RECOMMENDING JAVA API METHODS BASED ON PROGRAMMING TASK DESCRIPTIONS BY NOVICE PROGRAMMERS

Chun Jiann Lim[1], ***Moon Ting Su***[2*]

Department of Software Engineering, Faculty of Computer Science and Information Technology,
Universiti Malaya 50603 Kuala Lumpur, Malaysia

Email: chunjiann.lim3@gmail.com[1], smting@um.edu.my[2 *] (corresponding author)

DOI: https://doi.org/10.22452/mjcs.vol36no2.3

ABSTRACT

The overwhelming number of Application Programming Interfaces (APIs) and the lexical gap between novices' programming task descriptions in their search queries and API documentations deter novice programmers from finding suitable API methods to be used in their code. To address the lexical gap, this study investigated novice programmers' descriptions of their programming tasks and used the found insights in a novel approach (APIFind) for recommending relevant API methods for the programming tasks. Queries written by novice programmers were collected and analysed using term frequency and constituency parsing. Four common patterns related to the return type of an API method and/or API class that provides an implementation for the API method were found and captured in the Novice Programming Task Description Model (NPTDM). APIFind uses NPTDM that was operationalised in a rule-based module, a WordNet map of API word-synonyms, a programming task dataset comprising the collected queries, a Java API class and method repository, a Stack Overflow Q&A thread repository, and the BM25 model in Apache Lucene, to produce the top-5 API methods relevant to a search query. Benchmarking results using mean average precision @ 5 and mean reciprocal rank @ 5 as the evaluation metrics show that APIFind outperformed BIKER and CROKAGE when the novice queries test dataset was used. It performed slightly better than BIKER but slightly worse than CROKAGE when the reduced BIKER test dataset was used. In conclusion, common patterns exist in novice programmers' search queries and can be used in API recommendations for novice programmers.

Keywords: API Recommendation, Novice Programmer, Novice Programming Task Description Model, Rule-Based, Best Match 25.

1.0 INTRODUCTION

The use of software libraries or frameworks in software development has become more and more inevitable. The functionalities provided by these libraries and frameworks are exposed through application programming interfaces (APIs) [1]. Some key benefits of using APIs and the underlying software libraries or frameworks are shorter development time and better software quality [2], code reuse, and access to system resources such as graphics, networking, and the file system [3].

APIs are growing in quantity and complexity to provide advanced functions in applications [4]. However, it has been found that APIs are difficult to use for all levels of programmers [3]. Novice programmers, in particular, struggled with using software libraries [5]. They face a selection barrier, where they find it hard to know which APIs to use for their programming tasks [6]. Existing studies used the term "novice programmers" to refer to first-year undergraduate computer science students [2, 5, 7].

To apply a functionality exposed by an API, a programmer typically searches an electronic copy of the official documentation of the API for the relevant method to be used in his/her code. With APIs having a huge number of classes and an even greater number of methods, finding the correct method of the correct class to be used is difficult, even for experienced developers [8]. The success of such a search is very much dependent on the terms the programmer uses in his or her search query (also known as a user query). Studies have found that programmers

describe their programming tasks using terms different from the terms used in the API documentation, causing a lexical gap between programming task descriptions in search queries and API documentation [9], which is also known as a vocabulary problem [8]. The mismatch between the terms used in search queries and API descriptions leads to a futile search.

Novice programmers seem to experience a more prominent lexical gap compared to more expert programmers. For instance, a novice programmer stated the term "form" in his/her search query to find out how to create a window in a Java application [10] instead of the term "window" typically used in Java programming and by more experienced Java programmers. In addition, a study targeting novice programmers found that, although having the specific name of a relevant API element (such as a package, class, data type, or method) as a keyword in a search query is effective in finding relevant code examples, novice programmers fail to provide such an element's name in their search queries [2]. For example, when given a programming task to verify whether the structure of an XML file conforms to a given XML schema, a novice programmer started with the search query "java xml processing tutorials", but the search query failed to return any relevant code examples. The novice programmer then resorted to reading the API documentation, and discovered that the *Schema* class is relevant to the programming task. Finally, the novice programmer stated the API class *Schema* in his or her reformulated search query "java xml validation against schema", and found relevant code examples [2].

The lexical gap problem is mainly caused by the inability of programmers to use suitable or domain-specific terms in their search queries. On the other hand, API documentations that aim at helping programmers understand and learn the structure and functionalities of APIs [1] also have limitations. They typically exhibit a number of content and presentation issues, such as incompleteness and insufficiency of information, lack of usage examples, ambiguous, unclear, and verbose descriptions [1]. These weaknesses of API documentations greatly affect the usability of the APIs.

In the last decade, several studies have developed code search or API recommendation tools to return more relevant results to user queries that are related to programming tasks. These API recommendation studies, namely, RACK [11, 12], NLP2API [13, 14], BIKER [9, 15], CROKAGE [16] [17], and APIRecJ [18], used Stack Overflow (SO) threads [19] for training models or mining associations that would be used in their approaches. SO was used in these studies as a source of crowdsourced documentation since many programming questions have been posted and answered on this Community Question and Answer site. Except for RACK, all the studies mentioned earlier in this paragraph constructed word embedding models (such as Word2Vec, FastText, Doc2Vec) from the SO threads and used the models in their approaches. All studies (except RACK and APIRecJ) used term weighting techniques such as inverse document frequency (IDF), term frequency-inverse document frequency (TF-IDF), and Best Match 25 (BM 25). Details of other techniques used by each of the studies can be found in Section 9.0 of this paper.

None of the API recommendation studies mentioned above has any technique, approach, vocabulary, or data in API recommendation that caters to the queries from novice programmers, who may describe their programming tasks using terms and phrases different from those used in the respective domain. This motivated this study to address the aforementioned lexical gap by investigating novice programmers' descriptions of their programming tasks in search queries and how the outcomes of the investigation could be used to recommend desired and relevant API methods for programming tasks, to break the API method selection barrier faced by novice programmers. Similar to some existing API recommendation studies [9] [12] [16], this study used API search queries as programming task descriptions to search for relevant API methods or classes.

This study collected novice programmers' descriptions of their programming tasks in the form of natural language (NL) search queries that they wrote to look for API methods that are relevant to some pre-defined programming tasks. The collected queries were analysed to identify common patterns that might exist in novice programmers' descriptions of their programming tasks. The analysis was done using two natural language processing (NLP) techniques, namely, term frequency and constituency parsing. Four common patterns related to the return type of an API method and/or the API class that provides the implementation for the API method were discovered and captured in **N**ovice **P**rogramming **T**ask **D**escription **M**odel (NPTDM).

This study also developed a novel API recommendation approach, APIFind. APIFind utilises the collected novice programmer queries (in the form of a programming task dataset) and NPTDM that has been operationalised in a rule-based module (RBM). APIFind expands a search query by appending it with API words found from the WordNet map

of API word-synonyms; calculates cosine similarity between the expanded query and the novice programmer queries in the programming task dataset to identify relevant API phrases (refer to Section 2.6 of this paper); extracts API methods that correspond to the identified API phrases and extracts the descriptions of these API methods from the Java API class and method repository (derived from official Javadoc documentation); ranks the relevant API methods based on cosine similarity between the expanded query and API methods' descriptions; filters the ranked list of API methods using the return type and/or the API class identified by the RBM; and finds the hyperlinks to related SO threads from the SO question and answer (Q&A) thread repository. APIFind uses the BM25 model to calculate cosine similarity where applicable. The benchmarking results of the performance of APIFind against two state-of-the-art studies using two test datasets show the potential of APIFind in recommending relevant API methods.

This study also developed APIFind into a recommender tool comprising an Eclipse plug-in as the front-end and a server component that implements the recommendation phase of APIFind. Novice programmers were recruited to evaluate the recommender tool.

The key contributions of this paper are:

- A novice queries dataset with ground truth Java API methods (Section 2.0 of this paper). It can be used by other studies that focus on novice programmers.
- NPTDM, which distills four common patterns that exist in novice programmers' descriptions of their programming tasks in NL search queries. NPTDM provides some insights on how novice programmers compose their search queries and can be used as a reference for further research in NLP of novice programmers' search queries. In addition, NPTDM can be operationalised for actual use in API recommendation tools, as exemplified by our RBM.
- APIFind, a novel approach for recommending relevant Java API methods for novice programmer queries.

The rest of the paper is organised as follows: Section 2.0 describes the construction of the novice queries dataset. Section 3.0 describes the construction of NPTDM. Section 4.0 gives an exemplar use case of NPTDM. Section 5.0 explains the proposed approach, APIFind. Section 6.0 presents the benchmarking of the performance of APIFind. Section 7.0 outlines the implemented recommender tool. Section 8.0 discusses the threats to the validity of results. Section 9.0 compares this study with existing related work. Section 10.0 provides the conclusion and future work.

2.0 CONSTRUCTION OF NOVICE QUERIES DATASET

This study employed a questionnaire survey to collect programming task descriptions in the form of search queries written by novice programmers. This section details the survey instrument design, participant selection criteria, pilot test, participant recruitment and administration of the survey, survey results, programming tasks, API methods, API phrases, API classes, and the novice queries dataset constructed from the collected queries.

2.1 Survey Instrument Design

The survey instrument, or form, consists of a cover page and two main sections. The cover page includes a brief overview of the study, participant selection criteria, instructions for each section of the survey form, the expected participation duration of about an hour, a participation honorarium of RM15, terms and conditions of participation, and contact details of the researchers.

In Section 1 (Programming Tasks) of the survey form, participants were presented with some programming tasks. Refer to Section 2.6 of this paper for details. Each programming task was given as an executable, partially-completed Java program which contains the main method *public static void main(String[] args)*. The *main* method invokes a methodX with specific value(s) passed as the argument(s) for the parameter(s) of methodX and displays the output of the invocation of methodX in the console. The *main* method also provides comments that state the expected output of the invocation of methodX. methodX is defined with a specific return type and specific types of parameter(s). The participants were asked to write the search queries that they would use in search engines or websites to find the Java API method(s) that they would need to use inside methodX. They were also asked to state whether they had previously encountered a programming problem that is similar to the given programming task. Each programming task was described in the form of a partially-completed Java program instead of in NL sentences so that the task description would not influence the participants' choices of the words or terms to use in their search queries.

Section 2 of the survey form contains two parts. Section 2 Part A (Participant details) asks for the participant's name, degree, year of study, programming experience, Java programming experience, and level of English proficiency. Section 2 Part B (Participant Experience and Preferences) asks about the difficulties the participant faced while learning to program, and the type of information that would help him or her carry out the programming tasks given in Section 1 of the survey form.

2.2 Participant Selection Criteria

As mentioned earlier, existing studies have used the term "novice programmers" to refer to first-year undergraduate computer science students [2, 5, 7]. The recruitment of novice programmers for our survey followed stricter participant selection criteria as stated here: First-year undergraduate computer science students with less than nine months of programming experience, learning Java as their first programming language and have not mastered the Java methods or classes in *java.io, java.lang,* and *java.util* packages.

2.3 Pilot Test

Two participants took part in a pilot study to test the suitability of the survey instrument. Both participants were first-year undergraduate students pursuing a Bachelor of Computer Science (Information Systems) degree with less than nine months of programming and Java programming experience. The participants completed the survey in a face-to-face meeting.

The participants' feedback shows that the survey was too long and some instructions were ambiguous. As a result, the number of programming tasks in Section 1 of the survey was reduced from 27 to 17. The instructions in Section 1 of the survey were also rewritten to be clearer.

2.4 Participant Recruitment and Administration of the Survey

After improving the survey instrument, this study managed to recruit 17 participants to participate in the survey. The participants were recruited through announcements made by lecturers teaching programming classes at Universiti Malaya and Universiti Putra Malaysia. Although the call for participation was advertised several times and an honorarium of RM15 was offered for participation, only 17 participants were interested and recruited for the survey. This could be due to the long participation duration of about an hour.

Fourteen participants used their laptops to complete the survey in softcopy in front of the researcher. Due to the long distance, three participants were unable to meet face-to-face, so they completed the survey in softcopy remotely. The researcher briefed all the participants on the purpose of the study and the instructions in the survey instrument before they started the survey.

2.5 Survey Results

In Section 1 (Programming Tasks) of the survey, 17 search queries were collected for each of the 17 programming tasks. Apart from the search queries, the data collected from this section of the survey shows that the participants had encountered programming problems that are similar to the given programming tasks before. More than half of the participants (ranging from 52.9% to 100%) had come across tasks that are similar to 15 of the given programming tasks. 31.4% and 41.2% of the participants had come across tasks that are similar to programming Task 16 (*hasNext(), next(), nextLine()*), and Task 10 (*clear()*), respectively. This shows that all the given programming tasks are at the level of novice programmers, fulfilling the scope of this study, which targeted novice programmers.

The data collected from Section 2 Part A (Participant details) of the survey shows that all recruited participants were first-year undergraduate computer science students with less than six months of programming and Java programming experience. Therefore, all the participants are novice programmers, as outlined in the participant selection criteria.

In terms of English proficiency, 13 students scored MUET Band 2 – 4 / IELTS Band 5.5 - 7.5 / TOEFL 46 – 109 and 4 students scored MUET Band 5 – 6 / IELTS Band 8 – 9 / TOEFL 110 – 120. With these higher levels of English

proficiency, we assumed that the use of the English language in the survey would not impede these participants' understanding of the tasks to be completed.

In Section 2 Part B (Participant Experience and Preferences) of the survey, participants were asked about their programming experience and preferences. Question 1 asks about the difficulties that they faced while learning programming. The participants were allowed to choose more than one option. Fig. 1 shows the participants' responses to Question 1. The majority of the participants (12 out of 17) selected the option "I don't know how to combine the methods", indicating a coordination barrier. It was followed by a selection barrier where 10 participants chose "I don't know which methods to use" and an understanding barrier where 10 participants chose "the methods did not work as expected". These three barriers were described by Ko et al. [6]. The results show that the API method selection barrier is among the top barriers faced by novice programmers. This provides additional justification for this study's aim to mitigate the API method selection barrier by addressing the lexical gap through the recommendation of relevant API methods.

Question 2 states "For the search queries you have written in Section 1, which of the following information would help you to carry out the respective programming task?". The participants could choose more than one option. Fig. 2 shows the participants' responses to Question 2. The number of participants who chose the Java API method (12 participants) doubled the number who chose the Java API class (6 participants). This indicates a general preference for a method-level recommendation over a class-level recommendation. This supports this study's focus on the recommendation of API methods. The responses from the participants also indicated that the following information would help them to carry out the respective programming tasks: the URLs or links to relevant web pages (13 participants), similar questions asked by others (12 participants), related code snippets (11 participants), the official documentation for the method (10 participants), the parameters of the method (6 participants), and other related methods (4 participants). We took the findings here into consideration when designing the information to be returned together with the API methods recommended by our approach, APIFind.

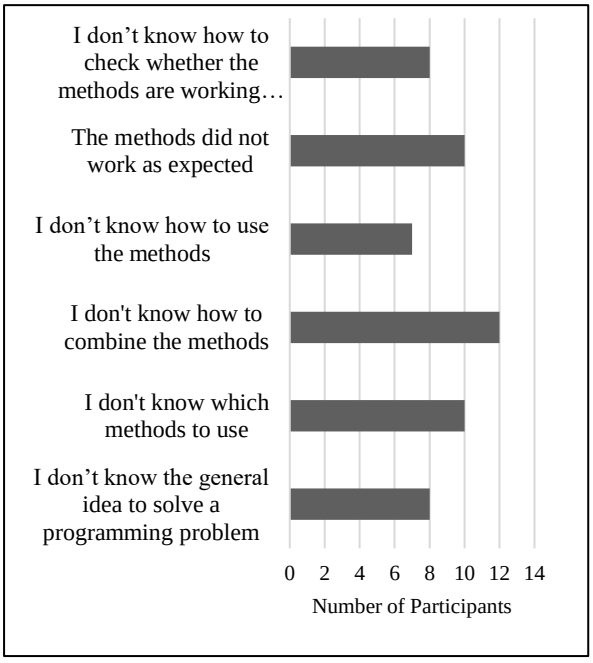

Fig. 1. Participants' responses to Question 1 (Difficulties faced while learning programming)

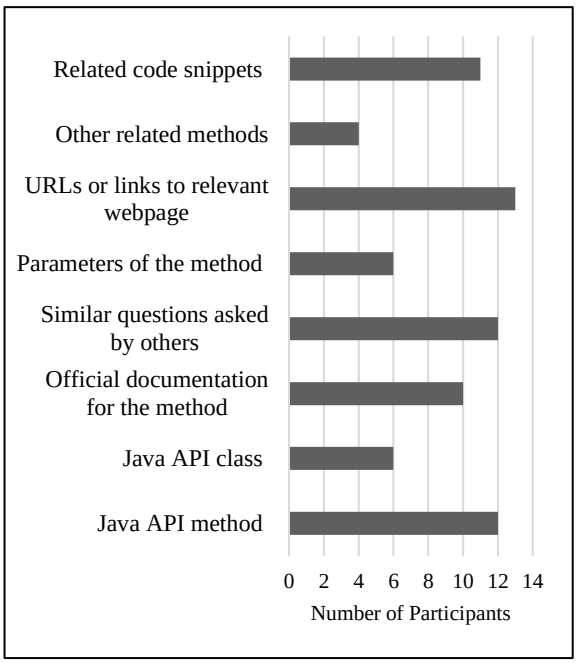

Fig. 2. Participants' responses to Question 2 (Information that would help to carry out the respective programming task)

2.6 Programming Tasks, API Methods, API Phrases, and Implementation API Classes

In the Java Development Kit (JDK), a number of Java classes might implement methods or provide implementations for methods that have the same name and the same parameter(s). The *java.lang.StringBuilder*, *java.lang.String*, and *java.lang.StringBuffer* classes, for example, all provide a *charAt(int index)* method. The three versions of the *charAt(int index)* method perform the same functionality of retrieving the character at position *index*, but for different types of strings. We leveraged this characteristic and derived what we termed "API phrases" from the actual Java API methods. Following that, an API phrase is used in this study to represent a set of methods with the same name and the same parameter(s) that are implemented by different classes. It consists of the name and parameter(s), but not the return type of the respective Java API method(s). For instance, the API phrase *charAt(int index)* is used to represent the set of *charAt(int index)* methods provided by *java.lang.StringBuilder*, *java.lang.String*, and *java.lang.StringBuffer* classes. Except for the API phrase *parse(String s)*, which is used to represent the *parseInt(String s)*, *parseDouble(String s)*, *parseFloat(String s)*, and *parseLong(String s)* methods of the respective *Wrapper* classes, all of the API phrases in this study resemble the Java API method(s) that they represent.

By using API phrases, search queries could be collected for an API phrase instead of for each version of the same method. For example, search queries could be collected for the API phrase *charAt(int index)* instead of collected separately for each *charAt(int index)* method provided by the *java.lang.StringBuilder*, *java.lang.String*, and *java.lang.StringBuffer* classes. As API phrases contain the names of the respective API methods, which are essential verbs that denote actions, they could be used to represent core functionalities provided by Java APIs.

We referred to the list of Java API methods and classes covered in the College Board's Advanced Placement Computer Science A Exam (Java Subset) [20], as outlined in the Java Quick Reference [21], to decide on the Java API methods to be covered by this study. The College Board is an educational organisation that prepares students to pursue college-level education through programs mainly in the United States and Canada. One such program is the Advanced Placement Computer Science A program which includes an introductory college-level (undergraduate level) computer science course in the Java programming language.

Covering all the API methods from the College Board's Advanced Placement Computer Science A Exam would require participants to spend a significant amount of time completing Section 1 (Programming Task) of the survey. This could cause a decline in participation and also respondent fatigue, which would reduce the quality of the data provided by participants. To avoid that, we excluded API methods whose names are exact terms that are generally used outside of the programming domain. For example, *sqrt(double x)* and *abs(double x)* were excluded from the survey because the term *sqrt* is commonly used as the symbol or abbreviation for square root, whereas the term abs is commonly used to represent the operation to obtain the absolute value of a number. *random()* was also excluded because the method name is the same term used in English to mean "out-of-pattern" or arbitrary. We excluded abstract methods but included concrete methods because concrete methods but not abstract methods provide the implementations for the functionalities required.

Thirty API phrases were derived based on the remaining API methods. Initially, 27 programming tasks were defined to cover the 30 API phrases in the survey instrument. However, the number of programming tasks was reduced to 17 (covering 22 API phrases) following the feedback from the pilot study's participants that the survey was too long. We removed those programming tasks where both participants in the pilot study stated the method name of the corresponding API phrase in the search queries they wrote for the task. For example, the two participants wrote the search queries "How to convert string to upper case" and "convert lower case letter to upper case letter" for the programming task related to the API phrase *toUpperCase()*. This task was removed because the words in the API phrase *toUpperCase()* appeared in the search queries written by both participants, indicating there is no lexical gap between the API phrase and the queries. The programming tasks related to the following API phrases were removed from the final survey instrument: *toUpperCase()*, *toLowerCase()*, *equalsIgnoreCase(String anotherString)*, *add(E e)*, *add(int index, E element)*, *remove(Object o)*, *remove(int index)*, and *indexOf(Object o)*.

Table 1 shows the 17 programming tasks, their corresponding 22 API phrases, and the API classes that provide implementations for the corresponding API methods. For simplicity's sake, these API classes will be termed "implementation API classes" from this point onwards. The 22 API phrases cover 598 actual Java methods in JDK 13. Some API phrases are closely related and are covered in the same programming task. In particular, programming tasks 4, 8, and 17 cover two API phrases each, and programming task 16 covers 3 API phrases.

2.7 Novice Queries Dataset

Since 17 novice programmers participated in the survey, 17 search queries were collected for each of the 17 programming tasks. The search queries for a programming task that covers more than one API phrase were duplicated for each of the API phrases covered by the programming task.

The search queries were mapped to their respective ground truth API phrases. This resulted in 22 sets of search queries, namely, one set of 17 search queries for each of the 22 API phrases, which is 374 search queries in total.

We termed the above sets of search queries as the novice queries dataset. We divided the novice queries dataset into a novice queries training dataset for use in our APIFind approach and a novice queries test dataset for use in evaluating the performance of APIFind. Five queries from each set of queries were selected randomly to populate the novice queries test dataset (110 queries = 5 queries × 22 API phrases). The remaining 264 queries were used as the novice queries training dataset.

Table 1: Programming tasks, API phrases and implementation API classes

Programming Task	API Phrases	Number of Implementation API Class (Class name)
Task 1	charAt(int index)	3 (java.lang.StringBuilder, java.lang.String, java.lang.StringBuffer)
Task 2	parse(String s)	4 (java.lang.Integer, java.lang.Double, java.lang.Float, java.lang.Long)
Task 3	length()	10 (java.lang.String, java.lang.StringJoiner, java.io.file, etc.)
Task 4	substring(int beginIndex); substring(int beginIndex, int endIndex)	1 (java.lang.String); 1 (java.lang.String)
Task 5	contains(CharSequence s)	1 (java.lang.String)
Task 6	replace(CharSequence target, CharSequence replacement)	1 (java.lang.String)
Task 7	split(String regex)	1 (java.lang.String)
Task 8	isLetter(char ch); isDigit(char ch)	1 (java.lang.Character); 1 (java.lang.Character)
Task 9	addAll(Collection< ? extends E> c)	21 (java.util.ArrayList, java.util.HashSet, java.util.LinkedList, etc.)
Task 10	clear()	40 (java.util.ArrayList, java.util.HashMap, java.util.Hashtable, etc.)
Task 11	contains(Object o)	22 (java.util.ArrayList, java.util.HashSet, java.util.LinkedList, etc.)
Task 12	size()	39 (java.util.ArrayList, java.util.HashMap, java.util.Hashtable, etc.)

Task 13	get(int index)	5 (java.util.ArrayList, java.util.Stack, java.util.LinkedList, etc.)
Task 14	set(int index, E element)	5 (java.util.ArrayList, java.util.Stack, java.util.LinkedList, etc.)
Task 15	nextInt(int bound)	3 (java.util.Random, java.util.SplittableRandom, java.util.concurrent.ThreadLocalRandom)

Task 16	hasNext();
	next();
	nextLine()
		1 (java.util.Scanner);
		1 (java.util.Scanner);
		1 (java.util.Scanner);

Task 17	equals(Object o); clone()	40 (java.util.TreeMap, java.util.Vector, java.util.Properties, etc.); 396 (java.util.Base64, java.io.File, java.lang.Double, etc.)

3.0 CONSTRUCTION OF NOVICE PROGRAMMING TASK DESCRIPTION MODEL (NPTDM)

This section explains the analysis performed on the novice queries dataset, the use of NLP techniques (namely, term frequency and constituency parsing) for identifying common patterns in the queries collected, and how the patterns are represented in NPTDM.

3.1 Preliminary Analysis of the Novice Queries Dataset

A preliminary analysis of the novice queries dataset shows that 88% of the queries contain terms that represent either the return types of the ground truth API methods or the implementation API classes. The return types and the implementation API classes stated in a search query can be used to narrow down the API methods the user is searching for.

The preliminary analysis also shows that 98% of the queries are in the form of sentences. For example, “How to combine array lists”. Only 2% of the queries are not in the form of sentences. For example, “clear list, clear set” and “method true=false, false=true”. As a result of this finding, we analysed the collected queries at the sentence level using constituency parsing in addition to at the word level using term frequency.

3.2 Finding Common Patterns using Term Frequency

This study employed term frequency to find common patterns in the novice queries training dataset. Term frequency is the frequency of a term or word in a document [22]. In the context of this study, a “document” refers to the set of queries collected for a particular API phrase. Following that, term frequency in this study refers to the number of times a term appears in the set of queries collected for a particular API phrase.

As mentioned earlier, our preliminary analysis of the novice queries dataset shows that 88% of the queries contain terms used to indicate the return type of the sought API methods or the implementation API classes. Therefore, we used the return type of API method and the implementation API class to group queries together before analysing them to find the common terms in Steps 5 and 6 below.

The steps for finding common patterns in the collected queries by using the term frequency are:

1. Remove all the stop words (such as "a", "an", "and", and "are") from the queries.
   This is to reduce the noise in the calculation of the frequencies of the terms that appeared in the queries (or terms’ frequencies).
2. Calculate and record the frequencies of each remaining term that appeared in the set of queries collected for an API phrase.

This was done for each of the 22 API phrases' query sets. For example, it was found that the term "combine" appeared 10 times in the set of queries collected for the API phrase *addAll(Collection<? extends E> c)* and this was recorded.

3. Remove uncommon terms, namely terms that appeared only once in the set of queries collected for an API phrase.
   This was done for each of the 22 API phrases' query sets. Uncommon terms were removed because they are not representative of words that were commonly used by different novice programmers to describe the programming task related to the respective API phrase.
4. Remove terms that appeared in more than half of the 22 API phrases' query sets.
   Common terms (such as "Java", "method", and "use") that novice programmers frequently used in most of the queries, regardless of the API phrase the queries were for, were removed. The reason is that terms that occur frequently across the entire collection are less useful in discriminating among the documents [22]. This study arbitrarily chose half of the API phrases' query sets as the threshold value because there is no existing work available as a reference.
5. Identify common terms in the queries for the API phrases that correspond to the API methods that have the same return type.
   For example, the API phrases *equals(Object o), contains(Object o), contains(CharSequence s), isLetter(char ch), isDigit(char ch), hasNext()*, and *addAll(Collection<? extends E>* c) all correspond to API methods that produce a return value of the *boolean* data type. The terms that appeared in this group of seven API phrases' query sets were pooled together, and the total frequencies for each term were calculated by adding the frequencies of the same term from the seven API phrases' query sets.
   If a term appears in the queries for more than half of the group of the API phrases, it is a common term for this group. For the example above, the terms "check" and "whether" appeared in queries for more than half of the seven API phrases, i.e., 5 and 6 of the API phrases, respectively. Therefore, they were the common terms for this group of API phrases.
6. Identify common terms in the queries for the API phrases that correspond to the API methods that have the same implementation API classes.
   For example, API phrases *contains(CharSequence s), substring(int beginIndex), substring(int beginIndex, int endIndex), replace(CharSequence target , CharSequence replacement),* and *split(String regex)* correspond to API methods that have the same implementation API class *java.lang.String*. The terms that appeared in this group of five API phrases' query sets were pooled together, and the total frequencies for each term were calculated by adding the frequencies of the same term from the five API phrases' query sets. If a term appears in the queries for more than half of the group of API phrases, it is regarded as a common term for this group of API phrases.

Common terms and the respective pattern found using term frequency: While identifying common patterns using term frequency, the terms "check", "whether", "boolean", "true", or "false" were found exclusively in the queries of the group of seven API phrases that correspond to API methods having a *boolean* return type. The term "check" recorded the highest term frequency (49) in the group and appeared in queries for 6 out of 7 API phrases in the group. It was followed by the term "whether", with a term frequency of 14 and appeared in queries for 5 out of 7 API phrases in the group. Even though the terms "boolean", "true", and "false" scored lower term frequencies, which were 8, 9, and 6, respectively, these terms are closely related to the *boolean* return type, and appeared in queries for most of the API phrases in the group.

As a result, the first pattern established is the use of common terms "check", "whether", "boolean", "true", or "false" in queries to look for API methods with a *boolean* return type. For example, the term "check" appears in the query "check if input is part of string", written by one of the novice programmers to search for the API method *contains(CharSequence s)* that has a *boolean* return type. This pattern links non-API terms such as "check" and "whether" to API methods having a *boolean* return type. Refer to Pattern 1 in Section 3.4 of this paper for further details.

Apart from the above, there was no common term found in queries with respect to API classes and other return types.

3.3 Finding Common Patterns using Constituency Parsing

Constituency parsing analyzes a sentence in groups of words that may behave as a single unit or phrase (aka constituent) [22], such as a noun phrase (NP), a prepositional phrase (PP), a verb phrase (VP), and a conjunction phrase (CP). An NP refers to a sequence of words surrounding a central noun. A PP has a preposition (such as "to" and "into") followed by an NP. A VP consists of a verb and several other constituents. A CP consists of a word that connects words, phrases, or clauses, such as "whether".

The data type of the value returned by a Java API method is in the form of a noun. It can be either one of the primitive data types (such as *float* and *boolean*), void, an API class, an API interface, or a collection of one of these (such as an array of type *float*, *float[]*). The name of an implementation API class of a Java API method is typically a noun as well, for example, *String* and *ArrayList*. Nevertheless, our preliminary analysis of the novice queries dataset found that there are many occurrences where the return type of API methods and the implementation API classes were expressed in the form of a phrase, in particular PP or NP, instead of a single noun word in the collected queries. For example, most of the queries collected for programming task 12 (related to the *size()* method) contain the phrases "number of elements" or "number of value" which is essentially a PP.

Due to that, constituency parsing was performed to break the collected query sentences into phrases or constituents and classify the phrases. The identified phrases were then analysed to find common patterns that are related to the return type of API method and/or the implementation API class. This was done for those API phrase groups with no common terms found in their corresponding queries when term frequency was applied in the previous step (Section 3.2 of this paper).

The steps for using constituency parsing to find common patterns in the collected queries are:

1. Identify the types of phrases in the queries by performing constituency parsing on the queries using the Stanford CoreNLP API [23]. For example, "get a single character from a string" is one of the queries collected for the API phrase *charAt(int index)*. The word "get" forms a VP, the phrase "a single character" is an NP, and the phrase "from a string" is a PP. The result of constituency parsing on this query is VP NP PP.
2. Identify the positions of phrases that contain the return type and the implementation API class found in the queries.
   For example, in the query "get a single character from a string", the NP "a single character" contains the return type *char*, and the PP "from a string" contains the implementation API class *String*. The phrase containing the return type was positioned as the first noun phrase (NP1), and the phrase containing the implementation API class was positioned as the first prepositional phrase (PP1) in the query sentence. The numbering of the phrases was given because more than one occurrence of the same type of phrase might appear in a query, for example, there are NP1 and NP2 in the third pattern discovered (refer to Pattern 3 in Section 3.4 of this paper).
3. Identify common patterns of queries.
   Firstly, by analysing the types and positions of phrases containing the return type and the implementation API class in the queries, it was found that the return type and the implementation API class were mostly in either NP1 or PP1. Secondly, the frequently occurring words found in these phrases were identified. For instance, the prepositions "to", "into", "in", "inside", "from", and "of" were found in PP1 in most of the queries. Thirdly, the relative positions of these phrases were analysed and NP1 was mostly found positioned before PP1 in most of the queries. For the example given in step 2, NP1 contains the return type, PP1 contains the implementation API class, and NP1 is positioned before PP1.

Common patterns found using constituency parsing: It was found that 63% of the queries for the remaining 15 API phrases (other than queries for API phrases that correspond to API methods with a *boolean* return type) contain the return type or implementation API class in the first prepositional phrase (PP1) of the queries. Specifically, the return types were found in PP1, which contains the prepositions "to" and "into". Refer to Pattern 2 in Section 3.4 of this paper.

56% of the queries for the remaining 15 API phrases have PP1 containing the prepositions "of", "in", "inside", or "from". A slightly more complicated pattern was found for this group of queries. This third pattern is based on the

combination of NP1 followed by PP1. The return type can be found as NP1, and the implementation API class can be found as NP2 inside PP1, which contains the prepositions “of”, “in”, “inside”, or “from”. Refer to Pattern 3 in Section 3.4 of this paper.

The remaining 37% of queries for the remaining 15 API phrases were covered in the fourth pattern. If the queries do not contain any of the prepositions specified above, NP1 contains either a return type or an implementation API class. Refer to Pattern 4 in Section 3.4 of this paper.

3.4 Novice Programming Task Description Model (NPTDM)

The patterns discovered from analysing the collected queries using term frequency and constituency parsing are captured in NPTDM. Since NPDTM focuses on the terms and constituent structures used in queries, it could cater to other programming languages with constructs similar to Java.

NPTDM captures four common patterns that exist in novice programmers’ descriptions of their programming tasks in NL search queries written to look for API methods that are relevant to the programming tasks described. These common patterns are related specifically to the return type of the API method sought and the implementation API class.

The four patterns are described below. These patterns are stated in bracketed notation [22] that is used to represent a parse tree of sentences or, specifically, queries in the context of this study.

Pattern 1: [VP check] | [CP whether] | [NP boolean | true | false]

If query S is for finding an API method with a return type of *boolean* value, S contains either the verb phrase (VP) term “check”, the conjunction phrase (CP) term “whether”, or the noun phrase (NP) term “boolean”, “true”, or “false”.

The common terms novice programmers used in search queries to look for API methods with a return type of *boolean* value are “check”, “whether”, “boolean”, “true”, or “false”.

This study did not find any common terms that novice programmers use in search queries to look for API methods with other return types (such as *int*, *char*, and so on) and in search queries that contain implementation API classes.

Pattern 2: [PP1 [TO to | IN into] [NP returnType]]

If query S has its first preposition phrase (PP1) in the form above, i.e., PP1 contains the prepositions “to” or “into”, the noun phrase (NP) after the preposition is the return type (returnType) of the sought API method. “TO” is the part-of-speech tag for “to”. “IN” is the part-of-speech tag for “into”.

The common pattern novice programmers used to state the return type of the sought API method is to precede it with the preposition “to” or “into”.

Pattern 3: [NP1 returnType] [PP1 [IN of | in | inside | from] [NP2 APIclass]]

If query S has its first noun phrase (NP1) followed by its first preposition phrase (PP1) in the form above, i.e., PP1 contains either the preposition “of”, “in”, “inside”, or “from”, then NP1 is the return type (returnType) of the sought API method and/or the noun phrase (NP2) after the preposition is the implementation API class (APIclass). “IN” is the part-of-speech tag for “of”, “in”, “inside”, and “from”.

The common pattern used by novice programmers to state the return type of the sought API method and/or the implementation API class is to succeed the return type with either the preposition “of”, “in”, “inside”, or “from”, or to precede the implementation API class with either one of these prepositions, or to connect the return type and the implementation API class with either one of these prepositions.

Pattern 4: [NP1 returnType | APIclass]

If none of the first 3 patterns is found in query S and query S has its first noun phrase (NP1) in the form above where NP1 does not contain the prepositions “to”, “into”, “of”, “in”, “inside”, or “from”, NP1 is either the return type (returnType) of the sought API method or the implementation API class (APIclass).

The common pattern used by novice programmers to state the return type of the sought API method or the implementation API class is to simply state the return type or the implementation API class.

NPTDM provides some insights into how novice programmers compose their search queries. In particular, NPTDM shows some common patterns that exist in their queries. NPTDM can be used as a reference for further research in NLP on novice programmers’ search queries.

In addition, NPTDM can be operationalised for use in API recommendation tools to analyse search queries submitted to these tools and improve the recommendations they produce for novice programmers. In particular, the operationalised NPTDM can be used to identify the return type of the sought API method and/or API class that provides the implementation for the method. These two pieces of information are important and can be used to narrow down API methods that are found relevant to the programming tasks described in the queries.

4.0 EXEMPLAR USE CASE OF NPTDM – THE RBM

The following presents an exemplar use case of NPTDM. The four common patterns in NPTDM were translated into the following if-then rules that were implemented in a rule-based module (RBM) that would be used in our APIFind approach:

Rule 1: IF the query contains any of the terms ‘check’, ‘whether’, ‘boolean’, ‘true’, and ‘false’, THEN the return type of the sought API method should be *boolean*.

Rule 2: IF the query contains prepositions, either ‘to’ or ‘into’ in PP1, THEN the noun phrase in PP1 should be the return type of the sought API method.

Rule 3: IF the query contains prepositions, either ‘of’, ‘in’, ‘inside’, or ‘from’ in PP1, THEN it should be either NP1 is the return type of the sought API method, or NP2 is the implementation API class, or both.

Rule 4: IF none of the rules above applies to the query, THEN NP1 should be either the return type of the sought API method or the implementation API class.

The RBM performs the following: checks the terms or words that appear in a user query (or expanded user query) against those terms that appear in Rule 1. If none of the terms in Rule 1 is found, the RBM then performs constituency parsing on the user query using the Stanford CoreNLP API and analyses the results of the parsing using the remaining three rules. The output of the RBM would be the return type of the sought API method and/or the implementation API class.

Our APIFind approach produces an initial list of ranked API methods found relevant to a user query. APIFind then uses the return type and/or implementation API class identified by the RBM as filters to remove API methods not having the identified return type or implementation API class from the initial ranked list of relevant API methods. If RBM does not identify any return type or implementation API class, APIFind will accept all the API methods in the ranked list without filtering them.

5.0 THE PROPOSED API RECOMMENDATION APPROACH (APIFIND)

The proposed approach, APIFind, consists of a preparation phase and a recommendation phase. In the preparation phase, the following were constructed: a programming task dataset, a WordNet map of API word-synonyms, a Java API class and method repository, and a SO Q&A thread repository. During the recommendation phase (Fig. 3), after

a user query is expanded using the WordNet map, API phrases relevant to the expanded query are determined from the programming task dataset by using the BM25 model in Apache Lucene [24]. Then, API methods (and their descriptions) that correspond to these API phrases are extracted from the Java API class and method repository and ranked. The ranked list of API methods is then filtered by removing those that do not have the return type or do not belong to the implementation API class identified by the RBM. SO Q&A thread repository is then used to find the hyperlinks to related SO threads for the top-5 of the filtered API methods. Finally, the top-5 API methods, together with the descriptions of the methods' functionality, parameter(s), and return type, the similar methods, and the hyperlinks to the related SO threads, are returned to the user.

5.1 The Preparation Phase

The purpose of the preparation phase is to pre-process and construct data that would be used in the recommendation phase. This includes the construction of a programming task dataset, a WordNet map of API word-synonyms, a Java API class and method repository, and a SO Q&A thread repository.

5.1.1 The Programming Task Dataset

The programming task dataset was constructed from the novice queries training dataset. All the queries collected for a particular API phrase were extracted from this dataset and stored in a text file dedicated to the API phrase. This was done for the 22 API phrases, resulting in a text file for each API phrase. The stop words in the queries in each text file were removed. The resulting 22 text files were stored in a folder, and these "documents" were indexed using the BM25 model in Apache Lucene. The indexing process involved identifying the unique terms or the "vocabulary" of the queries for each API phrase, and computing and storing the term frequency (TF) score and inverse document frequency (IDF) score for each term in the vocabulary. A correction factor by the length of the text in a file was also computed to address the significance of a term in a short text over a longer text [25].

The resulting programming task dataset is essentially a dataset that contains the terms that novice programmers used in the queries that they wrote for finding API methods that are relevant to the respective programming tasks. During the recommendation phase, the programming task dataset is used to identify API phrases that are relevant to an expanded query.

5.1.2 The WordNet Map of API Word-synonyms

As mentioned in the Introduction section, programmers face a "lexical gap", where they write search queries using terms that are different from the ones used in the API documentations. As a result, searching techniques based on term matching produce incorrect search results. To alleviate this limitation, this study constructed a WordNet map of API word-synonyms to enable the conversion of the terms used in search queries to terms that are used in the API documentations.

To construct the WordNet map of API word-synonyms, tokens (specifically words in this case) that appear in each API phrase were extracted from the API phrase. This was done by tokenizing the API phrase based on camel case and special characters "<", ">", "(", and ")". The reason for using camel case is that it is a common naming convention used in coding [26]. The resulting tokens were converted to lowercase letters. This study calls these tokens "API words" since they are words found in an API phrase. As an example, the API words derived from the *add(E e)* API phrase are *"*add*", "*E" and *"*e".

Using the Extended Java WordNet Library (extJWNL) [27], the API words found for each of the API phrases were then fed to the WordNet lexical dictionary to obtain their synonyms from the synsets (i.e., sets of synonyms) found in the WordNet. The API words were stored together with their corresponding synonyms as pairs of API word-synonyms in what we called WordNet map of API word-synonyms.

The synset for a particular API word contains possible NL terms programmers might use in the queries they write to search for API methods that correspond to the API word. During the recommendation phase, the WordNet map of API word-synonyms is used to identify the API word(s) that correspond to the terms used in the user query. The API word(s) found will be appended to the user query so that the query will contain words or terms used in the API documentation. Refer to Section 5.2.1 of this paper for an example.

5.1.3 The Java API Class and Method Repository

The Java API class and method repository was constructed to store the Javadoc descriptions of API classes and methods covered in the scope of this study. The JDK 13 documentation was downloaded in the form of the Javadoc HTML files [28]. jsoup was used to identify and extract concrete API classes that cover the API phrases of this study from the respective Javadoc HTML files. A total of 598 concrete API classes were extracted. For each concrete API class, all its methods and descriptions were extracted by identifying the HTML elements using jsoup. The extracted descriptions include the functions of the Java class; the functions of the methods; the parameters, return type, and name of the methods; the similar methods; and the related method suggestions. The similar methods were extracted from the "See Also" section of a method's description in the Javadoc HTML file. The descriptions extracted for each Java API method were stored in a separate text file. All the text files were then indexed using the BM25 model in Apache Lucene.

The Java API class and method repository is used to provide the API classes' and methods' descriptions needed in the recommendation phase.

5.1.4 The Stack Overflow Q&A Thread Repository

As mentioned in Section 2.5 of this paper, the results for Section 2 Part B Question 2 of the survey conducted to collect search queries from novice programmers show that novice programmers perceive that the following types of information would help them carry out the respective programming tasks described in their search queries: links to relevant web pages, similar questions asked by others, and code snippets. To provide this additional information together with the API methods recommended for a user query by our APIFind approach, a SO Q&A thread repository was constructed.

The SO Q&A threads (posts) were obtained from the Stack Exchange data dump in the form of an XML file. These threads were filtered to include only threads with questions that are annotated with a <java> tag and have a positive score. Non-Java threads were removed as they were not related. A positive score for a question is the number of upvotes given by other SO users for the question. It gives a positive indication of the usefulness or quality of the question. Threads with negative scores were discarded to exclude threads with poorly-described questions.

The next step was to identify SO Q&A threads that are related to those API methods that appear in the Java API class and method repository constructed earlier. This was done by checking the body text of the accepted answers of the SO Q&A threads against the list of API methods that appear in the Java API class and method repository. If the body text of the accepted answer of a SO Q&A thread contains any of the API methods that appear in the Java API class and method repository, the question ID, question description, corresponding API method(s), and accepted answer of the thread were extracted and saved in a text file. This was done for all such SO Q&A threads. The final text file was indexed using the BM25 model in Apache Lucene.

In the recommendation phase, the SO Q&A thread repository is used after the top-5 relevant API methods have been found for a user query.

5.2 The Recommendation Phase

The recommendation phase takes a user query as the input and produces a list of the top-5 API methods that are relevant to the user query, together with the descriptions of the methods' functionality, parameter(s) and return type of the methods, similar methods, and hyperlinks to related SO threads. This phase consists of six steps, labelled (1), (2), (3), (4), (5), and (6) in Fig. 3, which are explained in the respective sub-sections below.

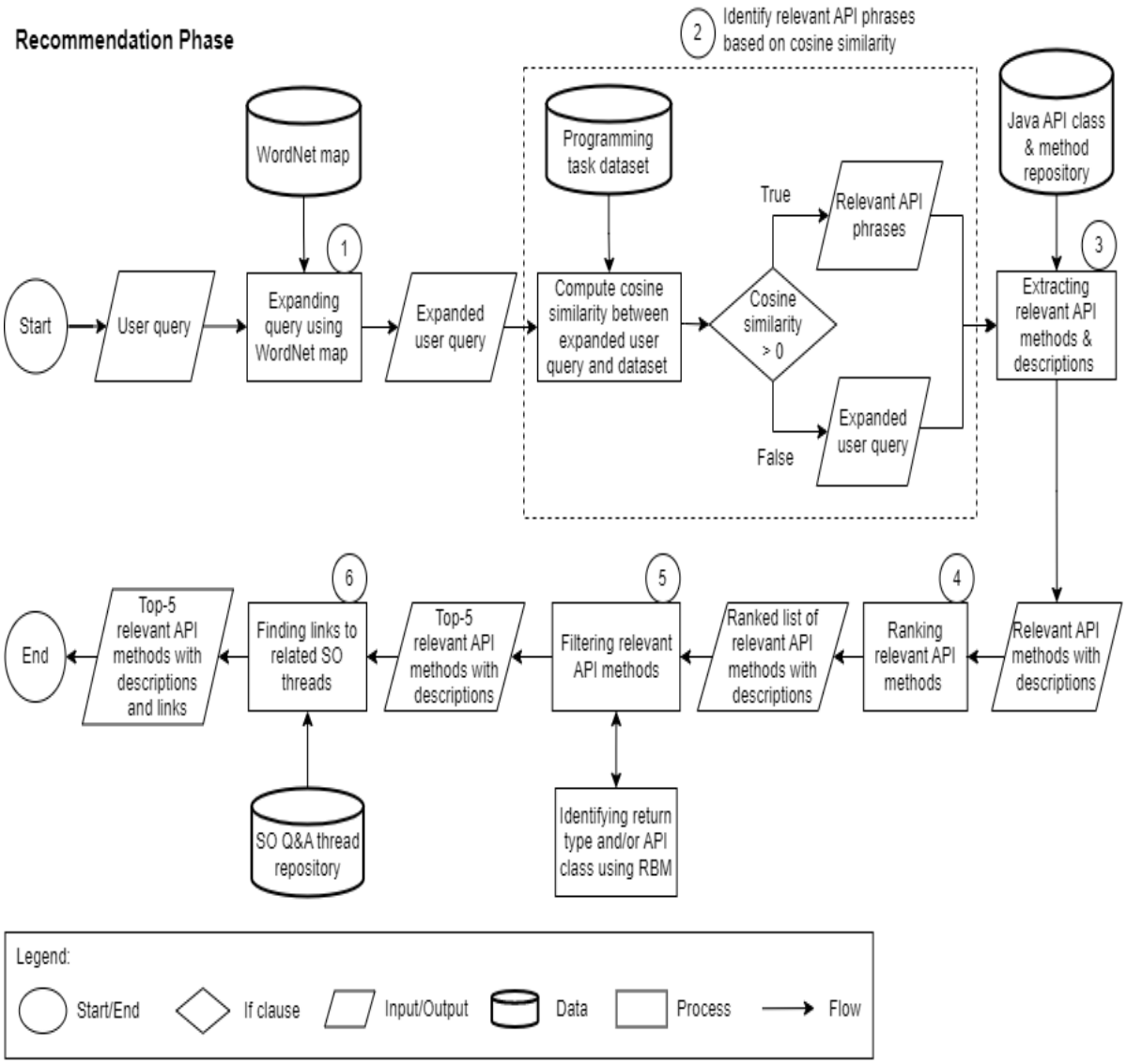

Fig. 3. The recommendation phase of the proposed approach

5.2.1 Step 1: Expanding the Query using the WordNet Map of API word-synonyms

This step expands a user query by appending it with API word(s) found in the WordNet map of API word-synonyms. Each word in the user query is checked against the WordNet map to see whether the word exists as a synonym for any API word in the WordNet map. If it does, the corresponding API word is appended to the user query. For example, for a user query "Replace target in a list by position number", the word "target" exists as a synonym for the API word "Object" in the WordNet map. The API word "Object" is then appended to the end of the user query, resulting in an expanded query "Replace target in a list by position number object".

5.2.2 Step 2: Identifying Relevant API Phrases for the Expanded Query based on Cosine Similarity

This step identifies relevant API phrases for the expanded query by computing cosine similarity between the expanded query and those queries in the programming task dataset by using the BM25 model in Apache Lucene. The cosine similarity for the BM25 model used in Apache Lucene is the product of the TF, IDF, and correction factor (CF) scores.

$$Cosine\ Similarity_{BM25} = TF_{BM25} \times IDF_{BM25} \times CF_{BM25}$$

An API phrase is taken as relevant to the expanded user query if the value of the cosine similarity is greater than zero and irrelevant if the value of the cosine similarity is zero. If relevant API phrases are found, they will be passed to the next step. Otherwise, the expanded user query will be passed to the next step instead.

5.2.3 Step 3: Extracting Relevant API Methods and Their Descriptions

This step extracts relevant API methods and their corresponding descriptions from the Java API class and method repository. The corresponding descriptions include information about the respective classes.

a) If relevant API phrases are found in the previous step, this step identifies the API methods which contain these relevant API phrases from the API methods in the Java API class and method repository, and then extracts the API methods' descriptions from the repository.

b) If no relevant API phrases are found in Step 2, this step makes use of the expanded user query instead and computes the cosine similarity values between the expanded user query and API methods' descriptions in the repository by using the BM25 model in Apache Lucene. This step then identifies API methods relevant to the expanded user query and extracts these API methods' descriptions from the repository. An API method is taken as relevant to the expanded user query if the value of the cosine similarity between the expanded user query and the method's description is greater than zero and irrelevant if the value of the cosine similarity is zero.

5.2.4 Step 4: Ranking Relevant API Methods

This step ranks the relevant API methods found in Step 3 based on cosine similarity values between these API methods' descriptions and the expanded user query. The relevant API method with the highest cosine similarity value is given the first position in the ranking, the relevant API method with the second highest cosine similarity value is given the second position in the ranking, and so on.

If relevant API phrases are found in Step 2, the cosine similarity values between the expanded user query and API methods' descriptions (extracted in Step 3.a) will be calculated first by using the BM25 model in Apache Lucene before the ranking of the relevant API methods is performed.

5.2.5 Step 5: Filtering Ranked List of API Methods using the Rule-Based Module (RBM)

This step filters the ranked list of API methods obtained from the previous step by removing those API methods with return types that are different from the return type identified by the RBM for the expanded user query. It also removes from the ranked list those API methods implemented by API classes that do not match the implementation API class identified by the RBM. The top-5 methods of the resulting ranked list of API methods (together with the methods' descriptions) are passed to the next step.

5.2.6 Step 6: Finding Hyperlinks to Related SO Threads

This step finds hyperlinks to SO threads related to the top-5 relevant API methods discovered in the previous step. It uses the BM25 model in Apache Lucene to compute the cosine similarity values between the expanded user query and question descriptions of SO threads that are in the SO Q&A thread repository. A SO thread is considered related to the expanded user query if the value of the cosine similarity between the expanded user query and the SO thread's question description is greater than zero, and not related if the value of the cosine similarity is zero.

This step then checks the API methods extracted from the accepted answers of the related SO threads against the top-5 relevant API methods produced by the previous step. If the accepted answer of a related SO thread has one of the top-5 relevant API methods, this step then extracts the question ID of the SO thread and uses the ID to construct a hyperlink to the original web page of the SO thread. The hyperlink is returned together with the respective top-5 relevant API method to the user. With the hyperlinks to the related SO threads, the user will be able to navigate to the SO threads where the user may find questions that are similar and code snippets that are relevant to the user query.

6.0 PERFORMANCE AND BENCHMARKING OF THE PROPOSED APPROACH

The benchmarking of APIFind was done by comparing its performance with that of BIKER [9, 15] and CROKAGE [16] which are state-of-the-art API recommendation approaches. Just like APIFind, they also produce method-level recommendations instead of class-level recommendations. Mean Average Precision (MAP@K) and Mean Reciprocal Rank (MRR@K) were used as the performance metrics.

6.1 Test Datasets

The performance of APIFind was evaluated using two test datasets, which are, the novice queries test dataset and a third-party test dataset obtained from the BIKER study [9]. The original test dataset used in the BIKER study was constructed by two researchers who manually screened the SO question titles collected in the BIKER study. They included those question titles that are related to searching APIs for programming tasks in their test dataset. Out of the 413 queries in the BIKER's test dataset, 48 queries are related to the API methods covered by the scope of our study. We took the 48 queries as a reduced BIKER test dataset and used it to evaluate the performance of APIFind.

6.2 Performance Metrics

This study used MAP and MRR for top-K recommendations as the metrics to evaluate the performance of APIFind. These metrics were used in both the state-of-the-art studies, BIKER and CROKAGE. BIKER produced the top-5 recommendation, and CROKAGE produced the top-1, top-5, and top-10 recommendations. To establish a common ground for comparison, K was set to 5 in this study.

MAP@K – Precision is calculated at the occurrence of every single relevant item in the ranked list [12]. Average precision is the average of the precision for all relevant items within the top-K results for a given query [12]. In this study, "item" refers to an API method.

$$\boldsymbol{AP@K} = \frac{\sum_{k=1}^{K} \boldsymbol{P_k} \times \boldsymbol{rel_k}}{|\boldsymbol{RR}|}$$

where

AP@K = Average precision for all relevant items within the top-K results
K = Number of top results considered
P_k = The precision at the k^{th} result
rel_k = Relevance function of the k^{th} result in the ranked list that returns either 1 (i.e., relevant) or 0 (i.e., non-relevant)
RR = Set of relevant results for a query

When there is no relevant item retrieved in the top-K results (RR = 0), AP is taken to be 0 [29].

Mean average precision is the mean of the average precisions for all the queries in the test dataset, as in the formula below [12].

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

where

MAP@K = Mean average precision for all queries from the test dataset
Q = Set of all queries in the test dataset

MRR@K - Reciprocal rank is the multiplicative inverse of the rank of the first relevant item in the top-K results for a particular query [12]. Mean reciprocal rank averages reciprocal ranks for all queries in the test dataset, as in the formula below [12].

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

where

MRR@K = Mean reciprocal rank within the top-K results for all queries in the test dataset.
rank(q, K) = Rank of the first correct API or the correct code segment from a ranked list of size K.

Since APIFind recommends API methods, rank(q, K) in this study refers to the rank of the first correct API method and not code segment, from a ranked list of size K. Rank(q, K) is taken as infinity if there is no relevant item returned in the top-K results for query q, resulting in the inverse of rank(q, K) to become zero.

6.3 Benchmarking using the Novice Queries Test Dataset

The queries from the novice queries test dataset were fed into APIFind, BIKER, and CROKAGE, and the recommended results were compared with the corresponding ground truth API methods in terms of MAP@5 and MRR@5. Table 2 shows the results, where APIFind outperformed both BIKER and CROKAGE, with a MAP@5 of 0.777 and a MRR@5 of 0.791 when the novice queries test dataset was used.

Table 2: Benchmarking results

Approach	**Novice queries test dataset**		**Reduced BIKER test dataset**	
	MAP@5	**MRR@5**	**MAP@5**	**MRR@5**
APIFind	0.777	0.791	0.520	0.527
BIKER	0.205	0.205	0.477	0.474
CROKAGE	0.351	0.353	0.602	0.613

A higher MAP@5 value indicates that, on average, the top-5 API methods produced by an approach for the set of queries in the test dataset are of higher precision. Following that, with APIFind achieving the highest MAP@5, the results show that, on average, APIFind was able to suggest more ground truth (or correct) API methods as the top-5 API methods for the set of queries in the test dataset compared to BIKER and CROKAGE.

A higher MRR@5 indicates that the respective approach on average returns ground truth API methods at higher positions in the top-5 API methods found for the set of queries in the test dataset. Following that, with APIFind achieving the highest MRR@5, the results show that APIFind was able to return ground truth API methods at higher positions (or ranks) in the top-5 API methods found for the set of queries in the test dataset compared to BIKER and CROKAGE.

APIFind uses the queries in the programming task dataset, which were written by novice programmers. As novices, the queries that they wrote may contain terms or phrases that differ from the actual specific terms used in the particular domain. On the other hand, the queries used in BIKER and CROKAGE came from SO, and the queries might not be written by novice programmers. SO queries tend to be written using terms that are closer to the specific terms used in the respective domain. The reason is that SO users are expected to "search and research" first before posting any questions [30].

When all three approaches were evaluated using the novice queries test dataset, which consists of queries written by novice programmers, it is not surprising that APIFind which uses novice programmers' queries, performed better than BIKER and CROKAGE which use queries that are not novice-specific.

6.4 Benchmarking using the Reduced BIKER Test Dataset

The evaluation was repeated on APIFind, BIKER, and CROKAGE using the reduced BIKER test dataset. When the reduced BIKER test dataset was used, one would expect BIKER to achieve better performance compared to the other two approaches. The results in Table 2 show otherwise, where CROKAGE performed better than APIFind, which performed slightly better than BIKER.

One reason for APIFind to perform better than BIKER could be that the 48 queries in the reduced BIKER test dataset are related specifically to the API methods covered by the scope of APIFind, whereas BIKER was built using SO queries for any Java API methods. One possible reason for CROKAGE to perform better than APIFind, which was built using novice programmers' queries, is that CROKAGE was built using SO queries that were not specifically written by novice programmers, and the queries in the reduced BIKER test dataset were also not specifically written by novice programmers, even though the ground truth API methods for these queries are methods that are more at the level of a novice programmer. As mentioned in Section 2.6 of this paper, these API methods were selected from the list of Java API methods covered in the College Board's Advanced Placement Computer Science A Exam (Java Subset).

As both CROKAGE and BIKER were built using non-novice queries, we could not think of a plausible reason for CROKAGE to perform better than BIKER when they were evaluated using the reduced BIKER test dataset.

The evaluation results using the reduced BIKER test dataset show that APIFind, which does not use any data from SO as input in finding API methods that are relevant to a user query, was able to achieve quite comparable performance to those of BIKER and CROKAGE that use SO data as input. Recall that APIFind uses its SO Q&A thread repository in its final step just to get additional information (i.e., hyperlinks to SO threads that are related) for the top-5 relevant API methods after these methods are found by the earlier steps.

In summary, the evaluation using the two test datasets shows the potential of APIFind in finding API methods that are relevant to novice programmers' queries (as in the novice queries test dataset) and that are relevant to SO queries or questions (as in the reduced BIKER test dataset).

7.0 ECLIPSE PLUG-IN AND THE BACK-END RECOMMENDER

APIFind was implemented in a recommender tool comprising an Eclipse plug-in that serves as the front end and a server component running at the back end. A user evaluation study was conducted where novice programmers were recruited to evaluate the usability of our recommender tool, BIKER and CROKAGE tools, using effectiveness, time-based efficiency, overall relative efficiency, and System Usability Scale (SUS) scores as evaluation metrics. In general, the evaluation results show that our recommender tool performed better than the BIKER and CROKAGE tools in terms of effectiveness and efficiency but slightly worse in terms of user satisfaction measured using SUS. Further details are given below.

The APIFind tool achieved 0.82 for effectiveness, 0.0108 for time-based efficiency, 0.808 for overall relative efficiency, and 66.0 for SUS. The BIKER tool achieved 0.44, 0.0049, 0.366, and 70.0 for the four metrics, respectively. The CROKAGE tool achieved 0.52, 0.0048, 0.533, and 69.5 for the four metrics, respectively.

The APIFind tool was able to achieve the highest effectiveness (where the participants were able to give the most correct answers by using the APIFind tool) and the highest overall relative efficiency (where the participants were able to get the correct API methods most of the time), probably because the information provided by the APIFind tool for each of the API methods it recommended is sufficient to help a novice programmer in identifying the most suitable API method to be used in his or her programming task. This information covers the implementation API class; the descriptions of the API method's functionality, parameter(s), and return type; the similar methods; and the hyperlinks to the related SO threads. One of the participants who evaluated the APIFind tool commented that "It makes me know more API methods, especially when solving problems. There are explanations that help me identify which methods are suitable for the question given."

On the other hand, the BIKER tool lacks information that could help a novice programmer find the most suitable API method for his or her programming task. This was pointed out by a participant who evaluated the BIKER tool: "I think

it needs more explanation on the API method instead of just showing relevant questions and code snippets. Because someone like me, who is not very good at programming, does not really understand the code snippets. I need to guess whether this method is relevant to what I am searching for based on the relevant questions section. Perhaps it could also include some input and output examples for better understanding." A participant who evaluated the CROKAGE tool made a similar observation: "The explanation could be improved, and the user interface will be much more comfortable if it is enhanced."

The APIFind tool also achieved the highest time-based efficiency (where the participants used the least time to find the correct API method for each task). The clean layout of the APIFind tool could have contributed to that. One participant who evaluated the APIFind tool stated that the APIFind tool is a "clean and neat tool for searching API methods for Java. A quick and useful tool for Java beginners." On the other hand, one of the participants who evaluated the CROKAGE tool commented that "the interface can be more attractive." This might be because the CROKAGE tool provides explanations and code snippets of all suggested solutions on a single web page, causing the page to become clustered and long. The BIKER tool has a similar interface, with the descriptions of the suggested API methods and the code snippets displayed on a single web page.

The APIFind tool achieved a slightly lower SUS score (66.0) compared to the BIKER (70.0) and the CROKAGE (69.5) tools, probably because the participants who evaluated the APIFind tool were required to download and install it as a plug-in for the Eclipse IDE. The participants who evaluated the BIKER and CROKAGE tools assessed them on their websites.

8.0 THREATS TO THE VALIDITY OF THE RESULTS

Internal validity: The following measures were taken to reduce the threats to the internal validity of the results. As mentioned earlier, the queries in the novice queries dataset were divided into a novice queries training dataset and a novice queries test dataset, and only the novice queries training dataset was used to construct the programming task dataset used in APIFind. The novice queries test dataset that was used for benchmarking the performance of APIFind was not used in APIFind so that the benchmarking could show how well APIFind would perform when given new queries.

The benchmarking of APIFind was also done using a second test dataset, namely, the reduced BIKER test dataset, to get a fairer comparison between APIFind, BIKER, and CROKAGE approaches. The quality of the queries in the reduced BIKER test dataset remained intact as these queries originated from the BIKER test dataset used in the evaluation of BIKER approach [9].

In addition, for performance benchmarking, this study evaluated the performance of BIKER and CROKAGE by using the tools implemented by BIKER and CROKAGE. This ensured that the BIKER and CROKAGE approaches evaluated are exactly the same as in their published papers.

External validity: The main threat to the external validity of this study is the use of small datasets in APIFind and in the benchmarking of APIFind. There was difficulty in recruiting novice programmers to write search queries. The novice queries training dataset used in the construction of the NPTDM and the construction of the programming task dataset employed by APIFind comprises 264 queries written by novice programmers. The novice queries test dataset used in the benchmarking comprises a different set of 110 queries written by novice programmers. Even though the datasets used are small, they are of high quality. A rigorous procedure was designed and followed in collecting the queries from the novice programmers and segregating the queries into the novice queries training and test datasets. In addition, the benchmarking of APIFind was also done by using a third-party test dataset, i.e., the reduced BIKER test dataset that comprises 48 queries.

Another threat is that the novice programmers' queries collected and used in this study cover only 598 methods in the JDK 13 API. Following that, the benchmarking results are not generalizable to other Java API methods not covered in this study. However, novice programmers' queries for more Java API methods can be included in APIFind using our current preprocessing code without affecting the core implementation of APIFind. This can be part of our future work.

Construct and conclusion validity: The performance of APIFind was benchmarked against BIKER and CROKAGE using MAP@K and MRR@K. Both MAP@K and MRR@K are established metrics that have been used in a number of existing API recommendation studies to measure the performance of their approaches or techniques [9, 13, 16]. The use of these reliable metrics aided in mitigating the threats to our study's construct and conclusion validity.

9.0 RELATED WORK

Bajracharya and Lopes analysed the usage log of Koders, a commercial code search engine, in terms of the usage data (such as duration and activities in each usage session) and the user queries [26]. They analysed the length of queries, the distribution and the types of the term (NL term versus code term) in the queries. They found the queries to be very short, and the terms used in them were quite diverse. To study the ways in which users express their queries, they did a qualitative analysis on 150 randomly selected search sessions from the query log. They discovered 5 different forms of lexical structure in the queries that show the different levels of verbosity queries can exhibit: 1) a verbose NL phrase; 2) a set of a few NL words; 3) a single word resembling a name used in the source code; 4) an acronym; and 5) a few lines of code. From the result types expressed in the queries, they found that users seek results at different levels of granularity, ranging from an entire system, feature, or entity (such as a class or method) to a certain line in code.

Our study differs from Bajracharya and Lopes' study [26] in that we analysed novice programmer queries, while they analysed non-novice queries. In addition, we analysed the novice programmer queries at the syntactic and semantic levels, and took the types and positions of the phrases in the queries into consideration when identifying the common patterns in the novice programmer queries, and we discovered more specific patterns compared to Bajracharya and Lopes' discovery of the different forms of the lexical structure and the different levels of granularity of results sought in queries.

The following presents the main existing studies on recommending relevant API methods or classes for NL queries.

RACK recommends the top-K relevant API classes for an NL query by using the token-/keyword-API associations mined from SO threads [11, 12]. RACK uses 3 heuristics to obtain API classes from its token-API pairs database. The heuristics are Keyword-API Co-occurrence (KAC) [11, 12], Keyword-Keyword Coherence (KKC) [11, 12], and Keyword Pair–API Co-occurrence (KPAC) [12]. The RACK study also derived API Co-occurrence Likelihood and API Coherence metrics based on the 3 heuristics. The top-K API classes are ranked based on the aggregated scores of the two metrics calculated for each API class. RACK was later outperformed by NLP2API [13, 14] and BIKER [9, 15].

Two of the researchers of RACK, Rahman and Roy, proposed the NLP2API [13, 14]. NLP2API used Apache Lucene to index SO threads as its corpus. Given a user query, NLP2API retrieves the top-K threads from its corpus and takes them as pseudo-relevance feedbacks that are further analysed to identify candidate API classes that will be used to reformulate the user query. NLP2API also applies TF-IDF and PageRank to produce four ranked lists of candidate API classes and calculates Borda scores for each candidate API class in the four ranked lists. It then uses a FastText word2vec word embedding model to calculate the similarity proximity between a query and each candidate API class. The similarity proximity score is added to the normalised Borda score to produce a final similarity score for the respective candidate API class. The candidate API classes are then ranked based on their final similarity scores.

The work on BIKER used SO Java threads to train a word2vec model, and construct a word IDF vocabulary and a knowledge base of API-related SO questions [9, 15]. BIKER identifies the top-50 question titles similar to a query from its knowledge base by using the word2vec model and the word IDF vocabulary. It also uses 2 heuristics to extract the API methods from the answers to these question titles. It then uses a harmonic mean of SimSO (similarity between the user query and SO threads in its knowledge base) and SimDoc (similarity between the query and the API description in the official API documentation) to rank the API methods. It returns the top-K API methods with the implementation API classes, the full path to the classes' packages, the official method descriptions, the top-3 similar question titles, and the code snippets as the results for a query.

CROKAGE recommends programming solutions (comprising code examples and explanations for the code examples) for a programming task query [16][17]. CROKAGE uses the BM25 to select the candidate answers for a query from

its Lucene-indexed knowledge base of SO question-answer pairs. It then calculates 4 similarity scores (i.e., semantic, API class-based, TF-IDF, and API method similarity) by making use of a FastText model, an IDF Map (containing pairs of word and the corresponding IDF value); a combined ranked list of API classes recommended by RACK, NLP2API, and BIKER tools; TF-IDF; and the frequency of API methods appearing across multiple candidate answers. CROKAGE calculates a final relevance score for each answer to a query by normalising the 4 similarity scores and summing the normalised values. It then uses the part-of-speech (POS) structure to compose an NL explanation for each suggested code example. CROKAGE was found to be better than BIKER in terms of the relevance of the suggested code example, the benefit of the code explanations, and the quality of the overall solution that is made up of code and explanation.

APIRecJ uses a Doc2Vec model and the Latent Dirichlet Allocation (LDA) topic modelling to recommend Java API classes for NL queries [18]. The Doc2Vec model was trained using SO question titles and is used to find question titles that are semantically similar to a query. LDA is applied to the Java API classes (found in the code snippets in the accepted answers to these similar questions) to extract a single topic comprising the top-10 Java API classes that are relevant to the query. APIRecJ is a less complex approach that makes use of some basic machine learning models, in particular, Doc2Vec and LDA. It was found to perform better than RACK and slightly better than NLP2API.

9.1 Comparison of Studies

This section explains how our study differs from the existing studies mentioned above in terms of the data and techniques used, the query reformulation, and the level of recommendation.

In terms of the data used, all the aforementioned existing studies, RACK [11, 12], NLP2API [13, 14], BIKER [9, 15], CROKAGE [16] [17], and APIRecJ [18] used SO threads as the main source of data for API recommendation. They utilised SO as crowdsourced knowledge in the API domain to train models or mine associations that would be used in their approaches. Our study did not use SO threads for the same purpose because it focused on novice programmer queries, and so far there is no feasible way to identify which SO questions were posted by novice programmers. Our study used queries that we collected directly from novice programmers. Nevertheless, our study used SO threads for a different purpose, namely, to construct hyperlinks to SO threads that are related to the top-5 relevant API methods found by the earlier steps of our APIFind approach.

The existing studies, including ours, used a mixture of techniques in their approaches. Techniques related to term weighting and word embedding were commonly used among the approaches. In terms of term weighting, NLP2API and CROKAGE used TF-IDF; BIKER and CROKAGE used IDF; CROKAGE and our APIFind used the BM25 model; and RACK and APIRecJ did not use any term weighting technique. In terms of word embedding, NLP2API and CROKAGE employed FastText, BIKER used Word2Vec, and APIRecJ used Doc2Vec. RACK and our APIFind did not use word embedding. Our APIFind approach used WordNet, the BM25 model, and a novel solution, namely, the RBM with the NPTDM.

RACK, NLP2API, CROKAGE, and our study all performed query reformulation, but BIKER and APIRecJ did not.

In terms of the level of recommendation, RACK, NLP2API, and APIRecJ recommend API classes, while BIKER, CROKAGE, and our APIFind recommend relevant API methods for a user query. Hence, for RACK, NLP2API, and APIRecJ, the user has to go through the methods provided by the recommended API classes to determine a suitable method to be used in his or her programming task. For method-level recommendation, BIKER focused on presenting information from the API documentation, such as the functionalities and return types of the API methods, as well as code snippets for the recommended API methods. CROKAGE emphasised more on code snippets and explanations, and the relevant API methods can be found in the code snippets. Our APIFind recommends relevant API methods together with the description of the methods' functionality, parameter(s), and return type, similar methods, and hyperlinks to the related SO threads.

Our APIFind was benchmarked against BIKER and CROKAGE since they also produce method-level instead of class-level recommendations.

9.2 Other Studies

The recent existing studies below are related to API recommendations in some ways, but they are not directly related to our work due to the differences in their main purposes and the nature of the techniques (i.e., neural network) used.

Gu et al. proposed DeepAPI, an approach that is based on deep learning to generate API usage sequences, which are the sequences of API method invocation, for an NL query [31]. DeepAPI adapts the RNN Encoder-Decoder neural language model, encodes a user query into a fixed-length context vector, and generates an API method invocation sequence based on the context vector. DeepAPI uses data from GitHub instead of SO.

DEEPANS is a neural network–based approach that addresses unresolved (no asker-accepted answer) and unanswered SO questions by finding the most relevant answer for them [32]. DEEPANS comprises three stages: 1) question boosting, where a clearer question generated (based on code snippets) using a trained sequence-to-sequence model is appended to the original question; 2) label establishment, where pairs of reformulated questions and corresponding answers are automatically labelled as either positive, neutral+, neutral−, or negative samples; and 3) answer recommendation, where a matching score between a question and an answer candidate is calculated for each of such pairs by using a weakly supervised neural network trained with the four groups of training samples. A higher matching score will have a higher chance of being chosen as the best answer. The same group of researchers also used the sequence-to-sequence model in the question-boosting phase of CODE2QUE [33], a web-based tool that generates question titles for code snippets entered by a user.

The work on CCBERT focused on generating SO question titles that could better reflect the corresponding question body [34]. A high-quality question title could increase the likelihood of the question being answered by the SO community. CCBERT is a model based on deep learning that can address rare tokens and long-range dependencies in the bi-modal (text descriptions and code snippets) context of the entire question body. CCBERT uses a pre-trained CodeBERT model to encode a question body into hidden representations and a stacked Transformer decoder to generate predicted tokens for the question title. CCBERT also uses a copy attention layer to refine the output distribution. Both the encoder and decoder perform the multi-head self-attention operation to improve the capturing of the long-range dependencies.

10.0 CONCLUSION AND FUTURE WORK

This study investigated novice programmers' descriptions of their programming tasks in search queries and used the discovered insight in recommending relevant API methods for the programming tasks to address the lexical gap between the novices' programming task descriptions and the API documentation, and subsequently the API method selection barrier faced by novice programmers. Queries written by novice programmers were collected and analysed using term frequency and constituency parsing to identify common patterns in the queries. Four common patterns related to the return type of an API method and/or API class that provides an implementation for the API method were discovered and captured in the NPTDM.

This study also developed a novel API recommendation approach, APIFind, that utilises the NPTDM, which has been operationalised in the RBM. APIFind uses the return type and/or implementation API class identified by the RBM to filter the API methods found relevant to a search query. APIFind also uses the WordNet map of API word-synonyms to expand a search query, a programming task dataset comprised of the collected novice programmers' queries, a Java API class and method repository derived from the official Javadoc documentation, a SO Q&A thread repository, and the BM25 model in Apache Lucene to produce the top-5 API methods relevant to a search query.

The benchmarking results show that APIFind with a MAP@5 of 0.777 and a MRR@5 of 0.791 performed much better than BIKER and CROKAGE when the novice queries test dataset was used. With a MAP@5 of 0.520 and a MRR@5 of 0.527, APIFind performed slightly better than BIKER but slightly worse than CROKAGE approaches when the reduced BIKER test dataset was used. The evaluation using the two test datasets shows the potential of APIFind in finding API methods that are relevant to novice programmers' queries (as in the novice queries test dataset) and that are relevant to SO queries or questions (as in the reduced BIKER test dataset). APIFind was built into a recommender tool, and evaluation by novice programmers shows that it exhibited better effectiveness and efficiency but slightly lower user satisfaction compared to the BIKER and CROKAGE tools.

In conclusion, common patterns exist in novice programmers' queries and can be used in API recommendations for novice programmers. Future work includes collecting more novice programmers' queries for more API methods, preferably by constructing a mechanism to identify and mine novice programmers' queries automatically from SO. With a larger dataset, more common patterns that exist in novice programmers' queries may be discovered and then used to expand the NPTDM. The larger dataset can also be used to improve APIFind.

ACKNOWLEDGMENT

We thank Universiti Malaya's Faculty Research Grant (GPF001D-2019) for supporting this research. We are grateful to Assoc. Prof. Dr. Ang Tan Fong, Dr. Unaizah Hanum binti Obaidellah, and Assoc. Prof. Dr. Rodziah binti Atan for their help in the recruitment of the participants, and to Mr. Qiao Huang for sharing the implemented tool for BIKER. We also thank all the participants who took part in this research.

REFERENCES

[1] G. Uddin and M. P. Robillard, "How API documentation fails". *IEEE Software*, Vol. 32, No. 4, 2015, pp. 68-75. https://doi.org/10.1109/MS.2014.80

[2] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study", in *2012 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, 2 - 9 June 2012, pp. 266-276. https://doi.org/10.1109/ICSE.2012.6227187

[3] B. A. Myers and J. Stylos, "Improving API usability". *Communications of the ACM*, Vol. 59, No. 6, 2016, pp. 62-69. https://doi.org/10.1145/2896587

[4] P. C. Evans and R. C. Basole, "Revealing the API ecosystem and enterprise strategy via visual analytics". *Communications of the ACM*, Vol. 59, No. 2, 2016, pp. 26-28. https://doi.org/10.1145/2856447

[5] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers". *ACM SIGCSE Bulletin*, Vol. 37, No. 3, 2005, pp. 14-18. https://doi.org/10.1145/1151954.1067453

[6] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems", in *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, Rome, Italy, 26 - 29 September 2004, pp. 199-206. https://doi.org/10.1109/VLHCC.2004.47

[7] C. Heiner and J. L. Zachary, "Improving Student Question Classification", in *International Conference on Educational Data Mining (EDM) (2nd)*, Cordoba, Spain, 1 - 3 July 2009, pp. 259-268. https://eric.ed.gov/?id=ED539086

[8] D. S. Eisenberg, J. Stylos, A. Faulring, and B. A. Myers, "Using association metrics to help users navigate API documentation", in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, Leganes, Spain, 21 - 25 September 2010, pp. 23-30. https://doi.org/10.1109/VLHCC.2010.13

[9] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap", in *33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier. France, 3-7 September 2018, pp. 293-304. https://doi.org/10.1145/3238147.3238191

[10] J. Stylos and B. A. Myers, "How programmers use internet resources to aid programming". 2005, Available: http://www.cs.cmu.edu/~jsstylos/stylos-2005.pdf.

[11] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge", in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, 14-18 March 2016, Vol. 1, pp. 349-359. https://doi.org/10.1109/SANER.2016.80

[12] M. M. Rahman, C. K. Roy, and D. Lo, "Automatic query reformulation for code search using crowdsourced knowledge". *Empirical Software Engineering*, Vol. 24, 2019, pp. 1869-1924. https://doi.org/10.1007/s10664-018-9671-0

[13] M. M. Rahman and C. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics", in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, Spain, 23-29 September 2018, pp. 473-484. https://doi.org/10.1109/ICSME.2018.00057

[14] M. M. Rahman and C. Roy, "NLP2API: Query Reformulation for Code Search Using Crowdsourced Knowledge and Extra-Large Data Analytics", in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, Spain, 23-29 September 2018, pp. 714-714. https://doi.org/10.1109/icsme.2018.00086

[15] L. Cai, H. Wang, Q. Huang, X. Xia, Z. Xing, and D. Lo, "BIKER: a tool for Bi-information source based API method recommendation", in *2019 27th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26-30 August 2019, pp. 1075-1079. https://doi.org/10.1145/3338906.3341174

[16] R. F. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia, "Recommending comprehensive solutions for programming tasks by mining crowd knowledge", in *27th International Conference on Program Comprehension*, Montreal, Quebec Canada, 25 May 2019, pp. 358-368. https://doi.org/10.1109/ICPC.2019.00054

[17] R. F. G. da Silva *et al.*, "CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge". *Empirical Software Engineering*, Vol. 25, No. 6, 2020, pp. 4707-4758. https://doi.org/10.1007/s10664-020-09863-2

[18] W. K. Lee and M. T. Su, "Mining Stack Overflow for API class recommendation using DOC2VEC and LDA". *IET Software*, Vol. 15, No. 5, 2021, pp. 308-322. https://doi.org/10.1049/sfw2.12023

[19] Stack Exchange Inc. (2022). *Stack Overflow*. Available: https://stackoverflow.com/

[20] College Board. (2014). *AP Computer Science Java Subset*. Available: https://secure-media.collegeboard.org/digitalServices/pdf/ap/ap-computer-science-a-java-subset.pdf

[21] College Board. (2019). *Java Quick Reference*. Available: https://apstudents.collegeboard.org/ap/pdf/ap-computer-science-a-java-quick-reference_0.pdf

[22] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 2008.

[23] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit", in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Baltimore, Maryland, USA, 23-24 June 2014, pp. 55-60. https://aclanthology.org/P14-5010

[24] The Apache Software Foundation. (2011). *Welcome to Apache Lucene*. Available: https://lucene.apache.org/

[25] D. Turnbull. (2015). *BM25 The Next Generation of Lucene Relevance*. Available: https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/

[26] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log". *Empirical Software Engineering*, Vol. 17, No. 4-5, 2012, pp. 424-466. https://doi.org/10.1007/s10664-010-9144-6

[27] YourKit. (2017). *Extended Java WordNet Library (extJWNL)*. Available: http://extjwnl.sourceforge.net/

[28] Oracle Corporation. (2020). *Java SE Development Kit 13 Documentation*. Available: https://www.oracle.com/java/technologies/javase-jdk13-doc-downloads.html

[29] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. USA: Cambridge University Press, 2008.

[30] Stack Exchange Inc. (2021). *How do I ask a good question?* Available: https://stackoverflow.com/help/how-to-ask

[31] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning", in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, WA, USA, 13-18 November 2016, pp. 631-642. https://doi.org/10.1145/2950290.2950334

[32] Z. Gao, X. Xia, D. Lo, and J. Grundy, "Technical Q&A Site Answer Recommendation via Question Boosting". *ACM Transactions on Software Engineering and Methodology*, Vol. 30, No. 1, 2021, pp. 1-34. https://doi.org/10.1145/3412845

[33] Z. Gao, X. Xia, D. Lo, J. Grundy, and Y.-F. Li, "Code2Que: A Tool for Improving Question Titles from Mined Code Snippets in Stack Overflow", in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 23-28 August 2021, pp. 1525–1529. https://doi.org/10.1145/3468264.3473114

[34] F. Zhang *et al.*, "Improving Stack Overflow question title generation with copying enhanced CodeBERT model and bi-modal information". *Information and Software Technology*, Vol. 148, 2022, p. 106922. https://doi.org/10.1016/j.infsof.2022.106922