APIBook – An Effective Approach for Finding APIs

Haibo Yu
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Minhang,
Shanghai 200240, China
+86-21-34204147
haibo_yu@sjtu.edu.cn

Wenhao Song
Sochool of Software,
Shanghai Jiao Tong University
800 Dongchuan Road, Minhang,
Shanghai 200240, China
+86-13817724887
me@cppdo.com

Tsunenori Mine
Faculty of Information Science and Electrical Engineering,
Kyushu University
744 Motooka, Nishi-ku,
Fukuoka 819-0395, Japan
+81-92-8023613
mine@ait.kyushu-u.ac.jp

ABSTRACT

Software libraries have become more and more complex in recent years. Developers usually have to rely on search engines to find API documents and then select suitable APIs to do relevant development when working on unfamiliar functions. However, the traditional search engines do not focus on searching APIs that make this process inconvenient and time consuming. Although a lot of efforts have been made on API understanding and code search in industry and academia, work and tools that can recommend API methods to users based on their description of API's functionality are still very limited.

In this paper, we propose a search-based recommendation algorithm on API methods. We call the algorithm APIBook and implement an API method recommendation tool based on the proposed algorithm. The algorithm can recommend relevant API methods to users based on user input written in natural language. This algorithm combines semantic relevance, type relevance and the extent of degree that API method is used to sort these API methods and rank those that are highly relevant and widely used in the top positions. Examples of codes in real projects are also provided to help users to learn and to understand the API method recommended. The API recommendation tool selects the Java Standard Library as well as 100 popular open source libraries as API recommending material. Users can input the API description via the Web interface, and view the search results with sample codes on screen.

The evaluation experiment is performed and the result shows that APIBook is more effective for finding APIs than traditional search models and it takes on average 0.7 seconds for finding relevant API methods which we think to be reasonable for satisfying daily query requirements.

CCS Concepts

• Software and its engineering → Software creation and management → Software development techniques → Reusability. • Information systems → Information systems applications。

Internetware '16, September 18 2016, Beijing, China
© 2016 ACM. ISBN 978-1-4503-4829-4/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2993717.2993727

Keywords

API Search; Information Retrieval; API Recommendation; Program Analysis

1. INTRODUCTION

Improvement of the efficiency of software development has long been a central issue in the field of software engineering. Code reusing is very important during software development. It has active effect on improving the efficiency and quality for software development [1][2][3]. With the development of the Internet and advances in computer science technology, development tasks are becoming more and more complex, and such situation causes the emergence of more complex program libraries aimed at helping developers simplify the development process. For example, the basic library of JDK8 has more than 140000 classes and methods. The research result has shown that the selection of API is one of the six learning barriers in end user programming systems [4].

Using search engines is one possible way for developers to solve the problems on programming. The search engines can find relevant information of APIs such as documents, blogs, forums and code snippets through searching and sorting. However, since the traditional search engines are not designed for searching programs, the search results generally include many irrelevant, imprecise, or unexpected ones. The selection and researching from these results will take a lot of extra time which reduces the efficiency of development.

There have been many researches for helping developers to use APIs efficiently. For example, Sourcerer [5] is a typical application for code search which is based on the structural information of open source code. Portfolio [6] searches for the relevant codes associated with a complicated task based on PageRank [7] and SAN [8] algorithms. However, these efforts focus on the recommendation of code snippets instead of APIs. Among the studies on API searching, Jungloid [9] can synthesize the usage of APIs based on the expected input types and output types. However, this research focuses on the usage of API and the problem of API selection is not solved. Mica [10] is the most relevant research with our work. It first obtains relevant pages through a traditional search engine and then matches the words on the pages with the names of classes and methods in JDK and finally finds the example codes for using APIs. However, the search results of Mica rely on the traditional search engines which cannot recognize the structural information of APIs, and hence the search results cannot be optimized based on the structural information of APIs.

In order to cover the deficiencies of the above methods, this paper proposes an algorithm called APIBook. The APIBook searches for APIs through multiple libraries using information retrieval technologies. The main characteristics and contributions of this paper are as follows:

1) Unlike traditional keyword search methods, our proposed algorithm takes natural language as input. This is a more natural way for users to express their searching attempts.
2) Combined with program analysis technologies, the matching and sorting algorithm considers all the aspects including semantics, types and the usage information of APIs, the algorithm can recommend the real highly matched and widely used APIs.
3) The output presents APIs as well as example code snippets of real programs that can help users to easily understand the usage of APIs.

The rest of this paper is organized as follows. We present the overall structure of APIBook in Section 2. The searching and sorting algorithms are described in Section 3. We describe the searching tool in section 4. The experiment and discussion are described in Section 5. Section 6 introduces related work and the conclusion of the paper is described in Section 7.

2. OVERALL STRUCTURE

The overall structure of APIBook is shown in Figure 1. First users will input their free-form query in natural language through the Web interface to describe their searching intents on the API methods. We call the user query "API description" which consists of a simple sentence or phrases.

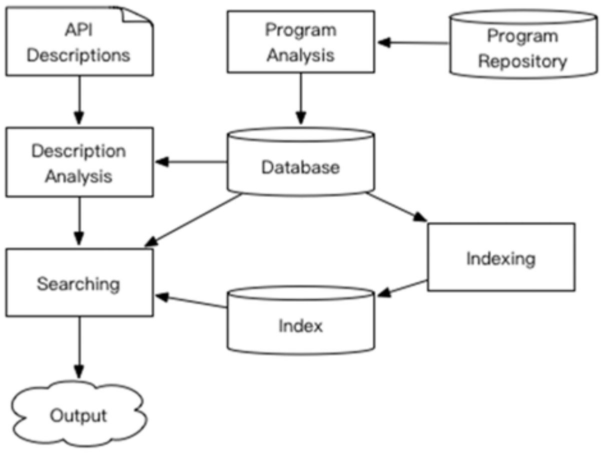
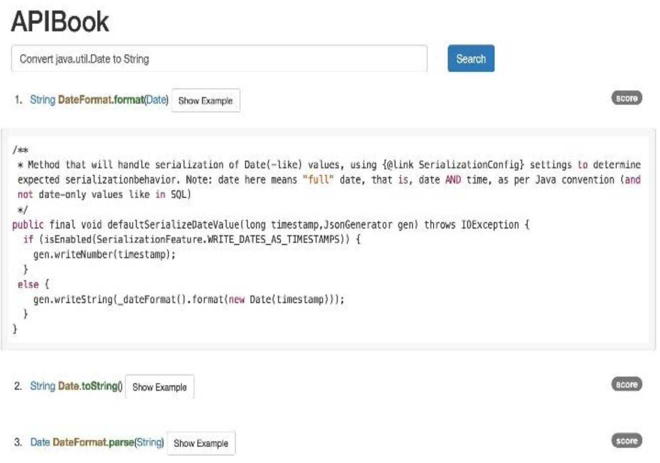


Figure 1. Overall Structure.

The user interface of APIBook with a search example is shown in Figure 2. In the example, we can see that the user inputs the API description "`Convert java.util.Date to String" in the search bar that attempts to search for APIs to convert the type of "`java.util.Date" into the type of "String". Here we can see that the input is in a free form. It includes not only the information of functionality "convert" but also the information of types "java.util" and "String" of the searched APIs. Users can input the functionality and type information mixed together and need not to learn special gramma for using APIBook.

After the user pressed the search button, the APIBook will perform description analysis that extracts relevant semantic and type information which represents the user's requirements to search for APIs. The accuracy of description analysis is the foundation of searching APIs correctly.


APIBook
Convert java.util.Date to String
Search
1. String DateFormat.format(Date)
Show Example
score
2. String Date.toString()
Show Example
score
3. Date DateFormat.parse(String)
Show Example
score

```
/**
 * Method that will handle serialization of Date(-like) values, using {@link SerializationConfig} settings to determine
 expected serializationbehavior. Note: date here means "full" date, that is, date AND time, as per Java convention (and
 not date-only values like in SQL)
 */
public final void defaultSerializeDateValue(long timestamp,JsonGenerator gen) throws IOException {
  if (isEnabled(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)) {
    gen.writeNumber(timestamp);
  }
 else {
    gen.writeString(_dateFormat().format(new Date(timestamp)));
  }
}
```

Figure 2. APIBook User Interface

The APIBook will reference the type information of APIs in the program repository when performing description analysis. Since the program can only be understood by computers after the analysis with program analysis techniques and the program analysis is time consuming, the APIBook analyzes the program repository in advance to extract relevant information and saves it into the database. In this way, the APIBook can save time for description analysis since it can use the information saved in the database directly when performing description analysis instead of doing that at run time.

After the description analysis, the semantic and type information extracted from the API description will be used as input of the searching module to search for relevant APIs from the program repository using our proposed matching and ranking algorithm. Finally, the list of matched APIs and the example code for using the APIs will be presented to users.

In Figure. 2, we can see that the output of APIBook was shown under the search bar. The matched APIs are sorted and listed based on the similarity order. Users can view the example code of a specific API by pressing the "show example" button on its right side.

In order to accelerate the searching and sorting algorithm, the APIBook constructs full-text index for certain data in the database. Compared to the index of database, the superiority of full-text index is that it has high efficiency when doing text matching. Since there are huge amount of APIs in the program repository, the optimization result of full-text indexing is obvious.

3. API RECOMEENDATION ALGORITHM

The API recommendation algorithm is the core part of this research. In this section, we will introduce the three main components of our API recommendation algorithm including the API description analysis, searching and sorting algorithm and the selection of example code.

3.1 API Description Analysis

As we mentioned in Section 2, that users input their free-form API description from the Web interface when they want to search for

APIs. In order to understand the user intent of searching, the APIBook analyzes their API description. In the APIBook, the information of API description is divided into two parts: semantic information and type information.

3.1.1 Semantic Information

The semantic information of an API description refers to the meaning of words in the API description. Each word has its meaning and will have a specific meaning when it is put into a certain sentence. In our approach, we use the word itself to represent its meaning and use the set of words extracted from the API description to represent the semantic information of the API description.

During the API description analysis, it needs to do the natural language processing which separates sentences and words in the API description and performs the part-of-speech tagging. We utilizes the open source tool CoreNLP [13] to apply the natural language processing to the API description.

A different word has its different importance in the API description. For example, in a description like “How can I clear or empty the StringBuilder?”, the words of “clear”, “empty” and “StringBuilder” are more important than the words: “How,” “can,” “I,” “or,” “a,” which are playing a supporting role in the description and hence be less important. According to the formation of English, nouns, verbs, adjectives usually have a relatively important semantics, and other words such as adverbs, prepositions mainly play a supporting role in the expression. In our approach, we use the nouns, verbs and adjectives as key semantic information of the API description and these expressions as the semantic information will be used for the semantic matching later.

3.1.2 Type Information

The type information of the API description refers to the information which concerns types. For example, in a description of “How can I clear or empty the StringBuilder?”, “StringBuilder” is the type information of this API description. The complete type name should be “java.lang.StringBuilder”. The type information plays an important role in an API description.

Since we allow users to input an API description in free-form and the users can add any type information freely such as using a short name, a complete name or even using a hidden way to represent types, therefore we need to extract relevant type information correctly and to apply normalization to them to a unified complete type name.

Since there is no corresponding declarations and implementations for base types such as “int,” “double,” “float,” ”long” in the library, the algorithm first checks all the nouns in the API description to see whether it is a Java base type or not. If it is a Java base type, the type name and the boxed type name will be added into the type information of the API description. Then the algorithm will form a regular expression for each noun of the API description and search it in the database to check if there is a matched type or not.

In order to avoid matching too much irrelevant types, the APIBook adopts a conservative policy that means only the exactly matched type name was judged to be successful matching. For example, "string" and "java.lang.String" can match "java.lang.String" type in the database, however they do not match the "com.example.FooString" type in the library.

3.2 Searching and Sorting Algorithm

After extracted relevant information from the API description through description analysis, the APIBook will perform matching using the extracted information to search for relevant API methods from the program repository.

The target of the searching algorithm is to find matched APIs which the user is searching for from the program repository. The searching process is shown in Figure 3.

In our searching algorithm, the information included in the API description input by the user was divided into two types: semantic information and type information. The semantic information represent the functionality or characteristics of APIs. It describes the basic searching attempts for the API. We use the collection of words in the API description to express the semantic information. The type information means the attempt on the type of searched API. For example, in the query of “convert InputStream to String”, the “InputStream” and “String” are the attempts on the type of searched API. There are some information whose type is implicit. For example, the “File” is the attempt on the type of the API in the query of “How to write to a file”.

The semantic matching and type matching are performed separately between the relevant information extracted from the API description and the APIs in the program repository. The matching scores of semantic matching and the type matching are calculated respectively. The usage score which reflects the usage extensiveness of the API will be retrieved from the library. The usage score is based on the assumption that users generally attempt implicitly to find the relevant APIs which are widely used.

The overall score is given based on the total consideration of the semantic score, type score and usage score and the search result is sorted based on the overall score.

3.2.1 Semantic Matching

3.2.1.1 Semantic Matching Dimensions

An API generally includes the semantic information of multiple dimensions such as the method name, class name and package name. The selection of semantic matching dimensions is very important in our searching algorithm.

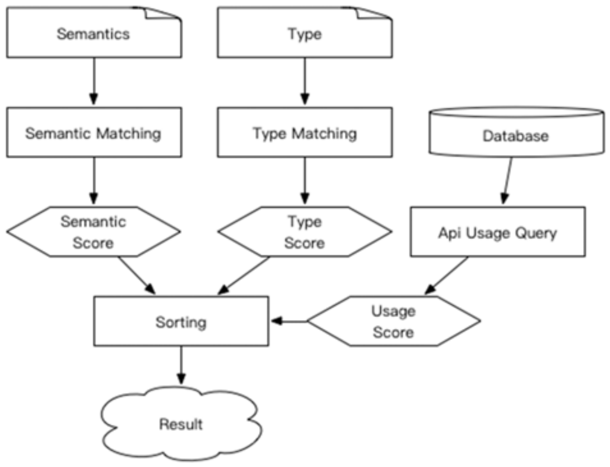


Figure 3. Searching & Sorting

The APIBook selects eight semantic matching dimensions: the method name, class name, types of input parameters, types of return parameters, comments for methods, comments for input parameters, comments for return parameters and the names of the parameters. The method name is the most precise description for an API. Therefore each word includes important information in it. The comment for the API is the detailed description about the functionalities of the API. It can provide more semantic information than the method name. However it may also include some words which have lower relevance with the API core functionalities, and at the same time this leads to be a lower importance than the method name. The class name reflects the working object and the function module. The type of input parameters, comments for the parameters, types of return parameters and comments for the return parameters reflect the semantic information of the inputs and outputs of the API.

3.2.1.2 Semantic Matching Score

The target of semantic matching is to find the most matched APIs from the semantic point of view.

First, the similarity between the API description and the APIs in the library was calculated for each dimension of semantic information of the API. The tf-idf [11] method is used for the calculation of the similarity of each dimension.

$$tf_{t,f} = \sqrt{freq_{t,f}} \qquad (1)$$

The $freq_{t,f}$ is the occurrence frequency of term t in dimension f.

We treat the collection of all the information from all dimensions as a document. idf_t means the uniqueness of term t in all the documents. As the number of documents including the term t becomes smaller, the value of idf_t becomes bigger. The idf_t is calculated using equation (2).

$$idf_t = 1 + \log(\frac{docs}{docs_t+1}) \qquad (2)$$

The $docs$ means the number of documents. The $docs_t$ means the number of documents which include the term t.

The similarity between a term and a document is calculated with an improved vector space mode (VSM) [12]. The difference between our improved VSM and traditional VSM is that the matching of one word in multiple dimensions is weaken than the matching of multiple words in one dimension.

We first calculate the similarity between term t and dimension f using equation (3).

$$sim_{t,f} = (idf_t^2 * tf_{t,f} * weight_f) / \sqrt{len_f} \qquad (3)$$

The $weight_f$ in equation (3) means the weight of dimension f, and the len_f means the number of terms in dimension f. The purpose of setting weights is to differentiate the importance of different dimensions. In the APIBook, the weight of the method name is set to 1.5 and the weight of other dimensions are set to 1 based on our experience.

Then the similarity between term t and document d is calculated using equation (4).

$$sim_{t,d} = (1 - c_0) * max_{f \in d} sim_{t,f} + c_0 * \sum_{f \in d} sim_t, f \qquad (4)$$

From the above equation (4), we can see that only the highest matching score of term t with all dimensions is reserved, the other similarity score is only added with weight c_0. In practice, the c_0 is set to 0.3. This can not only assure that the document which matches multiple words in different dimensions gets higher score, but also avoid the document which matches the same word in multiple dimensions gets higher score.

Finally, the semantic score of document d is calculated using equation (5).

$$S_{q,d} = coord_{q,d} * \sum_{t \in q} sim_{t,d} \qquad (5)$$

The $coord_{q,d}$ in equation (5) means the number of words which appears in the semantic information of API description q and document d. The coefficient $coord_{q,d}$ is used to assure that the document that can match more words can get higher score.

3.2.2 Type Matching

3.2.2.1 Type Matching Dimensions

Similar to semantic matching, the type matching also needs to select matching dimensions. In this paper, we select an API name, types of input parameters and types of return parameters as type matching dimensions. The class name reflects the working object and functional module. The types of input parameters and types of return parameters represent the types of inputs and outputs of the API. They all may relate with the type information in the API description which is described by the user.

3.2.2.2 Type Matching Score

The basic idea for type matching in this paper is that, for a type, if it can match with any type dimension of API means the API is related with that type. If there are multiple types in the API description, the more the types matched with the API, the stronger the API related with the type. In this paper, we use equation (6) to calculate the similarity between a type and a document.

$$\mathrm{sim}_{t,d} = max_{f \in d} \frac{freq_{t,f}}{terms_f} \qquad (6)$$

The $\mathrm{freq}_{t,f}$ in equation (6) means the frequency of a type t appeared in dimension f of the document. $terms_f$ means the total number of words in dimension f of the document. $\frac{freq_{t,f}}{terms_f}$ represents the percentage of a type in a certain dimension. For example, if an API has three input parameters, there are two parameters in type t, then the $\frac{freq_{t,f}}{terms_f}$ is 2/3. Since one type that matches the document in multiple dimensions does not always have stronger relation with users' searching attempt in that type, we select the maximum $\frac{freq_{t,f}}{terms_f}$ of each dimension f as the similarity of the word t with the document. This avoids that one type that matches the document in multiple dimensions gets too much high score.

The similarity between the type information q with document d is calculated using equation (7).

$$T_{q,d} = coord_{q,d} * \sum_{t \in q} \mathrm{sim}_{t,d} \qquad (7)$$

The $coord_{q,d}$ in equation (7) means the number of words appeared in both of the type information q and the document d. The coefficient $coord_{q,d}$ is used in order to assure that the API which matches with multiple types gets higher score.

3.2.3 Usage Score

The usage score reflects the popularity of an API. The usage score is calculated in this paper using equation (8).

$$\mathrm{U_d} = \log_{10}(\mathrm{uses_d} + 1) \qquad (8)$$

The uses_d in equation (8) means the number of classes which used the class of related APIs in document d. log_{10} is used to assure that the usage score affects the sorting of API at the magnitude of 10 times in order to avoid too large difference of API overall score caused by the usage score.

3.2.4 *Overall Score*

The overall score is calculated using equation (9) which is the product of the semantic matching score, type matching score and usage score.

$$score_{q,d} = (c_1 + T_{q,d}) * (c_1 + S_{q,d}) * (c_2 + U_d) \qquad (9)$$

In equation (9), c_1 and c_2 are small empirical baseline values used for avoiding certain documents will be under evaluated due to one item becoming too small value and loose the chance for being selected. In this paper, we set c_1 to 0.5, c_2 to 1. The reason for setting c_2 to be greater than c_1 is that the difference between the usage score is not very useful when its value is too small. Therefore the baseline for the usage score is set to a bigger one than the others.

3.3 API Example Code

The APIBook performs program analysis for all the source code in the program repository in advance. The call status for APIs of each method in each piece of source code is analyzed and the obtained results are stored into the database in the format of <API name, method> key pairs.

When it is necessary to present the example code for using the API, the APIBook will search in the database based on the API name to find relevant methods which use the API and then the example codes will be displayed to the user.

4. CONSTRUCTION OF PROGRAM REPOSITORY, DATABASE AND INDEX

In order to evaluate the effectiveness of our algorithm, we implemented an API searching and recommending tool based on our proposed searching algorithm described in Section 3.

The overall structure of our approach is shown in Figure 1 in Section 2. The API searching will be performed in the program repository based on the information extracted from the API description input by the user and the searched result of APIs will be presented to the user as a list as well as the example code for using the API. In this section, we will describe the method for the construction of program repository, database and index.

4.1 Program Repository Construction

The program repository of the APIBook consists of source code and compiled jar packages. Any library which has source code or jar package can be added into the APIBook program repository so as to be searched. The construction of program repository is convenient and flexible in this way compared to the construction of API documents.

In order to understand the usage status of APIs, the APIBook analyzes the program source code or binary code of third-parties and the API usage status can be obtained through the analyzation of the call relations in the libraries.

4.2 Database Construction

The APIBook extracts the API relevant information from the program repository using program analysis technologies.

The Eclipse JDT [14] program analysis tool is used for the program analysis for the libraries in the format of source code. The APIBook examines each API to obtain the information of its method name, class name, types of input parameters, types of return parameters, comments for methods, comments for input parameters, comments for return parameters and the parameter names. It also saves the information of APIs that the class called.

The ASM [15] binary code analysis framework is used for the program analysis for the library in the format of jar package. The information of APIs extracted from this kind of libraries is as same as the library in the source code format. However, since the comments of the methods, input parameters and return parameters of Java program are removed after compiling, these items will be set to blank.

The APIBook stores the relevant information of APIs in MongoDB [16] which is a kind of none relational database. The reason for using a non-relational database is its strong scalability. The storage can be extended easily only by adding extra machines when necessary and the searching efficiency will not be effected much. It's suitable for storying programs of a large number of third-party libraries.

4.3 Indexing Construction

It is necessary to do indexing in advance in order to accelerate the searching of APIs. In this paper, the indexing is based on Lucene [17] which is an open source searching framework. The APIBook constructs indexes for both semantic matching and type matching.

4.3.1 *Semantic Index*

The semantic index is used for the matching between the semantic key words in an API description input by a user and the semantic information of APIs in the program repository. Generally, the API type name, method name, variable name and comments are not suitable for being used as semantic indices directly. For example, for the method name "writeFile", the reasonable semantic index should be two separate words "write" and "file".

In order to construct the semantic index of APIs in program repository, the APIBook performs five preprocessing steps including segmentation, split of camel cased words, case normalization, stop word removal and stemming. Figure 4.shows the preprocessing flow with an example of "Files.isSameFile".

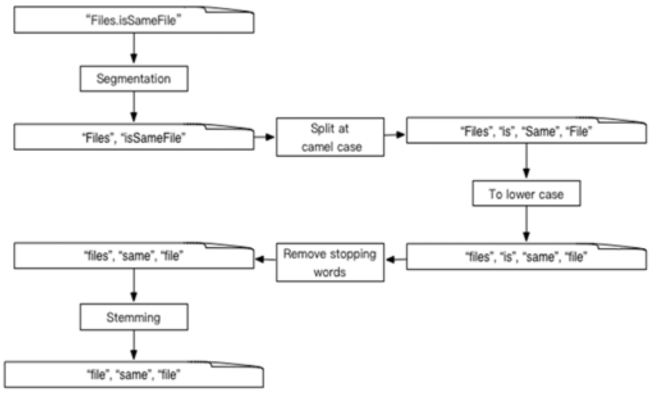


Figure 4. Example of pre-processing.

First, the APIBook performs segmentation of the document based on the place of punctuation and space to obtain single words. For example, the "Files.isSameFile" is separated to two words: "Files" and "isSameFile."

Java uses camel case naming convention to name class names, method names, and variable names in the program. The APIBook splits the camel cased words at the place where the case changes in order to extract key words from camel cased words. For example, "IsSameFile" will be split into three words: "is," "Same" and "File."

The camel cased words may produce a lot of mixed-case words after separation. However, for most of the words, the semantics of uppercase and lowercase express the same meaning such as ”Same” and “same.” It may result in mismatching for the words that have the same semantics if the indexing is performed using these mixed-case words directly. Therefore, the APIBook normalizes all the words into lowercase.

Stop words are those that do not have much information on the semantics such as "am," "is," "are," "a," "the," "what," "how" and so on in English. Since these words do not contain actual meaningful information, the APIBook removes these kind of words.

Stemming is the last step of preprocessing. There exist many words that have different forms such as singular/plural, root and derived words. However, these words which are in different forms have the similar semantics. In order to match these words as much as possible and do not introduce too much false positives at the same time, the APIBook performs stemming based on KStem algorithm [18].

After the preprocessing, the API semantic information will be indexed from multiple dimensions including method name, class name, types of input parameters, types of return parameters, method name, comments of parameters, comments of return parameters, parameter names.

4.3.2 Type Indexing

The type index of the APIBook is used for the type matching. The text preprocessing for the type information is relevantly simple compared to the semantic information. We only need to participate all the types of input parameters of the API. All the type information should be indexed using the full name which includes the package name such as “java.io.File.” This can assure the exact matching of types.

5. EVALUATION EXPERIMENT

In order to evaluate the effectiveness of our algorithm, we performed the evaluation experiment.

5.1 Experimental environment

The experimental environment is as follows: The Operating System is OSX 10.11.1. The CPU is Intel i5 2.7GHz. The memory is 8G. The version of Lucene is v 5.3.1. The version of MongoDB is v 3.0.7.

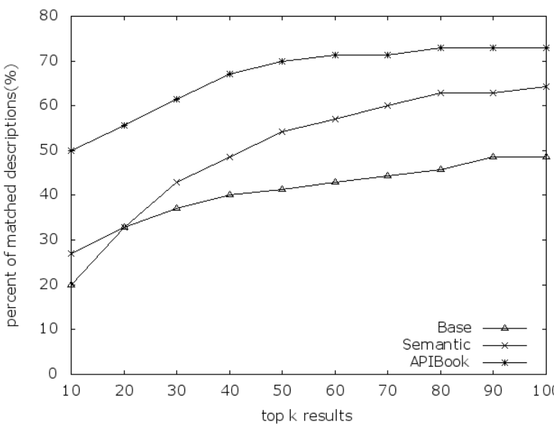


Figure 5. Experimental Result.

We choose JDK8 standard library and the top 100 most quoted program libraries in maven software repository as our program repository which includes 545683 API methods.

5.2 Selection of API Description

In order to evaluate the effectiveness of our algorithm, we selected 70 programming questions which are related with API usage and were asked over 150 times in Java section of StackOverflow. We use the title of these questions as our API description to describe the query. Some of the examples are shown below.

- Create ArrayList from array
- Generating random integers in a range with Java
- Converting String to Int in Java
- Iterate over each Entry in a Map
- How to generate a random alpha-numeric string?
- How can I convert a stack trace to a string?
- How do I discover memory usage of my application in Android?
- How can I generate an MD5 hash?
- How to round a number to n decimal places in Java
- How can I pad an integers with zeros on the left?

5.3 Evaluation Metrics

As for the evaluation metrics, we assume that an API fulfills the description if it can accomplish the programming task required in the API description. For example, the “Map.entrySet” and “Map.forEach” are judged to fulfill the API description of “Iterate over each Entry in a Map.”

Since there is few researches on API searching currently, Mica [10] does not provide quantized experimental results and the link is ineffective now, we choose the tf-idf weight and vector space model as our base method for the purpose of comparison. We also perform another comparison experiment which searches for relevant APIs only based on the semantic matching in order to conduct horizontal and vertical comparison.

5.4 Experimental Result on Searching

The experimental result is shown in Table 1. From Table 1, we can see that 50% of APIs can be found in top 10 search results based on the API descriptions using the APIBook method. The results are 30% higher than the base method. 72.9% of APIs can be found in top 100 search results using the APIBook method which are 25% higher than the base method. Even when we search for APIs only based on the semantic matching of the APIBook method, the search results are also better than the base method, and the search results are 7.1% and 15.7% higher than the base method for top 10 and top 100 results respectively. The experimental results show that the APIBook method can find APIs relevant to the description at almost, and the method is better than the base method even when only using the semantic matching. From Figure 5, we can see that the overall results of the APIBook method are better than those of the other two methods which means that the APIBook method is an effective method for API searching.

Table 1. Experimental Result

	Base Method	**Semantic Score Only**	**APIBook**
Top 10	20.0%	27.1%	50.0%
Top 30	37.1%	42,9%	61.4%
Top 50	41.4%	54.3%	70.0%
Top 100	48.6%	64.3%	72.9%

From Table 1, we can see that 27.1% of queries could not find relevant results in top 100 based on the API descriptions. After examining these API descriptions, we found that these API descriptions were characterized by the followings:

1) They include complicated logic descriptions such as "How to convert byte size into human readable format."
2) The function can only be implemented by using more than one APIs such as "java.util.Date to XMLGregorianCalendar."
3) The key type information from the API description cannot be obtained, such as "How can I concatenate two arrays in Java".

We will continue to do our research work to deal with the above problems in the future.

5.5 Efficiency Evaluation

Efficiency of searching is one of the important evaluation metric. Table 2 shows the experimental results of the average searching time of three searching methods. The base method is the method using the if-itf method for scoring. The semantic only method is the method that uses the APIBook method only employing semantic matching based on the semantic information. We can see that the average searching time of the base method, the semantic only method, and the APIBook method are 0.11s, 0.11s, and 0.73s respectively. Most of the searching time of the APIBook is spent on the calculation of API usage status. We think that 0.73s searching time is acceptable to users.

Table 2. Average Searching Time

Base Method	Semantic Only	APIBook
0.10s	0.11s	0.73s

6. RELATED WORK

In this section, we will introduce researches which are related to our work. Since the code search and recommendation has many common concerns with API search and recommendation, it is also introduced here.

There are some code or API search and recommendation algorithms based on the calculation of semantic similarity between a query and the target program. Shepherd et al. proposed Sando [19] which is a local code search framework integrated into Visual Studio IDE as a plug-in. Users can search for relevant code snippets from local project by inputting multiple key words. It reserved an interface for changing a searching algorithm and the default algorithm for similarity calculation uses the vector-space-model with the tf-idf weight. The searching algorithm of commercial code search engines KRUGLE [20] and OpenHub [21] are also based on traditional information retrieval techniques. They can not only support many programming languages and massive code libraries but also support online searching. Heinemann et al. proposed an API recommendation algorithm [22] which is based on context identifiers that utilizes the developer's intention embodied in names of variables, types and methods.

The main problem encountered by the above algorithms is the vocabulary problem [23] which means that the common vocabularies used in queries may be different from the vocabularies used in the methods of code or APIs because of the diversity of vocabularies. The difference may cause mismatching of words of the same meaning.

Some researches extends vocabularies of matching target in order to solve the above vocabulary problem. Chatterjee et al. proposed SNIFF [24] which is a free-form code search tool with improved success rate of code matching by using the documentation of the library methods to add plain English meaning to an undocumented user code. A set of small and highly relevant code snippets are generated by taking a type-based intersection of the candidate code snippets of obtained from a query search. McMillan et al. proposed Portfolio [6] which constructs the dependency graph based on the call relations of functions and then find the functions that matched with the input descriptions of the function based on the PageRank [7] and SAN [8] algorithm.

Some studies make use of the diversity of searching results of traditional search engines. Blueprint [25] is a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code. It automatically augments queries with code context, presents a code-centric view of search results, embeds the search experience into the editor, and retains a link between copied code and its source. Stylos et al. proposed Mica [10] which obtains relevant pages through traditional search engines and then matches the words extracted from the pages with class names and method names in JDK and presents the matched APIs to users. Hoffmann et al. proposed Assieme [26] which is a Web search interface that effectively supports common programming search tasks by combining information from Web-accessible Java Archive (JAR) files, API documentation, and pages that include explanatory text and sample code. Assieme uses a novel approach for finding and resolving implicit references to Java packages, types, and members within sample code on the Web.

Other studies are based on the other methods instead of semantic similarity. Code Conjurer [27] is a test driven code search engine which returns code snippets based on the test result. API Explorer [28] is a plugin of Eclipse IDE that recommend APIs for a user for his/her working condition by leveraging the structural relationships between API elements such as parameters, return values and inheritance to make API methods or types not accessible from a given API type more discoverable.

There are some research work that combined semantic and program structural information to recommend code or APIs. Bajracharya et al. proposed Sourcerer [5] which is a large scale code search engine that makes use of the semantic and structural information of program such as loops and branches. Bajracharya et al. also proposed Sourcerer API Search (SAS) [29] which is a search interface to find API usage examples in large code repositories on the base of Sourcerer. SAS extracts relevant APIs used in the code obtained through Sourcerer, creats Tag-cloud view of popular words to facilitate query reformulation and filtering out irrelevant search results. Reiss [30] lets users specify what they are looking for as precisely as possible using keywords, class or method signatures, test cases, contracts, and security constraints, and then obtains the initial search result based on the semantic matching of these information, and then uses an open set of program transformations such as changing the order of parameters to map retrieved code into what the user asked for. Chan et al. proposed an approach to recommend API methods through given textual phrases based on a greedy subgraph search algorithm and refinement techniques [31]. Thung et al. recommend API methods of a given target library and a textual description of the task based on the combination of information retrieval and collaborative filtering techniques [32].

There are some research work that are trying to help users to understand and use APIs. Zhong et al. proposed a framework MAPO [33] for mining the API usage patterns such as what APIs are often used together and their usage order. APIExample [34] is a tool that presents examples for using a specific Java API by collecting, extracting and integrating code snippets and descriptions. Baker [35] is an iterative, deductive method of linking source code examples to API documents. It analyzes the type information in the code based on a constrained solving method and constructs the by-directional link between code snippets and documents. Example code snippets can be added into an API dynamically based on this method.

The researches on code searching and recommendation which are introduced on the above cannot satisfy the requirements of API recommendation because they are used in different situations. That means code searching and recommendation is generally used under a relevantly complicated situation when users need to read and understand the functions and logics of the code. On the other hand, API searching and recommendation is mostly used to give a direct result when users want to realize a common algorithm or function.

The most relevant researches with our work on API searching are Mica [10], Assieme [26] and the work of Heinemann [22] whose API method recommendation algorithms are based on the context identifier. All of these methods recommend APIs for users using the target API description input by users. However, Mica has the problem that it depends on the search result of a traditional search engine too much and therefore the search results cannot be optimized based on the structural information of the APIs. The main problem of Assieme is that the analysis of reference is not accurate enough which causes irrelevant results. The method proposed by Heinemann et al. has some practical limitation because the semantic information used by the method is limited. On the other hand, our approach the APIBook combines semantic relevance, type relevance and the extent of degree that the API method is used, to sort API methods and hence leads to the real highly matched and widely used APIs, which can be ranked in the top positions.

7. CONCLUSION

In this paper, we proposed an API recommendation algorithm APIBook which performs searching based on the semantic, type and usage information of APIs. The experimental results showed that the searching algorithm of our proposed APIBook was effective. In the future, we will continue our research work on the following directions: try to understand API descriptions with complicated logic, to compose more than one APIs to satisfy an API description, and to optimize the calculation of usage information.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous Internetware reviewers for their valuable feedback on our earlier drafts of this paper. This research was sponsored in part by 973 Program in China (Grant No. 2015CB352203), the National Nature Science Foundation of China (Grant No. 61572312 and 61572313). This work is performed during Haibo Yu's visit to Kyushu University.

9. REFERENCES

[1] Victor R. Basili, Lionel C. Briand and, Walcélio L. Melo. How Reuse Influences Productivity in Object-oriented Systems. Communications of the ACM, 1996, 39(10), 104–116. DOI=10.1145/236156.236184.

[2] William B. Frakes, Kyo Kang. Software reuse research: Status and future. IEEE transactions on Software Engineering. 2005(7):529-536. DOI=10.1109/TSE.2005.85

[3] Stefan Haefliger, Georg von Krogh, Sebastian Spaeth. Code reuse in open source software. Management Science, 2008, 54(1):180-193. DOI=10.1287/mnsc.1070.0748.

[4] Andrew J. Ko, Brad A. Myers, Htet Htet Aung. Six learning barriers in end- user programming systems. In Proceedings of 2004 IEEE Symposium on Visual Languages and Human Centric Computing. 2004, 199-206. DOI=10.1109/VLHCC.2004.47.

[5] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In Proceedings of Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. 2006, 681-682 DOI=10.1145/1176617.1176671.

[6] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, Chen Fu. Portfolio: finding relevant functions and their usage. In Proceedings of the 33rd International Conference on Software Engineering, 2011, 111-120. DOI=10.1145/1985793.1985809.

[7] Lawrence Page, Sergey Brin, Rajeev Motwani and Terry Winograd. The PageRank citation ranking: bringing order to the Web. Technical Report, Stanford InfoLab, 1999.

[8] Scott Everett Preece. A spreading activation network model for information retrieval. Doctoral Dissertation, University of Illinois at Urbana-Champaign Champaign, IL, USA, 1981.

[9] David Mandelin, Lin Xu, Rastislav Bodík, Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, 40(6), 48-61. DOI=10.1145/1065010.1065018.

[10] Jeffrey Stylos, Brad A. Myers. Mica: A web-search tool for finding API components and examples. In Proceedings of the Visual Languages and Human-Centric Computing, VL/HCC 2006, 195-202. DOI=10.1109/VLHCC.2006.32.

[11] Juan Ramos. Using tf-idf to determine word relevance in document queries. Proceedings of the first instructional conference on ma- chine learning, 2003.

[12] G. Salton, A. Wong, C.S. Yang. A vector space model for automatic indexing. Communications of the ACM, 1975, 18(11), 613-620. DOI=10.1145/361219.361220.

[13] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard and David Mc-Closky. The Stanford CoreNLP Natural Language Processing Toolkit. In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. 2014, 55-60.

[14] R.M. Fuhrer, M.Keller, A.Kiezun. Advanced Refactoring in the eclipse jdt: Past, present, and future. In Proceedings of the First Workshop on Refactoring Tools. 2007, 31-32.

[15] Eric Bruneton, Romain Lenglet, Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems, 2002, 30.

[16] Kristina Chodorow. MongoDB: the definitive guide. O' Reilly Media, Inc., 2013.

[17] Michael McCandless, Erik Hatcher, Otis Gospodnetic. Lucene in Action: Covers Apache Lucene 3.0. Manning Publications Co., 2010.

[18] Andrew Wiese, Valerie Ho, Emily Hill. A comparison of stemmers on source code identifiers for software search. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, 496-499. DOI=10.1109/ICSM.2011.6080817.

[19] David Shepherd, Kostadin Damevski, Bartosz Ropski, and Thomas Fritz, 2012. Sando: An Extensible Local Code Search Framework, In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, 15:1-15:2. DOI=10.1145/2393596.2393612.

[20] KRUGLE: Source Code Search Engine, http://www. krugle.com.

[21] Open Hub Code Search, https://www.openhub.net.

[22] Lars Heinemann, Veronika Bauer, Markus Herrmannsdoerfer and Benjamin Hummel. Identifier-based context-dependent api method recommendation, In proceedings of 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, 31-40. DO= 10.1109/CSMR.2012.14.

[23] Furnas G. W., Landauer, T. K and Gomez L. M. and Dumais S. T., The vocabulary problem in human-system communication, Communications of the ACM, 1987, 30(11), 964-971. DOI=10.1145/32206.32212.

[24] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen, SNIFF: A Search Engine for Java Using Free-Form Queries, Fundamental Approaches to Software Engineering, 2009, Volume 5503 of the series LNCS, 385-400.

[25] Joel Brandt, Mira Dontcheva, Marcos Weskamp and Scott R. Klemmer, Example-centric programming: integrating web search into the development environment. In proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2010, 513-522. DOI=10.1145/1753326.1753402.

[26] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In Proceedings of the 20th annual ACM symposium on User interface software and technology, 2007, 13-22. DOI= 10.1145/1294211.1294216.

[27] Oliver Hummel, Werner Janjic, Colin Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. IEEE Software (Volume: 25, Issue: 5, Sept.-Oct. 2008), 45--52 . DOI= 10.1109/MS.2008.110.

[28] Ekwa Duala-Ekoko and Martin P. Robillard. Using Structure-Based Recommendations to Facilitate Discoverability in APIs. ECCOP 2011, Object-Oriented Programming, Volume 6813 of the series LNCS, 79-104. DOI=10.1007/978-3-642-22655-7_5.

[29] Sushil Bajracharya, Joel Ossher and Cristina Lopes, Searching API usage examples in code repositories with sourcerer API search. In Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation, 2010, 5-8. DOI=10.1145/1809175.1809177.

[30] Steven P. Reiss. Semantics-based code search. In Proceedings of the 31st International Conference on Software Engineering, 2009, 243—253. DOI=10.1109/ICSE.2009.5070525.

[31] Wing-Kwan Chan, Hong Cheng, David Lo. Searching connected API subgraph via text phrases. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, 10. DOI=10.1145/2393596.2393606.

[32] Ferdian Thung, Shaowei Wang, David Lo and Julia Lawall. Automatic recommendation of API methods from feature requests. In Proceedings of 2013 IEEE/ACM 28th International Conference on Automated Software Engineering, 2013. DOI=10.1109/ASE.2013.6693088.

[33] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei and Hong Mei. MAPO: Mining and recommending API usage patterns. In Proceedings of ECOOP 2009--Object-Oriented Programming, 2009, 318-343. DOI=10.1007/978-3-642-03013-0_15.

[34] Lijie Wang, Lu Fang, Leye Wang, Ge Li, Bing Xie, and Fuqing Yang. APIExample: An effective web search based usage example recommendation system for Java APIs. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, 2011, 592-595. DOI=10.1109/ASE.2011.6100133.

[35] Siddharth Subramanian, Laura Inozemtseva and Reid Holmes. Live API documentation. In proceedings of the 36th International Conference on Software Engineering, 2014, 643-652. DOI=10.1145/2568225.2568313.