API Method Recommendation via Explicit Matching of Functionality Verb Phrases

Wenkai Xie*
Fudan University
China

Xin Peng*†
Fudan University
China

Mingwei Liu*
Fudan University
China

Christoph Treude
The University of Adelaide
Australia

Zhenchang Xing
Australian National University
Australia

Xiaoxin Zhang*
Fudan University
China

Wenyun Zhao*
Fudan University
China

ABSTRACT

Due to the lexical gap between functionality descriptions and user queries, documentation-based API retrieval often produces poor results. Verb phrases and their phrase patterns are essential in both describing API functionalities and interpreting user queries. Thus we hypothesize that API retrieval can be facilitated by explicitly recognizing and matching between the fine-grained structures of functionality descriptions and user queries. To verify this hypothesis, we conducted a large-scale empirical study on the functionality descriptions of 14,733 JDK and Android API methods. We identified 356 different functionality verbs from the descriptions, which were grouped into 87 functionality categories, and we extracted 523 phrase patterns from the verb phrases of the descriptions. Building on these findings, we propose an API method recommendation approach based on explicit matching of functionality verb phrases in functionality descriptions and user queries, called PreMA. Our evaluation shows that PreMA can accurately recognize the functionality categories (92.8%) and phrase patterns (90.4%) of functionality description sentences; and when used for API retrieval tasks, PreMA can help participants complete their tasks more accurately and with fewer retries compared to a baseline approach.

CCS CONCEPTS

• **Software and its engineering** → *Documentation*; *Software development techniques*; • **Information systems** → **Query representation**; **Document representation**.

*W. Xie, X. Peng, M. Liu, X. Zhang and W. Zhao are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, and the Shanghai Institute of Intelligent Electronics & Systems, China.
†X. Peng is the corresponding author (pengxin@fudan.edu.cn).

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409731

KEYWORDS

API Retrieval, API Documentation, Functionality Description

ACM Reference Format:
Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API Method Recommendation via Explicit Matching of Functionality Verb Phrases. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409731

1 INTRODUCTION

Finding the right APIs that provide the desired functionalities is essential in many software development tasks. Popular API libraries such as JDK and Android provide reference documentation which includes functionality descriptions for API classes and methods. However, due to the lexical gap between functionality descriptions provided by API developers and search queries by API users, documentation-based API retrieval often produces poor results.

Researchers [27, 44] have tried to use word embedding techniques to bridge the lexical gap by learning the statistical relevance between words, such as "convert" and "transform", "image" and "color", "JSON" and "XML". However, these methods do not explicitly parse the fine-grained structures of functionality descriptions and user queries, neither do they explicitly match the semantic roles of different parts of functionality descriptions and user queries. As such, these methods may lead to poor matching results when fine-grained linguistic details of functionality descriptions and user queries must be taken into account to produce satisfactory matching. For example, existing methods cannot distinguish "convert Integer to String" from "convert String to Integer", because they do not understand the *source* and *goal* roles [11] of the two descriptions.

To battle this issue, some methods [21] use more advanced deep learning models (e.g., Recurrent Neural Network) to learn sequential patterns of natural language descriptions and API call sequences. They map the query-API matching problem as a machine translation task. However, these methods are supervised learning methods which require large-scale training data (pairs of method comments

and API call sequences), and can capture only the most frequent queries and API usage patterns. Therefore, for the long tail of less frequently used APIs, developers still have to resort to other means, such as documentation-based retrieval.

The lexical gap between API functionality descriptions and user queries is much wider than just sequence mismatching. For example, the API *java.lang.Integer.parseInt(String)* is the correct API for the query "convert String to Integer". Unfortunately, the functionality description of this API is "Parses the string argument as a signed decimal integer.". None of the deep learning methods can handle this wide gap between the functionality description and the user query. To match functionality descriptions and user queries on either side of this gap, we must understand the fine-grained structures of the sentences and the semantic roles of their parts, for example, the key verb phrases and the semantic roles "{source}" and "{goal}" involved in the phrases "convert {source} to {goal}" and "parse {source} as {goal}". Some researchers have attempted to address this gap by demanding external resources (e.g., Stack Overflow discussions) to augment user queries, for example Biker [27] and QECK [35], but these can only capture the most discussed APIs in external resources and suffer from information noise in these external resources.

In this work, we consider that *verb phrases* and their *phrase patterns* are essential in both describing the functionality of API methods and interpreting user queries. We call the verbs used to describe the method functionalities *functionality verbs*. Consequently, we argue that user queries and API descriptions should be matched in terms of *functionality verbs* and *phrase patterns*, according to the semantics they express rather than their lexical similarity. Previous research has also considered the importance of verb-object structures in source code identifiers [22, 24–26, 28, 29], and for software documentation, Treude et al. [40] focused on extracting development tasks, which are described by verbs from a predefined list, associated with a direct object and/or a prepositional phrase. However, little is known about how verb phrases are used in API functionality descriptions and user queries, and how inconsistencies in their use influence API retrieval.

We hypothesize that most API functionalities can be described with a limited number of commonly used *functionality verbs* and that the functionalities can be further classified into a small number of *functionality categories*. For example, the verb "return", when used in the phrase "return {source} as {goal}", expresses the functionalities of transformation and conversion, and these functionalities can be classified into the same category. We further hypothesize that the *functionality categories* and *phrase patterns* of user queries and API descriptions can be automatically recognized. Finally, we argue that fine-grained matching can be performed between user queries and API descriptions by aligning their *functionality categories* and *phrase patterns*. For example, both the query "convert String to Integer" and the API description "parses the string argument as a signed decimal integer" can be classified into the same *functionality category* "convert/transform/turn" and the participants (i.e., "{source}" and "{goal}") of their phrase pattens can be aligned by explicit matching.

In order to verify our hypotheses, we conduct an empirical study to investigate the *functionality verbs* and *functionality categories* as well as the *phrase patterns* present in the API functionality descriptions from the reference documentation of JDK and Android. We manually analyzed the functionality descriptions of 14,733 JDK and Android API methods. These descriptions contain 356 different *functionality verbs*, and these verbs can be grouped into 87 *functionality categories* based on their semantics in the description context. Each *functionality category* contains 1 to 28 (4.71 on average and 2.5 on median) different *functionality verbs*. For all *functionality categories*, 80% of the API descriptions are described by 1-5 *phrase patterns* (mean 2.3, median 2).

Building on our empirical findings on *functionality verbs*, we propose an API method recommendation approach based on explicit matching of functionality verb phrases in user queries and API descriptions, which is called PreMA. The approach trains a classifier to predict the *functionality categories* (summarized in our empirical study) of API descriptions. It then matches the description sentences with the *phrase patterns* of the corresponding *functionality categories* to determine the adopted *phrase patterns*. To recommend API methods, PreMA parses the API query issued by the user in the same way. It then matches the parsed user query with the parsed API description sentence by aligning their *functionality categories* and *phrase patterns*. Finally PreMA returns completely or partially (e.g., the same *functionality categories* but different operation objects) matched APIs as the query results, together with a linguistic explanation of the matching.

We evaluated the sentence analysis accuracy of PreMA based on the data annotated in the empirical study. The results show that the accuracy of *functionality category* classification and *phrase pattern* recognition is high (92.8% and 90.4% respectively). We also evaluated the performance of PreMA in documentation-based API retrieval by comparing it with a Word2Vec [33] based approach. The results show that the participants using PreMA completed their tasks more accurately (0.77 versus 0.54) with fewer retries (2.16 versus 3.30) and using less time (98.29s versus 113.42s).

Overall, this paper makes the following contributions:

- We conducted a large-scale empirical study on the functionality descriptions of 14,733 JDK and Android API methods. We identified 356 commonly used *functionality verbs*, 87 *functionality categories*, and 523 *phrase patterns* from the descriptions.
- We propose an API method recommendation approach based on explicit matching of functionality verb phrases in user queries and API descriptions.
- We evaluated the proposed approach in terms of the accuracy of API functionality description analysis and the performance of API retrieval.

The data and analysis results of the empirical study and evaluation are included in the replication package [7].

2 DEFINITIONS

We define the main concepts and relationships used for the explicit modeling of API functionality descriptions as shown in Figure 1.

The reference documentation contains for each API method a description of its main purpose. We call this description the *API functionality description* (or *f_description* for short). The *f_description* may include several sentences that describe the functionalities of the method, which we call *functionality sentences* (or *f_sentences*). Each *f_sentence* may include several verbs or verb phrases, but

exactly one *functionality verb* (or *f_verb*), which denotes the main action of the functionality. For example, the *f_sentence* "Attempts to cancel the execution of this task" includes two verbs ("attempt" and "cancel"), while only "cancel" is the *f_verb*. For a compound sentence that describes multiple functionalities we can split it into several *f_sentences*, each describing only one functionality.

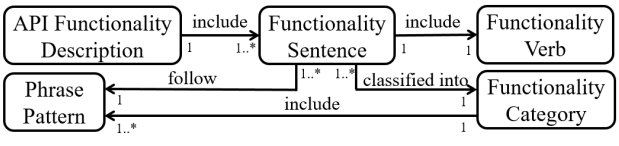

Figure 1: Terminology Model

A *f_sentence* can be classified into a *functionality category* (or *f_category*), based on the meaning of its *f_verb* in the current context. For example, the API *org.omg.CosNaming.NamingContextExtOperations.to_name(String)* contains this *f_sentence* in its *f_description*: "This operation converts a Stringfield Name into an equivalent array of Name Components." It includes the "convert" *f_verb* (underlined) and belongs to the "convert/transform/turn" *f_category*, defined as "transform something into other forms".

Two *f_sentences* may be classified into two distinct *f_categories*, even if they share the same *f_verb*. That is because the *f_verb* may have different meanings in different contexts. For example, "return" is widely used in *f_sentences* as *f_verbs* with different meanings, that are denoting different *f_categories*, such as: "get/return/obtain" (e.g., "Returns the state of this thread"), "check/test/determine" (e.g., "Return if the Type is a cube map"), or "convert/transform/parse" (e.g., "Returns the BigDecimal as a character array"). At the same time, one *f_category* can correspond to several distinct *f_verbs* that share meaning in different contexts. For example, in the JDK and Android reference documentation there are many *f_verbs* with the meaning captured by the "create/build/construct" *f_category*, such as: "create", "build", "produce", "construct", "generate", "establish", "make", "instantiate", etc.

The *f_sentences* from a *f_category* share a set of *phrase patterns* (or *p_patterns*) and each sentence conforms to one of them. A *p_pattern* consists of the following elements: 1) a *f_category*, e.g., "convert/transform/turn"; 2) prepositions, e.g., "in", "at", "from"; 3) semantic roles which depict conceptual relations among participants in the *p_pattern*, e.g., "location", "patient", "source", "goal"; 4) clause leaders, e.g., "that", "whether"; 5) clauses, e.g., a clause led by "whether"; 6) infinitives (e.g., "to be called") and gerunds (e.g., "reading a file"). For example, the *f_sentence* "This operation converts a Stringfield Name into an equivalent array of Name Components." follows the "V {source} to/as/into {goal}" *p_pattern*, where "V" indicates the *f_category*, while "{source}" and "{goal}" denote semantic roles. Note that semantic roles are consistent despite the alternations in the syntax [11], thus it is possible to align the semantic roles between different *p_patterns*.

VerbNet [10] is a domain-independent and broad-coverage verb lexicon for English. It contains about 5,800 verbs. They are classified into over 270 classes and each class contains a set of syntactic frames that the members of the class commonly use. We did not use the VerbNet classes and frames as our *f_categories* and *p_patterns* for the following two reasons. First, some verbs used in the API *f_sentences* (e.g., "unmarshal", "iterate") are not included in VerbNet. Second, the VerbNet classes are not defined by the meanings of verbs. For example, "open" and "close" are both in the same class "crane" in VerbNet, but are classified into different *f_categories*. We decided to annotate our own *f_categories* and *p_patterns*, but use the 12 semantic roles defined in VerbNet: "duration", "location", "goal", "source", "patient", "instrument", "beneficiary", "attribute", "theme", "material", "topic", "product".

3 EMPIRICAL STUDY

We conducted an empirical study to understand what *f_verbs*, *p_patterns*, and *f_categories* are present in API *f_sentences*. Specifically, we focus on answering the following research questions:

- RQ1. What verbs are used in the API *f_sentences*?
- RQ2. What *f_categories* can the *f_sentences* be classified into?
- RQ3. What *p_patterns* are used in the *f_sentences* from each *f_category*?

We focus on JDK and Android APIs based on the reference documentation of JDK 1.8 [6] and Android 27 [1]. We present and analyze the results of the empirical study, in order to answer the three research questions. Complete data and analysis results corresponding to the empirical study are included in the replication package [7], including complete *f_sentences*, *f_verbs*, *f_categories* and *p_patterns*.

3.1 Study Design

3.1.1 Verb Analysis (RQ1). From the semi-structured API declarations in the JDK and Android reference documentations, we extracted 38,819 and 29,125 API methods (including constructors) respectively using BeautifulSoup [4]. We filtered out the methods that have no *f_description* or with a *f_description* that meets one of the following conditions: (1) stating a method overriding, e.g., "Overrides hashCode"; (2) stating a method deprecation, e.g., "Deprecated. Use getTimeToLive instead"; (3) suggesting to check the description of another API, e.g., "See getenv()"; (4) suggesting another API with the same functionality, e.g., "Same as charCount(int)". After the filtering, we obtained 54,256 methods (31,618 JDK and 22,638 Android). For each remaining method we extract the first sentence of its description from the reference documentation as its *f_sentence*. If the sentence is a compound sentence, we split it into multiple *f_sentences*.

We used SpaCy [8] (an open-source library for natural language processing) to parse the *f_sentence* of each API method and identify the verbs used in the sentence. If a *f_sentence* includes more than one verb, we considered all of them for answering the first research question. We used SpaCy to lemmatize the identified verbs to their normal forms. Once extracted, we analyze the frequency and distribution of the verbs.

3.1.2 Functionality Category Analysis (RQ2). The identification of the *f_categories* and *f_verbs* from JDK and Android was done via the qualitative analysis of the *f_sentences*, using open coding. The coding was done in two phases: coding protocol definition phase and annotation phase. In the first phase experts identified and defined a set of *f_categories* (i.e., codes) based on a subset of API

descriptions. In the annotation phase, a larger group of annotators were trained to use the coding protocol developed in the first phase to code a larger set of *f_sentences*.

Selecting random *f_sentences* for annotation would likely result in a large number of sentences with "get/set" verbs, given their prevalence. Instead, for the first coding phase, we randomly sampled 10 *f_sentences* for each of the 116 most frequently used verbs at the root of the parse tree (i.e., the verbs of the main clause). We focus on these verbs, as they are more likely to be *f_verbs* (remember that RQ1 considered all verbs in the *f_sentences*). This number of verbs (i.e., 116) is selected based on the results of RQ1 (see Section 3.2). These verbs cover 84% of the *f_sentences* used in RQ1 - all of the 116 most frequent verbs identified in RQ1 are root-node verbs. After eliminating duplicates we obtained 1,139 *f_sentences* (590 from JDK and 549 from Android), used for the initial coding phase.

For the annotation phase we randomly sampled another 20,000 API *f_sentences* regardless of the frequency of the verbs. To have a more balanced dataset (i.e., not too many "get/set" sentences) we further refined the samples, based on their root-node verb. If a root-node verb appears in more than 1,000 of the sampled *f_sentences*, then we randomly selected and kept 1,000 of those *f_sentences*. We kept all the *f_sentences* for the root-node verbs that occur in fewer than 1,000 samples. In this way, *f_sentences* using unpopular verbs may also be included. Finally, we obtained 13,635 *f_sentences* for the annotation phase (7,716 from JDK and 5,919 from Android).

The initial coding was done by three of the authors, who are experts in Java and Android development, as follows. First, an API *f_sentence* is randomly allocated to an annotator. Second, the annotator examines the API and its *f_sentence* and identifies the *f_verb* used in it. Third, the annotator attempts to classify (i.e., annotate) the API *f_sentence* into an existing *f_category* (i.e., existing code). If no code is found, then a new code (i.e., *f_category*) is created and a definition provided. A special code "Unknown" is created to accommodate the sentences that were not actually *f_sentences* (e.g., "Equivalent to the codePointCount(char[], int, int) method, for convenience"). Each *f_sentence* is coded by two annotators independently and if their annotations (*f_verb* or *f_category*) are different, then a third annotator is assigned to resolve the conflict. The above process was repeated until all the samples were annotated.

The annotation was done by 10 students (2 PhD and 8 MS students), who are familiar with Java and Android development. Before annotation, the students were trained by the experts who defined the coding instrument. The training was conducted in group and took more than one hour. More material about training, including examples, is available in the replication package [7]. The annotation was done using the codes (i.e., *f_categories*) identified in the coding protocol definition phase and followed the same process. As in the first phase, the annotators could create new codes, when needed. The code for each *f_category* is one of the *f_verbs* (named the *label f_verb*), which is frequently used (but not necessarily the most frequently used), best reflects the meaning of the *f_category*, and is not used to label another *f_category*.

To facilitate the coding, we developed a web based annotation tool. The GUI of the annotation tool is available on the web [2].

3.1.3 Phrase Pattern Analysis (RQ3). As mentioned above, in order to capture the functionality of a method, the *f_verb* is not enough, and *p_patterns* are important to establish the context. We investigate the *p_patterns* used in each *f_category*, identified in RQ2, by annotating all the 14,774 (initial coding phase 1,139 and annotation phase 13,635) *f_sentences*. For each *f_sentence*, two authors annotated the *p_pattern* independently based on the VerbNet annotation guidelines [11]. If the annotations are different a third author was assigned to resolve the conflict by majority voting strategy.

3.2 Verb Analysis Results (RQ1)

We identify 931 different verbs from the *f_sentences* of the 54,256 JDK and Android API methods. Table 1 shows the top 30 most frequently used verbs and their occurrences. "Return" is by far the most used verb, followed by "set" and "get". These are not surprising, based on our experience with the code and documentation. An interesting observation is that these top 30 verbs are not domain specific, which implies that we expect them to occur in other libraries, from different domains, as well. We inspected all the verbs and we found that most of them are not domain specific (to Android or Java). For example, "dial" is a domain specific verb to Android. Followings are some of the identified domain specific verbs and their frequency: ("mute", 3), ("denigrate", 7), ("suffix", 7), ("prefix", 5), ("negotiate", 3), ("dial", 3), ("snooze", 1), ("absorb", 1), ("advertise", 7), ("roam", 6), ("introspect", 5).

Table 1: Top 30 Most Frequently Used Verbs

Verb	#Occu	Verb	#Occu	Verb	#Occu
return	16,268	indicate	730	do	478
set	5,406	determine	691	retrieve	458
get	3,624	write	676	give	446
call	2,472	change	602	contain	436
have	2,034	obtain	599	support	403
create	1,898	read	558	specify	400
use	1,837	check	524	convert	396
add	1,259	insert	520	update	390
remove	1,067	perform	518	describe	387
invoke	877	start	508	notify	383

Figure 2 shows the distribution of the 931 verbs (Y-axis in log scale). The distribution analysis reveals that the 87 (9.34%) most frequent verbs appear in 80% of the API *f_sentences*. If we exclude "return" as outlier, then the 115 (12.37%) most frequent verbs appear in 80% of the API *f_sentences* that do not include "return". If we include "return", then the 116 (12.46%) most frequent verbs appear in 84.28% of the API *f_sentences*. In other words, a relatively small number of verbs covers almost all API *f_sentences*.

3.3 Functionality Categories (RQ2)

Given the number of annotators and codes, we used Cohen's Kappa coefficient [32] to measure the agreement rate between the annotators. For the initial coding phase Kappa is 0.724 and for the second annotation phase it is 0.700. The annotators identified 87 *f_categories* (not including "Unknown"), of which 50 were identified in first coding phase and 37 in the second one. Among the 87 *f_categories*, 65 cover both JDK APIs and Android APIs, 10 cover only JDK APIs, and 12 cover only Android APIs.

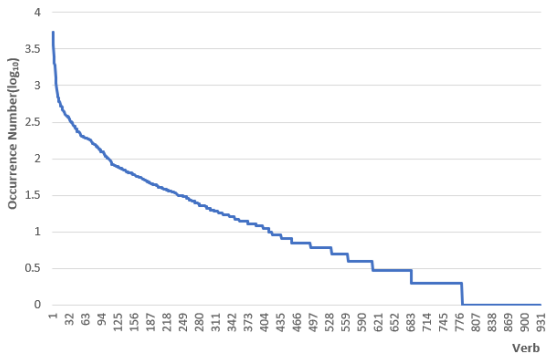

Figure 2: Distribution of the Verb Occurrences

The 87 *f_categories* contain 356 *f_verbs*, eight of which appear in more than two *f_categories*. "Return" appears in seven *f_categories*, "determine" appears in six *f_categories*, "indicate" appears in five, "tell", "retrieve", and "give" appear in four, while "get" and "notify" appear in three. There are 26 *f_verbs* that appear in two *f_categories*. All other 322 appear in a single *f_category*. On average, a *f_category* contains 4.71 *f_verbs* (median 2.5). The "convert/transform/..." *f_category* contains the most *f_verbs* (i.e., 28), while 30 (34.48%) *f_categories* contain a single *f_verb*. Note that unpopular verbs may be included in a *f_category* together with popular verbs.

We compared the 356 *f_verbs* with the list of 202 programming actions published by Treude et al. [40]. Their programming tasks have similar semantics to our *f_verbs*, but are based on a much smaller data set. We find that 121 of our 356 *f_verbs* (34.0%) were also identified by them as programming actions, while our empirical study identified an additional 235 *f_verbs*.

We define the label *f_verb* as a representative *f_verb* of one *f_category*. For each *f_category*, three of the co-authors chose the one *f_verb* from all *f_verbs* in this *f_category* as the label *f_verb* of this *f_category* through discussion. Two heuristics were used for choosing the label *f_verb*: (1) The co-authors check *f_verbs* in the *f_category* by frequency from high to low until the label *f_verb* is determined. (2) A *f_verb* is only considered as label *f_verb* if its meaning covers the meaning of the *f_category* and there is no confusion with another *f_category*.

The top 10 *f_categories* based on the number of *f_sentences* and their label *f_verbs* are shown in Table 2.

3.4 Phrase Patterns (RQ3)

Two annotators were considered to reach an agreement if their *p_pattern* annotations for a *f_sentence* are the same. As a result, the agreement rate for *p_pattern* annotation is 90.2% (i.e., almost perfect agreement). The *p_patterns* identified from the *f_sentences* of the same *f_category* are aggregated to get the *p_patterns* of each *f_category*. Note that *p_patterns* that belong to the same *f_category* and only differ in prepositions or clause leaders were merged into one, for example "V {source} to/as/into {goal}" for the "convert/transform/parse" *f_category*.

Table 2: Top 10 Most Frequently Used *f_categories*

f_categories	Label *f_verb*	#*f_sentences*
get/return/obtain/...	get	3021
set/control/configure/...	set	1303
check/test/determine/...	check	977
create/build/construct/...	create	784
append/add/insert/...	append	777
call/invoke/notify/...	call	762
perform/execute/run/...	perform	409
convert/transform/parse/...	convert	393
remove/delete/exclude/...	remove	348
write/record/output /..	write	293

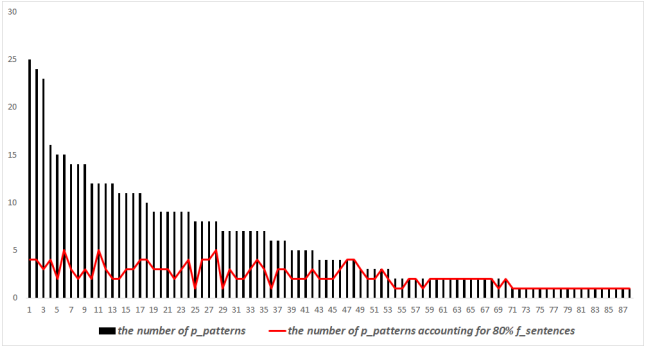

Figure 3: Distribution of *p_pattern* over *f_categories*

Figure 3 shows the distribution of *p_pattern* numbers over *f_categories*. The number of *p_patterns* for each of the 87 *f_categories* varies between 1 and 25 (mean 6, median 4). The top 3 *f_categories* that have the most *p_patterns* are "set/control/configure" (25), "append/put/add" (24), and "get/return/obtain" (23). There are 18 *f_categories* that have only one *p_pattern*, e.g., "lock", "touch/press", "collect/recycle/sample".

For each *f_category* we analyzed the number of *p_patterns* that cover 80% of the *f_sentences*, indicated by the red line in Figure 3. We found that, for all the *f_categories* 80% of the *f_sentences* are described by 1-5 *p_patterns* (mean 2.3, median 2). For example, the "append/add/insert" *f_category* has 24 *p_patterns*, while 4 of them cover 80% of the *f_sentences*.

The identified *p_patterns* have 0-4 semantic roles. The numbers of *p_patterns* that have 0, 1, 2, 3, 4 semantic roles are 28 (5.4%), 166 (31.7%), 216 (41.3%), 110 (21.0%) and 3 (0.6%), respectively. We can see that 73% of *p_patterns* are simple ones with 1 or 2 semantic roles. An example of *p_patterns* with 3 semantic roles is "V {patient} from {source} as/into/to {goal}" for the "convert/transform/parse" *f_category*; a *f_sentence* following this *p_pattern* is "Convert a long datetime from the given time scale to the universal time scale.".

4 APPROACH

Our empirical study shows that most of the JDK and Android API functionality sentences can be classified into a limited set of 87 *f_categories* with 1-5 *p_patterns* (2.33 on average) used for each *f_category*. These findings imply how verb analysis can be used

for matching between an API query and a *f_sentence*: first recognize their *f_categories* and *p_patterns*; then align them based on *f_categories* and *p_patterns* for fine-grained matching between corresponding participants.

Based on this idea we propose an approach PreMA for matching of API functionality descriptions as shown in Figure 4. Given *f_sentences* from API reference documentation, the approach parses the sentences by analyzing their *f_categories* and *p_patterns*. The parsed *f_sentences* are then stored for further analysis. When used for API searching, PreMA parses the API query issued by the developer in a similar way. It then matches the parsed API query with the parsed *f_sentences* by aligning them based on *f_categories* and *p_patterns*. The API matching results include completely or partially (e.g., the same *f_categories* but different participants) matched APIs and their *f_sentences*, together with explanations of the matching.

Our implementation uses BeautifulSoup to parse the HTML pages of API reference documentation. It extracts all API methods with their *f_description* and filters out invalid API methods using rules (same as in Section 3.1.1). For each remaining method, we extract the first sentence of its description from the reference documentation as its *f_sentence* for functionality sentence parsing.

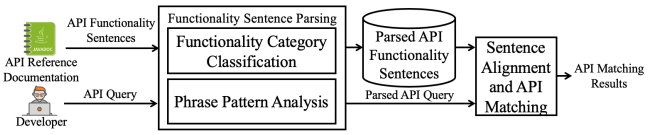

Figure 4: Overview of PreMA

4.1 Functionality Category Classification

The 87 *f_categories* and 14,774 annotated *f_sentences* provided by our empirical study enable automated classification of *f_sentences* into *f_categories*. We treat *f_category* classification as a text classification task and use the *f_sentence* annotation data to train a classifier for the task. The classifier takes a sentence (*f_sentence* or query) as input and returns one of the 87 *f_categories* or the category "Unknown" as the output.

We implement the classifier based on BERT [18] (Bidirectional Encoder Representations from Transformers), a state-of-the-art language model. The model is used for learning representations of sentences: it takes as input a sequence of words, and outputs the distributed vector representation of the word sequence [41]. Google provides two pre-trained BERT models (BERT-base, BERT-large), which were trained on a large-scale unlabelled corpus to capture rich semantic features. The pre-trained models can be customized by adding an output layer and fine-tuned based on labelled data for specific NLP (Natural Language Processing) tasks such as text classification and question answering. We use the pre-trained BERT-base model and add a classification layer (fully-connected layer) with the *f_categories* identified in our empirical study, and then fine-tune the model based on a set of training data consisting of *f_sentence-f_category* pairs.

4.2 Phrase Pattern Analysis

Given a sentence (*f_sentence* or query) and its *f_category*, phrase pattern analysis determines the *p_pattern* used in it. As our empirical study has identified a set of *p_patterns* for each *f_category*, the analysis only needs to match the sentence with the *p_patterns* of the *f_category* that it belongs to. We use Spacy [8], which performs well on Java API documentation [13], to do POS (Part of Speech) tagging and dependency parsing of the sentence. After that, we identify the *f_verb* used in the sentence and then the *p_pattern*.

4.2.1 Functionality Verb Identification. The functionality category classification does not identify the *f_verb* used in the sentence, so we need to identify the *f_verb* based on POS tagging and dependency parsing. Our empirical study identified a set of *f_verbs* for each *f_category* and multiple of them may appear in the sentence. Thus functionality verb identification just needs to choose from the *f_verbs* of the *f_category* that the given sentence belongs to. We use a heuristic-based approach to choose from the candidate verbs. Given a sentence we traverse its dependency tree in preorder. The first candidate verb that is traversed is considered as the *f_verb* of the sentence. If no candidate verb is found after traversing the entire dependency tree, the first verb traversed in the sentence is considered as the *f_verb*, indicating a new *f_verb* for the *f_category* that was not identified in the empirical study. Figure 5 shows a dependency tree used as an example of functionality verb identification. The sentence belongs to the *f_category* "get/return/obtain". The arrows from a token indicate the syntactic children that appear before and after the token and the labels on the arrows indicate the dependency types. For example, "dobj", "pobj" and "xcomp" refer to direct object, object of the preposition, and open clausal complement respectively. This sentence has two verbs (i.e., "use" and "get"). When traversing the dependency tree in preorder, "use" is the first traversed verb as it is the root node, but it is not in the *f_verb* set of the *f_category* "get/return/obtain"; "get" is the second traversed verb and it is in the *f_verb* set of the *f_category*, so we choose it as the *f_verb* of the sentence.

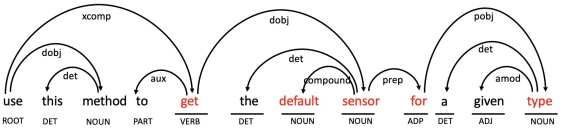

Figure 5: An Example of Functionality Verb Identification

4.2.2 Phrase Pattern Identification. To identify the *p_pattern* we need to match the given sentence with the *p_patterns* of the *f_category* that the sentence belongs to.

To do so we need to first identify the core clause of the given sentence that describes the functionality of the API method. This can be done by finding the subtree of the dependency tree rooted at the *f_verb*. For example, for the *f_sentence* shown in Figure 5, the "get" clause (underlined) is the core clause that describes the functionality. Then from the core clause, we extract the syntactic pattern *SP* by analyzing the dependency tree of the core clause. We replace the words in the core clause with a placeholder for

syntactic components using the following rules: 1) replace verb with "V"; 2) replace nouns and noun phrases with "NP"; 3) replace the object clause and adverbial clause with "S"; 4) replace gerunds with "S_ING"; and 5) replace infinitive with "S_INF". Clause leaders (e.g., "that", "whether") and prepositions (e.g., "in", "at", "to") in the core clause are retained. The replacement is done by recursively visiting the subtree of the core clause. For example, the syntactic pattern identified for "This method will start profiling if isProfiling() returns true." is "V S_ING if S" (Rule 1, 3 and 4); and the syntactic pattern identified for "Registers the parameter named parameterName to be of JDBC type sqlType." is "V NP S_INF" (Rule 1, 2 and 5).

After obtaining the syntactic pattern *SP* of a sentence, we find the most similar *p_pattern* among candidate *p_patterns* which are all *p_patterns* of the *f_category*. We split *SP* and *p_patterns* into small components for this comparison. Prepositions with subsequent "NP" or semantic roles are considered as a component. For example, "V {patient} for {beneficiary}" can be split into three components "V", "{patient}" and "for {beneficiary}". We compare each *p_pattern* with *SP* by the order of components to get the matching number of components in the *SP*. "NP" could match with any semantic role in *p_pattern*. If not all components in *p_pattern* could be matched in *SP*, we remove this *p_pattern* from the candidate *p_patterns*. Finally, we choose the *p_pattern* from the candidate *p_patterns* with the highest number of matching components as the *p_pattern* of the sentence. For example, for the sentence in Figure 5, after matching "V NP for NP" with all *p_patterns* in the *f_category* "get/return/obtain', we obtain two candidates "V {patient}" and "V {patient} for {beneficiary}" and the matched number of components is 2 and 3, respectively. Thus, we choose "V {patient} for {beneficiary}" as the *p_pattern*.

4.3 Sentence Alignment and API Matching

Given a parsed API query Q we match it with each parsed *f_sentence* FS and calculate their matching score by aligning them based on *f_categories* and *p_patterns*. The matching score is calculated with Equation 1 by combining three different similarities: *f_category* similarity, semantic role similarity, and text similarity. Then we rank API methods by matching score and generate the explanation for each method.

$$Score(Q, FS) = Sim_C(Q, FS) + Sim_R(Q, FS) + Sim_T(Q, FS) \quad (1)$$

4.3.1 Functionality Category Similarity Calculation. The *f_category* similarity $Sim_C(Q, FS)$ measures whether the *f_categories* of Q and FS are the same. If Q and FS are classified into the same *f_category*, $SimC(Q, FS) = 1$; otherwise, $SimC(Q, FS) = 0$.

4.3.2 Semantic Role Similarity Calculation. The semantic role similarity $Sim_R(Q, FS)$ measures the similarity between corresponding semantic roles of Q and FS using entity based matching. If Q and FS are classified into different *f_categories*, $SimR(Q, FS) = 0$. Otherwise, we calculate $Sim_R(Q, FS)$ in three steps, i.e., entity alignment, entity linking, and entity matching.

First, we align the corresponding entities between Q and FS based on semantic roles. An entity is a noun phrase in Q or FS. An entity E_Q in Q and an entity E_{FS} in FS can be aligned if and only if they play the same semantic role in Q and FS. Figure 6 shows an example of entity alignment. In this example, "the string argument"

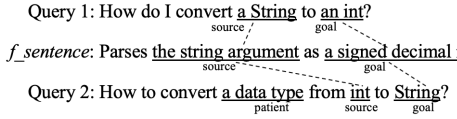

Figure 6: An Example of Entity Alignment

in the *f_sentence* is aligned with "a String" in Query 1 and "int" in Query 2; "a signed decimal integer" in the *f_sentence* is aligned with "an int" in Query 1 and "String" in Query 2. In this way it is easy to determine that Query 1 matches better with the *f_sentence* than Query 2. If an entity E_Q in Q has two corresponding entities E_{FS1} and E_{FS2} in FS, E_Q is aligned with both E_{FS1} and E_{FS2}, and vice versa.

Second, we link entities in Q and FS to the corresponding entities in a general knowledge graph (Wikidata [42] in the current implementation). Based on the linking we can use the knowledge in the general knowledge graph to calculate the similarity between the entities in Q and FS. For example, Wikidata provides knowledge like "string is a sequence of characters and a data type" and "str is an alias of string". To consider the linking between an entity E_S in Q or FS and an entity E_W in the general knowledge graph, we first do preprocessing (tokenization, stop word removal, and lemmatization) on the noun phrase of E_S and then calculate the following two similarities between E_S and E_W: 1) morphological similarity that can be measured based on the minimum edit distance between the preprocessed noun phrase of E_S and any alias of E_W: 2) context similarity that can be measured by the text similarity between the sentence that E_S appears in (i.e., Q or FS) and the definition sentences of E_W provided by the general knowledge graph. Finally E_S is linked to an entity in the general knowledge graph that has the highest combined similarity with E_S.

Third, we match between the aligned entities based on entity linking results. For an entity E_Q in Q and an aligned entity E_{FS} in FS, we calculate their matching score in the following way: 1) if E_Q and E_{FS} are linked to the same entity in the general knowledge graph (e.g., "string" and "char sequence"), they are equal and their matching score is 1; 2) if E_Q and E_{FS} are linked to two entities with hyponymy (e.g., "instance of ", "subclass of", or "part of") relationship in the general knowledge graph (e.g., "int" and "primary type"), their matching score is 1; 3) if one of E_Q and E_{FS} is the prefix or suffix of the other one (e.g., "Integer value" and "Integer"), they are in a hyponymy relationship and their matching score is 1; 4) otherwise, the matching score is 0.

Finally, $Sim_R(Q, FS)$ is calculated by summing the matching scores of all the aligned entity pairs between Q and FS. Note that if an entity E in Q (or FS) has several aligned entities in FS (or Q) we only consider the entity pair that has the highest matching score.

4.3.3 Text Similarity Calculation. The text similarity $Sim_T(Q, FS)$ measures the overall text similarity between Q and FS to cover other sentence parts (e.g., clauses, gerunds, and infinitive). The similarity is calculated based on the Word2Vec [33] model. We use a 100-dimensional Word2Vec model pre-trained on the Wikipedia corpus [12] and tune the model based on the corpus of all *f_sentences* extracted in our empirical study using gensim [5]. Then we calculate the similarity in the following way: 1) preprocess Q and FS by

tokenization, stop word removal, and lemmatization; 2) generate a vector for Q and FS respectively by averaging the vectors of all their words produced by the Word2Vec model; 3) calculate the normalized cosine similarity between the vectors of Q and FS.

4.3.4 Matching Result Generation. Given a query we rank the *f_sentences* by their matching scores from high to low. For each *f_sentence*, we generate a linguistic explanation for matching by describing: 1) the *f_category* that the query and the *f_sentence* belong to, 2) the semantic roles in the query and the *f_sentence*, and 3) all matched entity pairs and their matching degrees. For example, one matching result for the query "How do you crash a JVM?" [9] would be "Terminates the currently running Java Virtual Machine" of *java.lang.System.exit(int)*. We can explain this matching as follows: both belong to the *f_category* "stop/quit/terminate"; "JVM" matches with "Java Virtual Machine" at an "Equal" level, and they share the semantic role "patient".

5 EVALUATION

Our evaluation includes two parts. In the first part, we evaluate the accuracy of *f_sentence* parsing, including *f_category* classification and *p_pattern* analysis. In the second part, we evaluate the performance of PreMA in documentation-based API retrieval tasks by comparing it with a deep learning based approach.

5.1 Accuracy of Functionality Sentence Parsing

To evaluate the accuracy of the *f_category* classification, we used the 14,774 *f_sentences* annotated in the empirical study to do a 5-fold cross validation. The average accuracy on the test set is 92.8% (with 92.6% minimum accuracy). Our analysis shows that most of the misclassified *f_sentences* use rare verbs such as "pin" and "compile", which have very few samples in the annotated *f_sentences*.

To evaluate the accuracy of the *p_pattern* analysis, we removed those belonging to the "Unknown" *f_category* from the 14,774 *f_sentences* and used the remaining 11,074 *f_sentences* and their annotated *p_patterns* as the data set. For each *f_sentence*, we used our approach to identify the *p_pattern* and compared it with the annotation. The results show that the accuracy of the *p_pattern* analysis is 90.4%. Our analysis shows that most of the mistakes were caused by the POS tagging and dependency parsing implemented by Spacy. For example, gerunds are sometimes recognized as noun phrases, e.g., "Starts looping playback from the current position", and "to" in an infinitives is sometimes recognized as preposition, e.g., "Marshals to output the value in the Holder".

The above evaluation is based on the *f_sentences* extracted from the JDK and Android reference documentation. To confirm whether the classifier and analyzer can be applied to other libraries, we further evaluate the accuracy of *f_sentence* parsing on Apache POI (a Java library for processing Microsoft Office documents) [3]. We randomly selected 100 *f_sentences* from the POI reference documentation and invited three MS students to annotate their *f_categories* and *p_patterns* in a similar way to the empirical study. The annotation process did not produce new *f_categories*. We used all the 14,774 *f_sentences* annotated in the empirical study to train a *f_category* classifier and used it to classify the 100 POI sentences. The *f_category* classification accuracy on POI sentences is 97%. We further use PreMA to analyze the *p_patterns* of the 100 POI sentences and the accuracy is 95%. The results show that the *f_category* classifier and the *p_pattern* analyzer trained on JDK and Android reference documentation also work well for POI.

Table 3 shows results of functionality sentence parsing produced by PreMA, where the bold italic words and subscripts in *f_sentences* denote the skeletons and semantic roles (clauses) of *p_patterns*. We can see that *f_sentences* with the same *f_verbs* (e.g., "return") can be classified into different *f_categories*. The *p_patterns* of the same *f_categories* may have different numbers of semantic roles, for example the *f_category* "convert/transform/turn" has both *p_patterns* with 2 and 3 semantic roles. Based on the recognized *p_patterns* the participants of different *f_sentences* of the same *f_categories* can be aligned based on semantic roles (e.g., source, goal) and clauses.

5.2 Performance of API Method Retrieval

We implemented a deep learning based approach as the baseline tool for API method retrieval. The tool uses Word2Vec [33] to produce vector representation of words and sentences. It matches a user query with an API description without explicit analysis of functionality verb phrases. It uses a 100-dimensional Word2Vec model pre-trained on the Wikipedia corpus [12] and tunes the model based on the corpus of all *f_sentences* using gensim [5]. It generates a vector for the user query by averaging the vectors of all its words after preprocessing (i.e., tokenization, stop word removal, and lemmatization) and a vector for the API description in a similar way. Finally it calculates the cosine similarity between the vector of the query and the vector of the description of each API method, and ranks the candidate API methods by the similarity. We tried and tested two strategies for API matching, i.e., using the full description of each API method or only the first sentence of its description, based on a manually constructed dataset of user query and API method pairs. The results show that the implementation using the first sentence of each method description has better performance, thus we chose it as the baseline. We did not consider BERT-based approach as baseline, as we need to train a binary classifier for query-document relevance based on fine-tuned BERT model and additional training data of relevant/irrelevant query-document pairs.

We selected API retrieval tasks from Stack Overflow questions that are tagged with "Java" or "Android" based on the following criteria: the questions ask for APIs implementing specific functionalities and have at least one accepted answer that recommends a single JDK or Android API method. We ranked the questions meeting the above criteria by their votes and randomly selected 30 questions from the highly ranked ones. For each selected question we generated an API retrieval task that uses the question title and body as the task description.

We invited two PhD and ten MS students who are familiar with Java and Android development to complete the 30 tasks. For each task they can formulate and try different queries based on their understanding of the task description. Both PreMA and the baseline approach return the top-10 API methods together with their class descriptions as the context for the user to select. The participants finish a task when they find an API method that matches the task description. They were divided into two groups (G_1 and G_2) based

Table 3: Examples of Functionality Sentence Parsing

API Method	*f_sentence*	*f_category*	Library
javax.xml.transform.Transformer.transform()	***Transform*** {the XML Source}$_s$ ***to*** {a Result}$_g$.	convert/transform/turn	JDK
android.graphics.Color.HSVToColor()	***Convert*** {HSV components}$_s$ ***to*** {an ARGB color}$_g$.	convert/transform/turn	Android
org.apache.poi.hpsf.PropertySet.toInputStream()	***Returns*** {the contents of this property set stream}$_s$ **as** {an input stream}$_g$.	convert/transform/turn	POI
android.icu.util.UniversalTimeScale.from()	***Convert*** {a long datetime}$_p$ ***from*** {the given time scale}$_s$ ***to*** {the universal time scale}$_g$.	convert/transform/turn	Android
java.text.CharacterIterator.getBeginIndex()	***Returns*** {the start index of the text}$_p$.	get/return/obtain	JDK
java.sql.Statement.getResultSetType()	***Retrieves*** {the result set type}$_p$ ***for*** {ResultSet objects generated by this Statement object}$_b$.	get/return/obtain	JDK
org.apache.poi.ss.formula.eval.AreaEval.containsColumn()	***Returns*** {true}$_p$ ***if*** {the specified col is in range}$_c$.	check/test/determine	POI
org.apache.poi.xssf.usermodel.XSSFWorkbook.isSheetHidden()	***Check whether*** {a sheet is very hidden}$_c$.	check/test/determine	POI

Participants of *p_patterns*: s (source), g (goal), p (patient), b (beneficiary), c (clause).

on a pre-experiment survey on their programming experience, balancing the experience in both groups. The 30 tasks were randomly divided into two groups (T_1 and T_2). The experiment was conducted in two phases. In the first phase, the participants in G_1 and G_2 were asked to complete the tasks in T_1 with PreMA and the baseline tool respectively. In the second phase, the two groups exchanged the tools to complete the tasks in T_2. All participants were required to run a full-screen recorder to record their API retrieval processes.

For each task we recorded the correctness, number of retries, and completion time of each participant and calculated the accuracy (i.e., the ratio of participants who selected the right API for the task) and average number of retries and completion time of the two participant groups. Figure 7 shows the performance of the groups using PreMA and the baseline tool over the 30 tasks. Participants using PreMA completed the tasks more accurately, required fewer retries, and used less time than those using the baseline tool. On average, the accuracy, number of retires, and completion time (by seconds) of the participants using PreMA and the baseline tool are 0.77, 2.16, 98.29 versus 0.54, 3.30, 113.42 respectively. We used Welch's T-test for verifying the statistical significance of the differences and the p-values for accuracy, number of retires, and time are 0.0032, 0.0064, and 0.2832 respectively. We can see the differences in accuracy and number of retires are statistically significant ($p << 0.05$), while the difference in time is not significant.

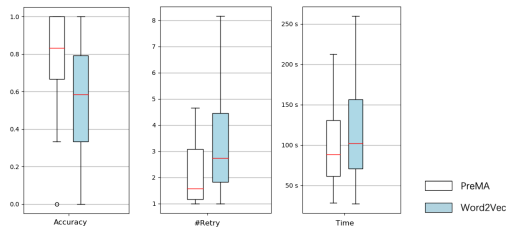

Figure 7: Performance of API Method Retrieval

The analysis of the screen recordings revealed that our tool performs particularly well for tasks that are order-sensitive. For example, for the task "converting array to list in Java", participants were able to find correct APIs quickly with our tool while the results returned by the baseline tool included APIs that turn lists into arrays (i.e., the reverse order). For tasks that involve concepts related to the software domain, our approach can also recommend better results. For example, for the task "how do you crash a jvm", our tool can correctly recommend the APIs *java.lang.System.exit(int)* and *java.lang.Runtime.halt(int)* while the results returned by the baseline tool are not related to "jvm". We also found *f_categories* to be helpful for solving tasks. For example, for the task "check whether a string is not null and not empty", the *f_sentences* recommended by our tool are in the same *f_category* "check/test/determine" as the question. However, *f_sentences* recommended by the baseline tool tend to contain the keyword "null", and the verb used in the question is ignored.

One concern is that our approach may not be able to search for APIs that use unpopular verbs. Some unpopular verbs express domain specific meanings (e.g., "mute", "roam", "dial") and can be easily discriminated by themselves. Some other unpopular verbs can be categorized together with more popular verbs and thus can also be supported. Among the 30 tasks, there are 3 tasks whose corresponding APIs (i.e., *java.util.Collections.sort()*, *java.lang.System.exit()*, *java.lang.String.split()*) use unpopular verbs (i.e., "sort", "terminate", "split"). These verbs have been included in the identified *f_categories*, for example "terminate" is included in a *f_category* with popular verbs "stop" and "end".

From interviewing participants after the experiment, we learned that they perceived our tool to provide them with correct and relevant answers more easily while the baseline tool required more queries. A participant said that our tool could understand queries better than the baseline tool. For example, when participants entered "check" as part of a query, our tool is able to recommend APIs whose *f_sentences* contain verbs such as "test" and "determine". Participants also suggested improvements to the tool, such as better ranking of results and showing all variants of overloaded methods.

6 THREATS TO VALIDITY

Internal validity. A potential threat to internal validity stems from the use of the natural language processing library, SpaCy. Some of our analyses are based on SpaCy's natural language processing of sentences (e.g., RQ1 and RQ3). No natural language processing library achieves 100% accuracy on any large data set, and SpaCy's perfomance was found to be on par with the state of the art [17] and outperforming other libraries when applied to software documentation [13]. SpaCy was not designed specifically for software text, i.e., text containing code elements, incomplete sentences, or grammar errors which are common on Stack Overflow. Currently there is no natural language processing tool specialized for parsing software development related text and we have to rely on general-purpose natural language processing tools. To mitigate this threat, we use heuristic rules to correct some common mistakes. This process is similar to related work [40].

Another threat may arise from the scale of the open coding. For the analysis of *f_categories* in RQ2, we only coded the *f_sentences* of 14,733 JDK and Android API methods, not all of the 54,256 *f_sentences*. One concern is that the *f_verbs* and *f_categories* we identified may not cover all *f_verbs* and *f_categories* in JDK and

Android. Open coding of the full set is beyond our capabilities, and we try our best to cover as many *f_verbs* and *f_categories* as possible by our sampling strategy. Based on the findings of RQ1, the 116 most frequent verbs appear in approximately 84% of the *f_sentences*. Thus, in the initial coding phase, we sample 10 sentences for each of the 116 most frequent verbs to cover as many common *f_verbs* and *f_categories* as possible with a relatively small sampling size (1,139). In the second phase, we create a larger random sample (13,635) to cover uncommon *f_verbs* and *f_categories* that were not covered by the first sample.

An additional threat is related to the quality of open coding. We mitigate this by separating the two phases of coding and training all coders before coding. We report Cohen's Kappa for all open coding to provide evidence that our coding results are reliable.

External validity. A major concern is the extent to which our automated detection tools are generalizable. We provide evidence for their generalizability by evaluating them on JDK and Android *f_sentences*, Stack Overflow questions, and POI *f_sentences*. New *f_categories* and *f_patterns* may need to be revealed using similar empirical study process when using our approach for other libraries.

7 RELATED WORK

7.1 Knowledge about Functionality

Functionality is an important knowledge type required for software development tasks such as features implementation and maintenance. Kirk et al. [30] studied the "reuse problems" faced when developing applications based on a framework and identified four main categories of framework reuse problems—"Functionality" is one of them. Erdos and Sneed [19] identified seven questions developers need to ask during software maintenance tasks. All questions are about understanding the behavior of the program and therefore about functional knowledge.

Other related work targets functionality descriptions in software documentation. Maalej and Robillard [31] reported on a study of knowledge patterns in API documentation, such as Functionality, Concepts, and Directives. The authors found that functionality accounts for a large part of API documentation, but they do not offer further categorization of the functionality descriptions in API documentation. Based on Maalej and Robillard's results, Fucci et al. [20] attempted to use machine learning techniques to classify the knowledge types of sentences in API documentation. They reported that the most frequent knowledge types are Functionality and Non-information. In our work, we focus on the most common knowledge type—Functionality—and classify and analyze API functionality descriptions based on functionality verbs.

7.2 Verb Phrases in Software Engineering

In many programming languages, method names are used to describe the implementation of a method at a high level. Past research has found that source code will be more readable if every method has an appropriate name [25]. Since the method name usually consists of verb phrases, Høst and Østvold [24] constructed a lexicon containing frequently used verbs in Java method names and reported characteristics of method names based on their verbs. Hayase et al. [22] built a domain-specific dictionary of verb-object relations from identifiers appearing in source code files. Kashiwabara et al. [29] focused on recommending similar verbs for a method name so that developers can use consistent verbs for method names. However, their focus was on method names instead of natural language descriptions of methods.

Shepherd et al. [38] proposed an approach for query expansion and code search. This method uses <verb, direct object> (V-DO) pairs from method signatures and comments to find actions that cross-cut object-oriented systems. Hill et al. [23] proposed an approach to automatically extract and generate noun, verb, and prepositional phrases from method and field signatures, capturing word context of natural language queries for maintenance and reuse.

Treude et al. [40] focused on natural language descriptions and extracted development task phrases from software documentation. However, they extracted all task phrases from sentences. In their work, one sentence can contain more than one task phrase and they did not distinguish them based on importance. Also, they only used a small set of predefined verbs to define task phrases and did not consider synonyms in a systematic way. In this work, we focus on functionality descriptions of Java and Android API methods from API reference documentation and classify functionality verbs into functionality categories, thus providing a systematic way for exploring synonyms in a functionality category.

7.3 API Recommendation

Current API recommendation approaches typically use context information to recommend APIs, e.g., API dependency graphs [16], feature request history [39], and question-and-answer websites and documents [27, 37]. Current approaches can not only recommend API methods and classes from third-party libraries [27, 37], but also support project-specific APIs [43].

Rahman et al. [37] proposed an approach called RACK and also constructed a corpus to map keywords from Stack Overflow questions to API documentation. Based on this corpus, RACK can recommend APIs for a given query. Huang et al. [27] combined Stack Overflow knowledge with API documentation and proposed BIKER, which can also recommend APIs for a given query. However, these approaches can only work for APIs which have been discussed extensively on sites such as Stack Overflow and suffer from information noise in these external resources. Previous work has shown that Stack Overflow tends to be slow at covering new APIs [36] and can ignore significant parts of an API. In contrast, our approach does not rely on external resources.

The approach by Hill et al. [23] can automatically categorize extracted phrases into a hierarchy based on partial phrase matching, to help software maintainers quickly discriminate between relevant and irrelevant search results and reformulate queries. However, their approach can not deal with the problem of lexical gaps between queries and documentation.

Other approaches in the area of API recommendations do not focus on recommending methods, but for example on code snippets [15, 34, 35, 45] or parameters [14] instead.

8 CONCLUSION

In this paper, we conducted a large-scale empirical study on the functionality descriptions of 14,733 JDK and Android API methods.

We identified 356 different *functionality verbs* from the descriptions, and these verbs can be grouped into 87 *functionality categories* based on their semantics in the description context. We also extracted 523 phrase patterns from the verb phrases of the descriptions. Building on these findings, we propose an API method recommendation approach based on explicit matching of functionality verb phrases in functionality descriptions and user queries, which is called PreMA. We conducted experimental studies to evaluate the functionality analysis accuracy and API retrieval performance of PreMA. The results show that PreMA can accurately recognize the functionality categories (92.8%) and phrase patterns (90.4%) of functionality description sentences; and the participants using PreMA completed their tasks more accurately (0.77 versus 0.54) with fewer retries (2.16 versus 3.30) and using less time (98.29s versus 113.42s).

Future work will be devoted to applying the approaches for automatically recognizing *functionality categories* and associated *functionality verbs* and *phrase patterns* to other software engineering problems, such as documentation quality and information retrieval. In addition, we will further improve the context analysis capability (e.g., by considering the class descriptions and method descriptions beyond functionalities) PreMA to achieve more precise API matching. All data from this work will be turned into archived open data after acceptance.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

REFERENCES

[1] 2020. *Android 27 Reference Documentation.* Retrieved September 10, 2020 from https://developer.android.com/reference/packages
[2] 2020. *Annotation Tool GUI.* Retrieved September 10, 2020 from https://i.loli.net/2019/05/12/5cd7d3ed9a2e6.png
[3] 2020. *Apache POI.* Retrieved September 10, 2020 from https://poi.apache.org/
[4] 2020. *BeautifulSoup.* Retrieved September 10, 2020 from https://www.crummy.com/software/BeautifulSoup/
[5] 2020. *gensim.* Retrieved September 10, 2020 from https://radimrehurek.com/gensim/
[6] 2020. *JDK 1.8 Reference Documentation.* Retrieved September 10, 2020 from https://docs.oracle.com/javase/8/docs/api/overview-summary.html
[7] 2020. *Replication Package.* Retrieved September 9, 2020 from https://fudanselab.github.io/Research-FSE2020-FuncVerb/
[8] 2020. *SpaCy.* Retrieved September 10, 2020 from https://spacy.io
[9] 2020. *Stack Overflow Question 65200.* Retrieved September 10, 2020 from https://stackoverflow.com/questions/65200/
[10] 2020. *VerbNet.* Retrieved September 10, 2020 from http://verbs.colorado.edu/~mpalmer/projects/verbnet.html
[11] 2020. *VerbNet Annotation Guidelines.* Retrieved September 10, 2020 from https://verbs.colorado.edu/verb-index/VerbNet_Guidelines.pdf
[12] 2020. *word2vec-api.* Retrieved September 10, 2020 from https://github.com/3Top/word2vec-api
[13] Fouad Nasser A Al Omran and Christoph Treude. 2017. Choosing an NLP Library for Analyzing Software Documentation: A Systematic Literature Review and a Series of Experiments. In *Proceedings of the International Conference on Mining Software Repositories*. 187–197.
[14] Muhammad Asaduzzaman, Chanchal K. Roy, Samiul Monir, and Kevin A. Schneider. 2015. Exploring API method parameter recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 271–280.
[15] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 628–632.
[16] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 10.
[17] Jinho D. Choi, Joel R. Tetreault, and Amanda Stent. 2015. It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. 387–396.
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).
[19] Katalin Erdös and Harry M. Sneed. 1998. Partial Comprehension of Complex Programs (enough to perform maintenance). In *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, 98–105.
[20] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. 2019. On using machine learning to identify knowledge in API reference documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 109–119.
[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 631–642.
[22] Yasuhiro Hayase, Yu Kashima, Yuki Manabe, and Katsuro Inoue. 2011. Building Domain Specific Dictionaries of Verb-Object Relation from Source Code. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. 93–100.
[23] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 232–242.
[24] Einar W. Høst and Bjarte M. Østvold. 2007. The Programmer's Lexicon, Volume I: The Verbs. In *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*. 193–202.
[25] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. 294–317.
[26] Einar W. Høst and Bjarte M. Østvold. 2010. Canonical Method Names for Java - Using Implementation Semantics to Identify Synonymous Verbs. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. 226–245.
[27] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 293–304.
[28] Yuki Kashiwabara, Takashi Ishio, Hideaki Hata, and Katsuro Inoue. 2015. Method Verb Recommendation Using Association Rule Mining in a Set of Existing Projects. *IEICE Transactions* 98-D, 3 (2015), 627–636.
[29] Yuki Kashiwabara, Yuya Onizuka, Takashi Ishio, Yasuhiro Hayase, Tetsuo Yamamoto, and Katsuro Inoue. 2014. Recommending verbs for rename method using association rule mining. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*. 323–327.
[30] Douglas Samuel Kirk, Marc Roper, and Murray Wood. 2007. Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering* 12, 3 (2007), 243–274.
[31] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282.
[32] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.
[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.
[34] Anh Nguyen, Peter C. Rigby, Thanh Van Nguyen, Dharani Palani, Mark Karanfil, and Tien N. Nguyen. 2018. Statistical Translation of English Texts to API Code Templates. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 194–205.
[35] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query Expansion Based on Crowd Knowledge for Code Search. *IEEE Trans. Services Computing* 9, 5 (2016), 771–783.
[36] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Institute of Technology, Tech. Rep* 11 (2012).

[37] Mohammad Masudur Rahman, Chanchal Kumar Roy, and David Lo. 2016. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 349–359.

[38] David C. Shepherd, Zachary P. Fry, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. 2007. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007 (ACM International Conference Proceeding Series, Vol. 208)*, Brian M. Barry and Oege de Moor (Eds.). ACM, 212–224.

[39] Ferdian Thung, Shaowei Wang, David Lo, and Julia L. Lawall. 2013. Automatic recommendation of API methods from feature requests. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 290–300.

[40] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. 2015. Extracting Development Tasks to Navigate Software Documentation. *IEEE Trans. Software Eng.* 41, 6 (2015), 565–581.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[42] Denny Vrandecic. 2013. The Rise of Wikidata. *IEEE Intelligent Systems* 28, 4 (2013), 90–95.

[43] Rensong Xie, Xianglong Kong, Lulu Wang, Ying Zhou, and Bixin Li. 2019. HiRec: API Recommendation using Hierarchical Context. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ivaki, and Nuno Laranjeiro (Eds.). IEEE, 369–379.

[44] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 404–415.

[45] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 956–961.