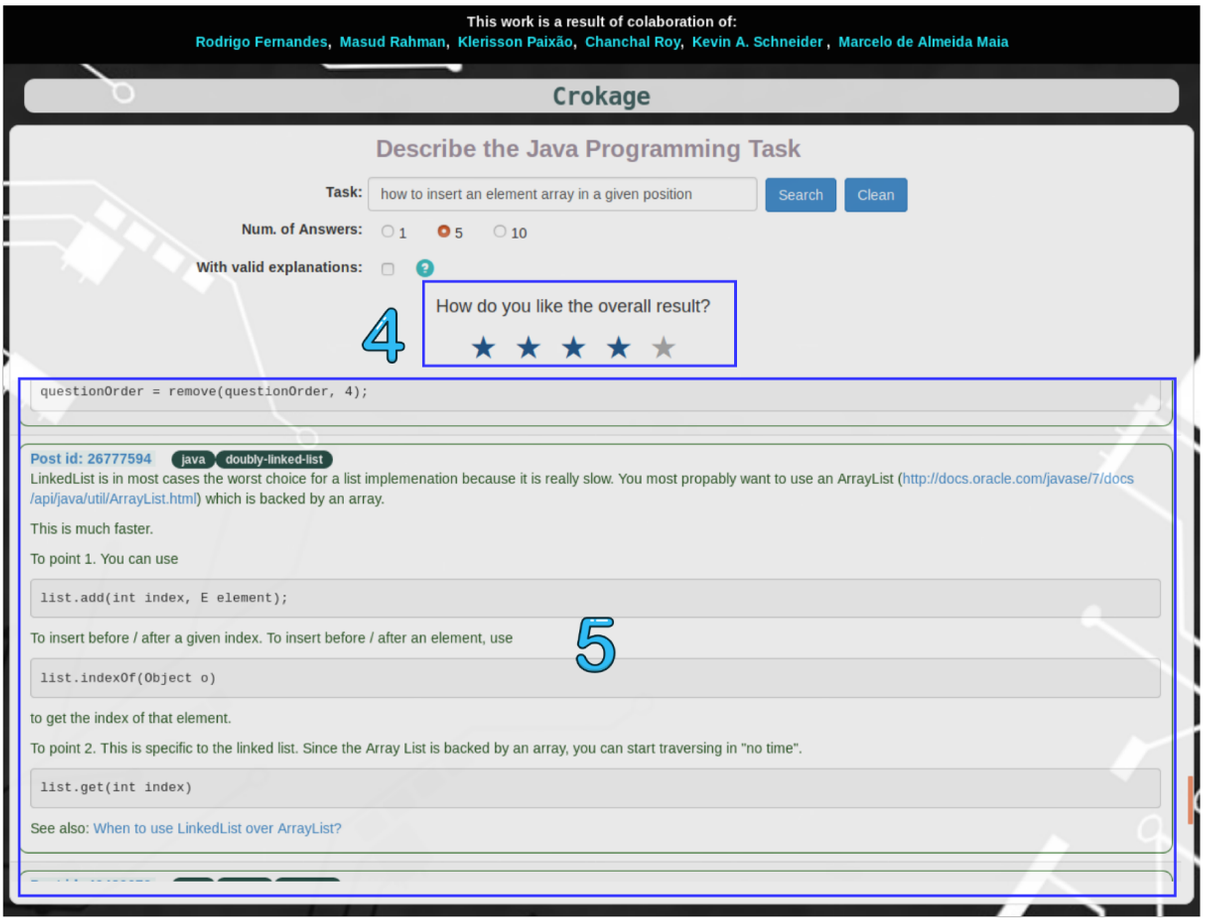


Figure 8 – Search Results: 4) The user feedback about the overall solution. 5) The top n answers related to the parameters, where n is the number of answers selected by the user

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).

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 3.2.

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*

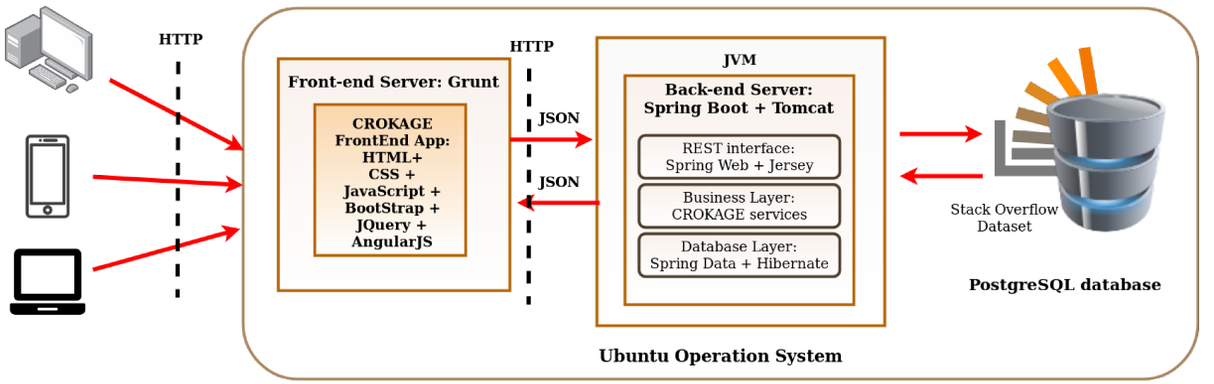


Figure 9 – CROKAGE REST architecture composed by a Front-end Server, a Back-end Server and a database.

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 3.3.

5.3 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 across 83 countries during that interval. Figure 10 shows the countries where the searches originate from and the density of the searches in those countries. We notice that around half of the searches (49.4%) are concentrated in five countries: United States (16.72%), Brazil (12.22%), Germany (11.25%), India (9.18%), and Canada (6.08%).

5.3.1 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%).

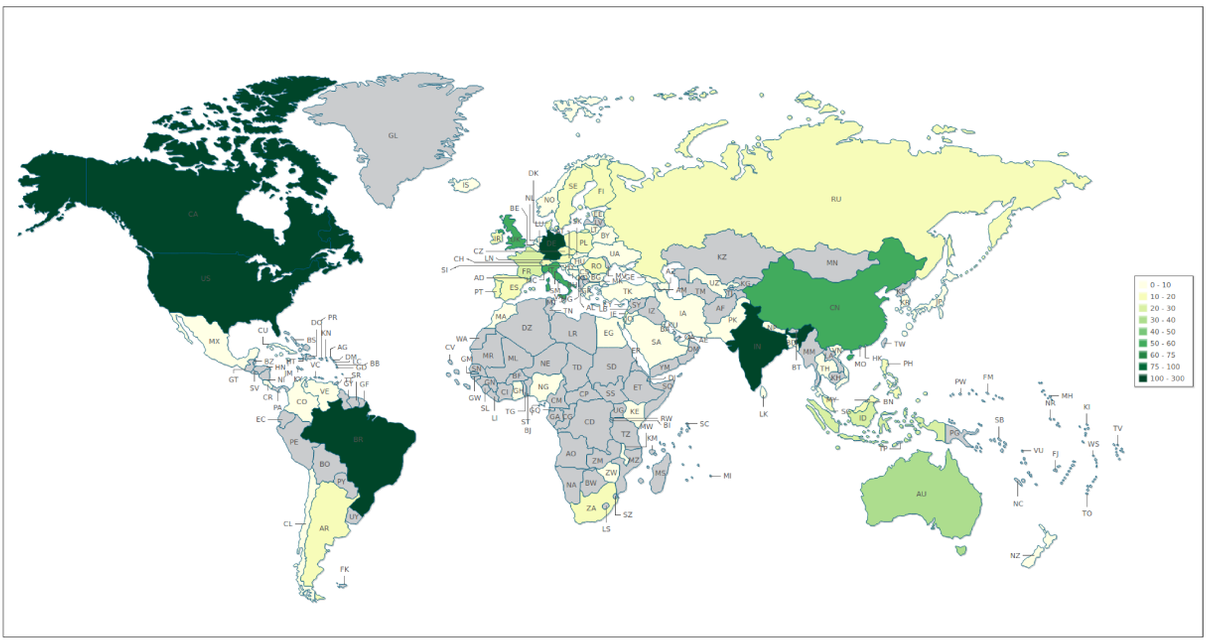


Figure 10 – Countries of the developers who used CROKAGE and the density of the searches: the darker the color of the country, the higher the usage of CROKAGE. No access has been registered by countries in gray

Table 15 – 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

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[2]. 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 15 summarizes the overview of the queries.

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.

[2] 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

Number of Tokens: we classify the valid queries according to their number of tokens[3] as showed in Figure 11. 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"*.

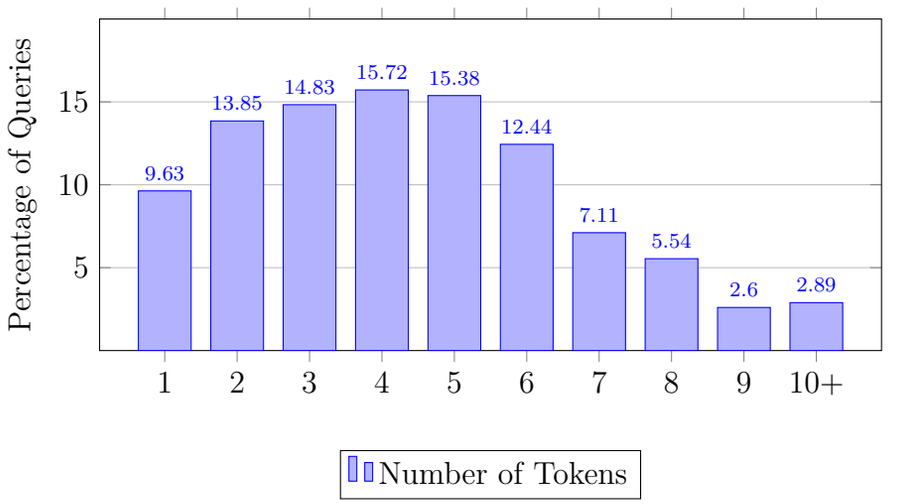


Figure 11 – Composition of developers' queries in terms of the number of tokens and their proportion (in percentage)

Number of Stop Words: we also classify the queries according to the number of stop words[4] as showed in Figure 12. 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, such as *"**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 Figure 13. 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**?"*.

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

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

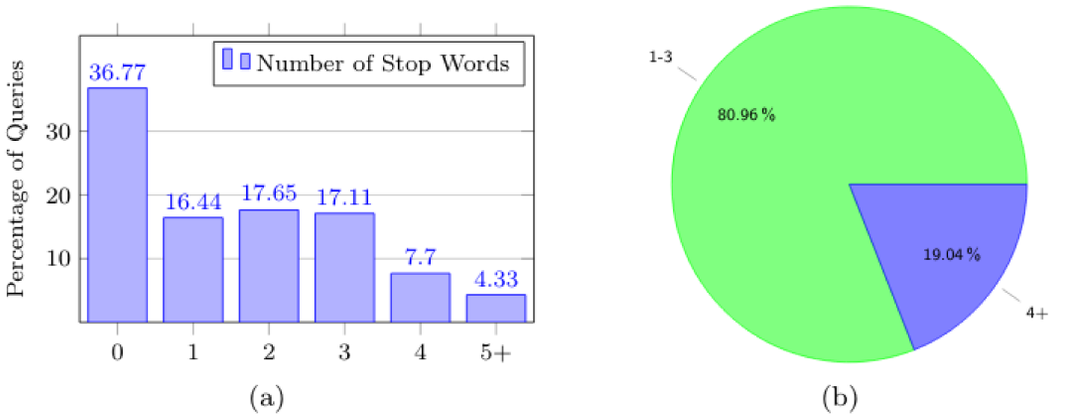


Figure 12 – (a) Composition of developers' queries in terms of the number of stop words and their proportion in percentage and (b) the proportion of stop words in the queries containing stop words. The queries are divided in two groups: 1-3 and 4+

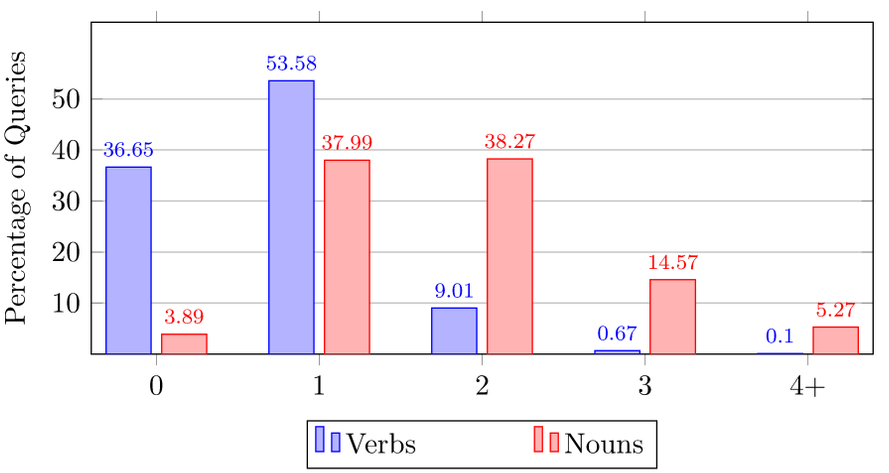


Figure 13 – Composition of developers' queries in terms of the number of verbs and nouns in their POS composition

5.3.2 User Feedback

CROKAGE collects the feedback provided by the users in a 5-star rating (Section 5.2). 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 developers for the 15,865 provided solutions, which corresponds to a feedback rate of 4.9%. The ratings are showed in Figure 14.

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 Figure 14-b, correspond to 55.86% while the negative ones correspond to 35.07%. In 9.43% of the cases, 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

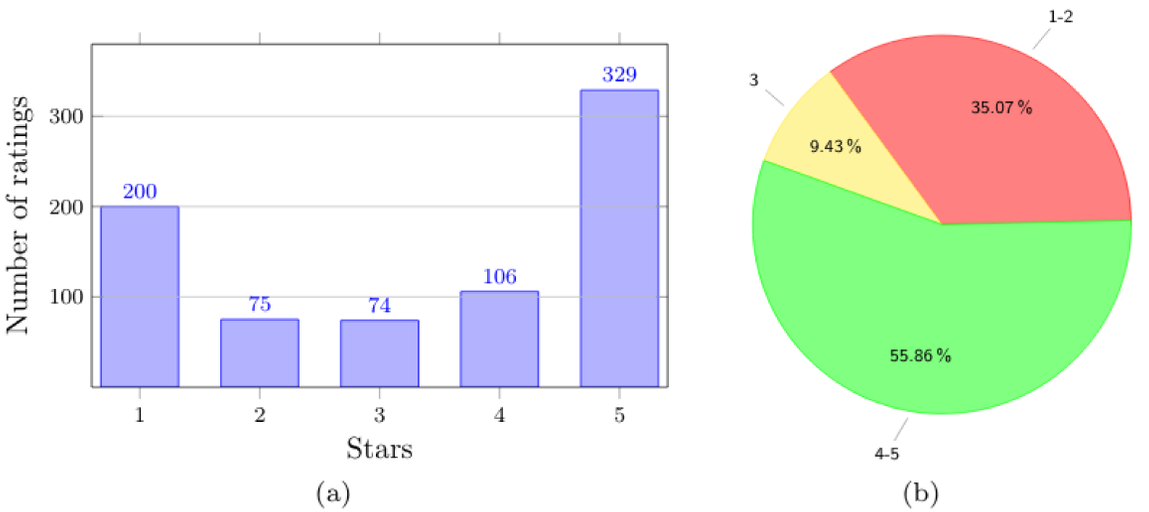


Figure 14 – (a) Number of ratings by number of stars in absolute numbers, and (b) the number of grouped ratings by group 1-2 (poor solution), 3 (related but incomplete) and 4-5 (good solution)

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 16. In the end, nine labels/categories are found as described below:

- **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"*.

Table 16 – 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

- **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"*.
- **Other Language**: queries not related to Java[5], 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"*.

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 for a query that is adherent to CROKAGE's purposes. While 16.9% of the queries do not belong to Java language, the other 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.

We extend Figure 14 to represent the ratings, excluding the ratings for queries classified in all categories except "No obvious reason". That is, we remove from this Figure, the ratings for queries that are not adherent to CROKAGE's purposes. The result is showed in Figure 15. We notice that after removing the ratings for these queries, the proportion of the good solutions rises to 0.75.

5.3.3 Conclusions on the tool usage

The observations made by the two evaluators after the ratings confirm the results obtained in Section 5.3.1: in general, during the code search, the developers tend to use

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

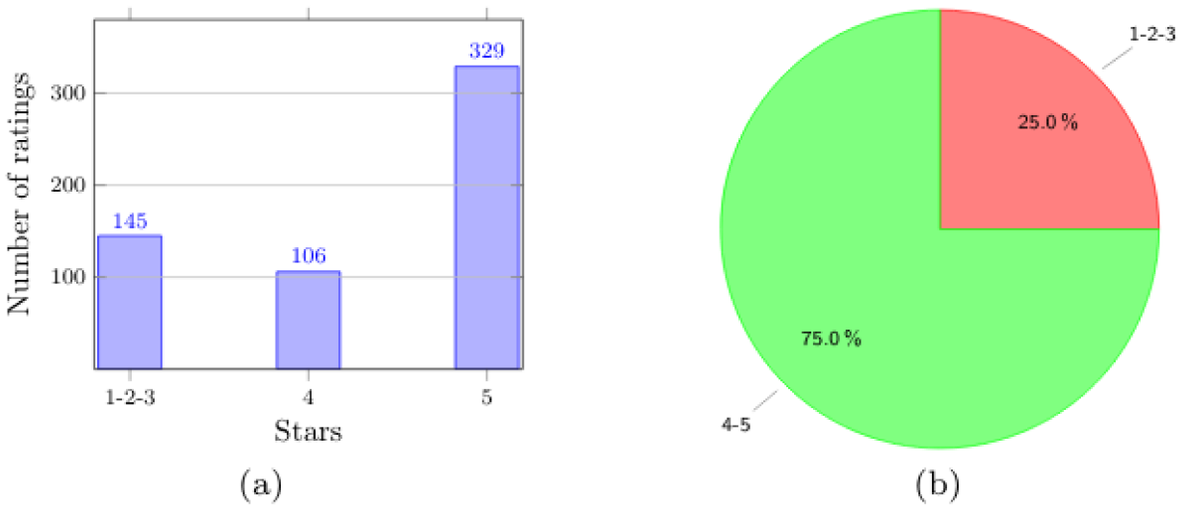


Figure 15 – (a) Number of ratings by number of stars in absolute numbers, and (b) the number of grouped ratings by group 1-2-3 (poor and incomplete solutions) and 4-5 (good solutions)

short queries, containing stop words, and containing one or two nouns, but not always a verb. Furthermore, the words may be misspelled. However, if we consider only the queries that represent *how-to* programming tasks, written in such a way to convey a well defined problem that can be solved by code, and belonging to Java language, the accuracy of the tool in practice rises from 0.55 to 0.75. That is, the Top-K Accuracy of the tool would differ only 0.06 of the 0.81 obtained in RQ2 (Table 8), confirming the efficiency of the tool to retrieve relevant answers among the Top-10 results.

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.

5.3.4 Threats to Validity

Herein, we identify the threats related to the usage analysis of our tool.

Threats to internal validity: three threats can exist. The first threat 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 second threat is related to the quality of the inputted queries by the users. We mitigate this threat by applying a series of filters in order to guarantee that the analyzed queries that are not duplicated, belonging to Java language and containing only ascii characters. The third 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: we rely on the feedback provided by the users in order to perform the open coding and manually classify the queries with low ratings. Hence, a possible threat is related to incorrect ratings for the provided solutions, that could affect our conclusions on the reasons of the developer's dissatisfaction. Although we have no control over the user feedback, it could be the case where the same user provide more than one rating for the same query and solution. Thus, for those cases, we leave only the last rating.

Threats to construct validity: the feedback provided by the users are in the form of a 5-star rating. We found this metric suitable for collecting the user feedback because of its simplicity. Even though, our feedback rate is only 4.9%, which would probably be lower if we have adopted a more sophisticated rating system.

Chapter 6

Related Work

In this chapter we present works related to our approach. We divide them into three main aspects. First, in Section 6.1 we present works related to the exploration of Q&A features to mine knowledge from Stack Overflow. Next, in Section 6.2 we present works related to the provision of code examples. And finally, in Section 6.3 we present works related to the generation of code explanations.

6.1 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. 2017, Zhang et al. 2017, Hoogeveen et al. 2018, Silva, Paixao e Maia 2018, Diamantopoulos e Symeonidis 2015]. Zhang et al. [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.[Ahasanuzzaman et al. 2016] combine the power of the state-of-art information retrieval technique BM25 [Robertson e 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, Paixao e Maia 2018] does not assert their findings. Zhang et al. [Zhang et al. 2017] 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 work (i.e., DupPredictor and Dupe). In another work, Zhang et al. [Zhang et al. 2017] 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.[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 [Diamantopoulos e 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 information retrieval 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. Our input query however, carry much less information than a question. While a question contains different fields of information (e.g., title and description) and associated meta data (e.g., date, tag, upvotes, etc), the query contains only a limited bag of words, limiting the power of information retrieval techniques to match the similarities.

Similarly to detect duplicated questions, existing studies [Xu et al. 2016, Fu e 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. [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 [Fu e Menzies 2017] who presented a simpler and faster method based on support-vector machine (SVM). Later, Xu et al. [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, Bacchelli e Lanza 2013, Ponzanelli et al. 2014, Ponzanelli et al. 2014, Ponzanelli, Bacchelli e Lanza 2013, Ponzanelli et al. 2014]. Ponzanelli et al. [Ponzanelli, Bacchelli e Lanza 2013] 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, enabling the user to import the code from the crowd to the IDE. They implement their approach called Seahawk in form of a Eclipse plugin [Ponzanelli, Bacchelli e Lanza 2013]. Later on, they improve their work to consider the context of the IDE in the search for relevant Q&A threads [Ponzanelli et al. 2014]. 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. 2014], enables 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).

6.2 Code Example Suggestion

There have been several studies [Nguyen et al. 2016, Bajracharya, Ossher e Lopes 2010, Zagalsky, Barzilay e Yehudai 2012, Campbell e Treude 2017, Wang et al. 2016, Gvero e Kuncak 2015, Raghothaman, Wei e Hamadi 2016, Gu et al. 2016, Campos, Souza e Maia 2014, McMillan et al. 2011, Rahman, Roy e Lo 2017, Gu, Zhang e Kim 2018, Huang et al. 2018] that return relevant code against natural language queries. McMillan et al. [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 [Campbell e Treude 2017] develop a tool that assists users providing suggestions to the queries. Rahman et al. [Rahman, Roy e Lo 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 number of searches in a period of time).

Some works [Nguyen et al. 2016, Gu et al. 2016, Raghothaman, Wei e Hamadi 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, Wei e Hamadi 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 distinguish code aspects like method and class names. This concern has been also addressed by DeepCS [Gu, Zhang e Kim 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, Souza e Maia 2014, Lv et al. 2015, Zagalsky, Barzilay e Yehudai 2012, Bajracharya, Ossher e Lopes 2010] rely on lexical similarities to retrieve relevant code. Campos et al. [Campos, Souza e Maia 2014] rank related code documents by applying a combination of Vector Space and Boolean models. The same idea is used by Lv et al. [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. [Zagalsky, Barzilay e Yehudai 2012] propose a tool to retrieve source code based on keywords using TF-IDF to score code documents. Bajracharya et al. [Bajracharya, Ossher e Lopes 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 Chapter 4.

6.3 Code Explanation Generation

Several early studies [Wong, Yang e Tan 2013, Rahman, Roy e Keivanloo 2015, Xu et al. 2017, Chatterjee et al. 2017, Hu et al. 2018, Wong, Liu e Tan 2015] propose automatic approaches to extract explanations to code. For this, they explore lexical properties usually in combination with strategies like clone detection [Wong, Yang e Tan 2013, Wong, Liu e Tan 2015], topics (like LDA) [Rahman, Roy e Keivanloo 2015], word embeddings [Xu et al. 2017], machine learning [Chatterjee et al. 2017] and deep learning [Hu et al. 2018]. Wong et al. [Wong, Yang e Tan 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. [Wong, Liu e Tan 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. [Rahman, Roy e Keivanloo 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. [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. [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. [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 comprehensive survey by Wang et al. [Wang, Peng e Zhang 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.

Chapter 7

Conclusion

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 comprehensive solutions.

The search for relevant answers is composed by three stages. We first employ a fine tuned information retrieval technique called BM25 [Robertson e Walker 1994] to select candidate Q&A pairs from Stack Overflow. Then, we calculate scores between the query and the candidate Q&A pairs according to four similarity factors based on: the API classes related to the query, the API methods common in the pair's answers, the semantic similarity (i.e., word embeddings) and the lexical similarity (i.e., TF-IDF). After scoring the candidate Q&A pairs, we associate appropriate weights to the factors and combine them to rank the pairs. Next, we extract the Top-K relevant answers from the top scored pairs to compose the solutions.

Regarding the choice of the appropriate IR technique, we evaluate 11 IR techniques and find that BM25 [Robertson e Walker 1994] performs among the best. Likewise, we investigate each relevance factor individually for the ranking of candidate answers. We find that the use of API classes (i.e., API Class factor) does not influence the performance significantly, even after obtaining them from three state-of-art API recommendation systems. On the other hand we observe that the union of the factors (i.e., CROKAGE) is statistically superior than any proposed factor alone. Indeed, our findings show that CROKAGE outperforms several other baselines to retrieve relevant answers for programming tasks, including a state-of-art one. 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 relevant answers.

After selecting top quality answers from Stack Overflow for the programming task,

we use them to compose comprehensive solutions. For this, CROKAGE employs natural language processing to remove irrelevant sentences from the answers. Our intuition is that removing these sentences may help developers with a more concise explanation. We develop the tool in such a way that the developer is able to specify the desired number of top quality answers to be used in the solutions composition. The effectiveness of CROKAGE to provide quality solutions for programming tasks is demonstrated by a user study. Our findings show that developers prefer CROKAGE over the state-of-art to find solutions for programming tasks. Furthermore, since CROKAGE builds the solutions from the crowd (i.e., Stack Overflow) whereas the state-of-art retrieves them from the official documentation, we also conclude that developers prefer explanations provided by other users instead of the official ones.

The effectiveness of CROKAGE has also been demonstrated in practical use. The usage analysis of the tool after five months of operation shows that most of the developers are satisfied with the provided solutions. This analysis also reveals that during the code search, the developers tend to use short queries, containing stop words along with one or two nouns, but not necessarily containing verbs. Furthermore, the words may be misspelled. Our findings suggest that code search engines must address these issues in order to be effective on their recommendations.

7.1 Summary of the Contributions

We summarize our contributions as follows:

- ❑ We developed a novel approach –CROKAGE– that assists developers in obtaining solutions for their daily programming tasks. Unlike traditional search code engines such as Google or Bing that retrieves documents in different formats, CROKAGE retrieves solutions in the same format. Moreover, the results of the traditional search code engines do not necessarily contain code and explanations. CROKAGE guarantees that all results contain reusable code followed by explanations. Our tool also remove not important sentences from the answers, making the final solution more concise.

- ❑ We performed an empirical evaluation of CROKAGE on the suggestion of relevant code examples for programming tasks and a comparison against seven baselines, including the state-of-art [Huang et al. 2018]. We showed that CROKAGE outperforms the seven baselines in four performance metrics. Furthermore, we showed that the improvements are found for three popular programming languages.

- ❑ We compared the performance of 11 IR techniques to provide relevant solutions to programming tasks. We showed that the IR techniques perform differently for this

domain problem and that the choice of the technique to retrieve candidate answers for programming tasks must take into consideration the programming language. BM25 [Robertson e Walker 1994], in particular, fits to such a task, achieving 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.

- ❏ We showed an empirical evaluation of four relevance factors: API Class factor, Method, fastText and TF-IDF. We compare these factors in terms of performance to provide relevant solutions to programming tasks. Our findings show that the API Class factor is irrelevant for such a task and its absence does not affect the performance significantly. Furthermore, the cost associated to its use is high, making its use dispensable. On the other hand, the three other factors show improvements for such a task, mostly when combined, and their importance vary according to the language.

- ❏ We compared CROKAGE against the state-of-art work BIKER [Huang et al. 2018] in a user study involving 29 developers. The developers evaluated the solutions generated by the two tools in terms of relevance of the suggested code examples, the benefit of the code explanations and the overall solution quality (code + explanation). The study showed that the developers prefer CROKAGE over BIKER for the three aspects by a significant margin.

- ❏ We constructed a 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 Java and PHP answers were manually evaluated by two professional developers spending 94 man hours. For Python, we leveraged a manually curated dataset provided by a previous work [Yin et al. 2018] and built a process to extract the relevant answers for the programming tasks.

- ❏ We built a replication package[1] containing CROKAGE's prototype, detailed results of our user study and our used dataset for replication or third party reuse. We provide this replication package for the community so that other developers can replicate CROKAGE.

- ❏ We implemented CROKAGE in form of a tool to support developers with their programming tasks. We provide this service free of charge to the community. The tool can be invoked either by a web browser[2] or by any REST (Representational State Transfer) service, whose documentation is publicly availabe[3].

[1] https://github.com/muldon/crokage-emse-replication-package
[2] http://isel.ufu.br:9000/
[3] https://github.com/muldon/CROKAGE-replication-package

- We analyzed the usage of CROKAGE by the community five months after its release. We provided a report on the composition of the queries under four aspects. We also performed an open coding involving two professional developers who manually classified 349 queries with low ratings. The two developers discovered nine categories of queries and found that most of the queries (58.4%) 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. That is, we showed that the code search engines must implement on-the-fly improvements so that such problems do not negatively impact the retrieval of relevant results.

7.2 Future Work

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

- *Flexible Weighting Scheme:* 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. [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. We believe that this idea has potential to generate a better search mechanism that could outperform 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. Furthermore, as verified by our usage analysis of CROKAGE, the queries can contain misspelled words. A quality analysis of queries could show directions of how to reformulate those queries on-the-fly and thus obtain more relevant results. Query reformulation has been proposed before by previous previous work [Nie et al. 2016, Wang, Lo e Jiang 2014, Li et al. 2016, Rahman, Roy e Lo 2017, Rahman e Roy 2018, Rahman, Roy e Lo 2016]. Our similarity factors (TF-IDF, Semantic and Method) could possibly be leveraged to determine the query quality and propose reformulations.

- *Other Features:* CROKAGE relies on four weighted factors/features to estimate the similarity between a programming task and a Stack Overflow question-answer (i.e., Q&A) pair. More investigation is warranted whether other features could better estimate this similarity or dissimilarity so that more relevant pairs could be recommended. We believe that unexplored features such as synonyms, antonyms and social features related to the Stack Overflow discussions have potential to improve the recommendations. Among the social features, we highlight: Answer Count (i.e., the number of answers of a thread/question), Question Score (i.e., the number of upvotes of the question/thread), and the Total Thread Score representing the sum of all answers scores of a thread/question. Our assumption is that these features could help to estimate the similarity/dissimilarity between the query and answers or threads (i.e., discussions).

- *Search for Threads First:* CROKAGE searches for relevant answers directly against a query, disconsidering the discussion as a whole (i.e., the thread). Thus, relevant answers contained in more important threads may be missed, while non relevant answers could be promoted. CROKAGE misses important information from the threads, like their popularity and their relevance against the query. We envision that a search mechanism that would first retrieve the relevant threads against a query and then, of those threads, retrieve the relevant answers, has potential to outperform the current mechanism.

Finally, we conclude this work having presented a new approach -CROKAGE- to assist developers in obtaining solutions for their programming issues. Our tool have outperformed the state-of-art in retrieving relevant answers for given programming tasks by a significant margin. We also have shown the superiority of CROKAGE in comparison with the state-of-art in a user study. We also have shown the effectiveness of CROKAGE in practice through the feedback of the community that access our tool.

Bibliography

AHASANUZZAMAN, M. et al. Mining duplicate questions in stack overflow. In: **Proc. MSR**. [s.n.], 2016. p. 402–412. Disponível em: <https://doi.org/10.1145/2901739.2901770>.

AN, L. et al. Stack Overflow: A code laundering platform? In: **Proc. SANER**. [s.n.], 2017. p. 283–293. Disponível em: <https://doi.org/10.1109/SANER.2017.7884629>.

APACHE. **Lucene**. <http://lucene.apache.org/>.

BAEZA-YATES, R.; RIBEIRO-NETO, B. et al. **Modern information retrieval**. [S.l.]: ACM press New York, 1999. v. 463.

BAJRACHARYA, S.; OSSHER, J.; LOPES, C. Searching API usage examples in code repositories with sourcerer API search. In: **Workshop on Search-driven Development**. [s.n.], 2010. p. 5–8. Disponível em: <https://doi.org/10.1145/1809175.1809177>.

BEGINNERSBOOK. **BeginnersBook**. <http://beginnersbook.com>.

BINKLEY, D.; LAWRIE, D. Information retrieval applications in software development. **Encyclopedia of Software Engineering**, Taylor & Francis LLC, p. 231–242, 2010.

BOJANOWSKI, P. et al. Enriching word vectors with subword information. **Proc. TACL**, MIT Press, v. 5, p. 135–146, 2017. Disponível em: <https://doi.org/10.1162/tacl_a_00051>.

BRANDT, J. et al. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: ACM. **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. 2009. p. 1589–1598. Disponível em: <https://doi.org/10.1145/1518701.1518944>.

BÜTTCHER, S.; CLARKE, C. L.; CORMACK, G. V. **Information retrieval: Implementing and evaluating search engines**. Mit Press, 2016. Disponível em: <https://doi.org/10.1108/02640471111188088>.

Campbell, B. A.; Treude, C. Nlp2code: Code snippet content assist via natural language tasks. In: **Proc. ICSME**. [s.n.], 2017. p. 628–632. Disponível em: <https://doi.org/10.1109/ICSME.2017.56>.

CAMPOS, E. C.; SOUZA, L. B. L. D.; MAIA, M. A. Nuggets miner: Assisting developers by harnessing the Stack Overflow crowd knowledge and the github traceability. In: **Proc. CBSoft-Tool Session**. [S.l.: s.n.], 2014.

CHATTERJEE, P. et al. Extracting code segments and their descriptions from research articles. In: **Proc. MSR**. [s.n.], 2017. p. 91–101. Disponível em: <https://doi.org/10.1109/MSR.2017.10>.

CHEN, C. et al. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. **Proc. TSE**, IEEE, 2019. Disponível em: <https://doi.org/10.1109/TSE.2019.2896123>.

CIBOROWSKA, A.; KRAFT, N. A.; DAMEVSKI, K. Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the web. In: **Proc. MSR**. [s.n.], 2018. p. 94–97. Disponível em: <https://doi.org/10.1145/3196398.3196467>.

DANIELSSON, P.-E. Euclidean distance mapping. **Computer Graphics and image processing**, Elsevier, v. 14, n. 3, p. 227–248, 1980. Disponível em: <https://doi.org/10.1016/0146-664X(80)90054-4>.

DIAMANTOPOULOS, T.; SYMEONIDIS, A. L. Employing source code information to improve question-answering in Stack Overflow. In: **Proc. MSR**. [s.n.], 2015. p. 454–457. Disponível em: <https://doi.org/10.1109/MSR.2015.62>.

EFSTATHIOU, V.; SPINELLIS, D. Semantic source code models using identifier embeddings. In: IEEE PRESS. **Proc. MSR**. 2019. p. 29–33. Disponível em: <https://doi.org/10.1109/MSR.2019.00015>.

FANG, H.; ZHAI, C. An exploration of axiomatic approaches to information retrieval. In: ACM. **Proc. SIGIR**. 2005. p. 480–487. Disponível em: <https://doi.org/10.1145/1076034.1076116>.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern web architecture. **ACM Transactions on Internet Technology (TOIT)**, ACM, v. 2, n. 2, p. 115–150, 2002. Disponível em: <https://doi.org/10.1145/514183.514185>.

FRITZ, C.; PETER, E.; RICHLER, J. Effect size estimates: current use, calculations, and interpretation. **JEPG**, American Psychological Association, v. 141, n. 1, p. 2–18, 2012. Disponível em: <https://doi.org/10.1037/a0024338>.

FU, W.; MENZIES, T. Easy over hard: A case study on deep learning. In: **Proc. ESEC/FSE**. [s.n.], 2017. p. 49–60. Disponível em: <https://doi.org/10.1145/3106237.3106256>.

Google Inc. **Google Search Engine**. <http://google.com>.

GU, X.; ZHANG, H.; KIM, S. Deep code search. In: **Proc. ICSE**. [s.n.], 2018. p. 933–944. Disponível em: <https://doi.org/10.1145/3180155.3180167>.

GU, X. et al. Deep API learning. In: **Proc. FSE**. [s.n.], 2016. p. 631–642. Disponível em: <https://doi.org/10.1145/2950290.2950334>.

GVERO, T.; KUNCAK, V. Interactive synthesis using free-form queries. In: **Proc. ICSE**. [s.n.], 2015. p. 689–692. Disponível em: <https://doi.org/10.1109/ICSE.2015.224>.

HAN, J.; PEI, J.; KAMBER, M. **Data mining: concepts and techniques**. Elsevier, 2011. Disponível em: <https://doi.org/10.1145/565117.565130>.

Hill, E.; Rao, S.; Kak, A. On the use of stemming for concern location and bug localization in java. In: **Proc. SCAM**. [s.n.], 2012. p. 184–193. Disponível em: <https://doi.org/10.1109/SCAM.2012.29>.

HOOGEVEEN, D. et al. Detecting misflagged duplicate questions in community question-answering archives. In: **Proc. ICWSM**. [S.l.: s.n.], 2018. p. 112–120.

HU, X. et al. Deep code comment generation. In: **Proc. ICPC**. [s.n.], 2018. p. 200–210. Disponível em: <https://doi.org/10.1145/3196321.3196334>.

HUANG, Q. et al. API method recommendation without worrying about the task-API knowledge gap. In: **Proc. ASE**. [s.n.], 2018. p. 293–304. Disponível em: <https://doi.org/10.1145/3238147.3238191>.

JAVA2S. **Java2s**. <http://java2s.com>.

JELINEK, F. Interpolated estimation of markov source parameters from sparse data. In: **Proc. Workshop on Pattern Recognition in Practice, 1980**. [S.l.: s.n.], 1980.

JSOUP. **Java HTML Parser**. <http://jsoup.org>.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **Biometrics**, v. 33, n. 1, p. 159–174, 1977. Disponível em: <https://doi.org/10.2307/2529310>.

LI, Z. et al. Query reformulation by leveraging crowd wisdom for scenario-based software search. In: **Proceedings of the 8th Asia-Pacific Symposium on Internetware**. [s.n.], 2016. p. 36–44. Disponível em: <https://doi.org/10.1145/2993717.2993723>.

LV, F. et al. Codehow: Effective code search based on API understanding and extended boolean model (e). In: **Proc. ASE**. [s.n.], 2015. p. 260–270. Disponível em: <https://doi.org/10.1109/ASE.2015.42>.

MANNING, C.; RAGHAVAN, P.; SCHÜTZE, H. Introduction to information retrieval. **Natural Language Engineering**, Cambridge university press, v. 16, n. 1, p. 100–103, 2010. Disponível em: <https://doi.org/10.1017/S1351324909005129>.

MCMILLAN, C. et al. Portfolio: finding relevant functions and their usage. In: **Proc. ICSE**. [s.n.], 2011. p. 111–120. Disponível em: <https://doi.org/10.1145/1985793.1985809>.

Microsoft Inc. **Bing Search Engine**. <http://bing.com>.

MIHALCEA, R. et al. Corpus-based and knowledge-based measures of text semantic similarity. In: **Aaai**. [S.l.: s.n.], 2006. v. 6, n. 2006, p. 775–780.

Mikolov et al. **Fast Text**. <https://fasttext.cc/docs/en/unsupervised-tutorial.html>.

MIKOLOV, T. et al. Efficient estimation of word representations in vector space. **arXiv preprint arXiv:1301.3781**, 2013.

_____. Distributed representations of words and phrases and their compositionality. In: **Proc. NIPS**. [S.l.: s.n.], 2013. p. 3111–3119.

NASEHI, S. M. et al. What makes a good code example?: A study of programming q&a in stackoverflow. In: IEEE. **Proc. ICSM**. 2012. p. 25–34. Disponível em: <https://doi.org/10.1109/ICSM.2012.6405249>.

NGUYEN, T. et al. T2API: synthesizing API code usage templates from english texts with statistical translation. In: **Proc. FSE**. [s.n.], 2016. p. 1013–1017. Disponível em: <https://doi.org/10.1145/2950290.2983931>.

NGUYEN, T. V. et al. Combining word2vec with revised vector space model for better code retrieval. In: IEEE PRESS. **Proc. ICSE**. 2017. p. 183–185. Disponível em: <https://doi.org/10.1109/ICSE-C.2017.90>.

NIE, L. et al. Query expansion based on crowd knowledge for code search. **IEEE Transactions on Services Computing**, IEEE, v. 9, n. 5, p. 771–783, 2016. Disponível em: <https://doi.org/10.1109/TSC.2016.2560165>.

PAGLIARDINI, M.; GUPTA, P.; JAGGI, M. Unsupervised learning of sentence embeddings using compositional n-gram features. **arXiv preprint arXiv:1703.02507**, 2017. Disponível em: <https://doi.org/10.18653/v1/N18-1049>.

PONZANELLI, L.; BACCHELLI, A.; LANZA, M. Leveraging crowd knowledge for software comprehension and development. In: **Proc. CSMR**. [s.n.], 2013. p. 57–66. Disponível em: <https://doi.org/10.1109/CSMR.2013.16>.

_____. Seahawk: Stack Overflow in the IDE. In: **International Conference on Software Engineering (ICSE)**. [s.n.], 2013. p. 1295–1298. Disponível em: <https://doi.org/10.1109/ICSE.2013.6606701>.

PONZANELLI, L. et al. Mining Stack Overflow to turn the IDE into a self-confident programming prompter. In: **Proc. MSR**. [s.n.], 2014. p. 102–111. Disponível em: <https://doi.org/10.1145/2597073.2597077>.

_____. Prompter: A self-confident recommender system. In: IEEE. **Proc. ICSME**. 2014. p. 577–580. Disponível em: <https://doi.org/10.1109/ICSME.2014.99>.

RAGHOTHAMAN, M.; WEI, Y.; HAMADI, Y. SWIM: Synthesizing what I mean-code search and idiomatic snippet synthesis. In: **Proc. ICSE**. [s.n.], 2016. p. 357–367. Disponível em: <https://doi.org/10.1145/2884781.2884808>.

RAGKHITWETSAGUL, C. et al. Toxic code snippets on Stack Overflow. **arXiv:1806.07659**, 2018. Disponível em: <https://doi.org/10.1109/TSE.2019.2900307>.

RAHMAN, M. M.; ROY, C. K. STRICT: Information retrieval based search term identification for concept location. In: **Proc. SANER**. [s.n.], 2017. p. 79–90. Disponível em: <https://doi.org/10.1109/SANER.2017.7884611>.

_____. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In: **Proc. ICSME**. [s.n.], 2018. p. 473–484. Disponível em: <https://doi.org/10.1109/ICSME.2018.00057>.

RAHMAN, M. M.; ROY, C. K.; KEIVANLOO, I. Recommending insightful comments for source code using crowdsourced knowledge. In: **Proc. SCAM**. [s.n.], 2015. p. 81–90. Disponível em: <https://doi.org/10.1109/SCAM.2015.7335404>.

RAHMAN, M. M.; ROY, C. K.; Lo, D. Rack: Automatic api recommendation using crowdsourced knowledge. In: **Proc. SANER**. [s.n.], 2016. p. 349–359. Disponível em: <https://doi.org/10.1109/SANER.2016.80>.

RAHMAN, M. M.; ROY, C. K.; LO, D. Rack: Code search in the IDE using crowdsourced knowledge. In: **Proc. ICSE**. [s.n.], 2017. p. 51–54. Disponível em: <https://doi.org/10.1109/ICSE-C.2017.11>.

ROBERTSON, S. E.; JONES, K. S. Relevance weighting of search terms. **Journal of the American Society for Information science**, Wiley Online Library, v. 27, n. 3, p. 129–146, 1976. Disponível em: <https://doi.org/10.1002/asi.4630270302>.

ROBERTSON, S. E.; WALKER, S. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In: **Proc. ACM SIGIR**. [s.n.], 1994. p. 232–241. Disponível em: <https://doi.org/10.1007/978-1-4471-2099-5_24>.

SARYADA, W. **KodeJava**. <http://kodejava.org>.

SILVA, R. F. et al. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In: IEEE PRESS. **Proceedings of the 27th International Conference on Program Comprehension**. 2019. p. 358–368. Disponível em: <https://doi.org/10.1109/ICPC.2019.00054>.

Silva, R. F. G.; Paixao, K. V. R.; MAIA, M. A. Duplicate question detection in stack overflow: A reproducibility study. In: **Proc. SANER**. [s.n.], 2018. p. 572–581. Disponível em: <https://doi.org/10.7287/peerj.preprints.26555>.

Stack Exchange Inc. **Stack Overflow Search Engine**. <http://stackoverflow.com>.

WANG, S.; LO, D.; JIANG, L. Active code search: incorporating user feedback to improve code search relevance. In: ACM. **Proceedings of the 29th ACM/IEEE international conference on Automated software engineering**. 2014. p. 677–682. Disponível em: <https://doi.org/10.1145/2642937.2642947>.

WANG, X.; PENG, Y.; ZHANG, B. Comment generation for source code: State of the art, challenges and opportunities. **arXiv:1802.02971**, 2018.

WANG, Y. et al. Hunter: next-generation code reuse for java. In: **Proc. FSE**. [s.n.], 2016. p. 1028–1032. Disponível em: <https://doi.org/10.1145/2950290.2983934>.

WILCOXON, F. Individual comparisons by ranking methods. **Biometrics bulletin**, v. 1, n. 6, p. 80–83, 1945. Disponível em: <https://doi.org/10.2307/3001968>.

WONG, E.; LIU, T.; TAN, L. Clocom: Mining existing source code for automatic comment generation. In: **Proc. SANER**. [s.n.], 2015. p. 380–389. Disponível em: <https://doi.org/10.1109/SANER.2015.7081848>.

WONG, E.; YANG, J.; TAN, L. Autocomment: Mining question and answer sites for automatic comment generation. In: **Proc. ASE**. [s.n.], 2013. p. 562–567. Disponível em: <https://doi.org/10.1109/ASE.2013.6693113>.

XU, B. et al. Prediction of relatedness in stack overflow: deep learning vs. svm: a reproducibility study. In: ACM. **Proc. ESEM**. 2018. p. 21. Disponível em: <https://doi.org/10.1145/3239235.3240503>.

_____. Answerbot: Automated generation of answer summary to developersź technical questions. In: **Proc. ASE**. [s.n.], 2017. p. 706–716. Disponível em: <https://doi.org/10.1109/ASE.2017.8115681>.

_____. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: **Proc. ASE**. [s.n.], 2016. p. 51–62. Disponível em: <https://doi.org/10.1145/2970276.2970357>.

YANG, D. et al. Stack overflow in GitHub: any snippets there? In: **Proc. MSR**. [s.n.], 2017. p. 280–290. Disponível em: <https://doi.org/10.1109/MSR.2017.13>.

YE, X. et al. From word embeddings to document similarities for improved information retrieval in software engineering. In: **Proc. ICSE**. [s.n.], 2016. p. 404–415. Disponível em: <https://doi.org/10.1145/2884781.2884862>.

YIN, P. et al. Learning to mine aligned code and natural language pairs from stack overflow. In: **Proc. MSR**. ACM, 2018. (MSR), p. 476–486. Disponível em: <https://doi.org/10.1145/3196398.3196408>.

ZAGALSKY, A.; BARZILAY, O.; YEHUDAI, A. Example Overflow: Using social media for code recommendation. In: **Proc. RSSE**. [s.n.], 2012. p. 38–42. Disponível em: <https://doi.org/10.1109/RSSE.2012.6233407>.

ZAMANIRAD, S. et al. Programming bots by synthesizing natural language expressions into API invocations. In: **Proc. ASE**. [s.n.], 2017. p. 832–837. Disponível em: <https://doi.org/10.1109/ASE.2017.8115694>.

ZHAI, C.; LAFFERTY, J. A study of smoothing methods for language models applied to information retrieval. **TOIS**, ACM, v. 22, n. 2, p. 179–214, 2004. Disponível em: <https://doi.org/10.1145/984321.984322>.

ZHANG, W. E. et al. Detecting duplicate posts in programming qa communities via latent semantics and association rules. In: **Proc. WWW**. [S.l.: s.n.], 2017. p. 1221–1229.

_____. Feature analysis for duplicate detection in programming qa communities. In: SPRINGER. **Proc. ADMA**. 2017. p. 623–638. Disponível em: <https://doi.org/10.1007/978-3-319-69179-4_44>.

ZHANG, Y. et al. Multi-factor duplicate question detection in Stack Overflow. **JCST**, v. 30, n. 5, p. 981–997, Sep 2015. ISSN 1860-4749. Disponível em: <https://doi.org/10.1007/s11390-015-1576-4>.