Information and Software Technology 127 (2020) 106367

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

PostFinder: Mining Stack Overflow posts to support software developers

Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen*, Juri Di Rocco, Davide Di Ruscio

Università degli studi dell'Aquila, L'Aquila, Italy

ARTICLE INFO

Keywords:
Recommender systems
Mining Stack Overflow posts
Indexing posts

ABSTRACT

Context – During the development of complex software systems, programmers look for external resources to understand better how to use specific APIs and to get advice related to their current tasks. Stack Overflow provides developers with a broader insight into API usage as well as useful code examples. Given the circumstances, tools and techniques for mining Stack Overflow are highly desirable. **Objective –** In this paper, we introduce PostFinder, an approach that analyzes the project under development to extract suitable context, and allows developers to retrieve messages from Stack Overflow being relevant to the API function calls that have already been invoked. **Method –** PostFinder augments posts with additional data to make them more exposed to queries. On the client side, it boosts the context code with various factors to construct a query containing information needed for matching against the stored indexes. Multiple facets of the data available are used to optimize the search process, with the ultimate aim of recommending highly relevant SO posts. **Results –** The approach has been validated utilizing a user study involving a group of 12 developers to evaluate 500 posts for 50 contexts. Experimental results indicate the suitability of PostFinder to recommend relevant Stack Overflow posts and concurrently show that the tool outperforms a well-established baseline. **Conclusions –** We conclude that PostFinder can be deployed to assist developers in selecting relevant Stack Overflow posts while they are programming as well as to replace the module for searching posts in a code-to-code search engine.

1. Introduction

Developing complex software systems requires mastering several languages and technologies [1]. Thus, software developers need to devote effort to continuously understand how to use new third-party libraries even by consulting existing source code or heterogeneous sources of information. The time spent on discovering useful resources can have a dramatic productivity impact [2].

Over the last few years, many studies have been conducted to develop methods and tools being able to provide automated assistance to developers. The introduction of recommender systems to the domain of software development has brought substantial benefits. Among others, recommender systems assist the developer in navigating large information spaces and getting instant recommendations that might be helpful to solve the particular development problem at hand [3,4]. A recommender system in software engineering is defined as *"... a software application that provides information items estimated to be valuable for a software engineering task in a given context"* [1]. In general, developers have to master a vast number of information sources [5], often at a short time. In such a context, the problem is not the lack of information but instead an information overload coming from heterogeneous and rapidly evolving sources. Thus, recommender systems aim at giving developers recommendations, which may consist of different items, including code examples, issue reports, reusable source code, possible third-party components, and documentation.

Stack Overflow (SO) [6] is the most popular question-and-answer website [7], which is a good source of support for developers who seek for probable solutions from the Web [8,9]. SO discussion posts provide developers with a broader insight into API usage, and in some cases, with sound code examples. Moreover, in a recent development, Stack Overflow has been exploited by a code-to-code search engine to enrich code queries, with the ultimate aim of getting relevant source code. In particular, FaCoY [10] has been developed to recommend relevant GitHub code snippets to a project being developed. First, the system retrieves related SO posts to get more relevant source code. Afterwards, it exploits the newly obtained source code to expand the query and search from GitHub for more snippets, which are eventually introduced to developers. The module to retrieve posts plays a decisive role: it is a blocking issue in FaCoY's performance, if the module cannot retrieve any relevant posts, the system is unable to generate recommendations.

In this sense, we see the importance of getting related SO posts, given a code snippet as context. As the information space is huge, it is necessary to have tools that help narrow down the search scope as well as find the most relevant documentations [1]. However, how to construct

* Corresponding author.
E-mail addresses: riccardo.rubei@univaq.it (R. Rubei), claudio.disipio@univaq.it (C. Di Sipio), phuong.nguyen@univaq.it (P.T. Nguyen), juri.dirocco@univaq.it (J. Di Rocco), davide.diruscio@univaq.it (D. Di Ruscio).

https://doi.org/10.1016/j.infsof.2020.106367
Received 24 April 2020; Received in revised form 8 June 2020; Accepted 14 June 2020
Available online 25 June 2020
0950-5849/© 2020 Elsevier B.V. All rights reserved.

a query that best describes the developer's context and how to properly prepare SO data to be queried are still challenging tasks [4]. In particular, there is a need to enhance the quality of retrieved posts as well as to refine the input context to generate decent queries.

In this work, we propose PostFinder, a Stack Overflow posts recommender system, which is based on a two-phase approach to retrieve posts from Stack Overflow by taking various measures on both the data collection and query phases. To improve efficiency, we make use of Apache Lucene [11] to index the textual content and code coming from Stack Overflow. During the first phase, posts are retrieved and augmented with additional data to make them more exposed to queries. Afterwards, we boost the context code with different factors to construct a query that contains information needed for matching against the stored indexes. In a nutshell, we make use of *multiple facets* of the data available at hand to optimize the search process, with the ultimate aim of recommending highly relevant SO posts. Our work is twofold: *(i)* by providing SO posts, PostFinder can be used to replace FaCoY's SO module; more importantly, *(ii)* PostFinder can work as a standalone tool: given a snippet as context, the tool can provide highly relevant posts to the developer. Through a series of user studies, we demonstrate that our proposed approach considerably improves the recommendation performance, and thus outperforming the considered baseline. In this sense, our paper makes the following contributions:

- Identification of augmentation measures to automatically refine the considered input SO dump by considering various pieces of information;
- Characterizing the context code by automatically boosting the constituent terms to improve their exposure to the indexed data, and eventually build a proper query transparently for the developer;
- Two empirical evaluations of the proposed approach to evaluate the performance of PostFinder and to compare it with FaCoY;
- An implementation of the tool, which was successfully integrated into the Eclipse IDE, has been released together with the corresponding metadata to facilitate future research [12].

The paper is structured as follows. Section 2 provides background and describes the motivations for our work. In Section 3, we introduce PostFinder, the proposed approach to recommending SO posts. The evaluation is presented in Section 4. Section 5 analyzes the experimental results and discusses the threats to validity. In Section 6 we present related work and conclude the paper in Section 7.

2. Background

Over the last decade, several approaches have been conceived to leverage the use of crowdsourcing in software engineering [13]. Those exploiting Stack Overflow as the main source of information (e.g., [4,10,14–16]) can be classified into two main categories:

C1. approaches that focus on the automated creation of queries to be executed by search engines, and on the visualization of the retrieved posts according to some ranking model (e.g., [4,14,15]);
C2. approaches that deal with query creation and advanced indexing mechanisms specifically conceived for storing and retrieving SO posts (e.g., [10,16]).

PROMPTER [4] is among the most recent approaches falling in the first category above. It is an automatic tool, which is used to recommend SO posts given an input context built from source code. PROMPTER performs various processing steps to produce a query. First, it splits identifiers and removes stop words. Then, it ranks the terms according to their frequency by also considering the entropy in the entire SO dump. Once the query is built, the tool exploits a web service to perform the query via the Google and Bing search engines. Finally, a ranking model is employed to sort the results according to different metrics such as API similarity, tags analysis, and SO answers and questions.

```
1  package camelinaction;
2  import org.apache.camel.CamelContext;
3  import org.apache.camel.builder.RouteBuilder;
4  import org.apache.camel.impl.DefaultCamelContext;
5
6  public class FilePrinter {
7        public static void main(String args[]) throws Exception {
8              // create CamelContext
9              CamelContext context = new DefaultCamelContext();
10             // add our route to the CamelContext
11             context.addRoutes(new RouteBuilder() {
12                   public void configure() {
13                   }
14             });
15       }
16 }
```

Listing 1. Explanatory input context code.

FaCoY [10] is a recent code-to-code search engine that relies on Apache Lucene and provides developers with relevant GitHub code snippets. Two main phases are conducted to produce recommendations as follows. The first one is performed on the context code to get related SO posts from a local indexed dump. To this end, the system parses the context code and builds an initial query q_c to look for posts from Stack Overflow. From the set of retrieved posts, it parses natural language descriptive terms from questions to match against the question index of Q&A that has been built ex-ante to get more posts that contain relevant source code. Afterwards, a new query q_c' is formed from the newly obtained source code. The second phase is done on q_c' to search from GitHub for more snippets, which are finally introduced to developers. By focusing on the first phase, i.e., searching for SO posts by exploiting the input context code, FaCoY can be considered in category *C2* above. This module works like a bridge between the initial query q_c and the final results. In this sense, it has an important role to play since its performance considerably affects the final recommendation outcomes. The experimental results [10] demonstrate that FaCoY obtains a superior performance with regards to several baselines [17,18]

In the scope of this paper, we pay our attention to the FaCoY's module for searching SO posts. By a careful observation on the system, we found out that it suffers a setback for incomplete data as well as a brief input query. As we can see later in the paper, for many queries the system is unable to retrieve any SO posts, or for some contexts, it suggests irrelevant ones, i.e., false positives. To this end, we believe there is a need to overcome the limitations so as to enhance the overall performance of FaCoY.

As an example, we consider the explanatory code snippet shown in Listing 1. The code declares a `CamelContext` variable, and invokes functions `addRoutes()` and `configure()`. This illustrates the situation when a developer invokes Camel and searches for posts discussing how to use XML. The input code is pretty simple, and the developer would benefit from being suggested with SO posts that provide discussions related to the input source code.

In other words, it is expected that a search engine can recommend discussions that are relevant to the developer context, for instance the post[1] shown in Fig. 1. For the sake of clarity, we only capture the key information from the post, i.e., title, question, answer, code and display it in the figure. The post contains two answers and both of them are useful for the context. For instance, the depicted snippet contains class `CurrencyRoute` where `addRoutes()` is informative and the function `configure()` is completely defined. More importantly, this code shows that some additional packages are required,

[1] https://tinyurl.com/yydp8lwd

XML to object in Apache Camel

I'm trying to fetch www.dnb.no/portalfront/datafiles/miscellaneous/csv/kursliste_ws.xml and convert it to a Java object using xstrem.

0

I get this error:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
org/apache/camel/spi/DataFormatName Caused by: java.lang.ClassNotFoundException:
org.apache.camel.spi.DataFormatName
```

2 Answers

active oldest votes

1

This is the working route:

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;
import org.apache.log4j.BasicConfigurator;

import javax.jms.ConnectionFactory;

public class CurrencyRoute {

    public static void main(String args[]) throws Exception {
        // Log 4j
        BasicConfigurator.configure();

        // Create camel context
        CamelContext context = new DefaultCamelContext();

        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://
        context.addComponent("test-jms", JmsComponent.jmsComponentAutoAcknowledge(c
        // New route
        context.addRoutes(new RouteBuilder() {
            public void configure() {

                from("quartz://myTimer?trigger.repeatCount=0")
                        .log("### Quartz trigger ###")
                        .to("direct:readFile");
```

Fig. 1. A Stack Overflow post relevant to Listing 1.

e.g., `ActiveMQConnectionFactory` or `JmsComponent`. In this sense, the accompanying snippet is handy for supporting the development of the context code in Listing 1 since it provides a better insight into how to use the related APIs.

When the context code shown in Listing 1 is fed to FaCoY, the system fails to return any results. We anticipate that this is due to the lack of input data, i.e., the query code is very brief, and to the indexing process of FaCoY, which ignores some crucial components in source code when preparing the indexed data. Thus, we believe that there is still room for improvement. In this respect, Lucene offers a well-defined platform for managing and indexing data. However, it is incumbent upon the Lucene user to decide which data to index and which data to use as a query. To this end, we propose an approach to improve the module for searching SO posts of FaCoY. We also attempt to perform various refinement steps on the input SO dump as well as to polish the query code. By doing this, we are able to enrich the initial query with multiple pieces of information. Furthermore, we also increase the exposure of the background data collected from Stack Overflow by means of the boosting mechanism provided by Lucene. In fact, the post in Fig. 1 is recommended by PostFinder when we feed it with the query in Listing 1. This indicates that for the motivating example, our tool is more effective compared to FaCoY. In the next section, we are going to explain in detail the proposed approach.

3. Proposed approach

Given a user context consisting of the source code under development, we aim at searching for posts that contain highly relevant answers from Stack Overflow. We attempt to overcome the limitations of the existing approaches by properly indexing SO data and by processing the query by developers' side, exploiting various refinement techniques. In particular, we come up with a comprehensive approach named PostFinder, which takes into consideration three consecutive phases, i.e., *Index Creation, Query Creation*, and *Query Execution*. By *Index Creation*, we parse and organize an SO dump into a queryable format to facilitate future search operations. *Query Creation* is done at the de-

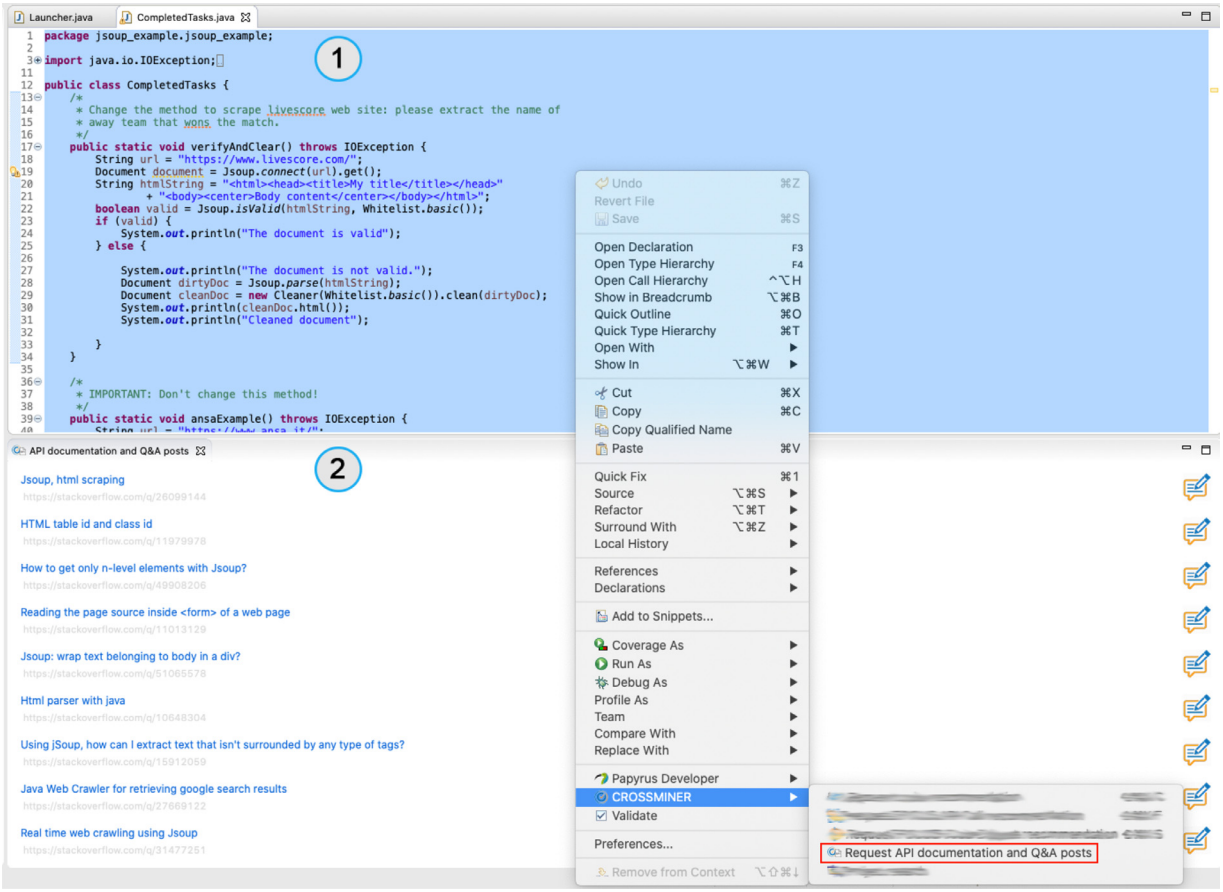


Fig. 2. PostFinder at work.

veloper's side to transform the current context into an informative query that can be used to search against the indexed data. Concerning *Query Execution*, the actual searching is performed employing Apache Lucene.

The index creation, query creation, and query execution phases are completely transparent to the user, who can ask for SO posts directly from the implemented Eclipse IDE plugin as shown in Fig. 2. In particular, the user selects the source code that she wants to include in the query ①, asks for the recommendation and gets a list of retrieved SO posts ②.

An overview of the PostFinder building components is depicted in Fig. 3. The three constituent phases, i.e., *Index Creation, Query Creation*, and *Query Execution* are described in the following subsections.

3.1. Index creation

Starting from an SO dump, the original data is loaded into MongoDB for further processing. Then, the data is parsed and transformed into a format that can be queried later on. At this point, it is necessary to use indexed data for future look up. We opt for Apache Lucene as it is a powerful IR tool widely used to manage and query vector data. For each SO post, the following components are excerpted: `Title`, `Body` and `Code`. Concerning the textual part, we extract questions, answers, and titles and index them using the `Indexer`. Meanwhile, code contents are parsed to extract useful artifacts before being fed to Lucene. In particular, the JDT parser is used to obtain six tokens as shown in the `Code` part of Table 1. However, code snippets in Stack Overflow posts may neither be complete nor compilable [19], and thus they cannot be found if being queried in their original format. Thus, to make them compilable, we propose two refinement steps, namely *Post cleaning and filtering* and *Code wrapping* as explained below.

Table 1
The used facets.

	Token	Description
Text	`Title`	The title of the post
	`Answer`	All answers contained in the post
	`Question`	The question
Code	`ImportDeclaration`	The directives used to invoke libraries
	`MethodDeclaration`	Method declarations with parameters
	`MethodInvocation`	API function calls
	`VariableType`	Types of all declared variables
	`VariableDec`	All declared variables
	`ClassInstance`	Class declarations

3.1.1. Post cleaning and filtering

In SO messages, a question is typically followed by answers and comments. However, many posts do not have any answers at all, and they are considered to be not useful for recommendation tasks. Thus, we filter out irrelevant posts as well as remove the low quality ones first by considering only those that have accepted answers. Then, only posts that contain the `code` tag[2] to include in their bodies Java source code are accepted for further processing.

[2] We are aware that there are SO posts that do not always use the `code` tag to include inline source code. Thus, relying only on such a tag might discard messages that instead should be kept. Natural Language Processing techniques can be exploited to make the employed cleaning and filtering phase less strict even though we defer this as future work.

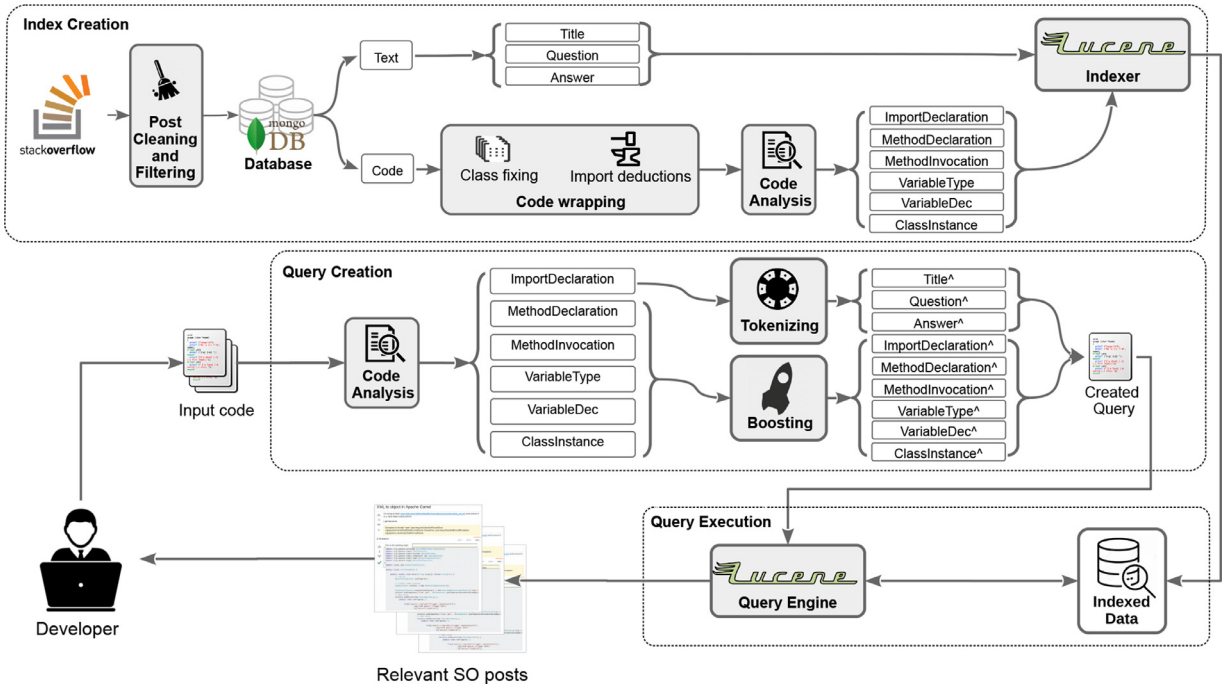


Fig. 3. The PostFinder architecture.

```
ASTParser parser = ASTParser.newParser(AST.JLS9);
parser.setResolveBindings(true);
parser.setKind(ASTParser.K_COMPILATION_UNIT);
parser.setSource(snippet.toCharArray());
Hashtable<String,String> options = JavaCore.getOptions();
options.put(JavaCore.COMPILER_DOC_COMMENT_SUPPORT, JavaCore.ENABLED);
parser.setCompilerOptions(options);
```

Listing 2. Original code snippet.

3.1.2. Code wrapping

To deal with incomplete and uncompilable snippets, e.g., those that have neither class structure nor import directives, we use the parsing option to yield more tokens. For code snippets without any imports, we wrap up with relevant classes to make them more informative. We exploit an archive made available by Benelallam et al. [20] to find the right library that an invocation belongs to, and thus, the corresponding `import` directives. The dataset contains more than 2.8M artifacts together with their dependencies as well as other relationships, e.g., versions. We count the frequency that each artifact is invoked, and sort into a ranked list in descending order. Then only top *N* class canonical names, i.e., the ones appear in an import statement, are selected. By parsing all API calls within a method declaration, we trace back to their original packages from the top *N* names. Nevertheless, given that a class instance is invoked without any declaration, more than one canonical name could be found there. In this case, we compute the Levenshtein distance from each name to the title and body text of the post and use it as a heuristic to extract the best matched one. Finally, the corresponding import directives are placed at the beginning of the code. It is important to remark that the choices related to the different design alternatives, threshold values, augmentation mechanisms, etc. have been performed iteratively and empirically to maximize the accuracy of the overall approach.

We consider an example as follows: Listing 2 depicts a code snippet extracted from SO. The code contains just function calls, and it is incomplete since there is neither class declaration nor import. If we use this code without any refinement to index Lucene, it might not be unearthed by the search engine due to the lack of data.

```
import java.util.HashMap;
import java.util.Hashtable;
import org.eclipse.jdt.core.JavaCore;
import org.eclipse.jdt.core.dom.AST;
import org.eclipse.jdt.core.dom.ASTParser;
import org.eclipse.jdt.core.dom.ASTVisitor;

public class fix(){
   ASTParser parser = ASTParser.newParser(AST.JLS9);
   parser.setResolveBindings(true);
   parser.setKind(ASTParser.K_COMPILATION_UNIT);
   parser.setSource(snippet.toCharArray());
   Hashtable<String, String> options = JavaCore.getOptions();
   options.put(JavaCore.COMPILER_DOC_COMMENT_SUPPORT, JavaCore.ENABLED);
   parser.setCompilerOptions(options);
}
```

Listing 3. Augmented code snippet.

By adding class fix and import, we obtain a new code snippet as shown in Listing 3. The snippet resembles a real hand-written code, which probably facilitates the matching process later on.

Once the refinement steps have been done, all terms corresponding to the tokens specified in the `Code` part of Table 1 are indexed and stored into Lucene for further look up.

3.2. Query creation

This phase is conducted on the client side, and the method declaration being developed is used as input context. A query can be formed by considering all terms extracted from the context code. It is evident that each term in posts has a different level of importance. Thus, the second phase is to equip a query with more information that better describes the current context, taking into account the terms' importance level. Fortunately, Lucene supports *boosting*, a scoring mechanism to assign a weight to each indexed token. Based on scoring, we perform two augmentation steps, i.e., *Boosting*, and *Tokenizing* as follows.

```
VariableDeclarationType: CamelContext^1.0 OR
VariableDeclaration: context^1.0 OR
MethodInvocation: addRoutes^1.0 OR
ClassInstance: DefaultCamelContext^1.0 OR
ImportDeclaration: org.apache.camel.impl.DefaultCamelContext^1.0 OR
ImportDeclaration: org.apache.camel.CamelContext^1.0 OR
ClassInstance: RouteBuilder^1.0 OR
MethodDeclaration: main^1.0 OR
ImportDeclaration: org.apache.camel.builder.RouteBuilder^1.0 OR
MethodDeclaration: configure^1.0 OR
Answer: apache^1.4 OR
Answer: camel^1.4 OR
Question: apache^1.4 OR
Question: camel^1.4 OR
Title: apache^4 OR
Title: camel^4
```

Listing 4. Sample query produced after the *Boosting* and *Tokenizing* phases.

3.2.1. Boosting

The original code is parsed to obtain the six tokens listed in the last half of Table 1. Each term in the code is assigned a concrete weight to boost the level of importance. Entropy [21] is exploited to compute the quantity of information of a document using the following formula: $H = -\sum p(x)\log p(x)$, where $p(x)$ is the probability of term x. An entropy value ranges from 0 to $log(n)$, where n is the number of terms within the document. We compute entropy for all terms in the original source code and rank them in a list of descending order. Then the first quarter of the list is assigned a boost value of 4. Similarly, the next 2nd, 3rd, and 4th quarters get the boost value of 3, 2, and 1, respectively. Finally, all the code terms are attached to their corresponding tokens to form the query.

3.2.2. Tokenizing

By the *Index Creation* phase in Section 3.1, nine different tokens have been populated (see Table 1). Among them, there are three textual tokens, i.e., `Title`, `Answer`, and `Question`. However, by the developer's side, the input context contains just code and there are no textual parts that can be used to match against the three tokens. Thus, given the input code, we attempt to generate textual tokens by exploiting the import directives embedded at the beginning of each source file. Starting from an import directive, we break it into smaller pieces and attach them to all the textual tokens. A previous work [22] shows that in an SO post, the title is more important than the description. In particular, the importance ratio between description and title of a given post is 1/3 [22]. Accordingly, we set a boost value of 4 to the title and 1.4 to both the answer and question. Empirical evaluations conducted on the dataset demonstrate that the boost value has a smaller impact than that of the ratio between the title and the body of the post. Thus, we integrate it into the best query configuration.

By considering the code in Listing 1, the query ready to be executed that PostFinder creates after the boosting and tokenizing phases is shown in Listing 4.

3.3. Query execution

Queries that are created as described in the previous section, are executed by means of Apache Lucene. Moreover, we exploit the Lucene built-in BM25 to rank indexed posts. In particular, BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of the inter-relationship between the query terms within a document, e.g., their relative proximity. The index is computed as given below.

$$R(q,d) = \sum_{t \in q} \frac{f_t^d}{k_1((1-b) + b\frac{l_d}{avgl_d} + f_t^d} \quad (1)$$

where f_t^d is the frequency of term t in document d; l_d is the length of the document d; $avgl_d$ is the document average length along the collection; k is a free parameter usually set to 2 and $b \in [0, 1]$. When $b = 0$, the normalization process is not considered and thus the document length does not affect the final score. In constrast, when $b = 1$, the full-length normalization is performed. In practice, b is normally set to 0.75. It has been shown that for ranking documents, BM25 works better than the standard TF-IDF one [23]. By considering the query in Listing 4 as input, the post shown in Fig. 1 is among the resulting ones.

In the following section we introduce two evaluations to examine if our proposed solution is beneficial to the matching of relevant SO posts.

4. Evaluation methods and materials

As mentioned above, PostFinder is a multi-purpose tool: On one side, it can work as an independent engine to search for suitable Stack Overflow posts to directly support developers while they are programming. On the other side, it can be used to replace the first module of FaCoY.

For the former, the most relevant system with PostFinder is PROMPTER [4] whose original implementation is, unfortunately, no longer functioning.[3] Considering the fact that the re-implementation of the tool is not straightforward and time consuming, we decided to investigate what queries would be potentially hard for PROMPTER to answer but for PostFinder would be easy, and vice versa by exploiting some use case studies from the original PROMPTER paper and apply PostFinder on them.

By the latter, since the source code of FaCoY is available, in this work we concentrate on comparing PostFinder with FaCoY by means of an evaluation on a common dataset. For the sake of representation, from now on the baseline is addressed as FaCoY (unless otherwise stated), despite the fact that it is the first module, albeit the most decisive one, of the whole FaCoY system [10].

The following subsections describe the performed evaluations in detail. In particular, we present a discussion about the high-level differences between PostFinder and PROMPTER in Section 4.1. The experimental configurations for supporting the comparison of PostFinder and FaCoY are explained in Section 4.2. Section 4.3 presents the dataset used in the evaluation. Section 4.4 explains the evaluation methodology, and Section 4.5 introduces the evaluation metrics. The research questions are introduced in Section 4.6.

4.1. Differences between PostFinder and PROMPTER

We investigate the differences between the first ranked recommended post of the two approaches by relying on a set of queries and the corresponding results obtained by PROMPTER, provided by its authors.[4] To make the comparison fair as much as possible, we filter PostFinder's posts by selecting only those that date back to the year 2014, i.e., the date of the dump used by PROMPTER [4]. As the first example, depicted in Listing 5, we present a particular developer's context in which PROMPTER performs better than PostFinder with respect to the retrieved first top rank posts. As we can see, the post[5] recommended by PROMPTER shown in Fig. 4 is more relevant given the context rather than the one[6] retrieved by PostFinder shown in Fig. 5. The rationale behind this fact can be found in the query's lines of code. In particular, PostFinder misses some valuable results when it is fed with a very small context, as there are few import statements in such cases.

[3] Through private communications, the authors of PROMPTER informed us that the tool had not been maintained for a long time, and they also had difficulties in making it run again.

[4] We gratefully acknowledge the data provided by Prof. Dr. Michele Lanza and Dr. Luca Ponzanelli.

[5] https://tinyurl.com/ybmny6p8

[6] https://tinyurl.com/y8zarqsb

Fig. 4. A post related to Listing 5 retrieved by PROMPTER.

```
1  /*
2  List Filesystem roots
3  */
4  import java.io.*;
5  public class ListFileSystemRoots {
6        public static void main(String[] args) {
7              File[] rootDirectories = File.listRoots();
8              System.out.println("Available root directories in filesystem are: ");
9              for(int i=0 ; i < rootDirectories.length ; i++)
10                   System.out.println(rootDirectories[i]);
11       }
12 }
```

Listing 5. Comparing PostFinder with PROMPTER: First use case.

In contrast, the example shown in Listing 6 demonstrates that PostFinder is capable of providing a more relevant post. Given a more detailed context, PROMPTER's post[7] reported in Fig. 6 misses some additional information that PostFinder can deliver[8] as shown in Fig. 7. Moreover, more results could be retrieved by PostFinder using the wrapping technique described in Section 3.1.2. We cannot directly verify this claim as the PROMPTER's queries included in the provided set do not fall into this category, i.e., they contain compilable snippets of code.

According to the performed light-weight comparison, we noticed that PostFinder performs better than PROMPTER if the input query contains more import statements, even when the code is not compilable. However, to deliver a fair and a thorough comparison of both approaches, it is necessary to re-implement the PROMPTER tool by strictly following the descriptions in the original paper [4], and we consider this as a possible future work.

```
1  package varius;
2
3  import javax.swing.JFrame;
4  import javax.swing.JPanel;
5  import javax.swing.JProgressBar;
6
7  //Define a vertical progress bar
8  public class ProgressBar extends JFrame {
9        JProgressBar current = new JProgressBar(JProgressBar.VERTICAL, 0, 2000);
10       int num = 0;
11       public ProgressBar() {
12             setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13             JPanel pane = new JPanel();
14             current.setValue(0);
15             current.setStringPainted(true);
16             pane.add(current);
17             setContentPane(pane);
18       }
19
20       public void iterate() {
21             while (num < 2000) {
22                   current.setValue(num);
23                   try {
24                         Thread.sleep(1000);
25                   } catch (InterruptedException e) { }
26                   num += 95;
27             }
28       }
29       public static void main(String[] arguments) {
30             ProgressBar frame = new ProgressBar();
31             frame.pack();
32             frame.setVisible(true);
33             frame.iterate();
34       }
35 }
```

Listing 6. Comparing PostFinder with PROMPTER: Second use case.

[7] https://tinyurl.com/ybe37xch
[8] https://tinyurl.com/y9o38kq6

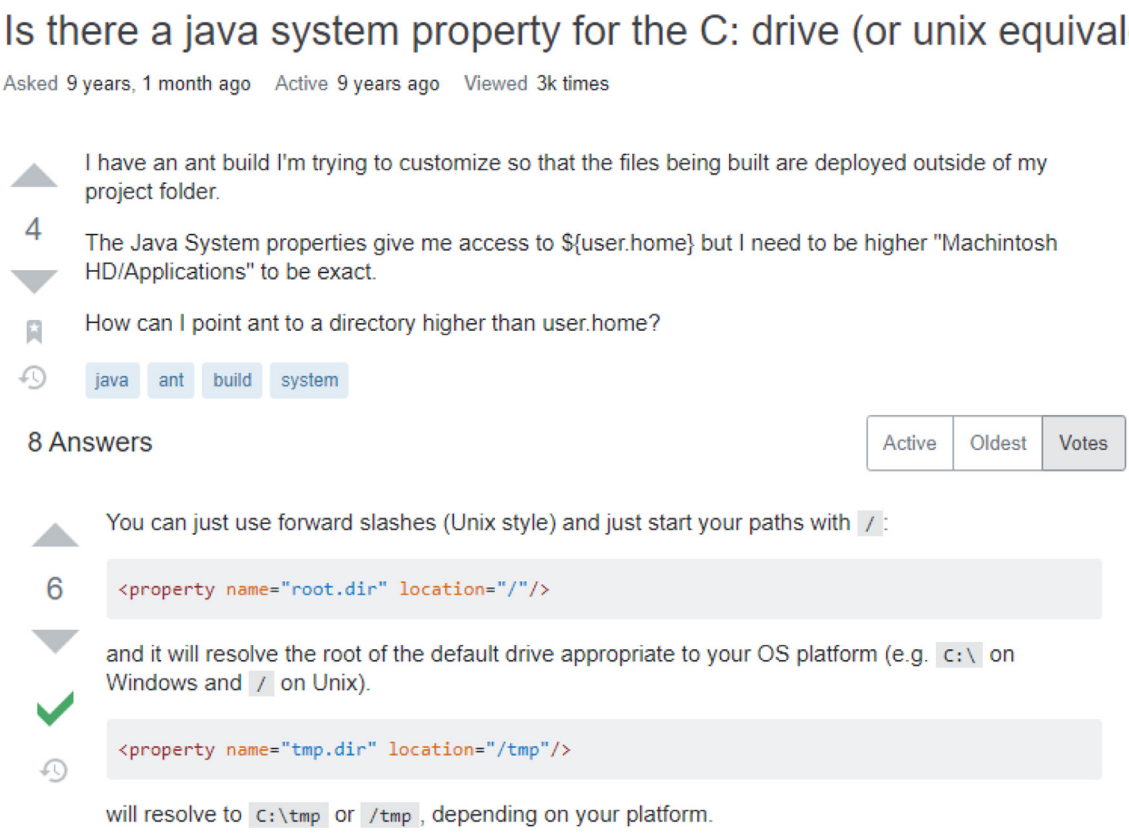


Fig. 5. A post related to Listing 5 retrieved by PostFinder.

Table 2
Experimental configurations.

Configuration	Wrapping	Boosting	Tokenizing	Ranking	# queries	Description
A	✗	✗	✗	✗	10	Flat queries, without considering any proposed augmentations
B	√	✗	✗	✗	10	Wrapping is introduced to queries in Configuration A
C	√	✗	✗	√	10	BM25 is used to rank the retrieved posts
D	√	√	✗	✗	10	Entropy is used to boost the queries in Conf. B
E	√	✗	√	✗	10	Transforming import directives to textual tokens for queries in Conf. B
F	√	√	√	√	10	Imposing all proposed augmentations
G	√	√	√	√	50	Imposing all proposed augmentations to compare PostFinder and FaCoY

4.2. Configurations

To study PostFinder's performance, we perform two main evaluations by means of user studies. The first one is done to evaluate the role of each augmentation proposed in Section 3. To this end, we consider six experimental configurations, i.e., **A**, **B**, **C**, **D**, **E**, **F** with ten queries for each (see Table 2). The second evaluation compares PostFinder with FaCoY, and this corresponds to the last configuration **G**. To thoroughly examine the difference in their performance, we consider 50 queries in **G**. The queries contain code snippets that invoke ten of the most popular Java libraries, i.e., Jackson, SWT, MongoDB driver, Javax Servlet, JDBC API, JDT core, Apache Camel, Apache Wicket, Twitter4j, Apache POI. The rationale behind the selection of such libraries is that through a careful observation, we realized that generally, feeding FaCoY with queries coming from random libraries might not yield any results, i.e., the anwer is blank, and such an outcome is not useful for any comparison. Thus, we have to select queries that return some posts which can eventually be used to evaluate both FaCoY and PostFinder.

The test configurations are explained in Table 2, whose the 2nd to 5th columns specify the presence of the techniques mentioned in Section 3, with the corresponding section being shown in parentheses. For example, the 2nd column **Wrapping** is a combination of *class fixing* and *library import deduction* (cf. Section 3.1). The column **Boosting** indicates the usage of the entropy for the calculation of the boosting values (cf. Section 3.2.1), **Tokenizing** refers to the usage of a *Maven dataset* in order to augment the imports (cf. Section 3.2.2). Finally column **Ranking** dictates the use of the BM-25 weighting scheme (cf. Section 3.3). To facilitate future research, we made available the PostFinder tool together with the related data in GitHub [12].

4.3. Dataset

To provide input for the evaluation, we exploited a Stack Overflow dump[9] of June 2017, which is an XML file of around 70GB in size, and contains more than 18 millions of posts. By filtering with tags, we obtained 757,439 posts containing Java source code. The resulting set has more than 1.2 millions of answers with 49.20% of them being already accepted, i.e., 552,458 answers. In such posts there were 32,578 snippets that have no imports. We fixed them as presented in Section 3.1.2. Eventually, we indexed and parsed all the posts following the paradigm described in Section 3.1. More details of the dataset used in our evaluation are shown in Table 3.

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

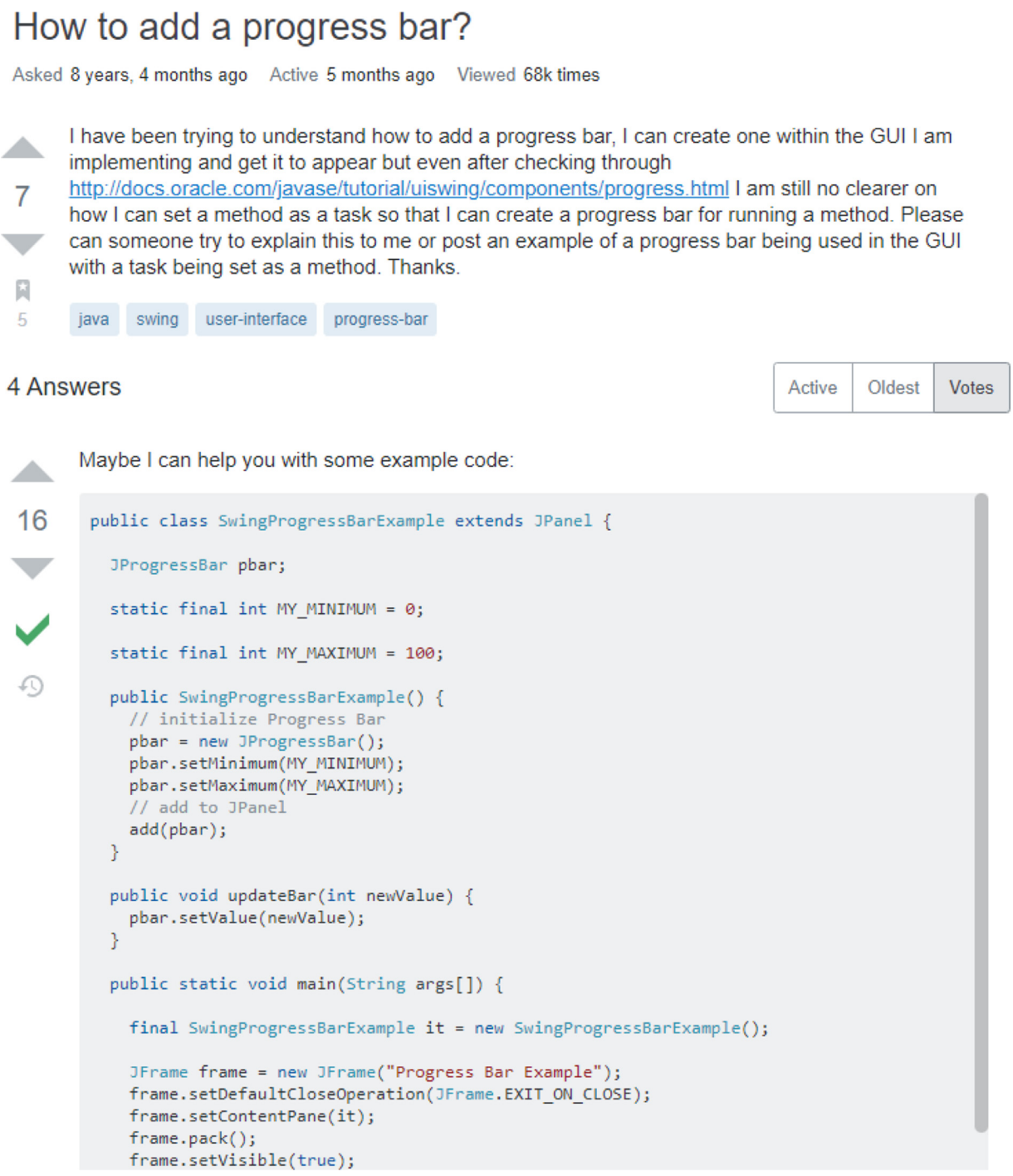


Fig. 6. A post related to Listing 6 retrieved by PROMPTER.

Table 3
A summary of the SO dump used in the evaluation.

Name	Value	Name	Value
Size	70GB	**# of answers**	1,122,789
# of posts	18,300,672	**# of acc. answers**	552,458
# of Java posts	757,439	**# of posts fixed**	32,578

4.4. User study

We resort to user studies as this is the only way to investigate whether the recommendation outcomes are really helpful to solve a specific task [4,24,25]. The input is expressed as a snippet of code together with a set of recommended posts, and the task is to judge how relevant the retrieved posts are, with respect to the input code. A group of 11 developers was asked to participate in the user studies. Six participants were master students attending a Software Engineering course. Three of them were 1st year PhD students, and the other two were postdoc researchers. Through a survey sent to each participant, we found out that more than a half of them had at least seven years of programming experience. Among these people, three participants have worked with programming for 15 years. All of them are capable of Java and at least another programming language, e.g., Python or C++. The evaluators use code search engines like GitHub, Stack Overflow, or Maven

how to include a progress bar in a program of file transfer using sockets in java

Asked 6 years, 11 months ago Active 6 years, 11 months ago Viewed 4k times

0

i am working on a project in java that transfers files from a client to a server. now i need to show the progress bar for each file transfer i.e the progress bar should automatically pop up as each transfer starts. i have made the program to create a progress bar but i am not able to merge it with the client-server programs. i would really appreciate if someone helps me with the loop to be used. thank you.

2 Answers

Active | Oldest | Votes

1

Swing is a single threaded framework. That means that all interactions with the UI are expected to be made within the context of this thread (AKA The Event Dispatching Thread).

This also means that if you performing any kind of time consuming/long running or blocking process within the EDT, you will prevent it from responding to events or updating the UI.

This is what your code is currently doing.

There are a number of mechanism available to you to over come this, in your case, the simplest is probably to use a `SwingWorker`

```
import java.awt.EventQueue;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;
import javax.swing.SwingWorker;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

public class ProgressBar extends JFrame {

    JProgressBar current = new JProgressBar(0, 100);
    int num = 0;

    public ProgressBar() {
        //exit button
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //create the panel to add the details
        JPanel pane = new JPanel();
        current.setValue(0);
```

Fig. 7. A post related to Listing 6 retrieved by PostFinder.

on a daily basis. Furthermore, they frequently re-use code fragments collected from these external sources. This is advantageous for the manual evaluation, since in the scope of this paper we consider only source code written in Java, and we assume that skilful Java developers shall have a better judgment about the relevance among the code being developed and the recommended posts. The knowledge of different programming languages, e.g., Perl, Python, is also a plus for the evaluation process.

Given a query, each system, i.e., FaCoY or PostFinder produces as outcome a ranked list of posts, considered to be relevant. To aim for a fair comparison, we mixed the top-5 results generated by each system in a single Google form and presented them to the evaluators together with the corresponding context code. This simulates a *taste test* [26] where users are asked to give feedback for a product, e.g., food or drink, without having a priori knowledge about what is being addressed. This aims to eliminate any possible bias or prejudice against a specific system. Each pair of code and post, i.e., $< query, retrieved\ post >$ is examined and evaluated by at least two participants. Each evaluator first inspects the input code snippet, then reads the post including text and source code to comprehend its purpose, and finally judges the relevance using the criteria listed in Table 4.

Apart from 11 developers mentioned before, we also involved one more researcher in validating the evaluation outcome by playing the role of a *mediator*. In case there is a substantial disagreement between any two participants, e.g., the first person assigned the score of 2 and the other one gave 4 to a same pair (cf. Table 4), the mediator examines the pair again to eventually reach a consensus. It is worth noting that to avoid bias, also in this phase the mediator did not know ex-ante where each pair comes from, i.e., FaCoY or PostFinder. In most cases, the evaluators agreed on the scores. The disagreement happened mainly within closely relevant scores: 1 with 2, or 3 with 4. Only when there were pairs with very different scores, e.g., 1 with 3, or 2 with 4, the mediator was asked to re-evaluate and judge. The ratio of such a disagreement was 15%. Moreover, an inter-rater agreement analysis using Cohen's Kappa test [27] revealed that the mediator substantially agreed with both evaluators, $\kappa(eval_1, mediator) = 0.677$ and $\kappa(eval_2, mediator) = 0.761$.

Table 4
Relevance Scores.

Score	Description
0	No results at all are returned
1	The post is totally irrelevant
2	The post contains some hints but it is still out of context
3	There are relevant suggestions but the key features are missing
4	The post provides proper recommendations and the related code snippets are useful considering the development's context

4.5. Evaluation metrics

As typically done in related work, the following metrics have been considered to evaluate the recommendation outcomes [24,25]:

- *Relevance*: it is the score given to a pair of $<$ *query, retrieved post* $>$ following Table 4;
- *Success rate*: if at least one of the top-5 retrieved posts receives 3 or 4 as score, the query is considered to be relevant. *Success rate* is the ratio of relevant queries to the total number of queries;
- *Precision*: it is computed as the ratio of pairs in the top-5 list that have a score of 3 or 4 to the total number of pairs, i.e., 5.

4.6. Research questions

The evaluations are conducted to answer the following research questions:

- **RQ$_1$**: *Which experimental configuration brings the best PostFinder performance?* We compare the flat configuration with the augmented ones to see which setting fosters the best recommendation outcome for PostFinder.
- **RQ$_2$**: *How does PostFinder compare with FaCoY?* Compared to FaCoY, our tool is equipped with various refinement techniques. By answering this question, we ascertain whether our proposed augmentations are useful for searching posts in comparison to the original approach FaCoY.
- **RQ$_3$**: *What are the reasons for the performance difference?* We are interested in understanding the factors that add up to the performance difference between the two systems.

The following section analyzes the systems' performance by addressing these research questions.

5. Experimental results

This section presents the results obtained from the experiments as well as related discussions. In Section 5.1, we analyze the outcomes obtained by performing PostFinder with six configurations (cf. Table 2), to answer **RQ$_1$**. Afterwards, we compare PostFinder with FaCoY by answering **RQ$_2$**. We attempt to reason what constitutes the performance differences between the two systems in **RQ$_3$**. Threats to validity of the performed experiments are discussed in Section 5.2.

5.1. Result analysis

RQ$_1$: *Which experimental configuration brings the best PostFinder performance?*

Every configuration is evaluated using ten queries, and each of them corresponds to five posts, resulting in 50 pairs of $<$ *query, retrieved post* $>$. We gather the relevance scores for the configurations and represent them in a Likert bar chart shown in Fig. 8. It is evident that performing PostFinder with flat queries, i.e., configuration **A**, yields the worst performance: 30% of the posts are either totally irrelevant, or irrelevant, and 70% of them are relevant or highly relevant. This suggests that feeding queries without incorporating any proposed augmentations

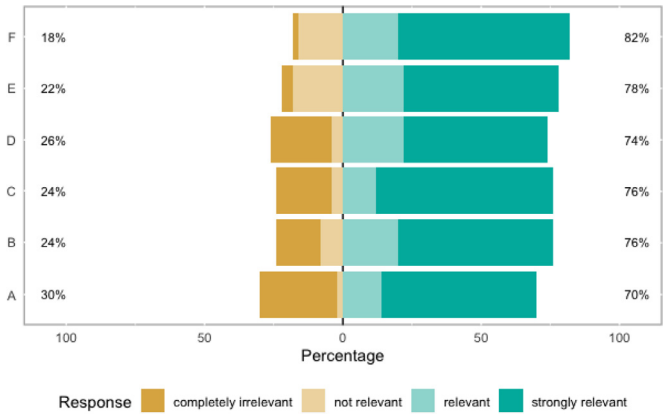


Fig. 8. Relevance scores for PostFinder.

Table 5
PostFinder success rate and precision.

Metric	Configuration					
	A	B	C	D	E	F
Success rate	0.90	0.90	0.90	0.90	1.00	1.00
Precision	0.60	0.66	0.68	0.74	0.78	0.82

brings least relevant posts. Meanwhile, the system obtains a better performance for configurations **B** (flat query plus wrapping) and **C** (flat query plus wrapping and ranking) with respect to **A**. Moreover, the two configurations **B, C** contribute to a comparable performance as their corresponding bars have a similar shape. Among others, the best relevance is seen when running PostFinder with **F**, i.e., all proposed augmentations are incorporated. In particular, no query pair gets 1 as the relevance score and 82% of them are considered as relevant or strongly relevant. This necessarily means that augmenting queries with all the proposed measures helps retrieve highly relevant posts.

Success rate is a superficial metric and it does not reflect well the outcome's traits. For instance, given a query, a system that gets one matched result has a success rate of 100%, which is exactly the same as that of another system which gets all five matched posts for the same query, however, the two systems are not equal in quality. Thus, further than *Success rate* we also measured *Precision* as shown in Table 5. Concerning *Precision*, we see that using flat query obtains the lowest precision, i.e., $Precision = 0.6$ and this is consistent with *Relevance* in Fig. 8. Again, the best *Precision*, i.e., 0.82 is obtained when all augmentations are imposed on the queries. Running PostFinder on the dataset always gets a minimum success rate of 0.90, regardless of the configuration.

According to the performed experiments, Configuration **F** seems to perform better than **E**. However, we anticipate that to assess to what extent Tokenizing (see Section 3.2.2) contributes to better results in a statistically significant manner, a larger dataset would be needed. In fact, tests that can be used for measuring the significance of the results, e.g., Wilcoxon rank sum test, do not work well for small samples [28]. Thus, according to the performed experiments, tokenizing contributes much to the matching of relevant posts. Understanding if this is statisti-

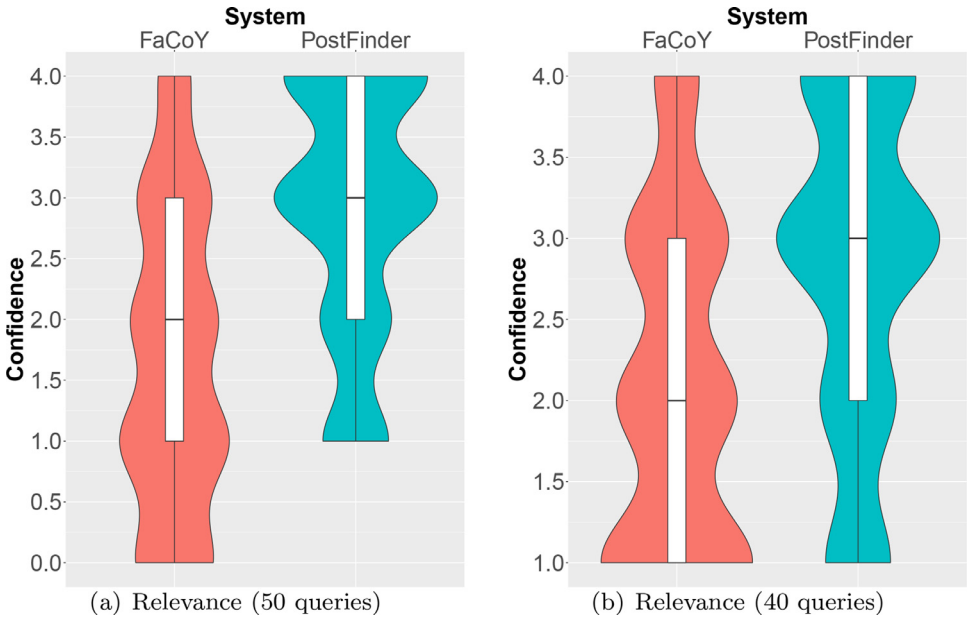


Fig. 9. Relevance for Configuration **G**.

cally significant needs a bigger dataset, and this is deferred as a future work.

> **Answer to RQ$_1$.** In summary, running PostFinder by deploying all proposed augmentations provides the best performance with respect to relevance, success rate, and precision.

RQ$_2$: *How does PostFinder compare with FaCoY?*

Considering the set of 50 queries, PostFinder returns 250 pairs of query-post. Each pair gets a score ranging from 1 to 4. However, FaCoY does not find any results for ten among the queries, i.e., the corresponding scores are 0 (see Table 4). We depict the relevance scores of both systems using violin boxplots in Fig. 9. A violin boxplot is a combination of boxplot and density traces which gives a more informative indication of the distribution's shape, or the magnitude of the density [29]. The boxplots demonstrate that PostFinder gains a considerably better relevance than that of FaCoY. In particular, PostFinder has more scores of 3 and 4, whereas FaCoY has more scores of 1 and 2. By inspecting the ten queries that yield no results, we found out that their input context code is considerably short. This supports our hypothesis in Section 2 that FaCoY is less effective given that input data is incomplete or missing.

To aim for a more reliable comparison, we remove the ten queries from the results of both systems and sketch the relevance scores in Fig. 9b. For this set of queries, the FaCoY's violin fluctuates starting from 3 down to 1. In contrast, the majority of the violin representing PostFinder lies on the upper part of the figure, starting from 3 in the vertical axis. We conclude that PostFinder obtains a better relevance compared to that of FaCoY.

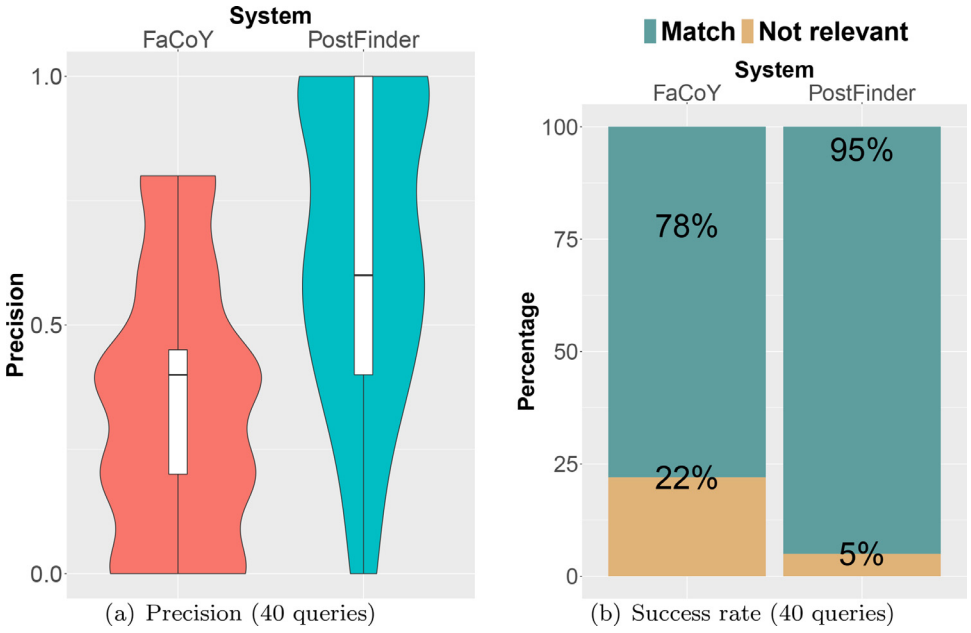
We further investigate the systems by considering Fig. 5.1 where the precision scores for 40 queries are depicted. By this metric, the performance difference between the two systems becomes more noticeable. To be more concrete, a larger part of the FaCoY's boxplot resides under the median horizontal line, implying that most of the queries get a precision lower than 0.5. In the opposite side, PostFinder gains better precisions that are larger than 0.5, and agglomerate to the upper bound, i.e., 1.0. The metric shows that, given a same query PostFinder returns more relevant posts than the baseline does.

The obtained success rates for both systems are shown in Fig. 5.1. Among 40 queries fed to FaCoY, 78% of them are successful, i.e., at least one pair of a query gets a value of 3 or 4. Meanwhile, PostFinder achieves a better percentage of success, 38 among 40 queries are successful, yielding a success rate of 95%.

It is important to understand if the performance difference is statistically significant. Thus, we performed a statistical analysis and the obtained results are shown in Table 6. We compute Wilcoxon rank sum test [30] on the scores obtained by the systems and get the following results: p-value for Success rate is *2.00e-02*; p-value for Precision is *8.90e-06*; p-value for Relevance is *1.08e-10*. The null hypothesis is that there are no differences between the performance of FaCoY and that of PostFinder. Using 95% as the significance level, or p-value $<$ 0.05 we see that by all quality indicators the *p*-values are always lower than *5e-02*. Thus, we reject the null hypothesis and conclude that the performance improvement obtained by PostFinder is statistically significant. Considering all the metrics, we see that the improvement gained by PostFinder is significant and meaningful.

Table 6
Statistical analysis for the used metrics.

	Success rate		Precision		Relevance	
	PostFinder	FaCoY	PostFinder	FaCoY	PostFinder	FaCoY
Mean	0.95	0.77	0.66	0.33	2.78	2.09
std	0.22	0.42	0.30	0.26	1.01	1.01
1st*quartile*	1.0	1.0	0.40	0.20	2.0	1.0
2nd*quartile*	1.0	1.0	0.60	0.40	3.0	2.0
3rd*quartile*	1.0	1.0	1.0	0.45	4.0	3.0
p-value	2.00e-02		8.90e-06		1.08e-10	



(a) Precision (40 queries) (b) Success rate (40 queries)

Fig. 10. Precision and success rate for Configuration **G**.

Table 7
Comparsion between FaCoY and PostFinder.

Technique	FaCoY	PostFinder
Code analysis	√	√
Class fixing	√	√
Import mining	✗	√
Entropy	✗	√
BM25	✗	√
Tokenizing	✗	√

Answer to RQ_2. PostFinder outperforms FaCoY in terms of relevance, success rate, and precision. Furthermore, the performance difference between the two systems is statistically significant.

RQ_3: *What are the reasons for the performance difference?*

We refer back to the example introduced in Section 2. Actually, the post in Fig. 1 is recommended by PostFinder when the code in Listing 1 is used as query. Compared to the baseline, PostFinder works better since it is capable of recommending a very relevant and helpful discussion, while it is not the case with FaCoY. By carefully investigating the query generated by PostFinder, we see that the transformation of import directives to produce textual tokens as shown in Listing 4 is beneficial to the search process: it equips the query with important terms which then match with the post's title. Table 7 distinguishes between the two systems by listing the facets exploited by each of them.

FaCoY exploits the Porter stemming algorithm and the English analyzing utilities provided by Lucene to perform a query. It parses the developer's source code as well as comments and uses extracted indexes described in Section 2 to search. As shown in $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$, a flat query containing only full-text is not sufficient to retrieve useful results. Though FaCoY employs Lucene as its indexer, it does not exhaustively exploit boosting which is considered to be the heart of Lucene. To this end, PostFinder attempts to improve the baseline by imposing various boosting measures. By the *Index Creation* phase, PostFinder enriches incomplete code snippets with class and import directives and then tokenizes them. By the *Query Creation* and *Execution* phases, PostFinder exploits import directives from source code to build indexes to match against indexed textual data. In other words, we make source code compilable, which was originally uncompilable, though the augmented code is inauthentic. This is important since a lot of code in Stack Overflow is uncompilable and cannot be indexed to Lucene. Since FaCoY does not perform these phases, it is unable to match source code with textual context in post. Furthermore, FaCoY cannot match input code with snippets stored in database but without import directives.

Answer to RQ_3. Altogether, the query boosting scheme and the considered facets for the creation of indexes are attributed to the performance difference between the two systems.

5.2. Threats to validity

We investigate the threats that may affect the validity of the experiments as well as the efforts made to minimize them.

Internal validity. It concerns any confounding factors that may have an influence on our results. We attempted to avoid any bias in the user studies by: *(i)* involving 11 developers with different levels of programming experience; (*ii*) simulating a taste test where users are not aware of what they are evaluating. Furthermore, the labeling results by two evaluators were then double-checked by an additional researcher to aim for soundness of the outcomes.

External validity. This refers to the generalizability of the obtained results and findings. To contrast and mitigate this threat, we enforced the following measures. The sets of code snippets that have been selected as queries invoke various Java libraries. Furthermore, the number of code lines of the queries ranges from 22 to 608, attempting to cover a wide range of possibilities in practice. Our approach is also applicable to other programming languages, however in the scope of this paper we restricted ourselves to perform evaluations on posts containing Java source code. The generalizability might be negatively affected by the conducted user study. In particular, the 11 selected participants have the same academic background, though their programming skills are of different levels. On one hand, generalizing the obtained results to a larger population that includes different backgrounds can contribute to enriching the results in terms of coverage. On the other hand, gathering people with a wide range of programming experiences is a real challenge. Thus, we limit ourselves to the mentioned study

to obtain a stable evaluation. Moreover, we mitigate this threat by involving one additional researcher who plays the role of mediator in the evaluation.

6. Related work

In this section, we summarize related work and associate our contributions to the literature in the following domains: *(i)* mining and usage of Stack Overflow; *(ii)* code recommender systems and search engines; and *(iii)* code wrapping technique.

6.1. Mining Stack Overflow to support software development

As discussed in the previous sections, SO can be exploited to support coding activities by providing developers with messages and code snippets therein that are relevant to the query explicitly or implicitly defined by the user. We review various studies that share some commonalities with PostFinder as follows.

Zagalsky et al. [16] introduce *Example overflow*, which allows developers to search for code snippets starting from provided keywords, which in turn are used by the system for retrieving code snippets from a local SO dump. Similarly to our approach, the search function is based on Apache Lucene even though the outcome of *Example overflow* consists of embeddable code, whereas PostFinder is able to retrieve full posts that are related to the user context.

Seahawk [15] has been developed to retrieve SO discussions, which are linked to the source code being developed. The search mechanism exploits code similarity techniques essentially based on TF-IDF. The PostFinder search mechanisms are instead based on different boosting features that are considered when creating queries and when executing them atop of Apache Lucene. de Souza et al. [14] present a tool being able to recommend a ranked list of pairs of SO questions and answers based on the user query, which consists of a list of terms. Furthermore, the approach also allows one to classify SO posts according to defined labels like *how-to, debug corrective*, etc. The main difference with PostFinder relies on the way queries are defined. In particular, in PostFinder queries consist of the whole developer context, instead of only lists of terms explicitly defined by the user.

AnswerBot [31] is an automatic approach aiming to summarize answers coming from Stack Overflow, given a specific question. The retrieval of relevant question is performed by exploiting word embeddings techniques combined with classical IDF metrics. To reduce information overloading, AnswerBot filters the answers by following different criteria, i.e., query related features, paragraph content features, and user related features.

Rigby and Robilliard [32] propose ACE, a tool that mines an input SO dump in order to find relevant elements in the code. ACE relies on a fully text-based analysis mechanism to identify and create indexes of the so-called salient element in the code. Different from PostFinder, ACE uses island parsers based on a set of regular expressions to approximate Java qualified statements (i.e., package definitions, class names, and so on).

MFISSO [33] is a system that exploits natural language processing and clustering techniques to obtain facets from the user's query, i.e., concepts expressed by the query. There are SO features with 8 facets, and seven of these are static and determined by NLP techniques and using Apache Lucene indexes. Clustering is applied to retrieve the dynamic facet by labeling titles, text, and tags of the SO posts. Once MFISSO retrieves the initial results, a final user can refine this search by interacting with the system, i.e., changing the facets to be considered. Reversely, PostFinder extracts the facets directly from source code.

PostFinder distinguishes itself from current approaches that deal with the mining of Stack Overflow as it addresses different phases of the whole searching process, i.e., Index Creation, Query Creation and Query Execution. To this end, PostFinder attempts to effectively exploit the well-defined indexing and searching mechanisms provided by Lucene to increase the exposure of queries to the indexed data. Nevertheless, we still believe that more investigations are needed to further improve PostFinder's performance, e.g., by better employing the boosting scheme. This is considered as an open research issue.

Table 8
Summary of SO mining approaches.

Tool/Engine	Code search	Mining SO	Availability
MFISSO [33]	√	√	✗
PROMPTER [4]	√	√	✗
AnswerBot [31]	✗	√	✗
Google/SO search	✗	√	√
Searchcode [17]	✗	✗	√
Krugle [18]	✗	✗	√
PublicWWW [34]	✗	√	√
ProgramCreek [35]	✗	✗	√
FaCoY [10]	√	√	√

Table 8 summarizes the functionalities as well as the availability of the related Stack Overflow search engines. The table also reveals the rationale behind the selection of FaCoY as baseline. The *Availability* column reports if the considered tool is available for download, including replication packages, and the legacy support provided by the authors. As PostFinder performs search on source code, *Code search* identifies tools that allow directly the usage of source code in the query. Furthermore, since some of the examined approaches do not search over Stack Overflow expressed by *Mining SO*, a comparison with them is not fair. Overall, the table suggests that only the comparison with FaCoY is feasible at the moment.

6.2. Code recommender systems and search engines

Moreno et al. introduce MUSE, a practical tool to recommend code examples related to a specific function [36]. MUSE parses source code to extract method usage, it simplifies examples and detects clones to group similar code snippets. Furthermore, it is able to rank recommendation outcomes according to various characteristics, i.e., reusability, understandability, and popularity.

Strathcona [37] is a recommendation tool, which analyzes developers context from the structural point of view and suggests a possible implementation related to the task that they are developing. Strathcona uses six heuristics based on inheritance hierarchy, field types method calls, and object usage in order to build the query. The built query is then executed on a repository containing all possible usage of the APIs and it is built automatically from the context. Finally, Strathcona retrieves code examples, which can be navigated by the developer both graphically and in a textual way.

Aroma [38] is a code recommendation tool which analyzes the code being developed, and proposes a set of strictly related snippets as suggestions. The first phase is a light-weight search which matches the featurized query and stored methods. The resulting snippets are then ranked and pruned to achieve the maximum similarity, which is computed using the Jaccard index. The snippets are eventually clustered and intersected to maximize the correlation with the original query as well as to get a succinct snippet representation. This aims at reducing irrelevant statements as much as possible.

In a recent work [39], we present CrossRec, a recommender system to assist open source software developers in selecting suitable third-party libraries. The system exploits a collaborative filtering technique to recommend libraries to developers by relying on the set of dependencies, which are currently included in the project being developed. Following the same line of reasoning, we developed FOCUS [3] to provide API function calls and code snippets. CrossRec uses a 2D ratings matrix to perform recommendation, while FOCUS exploits a 3D context-aware ratings matrix. We suppose that more dimensions can be added to the rating matrix, in order to incorporate more input data.

Sourcerer [40] is a code search tool mainly based on Lucene. Source code is indexed according to the included keywords and by considering fingerprints, which summarize code snippets in vectors. Then, Sourcerer maps the developers of each snippet in a matrix consisting of developer-document entries. Moreover, the tool can categorize developers with best skills according to their contributions.

CodeHow [41] is a code search engine specifically conceived to parse APIs online documentation by analyzing the user's query. The tool retrieves and parses information coming from the documentation by applying the standard NLP techniques, i.e., text normalization, stop words removal and stemming. Then, CodeHow finds similarities between the user's query and the related APIs by employing adoption of an information retrieval technique, called the Extended Boolean Model (EBM), and ElasticSearch was exploited as the main indexing and searching platform.

PostFinder can be combined with the previous tools and approaches with the aim of providing developers with recommendations consisting of both source code and related discussions retrieved from Stack Overflow.

6.3. Code wrapping technique

The code wrapping technique has been used in some other work to make source code more comprehensive. *Subramanian and Holmes* [42] developed an approach that can parse short code snippets to effectively identify API usage. The goal is to perform snippet analysis to extract structural information from short plain-text snippets that are often found in Stack Overflow. An opportune code wrap of methods and class is made to let the parser work properly.

Similarly, an iterative and deductive method of linking source code examples to API documentation has been proposed [43]. Starting from a Stack Overflow post, the tool is able to find the links among API usages and API documentation. Source code is properly wrapped using class and method declarations. To resolve the vocabulary mismatch problem when dealing with free-form code search queries, Sirres et al. [19] present an approach that leverages common developer questions and the associated expert answers to augment user queries with the relevant, but missing, structural code entities. This aims to improve the performance of matching relevant code examples within large code repositories. The tool removes ellipses and wraps code snippets by using a custom dummy class and method templates to make it able to parse by standard Java parsers.

PostFinder takes one step further to make code compilable by tracing back to the original libraries and augmenting the code with relevant imports. In this way, we are able to considerably expand the search scope. This explains why for many cases, PostFinder is able to provide relevant code while FaCoY cannot retrieve any results.

7. Conclusions

We introduced the PostFinder approach to provide highly relevant SO posts, given an input code snippet as the context. PostFinder addresses both the problem of adequately indexing SO posts, and that of automatically creating queries in a transparent manner for the developer. In particular, PostFinder performs different augmentations of SO posts for indexing them, and of input contexts for creating corresponding queries.

To study the performance of PostFinder, we performed two different user studies. The first study has been done in order to understand which combination of the conceived augmentations is the best one in terms of PostFinder performance. While the second one, which is a larger user study has been done to compare PostFinder with FaCoY. The experimental results show that PostFinder outperforms the corresponding module of FaCoY, which is devoted to searching SO posts that are relevant with input developer contexts.

The implementation of PostFinder is twofold. First, we already integrated it into the Eclipse IDE to directly support developers in real-world settings. Second, the tool can be used to substitute the module for searching Stack Overflow posts by FaCoY, and we are now working on the replacement, aiming to boost up the system's overall performance.

To showcase our contribution, we evaluated PostFinder using different metrics and assessed the relevance of the retrieved posts compared to those obtained by the baseline. The results confirm that the improvement of our tool is significant and meaningful. Our future research agenda focuses on performing further evaluations, especially to compare PostFinder with those approaches that rely on general-purpose search engines and that focus only on the query creation phase (e.g., PROMPTER [4]). Last but not least, though our proposed approach works well given the context, we still believe that its performance can be further improved, e.g., by better exploiting the boosting scheme. We consider this as future work.

Declaration of Competing Interest

The authors declare that they do not have any financial or nonfinancial conflict of interests

CRediT authorship contribution statement

Riccardo Rubei: Conceptualization, Methodology, Software, Validation. **Claudio Di Sipio:** Software, Data curation, Validation. **Phuong T. Nguyen:** Writing - original draft, Writing - review & editing. **Juri Di Rocco:** Data curation, Visualization. **Davide Di Ruscio:** Writing - review & editing, Supervision.

Acknowledgements

The research described in this paper has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant 732223.

References

[1] , Recommendation Systems in Software Engineering, M.P. Robillard, W. Maalej, R.J. Walker, T. Zimmermann (Eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, doi:10.1007/978-3-642-45135-5.

[2] E. Duala-Ekoko, M.P. Robillard, Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study, in: Proceedings of the 34th International Conference on Software Engineering, in: ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 266–276.

[3] P.T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, M. Di Penta, FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns, in: Proceedings of the 41st International Conference on Software Engineering, in: ICSE '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 1050–1060, doi:10.1109/ICSE.2019.00109.

[4] L. Ponzanelli, G. Bavota, M.D. Penta, R. Oliveto, M. Lanza, Prompter - turning the IDE into a self-confident programming assistant, Empir. Softw. Eng. 21 (5) (2016) 2190–2231, doi:10.1007/s10664-015-9397-1.

[5] B. Dagenais, H. Ossher, R.K.E. Bellamy, M.P. Robillard, J.P. de Vries, Moving into a new software project landscape, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, in: ICSE '10, ACM, New York, NY, USA, 2010, pp. 275–284, doi:10.1145/1806799.1806842.

[6] Stackoverflow, (https://stackoverflow.com/). last access 04.04.2019.

[7] S. Baltes, L. Dumani, C. Treude, S. Diehl, Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts, in: Proceedings of the 15th International Conference on Mining Software Repositories, in: MSR '18, ACM, New York, NY, USA, 2018, pp. 319–330, doi:10.1145/3196398.3196430.

[8] R. Abdalkareem, E. Shihab, J. Rilling, On code reuse from stackoverflow: an exploratory study on android apps, Inf. Softw. Technol. 88 (2017) 148–158, doi:10.1016/j.infsof.2017.04.005.

[9] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, How do api changes trigger stack overflow discussions? A study on the android SDK, in: Proceedings of the 22Nd International Conference on Program Comprehension, in: ICPC 2014, ACM, New York, NY, USA, 2014, pp. 83–94, doi:10.1145/2597008.2597155.

[10] K. Kim, D. Kim, T.F. Bissyandé, E. Choi, L. Li, J. Klein, Y.L. Traon, FaCoY: a code-to-code search engine, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, 2018, pp. 946–957, doi:10.1145/3180155.3180187.

[11] Apache Lucene Core, (https://lucene.apache.org/core/). last access 26.04.2019.

[12] R. Rubei, C. Di Sipio, P.T. Nguyen, J. Di Rocco, D. Di Ruscio, PostFinder - Online appendix, 2020.
[13] K. Mao, L. Capra, M. Harman, Y. Jia, A survey of the use of crowdsourcing in software engineering, J. Syst. Softw. 126 (2017) 57–84, doi:10.1016/j.jss.2016.09.015.
[14] L.B.L. de Souza, E.C. Campos, M.d.A. Maia, Ranking crowd knowledge to assist software development, in: Proceedings of the 22Nd International Conference on Program Comprehension, in: ICPC 2014, ACM, New York, NY, USA, 2014, pp. 72–82, doi:10.1145/2597008.2597146.
[15] L. Ponzanelli, A. Bacchelli, M. Lanza, Seahawk: Stack overflow in the ide, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 1295–1298, doi:10.1109/ICSE.2013.6606701.
[16] A. Zagalsky, O. Barzilay, A. Yehudai, Example overflow: Using social media for code recommendation, in: Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE), 2012, pp. 38–42.
[17] Searchcode | source code search engine.
[18] Home | Krugle - software development productivity.
[19] R. Sirres, T.F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, Y.L. Traon, Augmenting and structuring user queries to support efficient free-form code search, Empir. Softw. Eng. 23 (5) (2018) 2622–2654, doi:10.1007/s10664-017-9544-y.
[20] A. Benelallam, N. Harrand, C.S. Valero, B. Baudry, O. Barais, Maven central dependency graph, 2018. doi:10.5281/zenodo.1489120.
[21] C.E. Shannon, A mathematical theory of communication, Mobile Comput. Commun. Rev. 5 (1) (2001) 3–55, doi:10.1145/584091.584093.
[22] M. Borg, P. Runeson, J. Johansson, M.V. Mäntylä, A replicated study on duplicate detection: Using apache Lucene to search among android defects, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, in: ESEM '14, ACM, New York, NY, USA, 2014, pp. 8:1–8:4, doi:10.1145/2652524.2652556.
[23] J. Pérez-Iglesias, J.R. Pérez-Agüera, V. Fresno, Y.Z. Feinstein, Integrating the probabilistic models BM25/BM25F into Lucene, CoRR abs/0911.5046 (2009).
[24] D. Lo, L. Jiang, F. Thung, Detecting similar applications with collaborative tagging, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), in: ICSM '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 600–603, doi:10.1109/ICSM.2012.6405331.
[25] C. McMillan, M. Grechanik, D. Poshyvanyk, Detecting similar software applications, in: Proceedings of the 34th International Conference on Software Engineering, in: ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 364–374.
[26] S. Pettigrew, S. Charters, Tasting as a projective technique, Qual. Market Res.: Int. J. 11 (3) (2008) 331–343, doi:10.1108/13522750810879048.
[27] A.J. Viera, J.M. Garrett, Understanding interobserver agreement: the kappa statistic., Family Med. 37.5 (2005) 360–363.
[28] M. Happ, A. Bathke, E. Brunner, Optimal sample size planning for the Wilcoxon–Mann–Whitney test, Stat. Med. 38 (2018), doi:10.1002/sim.7983.
[29] J.L. Hintze, R.D. Nelson, Violin plots: a box plot-density trace synergism, Am. Stat. 52 (2) (1998) 181–184, doi:10.1080/00031305.1998.10480559.
[30] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer New York, New York, NY, pp. 196–202. doi:10.1007/978-1-4612-4380-9_16.
[31] B. Xu, Z. Xing, X. Xia, D. Lo, AnswerBot: Automated generation of answer summary to developers' technical questions, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Urbana, IL, 2017, pp. 706–716, doi:10.1109/ASE.2017.8115681.
[32] P.C. Rigby, M.P. Robillard, Discovering essential code elements in informal documentation, in: D. Notkin, B.H.C. Cheng, K. Pohl (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, IEEE Computer Society, 2013, pp. 832–841, doi:10.1109/ICSE.2013.6606629.
[33] M. Liu, X. Peng, Q. Jiang, A. Marcus, J. Yang, W. Zhao, Searching StackOverflow Questions with Multi-Faceted Categorization, in: Proceedings of the Tenth Asia-Pacific Symposium on Internetware - Internetware '18, ACM Press, Beijing, China, 2018, pp. 1–10, doi:10.1145/3275219.3275227.
[34] Search Engine for Source Code - Publicwww.com. https://publicwww.com/.
[35] Java Code Examples. https://www.programcreek.com/java-api-examples/.
[36] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, How Can I Use This Method? in: Proceedings of the 37th International Conference on Software Engineering, IEEE, Piscataway, 2015, pp. 880–890, doi:10.1109/ICSE.2015.98.
[37] R. Holmes, R.J. Walker, G.C. Murphy, Strathcona example recommendation tool, in: M. Wermelinger, H.C. Gall (Eds.), Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, ACM, 2005, pp. 237–240, doi:10.1145/1081706.1081744.
[38] S. Luan, D. Yang, K. Sen, S. Chandra, Aroma: Code recommendation via structural code search, CoRR abs/1812.01158 (2018).
[39] P.T. Nguyen, J. Di Rocco, D. Di Ruscio, M. Di Penta, CrossRec: Supporting Software Developers by Recommending Third-party Libraries, J. Syst. Softw. (2019) 110460, doi:10.1016/j.jss.2019.110460.
[40] E. Linstead, S.K. Bajracharya, T.C. Ngo, P. Rigor, C.V. Lopes, P. Baldi, Sourcerer: mining and searching internet-scale software repositories, Data Min. Knowl. Discov. 18 (2) (2009) 300–336, doi:10.1007/s10618-008-0118-x.
[41] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, J. Zhao, Codehow: Effective code search based on API understanding and extended boolean model (E), in: M.B. Cohen, L. Grunske, M. Whalen (Eds.), Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society, 2015, pp. 260–270, doi:10.1109/ASE.2015.42.
[42] S. Subramanian, R. Holmes, Making sense of online code snippets, in: Proceedings of the 10th Working Conference on Mining Software Repositories, in: MSR '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 85–88.
[43] S. Subramanian, L. Inozemtseva, R. Holmes, Live api documentation, in: Proceedings of the 36th International Conference on Software Engineering, in: ICSE 2014, ACM, New York, NY, USA, 2014, pp. 643–652, doi:10.1145/2568225.2568313.