Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping

Zejun Zhang
826320663@qq.com
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Minxue Pan*
mxp@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Tian Zhang*
ztluck@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Xinyu Zhou
161220181@smail.nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Xuandong Li
lxd@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

ABSTRACT

Application program interface (API) mapping is the key to the success of code migration. Leveraging API documentation to map APIs has been explored by previous studies, and recently, code-based learning approaches have become the mainstream approach and shown better results. However, learning approaches often require a large amount of training data (e.g., projects implemented using multiple languages or API mapping datasets), which are not widely available. In contrast, API documentation is usually available, but we have observed that much information in API documentation has been underexploited. Therefore, we develop a deep-dive approach to extensively explore API documentation to create improved API mapping methods. Our documentation exploration approach involves analyzing the functional description of APIs, and also considers the parameters and return values. The results of this analysis can be used to generate not only one-to-one API mapping, but also compatible API sequences, thereby enabling one-to-many API mapping. In addition, parameter-mapping relationships, which have often been ignored in previous approaches, can be produced. We apply this approach to map APIs from Java to Swift, and the experimental results indicate that our deep-dive analysis of API documentation leads to API mapping results that are superior to those generated by existing approaches.

KEYWORDS

API mapping, document process, code migration

ACM Reference Format:
Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. 2020. Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389282

*Corresponding authors.

1 INTRODUCTION

To adapt to a cross-platform and multi-language environment, different versions of software projects are often released. However, the rapid growth in the amount of software means that it is time-consuming and laborious to develop multiple versions of software projects. Therefore, to speed up the development of different versions of a software project, code-migration tools have been developed [1, 6, 7, 12, 21, 23, 25, 27, 30, 37, 38, 42]. These tools enable grammatical transformation, but they typically have little or no cross-language API-mapping functionality. For example, Java2-Csharp [7] transforms Java to C# and requires manually written application program interface (API)-mapping pairs, j2swift [30] transforms Java to Swift but does not support API mapping and transformation, and j2sINFERER [1] transforms Java to Swift and can support only a small amount of API mapping.

Many studies have been conducted to explore ways to reduce the manual effort of API mapping [1, 9–11, 17, 20, 26, 28, 34, 35, 40, 41, 43, 44]. Earlier studies were focused on leveraging API documentation to map APIs, but recent studies have focused on using code-based learning approaches to obtain better results. However, these approaches all require a large amount of training data. In addition, many approaches require bilingual projects, or even bilingual code fragments implementing the same functions as training data. However, as an experiment in [11] shows, such projects are only 0.1% of the total collected projects, and to find bilingual code fragments implementing the same functions can be even more difficult. Some other approaches [2, 28] require an API-mapping dataset to generate more mappings. However, currently only Java to C# API mapping [7] has such a dataset, which renders these other approaches difficult to utilize to obtain API mapping between other languages, such as Java and Swift.

Existing state-of-the-art approaches have another limitation, in that most generate only coarse API mappings: i.e., parameter binding is not handled or is ignored by the approaches [2, 11, 26, 28], and the many-to-many API mapping pairs contain much redundancy. For example, a many-to-many API mapping pair about reading files produced by DeepAM [11] contains multiple file APIs such as `new`, `open`, `close` and others. For code-migration purposes, this API-mapping set clearly needs to be modified to a less detailed form, such as an API mapping for an `open` operation, which is actually a one-to-one API mapping.

These challenges cannot easily be addressed by code learning-based approaches. Specifically, aside from the difficulty of acquiring sufficient training data and understanding API usage to a high level of detail, even obtaining the binding data types of APIs can be difficult when dealing with dynamic programming languages. For example, we could not extract the binding type information of a Swift API just by statically parsing the source code, as Swift lacks type information on the source level.

To address the above challenges, we carefully examined the API documentation for various programming languages, such as Java, C# and Swift, and observed that the API documentation authors had illustrated the method, the parameter and the return descriptions for an API in detail. Surprisingly, however, this abundant information contained in API documentation has not been fully exploited in documentation-based investigations [17, 34], wherein only the method description has been considered. Therefore, we hypothesize that we can also consider the parameter and the return value information of API documentation, as well as the method description, and thereby obtain the same or better API-mapping results relative to those obtained from code-based approaches. This approach has two advantages, namely that by also considering the parameter and the return value information, inaccurate or incomplete method descriptions may be avoided, and that by consideration of parameter information we can implement parameter mapping.

Thus, in this study we use a deep-dive approach to examine API documentation to achieve one-to-many API mapping. Our approach first involves extraction of the behavior, input and output information for an API from the API documentation. Second, we design a similarity computation to synthesize the total similarity for API mapping and use the Hungarian algorithm [24] to implement parameter mapping. Third, to achieve one-to-many API mapping, we build an API sequence graph (ASG) based on data type, and subsequently design a dynamic searching algorithm to generate target API sequences for a source API.

We use our approach to develop API mapping from Java to Swift, as the current Java to Swift code-migration tools do not offer manually-made API mapping datasets analogous to Java2CSharp [7], despite Java and Swift being popular programming languages, thus meaning that API mapping must be generated. We evaluate our approach on 15 diverse and widely used Java classes, and determine that the top-10 accuracy of the resulting one-to-one API mapping and one-to-many API mapping is 76% and 50% respectively. These are better results than TMAP [34] and SAR [2], and our approach can also obtain accurate parameter binding. In addition, our approach identifies many API mappings with different characteristics, such as one-to-one API mapping with different method names and one-to-many API mapping with different numbers of parameters.

The main contributions of this paper are as follows:

- We thoroughly examine API documentation to extract three-dimensional information for API mapping.
- We develop an approach that enables both one-to-one and one-to-many API mapping, and also the parameter binding.

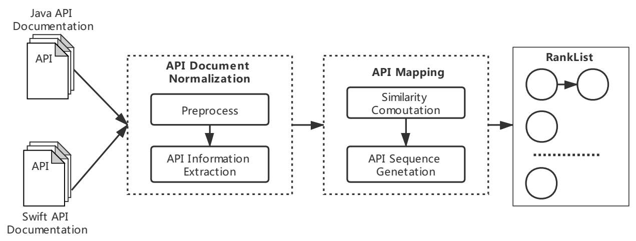

Figure 1: Approach overview

- Our experiment obtains 259 one-to-one and 28 one-to-many API mappings, which also contain the parameter-mapping relation, from Java to Swift. This approach is simpler than existing approaches, and yet gives superior results.

The rest of the paper is organized as follows: Section 2 presents the overview and the details of our approach. Section 3 describes the experimental evaluation. Section 4 discusses limitations and future work. Section 5 discusses related research. Finally, we give our conclusions in Section 6.

2 APPROACH IN ACTION

2.1 Approach Overview

We restrict API mapping to one-to-one and one-to-many in our study. Given a source API S, and a sequence T joined with several APIs of the target language, the $< S, T >$ is called an API mapping if it satisfies three conditions: (1) The S and T implement the same functionality; (2) The mapped parameters[1] represent the same entity; and (3) The final-states of the mapped parameters of S and T are the same, and the outputs of S and T represent the same entity.

Our approach comprises two phases, as shown in Figure 1. The first phase involves building the API information model of the source and target of APIs; this preprocesses API documentation and then extracts API information models. The second phase is to generate the API mapping; thus, a similarity computation is used to synthesize the similarity of an API mapping, and API sequence-generation is then used to manage one-to-many API mapping. Finally, we rank the generated API mappings.

We illustrate our approach, based on the above overview, by using an API mapping example from Java to Swift in Figure 2. For the API mapping $< S,T >$ in the example, S is a Java API `"startsWith(String prefix, int toffset)"` of String class. The T is Swift API sequences consisting of two APIs, which are `"suffix (from start: String.Index) -> Substring"` and `"starts (with possiblePrefix: Sequence) -> Bool"` of String class. API mapping involves returning whether the substring starting at the specified index has the specified prefix. Figure 2 shows part of the documentation of Java API S. There are three key parts to our approach, as outlined below.

The first step is to extract three dimensional information of the behavior, input and output from the informative API documentation. We observe that developers often refer to the method name, parameter list and return value type of an API signature to write behavior, input and output semantic information.

[1]The receiver of an API is considered as an implicit parameter.

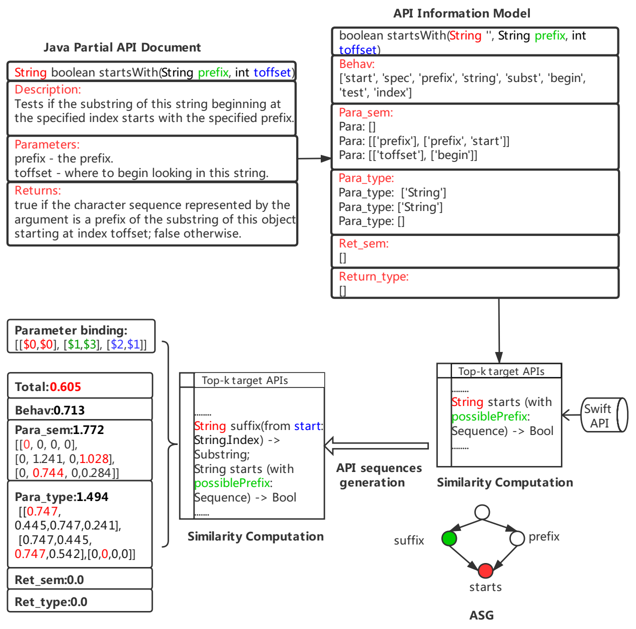
Java Partial API Document
String boolean startsWith(String prefix, int toffset)
Description:
Tests if the substring of this string beginning at the specified index starts with the specified prefix.
Parameters:
prefix - the prefix.
toffset - where to begin looking in this string.
Returns:
true if the character sequence represented by the argument is a prefix of the substring of this object starting at index toffset; false otherwise.
API Information Model
boolean startsWith(String '', String prefix, int toffset)
Behav:
['start', 'spec', 'prefix', 'string', 'subst', 'begin', 'test', 'index']
Para_sem:
Para: []
Para: [['prefix'], ['prefix', 'start']]
Para: [['toffset'], ['begin']]
Para_type:
Para_type: ['String']
Para_type: ['String']
Para_type: []
Ret_sem:
[]
Return_type:
[]
Parameter binding:
[[$0,$0], [$1,$3], [$2,$1]]
Total:0.605
Behav:0.713
Para_sem:1.772
[[0, 0, 0, 0],
[0, 1.241, 0,1.028],
[0, 0.744, 0,0.284]]
Para_type:1.494
[[0.747,
0.445,0.747,0.241],
[0.747,0.445,
0.747,0.542],[0,0,0,0]]
Ret_sem:0.0
Ret_type:0.0
Top-k target APIs
....
String suffix(from start: String.Index) -> Substring;
String starts (with possiblePrefix: Sequence) -> Bool
....
Similarity Computation
API sequences generation
Top-k target APIs
....
String starts (with possiblePrefix: Sequence) -> Bool
....
Swift API
Similarity Computation
suffix
prefix
starts
ASG

Figure 2: An illustrating example

Therefore, we use the API signature as a guide to extract the API information model. API method names encountered in the process of behavioral information extraction provide key information, so we use the names of all the API methods in the documentation as a keyword bag from which behavioral information originates. In Figure 2, the keyword list of the method information is *['start', 'spec', 'prefix', 'string', 'subst', 'begin', 'test', 'index']*. The input information comprises parameter type and parameter semantic information denoted by *"Para_sem"* box and *"Para_type"* box in Figure 2. Recall that the first parameter is the "this" object since the API is not static. When extracting semantic information of parameters, we find that the parameter name generally represents an entity. The parameter descriptions and method descriptions offer other keywords to further explain the meaning of this entity, so each piece of semantic parameter information can be represented by a two-tuple which consists of a parameter name and a modifying word for it. For example, a semantic unit of the parameter `prefix` is *[prefix, start]*. The extraction of the output information is similar to that of the input information. The details of this are given in Section 2.3.

The second step is to compute the similarity between APIs using the API information model. Here, the texts are converted to vector representations, and then the distance measure is used to compute the similarity of two short texts. The Word2Vec [22] is a simple and effective approach for word representation, and its vector representation exhibits a linear structure, so we are able to use simple vector arithmetic to represent a sentence. Thus, we choose Word2Vec and cosine similarity to calculate the similarity of each piece of dimensional information. Finally, we use a weighted summation method to synthesize the total similarity. Parameter mapping is solved using the Hungarian algorithm. In Figure 2, the total similarity is 0.605 and is denoted by the *"Total"* box, and the parameter-mapping relation is obtained, shown in the *"Parameter binding"* box. The details of this are given in Section 2.4.

The third step shows how to obtain the one-to-many mapping. That is, given a source API S, we need to find a reasonable and finite API sequence T in the target API set to ensure it is as equivalent as possible to S. We observe that API call sequences often exhibit data dependency, so we refer to these to build an ASG, and traverse the ASG to search for API sequences. Specifically, we need to determine a finite T corresponding to S. As our approach is to compute the similarity of $< S,T >$, we determine whether to include an API in T by comparing the previous and current API similarity. An API is added to T if including it can lead to an increase in the similarity reaching a threshold. The details are given in Section 2.5.

2.2 Preprocessor

The preprocessor is used to simplify and improve the quality of the subsequent extracted API information.

2.2.1 *Domain Knowledge Preprocessor.*

- **Datatype Normalization**: API documentation comprises redundant or special data types, and these may influence the data-type similarity computation. Therefore, we formulate three rules, based on our observations. The first rule is to collate all specific array data types into an "Array". The second rule is to replace the generic type parameters with the root class. The third rule is to normalize the lambda expression into "Function".
- **Parameter Supplementation**: A non-static API can be seen as having an implicit parameter "this" referring to its owner object. So, if the API modifier does not contain "static", we add the class type of the API into the parameter list.
- **Member Variable Transformation**: A source API may map to a member-variable of the target API. Therefore, we formulate a rule to transform the member variable into an API. Specifically, we take the variable name as the method name and the data type as the return type, and make the parameter list empty.

2.2.2 *Text Preprocessor.* We observe that a word of the API document may be a stop word or an aggregation of several words. Also, the various forms of the same word may influence the similarity computation. Therefore, we design a text-preprocessing algorithm.

This algorithm first parses the given sentence and then removes stop words. Next, it removes digital words and splits the remaining words by a regular expression "[A-Z][^A-Z]+". Splitting a word is very important, as some words in an API document can be split into several words. For example, an API method name is "equalsIgnoreCase" which be split into "equals", "ignore" and "case". Finally, we transform these words into lower case and use Lancaster [32] to stem words. For example, the word "representing" is converted to "repres".

2.3 API Information Extraction

The API signature generally consists of a method name, a parameter list and a return value. API documentation writers usually refer to this information to write the method, parameter and return value descriptions. Therefore, we extract three-dimensional information from the API signature, namely the behavior, input and output information, to build API information models.

An API information model *api* is denoted by a five-tuple $(beh, p_t, p_s, r_t, r_s)$, which is defined as follows:

- The behavior information *beh* represents a functional description of the API. It consists of several keywords.
- The input information consists of the parameter semantic information and the parameter type information, which are expressed as p_s, p_t. We use a dependency relation[2] and a data type set as the basic unit of p_s and p_t, respectively.
- The output information consists of return value semantic information and return type information, which are denoted as r_s, r_t. We use a keyword set to represent r_s and r_t.

2.3.1 Behavior Information Extraction. We find that the method name of the API signature provides information to concisely represent the functional behavior, and we use this to design an algorithm. The algorithm first places all preprocessed API method names into the keyword bag S. Next, it assigns the words of the preprocessed method name and the first sentence of preprocessed method descriptions[3] as the keyword bags W_1 and W_2, respectively, of an API. Finally, it uses the following operation to obtain the behavior information:

$$beh = W_1 \cup (W_2 \cap S).$$

2.3.2 Input Information Extraction. The input information consists of parameter semantic information p_s and parameter type information p_t. We observe that API documentation writers usually write parameter descriptions based on the parameter names of an API signature, with these parameter descriptions providing necessary information to explain parameter names. Therefore, to extract p_s, we represent the specific semantic information with a set of two-tuples, which consist of a parameter name and a keyword from the parameter description or method description. The keywords are determined by dependency relation [19] in NLP such that the parameter name directly depends on the keyword. This ensures selection of the most important words and obviates the need for various complex sentences. For example, Figure 3 shows the dependency relation of the method description of parameter "prefix" of the Java API `"startsWith(String prefix, int toffset)"` of String class. As "prefix" depends on "start", the unit of semantic information of the "prefix" parameter consists of "prefix" and "start". In particular, if the parameter name does not occur in the description, we use the parameter name and the first noun or first verb of the parameter description as the semantic information. To extract p_t, we first refer to the parameter type list in API signature, and then extract additional class type occurred in the parameter description. Considering that the primitive data types contain much less information than the non-primitive data types, we only incorporate the latter that occur in the parameter descriptions and the API signature into p_t.

2.3.3 Output Information Extraction. Output information comprises return type information and return semantic information.

[2]Each dependency relation is a two-tuple.
[3]Javadoc Comments [31] specifies that the first sentence in Javadoc provides good explanations for method's behavior.

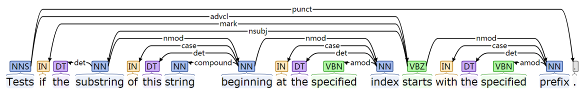
punct
advcl
mark
nsubj
nmod
case
det
det
compound
nmod
case
det
amod
nmod
case
det
amod
NNS IN DT NN IN DT NN NN IN DT VBN NN VBZ IN DT VBN NN .
Tests if the substring of this string beginning at the specified index starts with the specified prefix .

Figure 3: Dependency relation

The extraction algorithm is basically the same as that of the input information extraction 2.3.2. However, to extract the output semantic information we only retain keywords that exhibit a dependency relation with the "return" keyword.

2.4 Similarity Computation

To clarify the similarity computation of a source API api_j and a target API api_s, we first give some basic knowledge.

2.4.1 Vectorization of API Information.

Vectorization of Semantic Information. To generate the word embeddings for API information, TMAP (a documentation-based approach) uses the traditional TF-IDF [18] to vectorize the word. We employ the more modern technique Word2Vec, an unsupervised algorithm, to vectorize the API semantic information of behavior, input and output.

Vectorization of Class Type. The class type is the proper noun, which does not frequently occur in documentation. Therefore, we cannot use the Word2Vec to vectorize these. Fortunately, for each class type, the API documentation provides class description information for all its APIs. It is thus appropriate to use TF [18], a statistical method, to vectorize the class type.

2.4.2 Behavior Information Similarity. As the method descriptions are short texts, we adopt a straightforward method of similarity computation. The similarity is thus computed by averaging the embeddings of all keywords of behavior information for an API method, and then by calculating the cosine similarity between the averaged embedding for two such methods as follows:

$$sim_b = cos(\frac{1}{m}\sum_{i=1}^{m}\alpha(b_j[i]), \frac{1}{n}\sum_{k=1}^{n}\alpha(b_s[k]))$$

where $\alpha(\cdot)$ represents the Word2Vec vectorization, $\cos(\cdot)$ represents the cosine similarity, b_j is the behavior semantic information of a source API, the m is the number of keywords of b_j, and b_s and n of a target API are semantically equal to b_j and m.

2.4.3 Input and Output Information Similarities.

Parameters Binding. We now apply the Hungarian algorithm to the parameter binding. Assuming that we have a complete bipartite graph G, where one vertex set contains parameters of a source API, and the other contains parameters of a target API, an edge thereby means that two parameters exhibit a mapping relation, where its weight corresponds to its similarity of parameter mapping. Thus, we must find the parameter-mapping matrix B that satisfies the maximum-weight matching. Each row of B has two elements, i and j, which means the ith parameter of a source API must be mapped to the jth parameter of a target API. The sim_i is the maximum sum of parameter similarity. We use the $H_{ind}(\cdot)$ function to obtain the B and the $H_{sim}(\cdot)$ function to obtain the sim_i.

Input Information Similarity. The similarity of input information needs to be a combination of the similarity of the input type with the input semantic information. First, we compute the similarity matrices of parameter type A_t and of parameter semantic information A_s. Here, each value in the matrix is computed by the Hungarian algorithm to get the maximum similarity of the two parameters. Then, we use the Hungarian algorithm to obtain the maximum similarity of A_s and A_t. Finally, we average the two similarities to obtain the input information similarity sim_i, and determine parameter mapping B, as shown in the equation 1.

$$\begin{aligned} sim_i &= \frac{1}{n}(H_{sim}(A_t) + H_{sim}(A_s)) \\ B &= \begin{cases} H_{ind}(A_t) & H_{sim}(A_t) \geq H_{sim}(A_s) \\ H_{ind}(A_s) & otherwise \end{cases} \end{aligned} \quad (1)$$

where n is the number of the mapped-parameters whose parameter similarities are not zero.

Output Information Similarity. The computation of output information similarity sim_r is the same as the computation of input information similarity.

2.4.4 Total Information Similarity. After computing the similarities of three dimensional information, we use a weighted summation to compute the total similarity sim as follows:

$$sim = \frac{1}{3}(w_b \times sim_b + w_i \times sim_i + w_r \times sim_r),$$

where sim_b, sim_i, and sim_r represent the behavior, input, and output similarities, and w_b,w_i, and w_r are their weights. The values of the weights are set to 1.4, 1.0, and 0.3, which are obtained by experimental evaluation, as discussed in 3.3.1.

This computation is exemplified in Figure 2, in which the box with "Para_sem" specifies the matrix A_s and the box with "Para_type" specifies the matrix A_t. Here, $A_s(1, 3) = 1.028$, for example, represents the similarity of the second parameter "prefix" of the Java API `startswith` in `String` class and the fourth parameter "possiblePrefix" of the Swift API `starts` in `String` class. According to Equation 1, the input information similarity is computed as $(0 + 1.028 + 0.744 + 0.747 + 0.747 + 0)/4 \approx 0.817$ and the parameter-binding relation corresponds to $B = [(\$0, \$0), (\$1, \$3), (\$2, \$1)]$. The first element $(0, 0)$ in B represents that the first parameter "String" of the Java API is mapped to the first parameter "String" of the Swift API. The box with "Total" represents the total similarity, which is computed by $sim = (0.713 \times 1.4 + 0.817 \times 1.0 + 0 \times 0.3) \times \frac{1}{3} \approx 0.605$.

2.5 API Sequences Generation

For generating one-to-many API mapping, we need to find the target API sequence T that maps the S. This requires two problems to be solved: how to determine if two APIs have connectivity, and how to construct a finite sequence of APIs based on the S. We use the data dependency of APIs to determine their connectivity and use the validity and similarity of APIs to ensure the finity of the API sequence. Specifically, we first construct an ASG based on the data-dependency of APIs. All APIs mostly derive from the top-three target classes, like the class and the parameter types of S. Then, we design a search algorithm to construct T with finity and validity.

Algorithm 1: recursively generate one sequence

```
Input: $G$ - API sequence graph
       $api_j$ - a source API
       $api_s$ - a target API
       $flag$ - the parameter binding flag
       $path$ - a valid API sequence
       $sim_p$ - the parameter similarity
       $pSet$ - API sequences set
1  if $flag$ then
2      for $api$ in $G[api_s]$ do
3          $oldpath$=$path$
4          add $api$ to $path$
5          $p_t$=[],$p_s$=[]
6          for $api$ in $path$ do
7              add the parameter types of $api$ to $p_t$
8              add the parameter semantic information of $api$
                 to $p_s$
9          $oldsim_p$=$sim_p$
10         Use equation 1 to get $B$ and $sim_p$
11         $flag$=IsValid($path$,$B$)
12         if not $flag$ or $sim_p$ - $oldsim_p$ < 0.5 then
13             $pSet$=$pSet \cup oldpath$
14         else
15             GenPath($G$,$api_j$,$api$,$flag$,$path$,$sim_p$,$pSet$)
```

We first provide some basic concepts of this approach, and then describe the search algorithm 1.

2.5.1 Build ASG. For an ASG $G =< V, E >$ where V is the APIs set, $(api_1, api_2) \in E \Leftrightarrow$ there exist intersecting data types between the return type list of the api_1 and the parameter type list of the api_2.

2.5.2 Validity of the Parameter Binding. For a target API sequence T, if there exists an API in T having no parameter mapping with S, then the parameter binding is invalid in T. Otherwise, it is valid. It is denoted by $IsVaild(\cdot)$.

2.5.3 Generate Valid API Sequences. Algorithm 1 is used to generate an API sequence set $pSet$ for a source API api_j. First, we select an initial point api_s from the top-k target APIs of one-to-one API mappings. Then, we add its preceding API api into the API sequence $path$ (line2-4). Second, we orderly merge parameter types and parameter semantic information of each API in $path$ into a new parameter type list p_t and a new parameter semantic information list p_s (line5-8). Third, we compute parameter similarity sim_p and parameter binding matrix B according to equation 1. If the parameter binding is valid and the increment of parameter similarity is >0.5, we repeat the process by using api as the start point (line 15); otherwise we add $oldpath$ to $pSet$ (line 12-13).

2.5.4 Similarity of One-to-Many API Mapping. To generate top-k one-to-many API mappings, we need to compute their similarity with the S. The similarity computation is the same as that used in Section 2.4, but also needs input, output and behavior information.

We thus take the output and the behavior information of the last API in T as the output and the behavior information of T. Then, we orderly combine the parameter type list and the parameter semantic information of each API in T as the input information.

3 EVALUATION

Based on our approach, we conduct experiments to answer the following questions:

RQ1: What is the effectiveness of our approach in mining the one-to-one API mapping and the one-to-many API mapping? Specifically, we consider four metrics: the top-k accuracy of one-to-one API mappings and of one-to-many API mappings, the accuracy of parameter binding, and the characteristics of API mapping pairs.
RQ2: Is it necessary to consider the parameter and return-value information? In other words, is it reasonable that we use the three-dimensional information for implementing the API mapping?
RQ3: How does the effectiveness of our approach compare with that of existing documentation-based and code-based approaches?

3.1 Experimental Setup

All experiments are conducted on two computers: an Intel Core i7-6700, running a Windows OS at 3.40 GHZ with 16 GB RAM, and an Intel Core i5, running a MacOS at 2.3 GHz with 8 GB RAM. The Java and Swift API documentation are from Android Developer [5] and iOS Developer [13], respectively. The Java API is a public method of Android JDK classes, while the Swift API is a member variable or a method of classes in the Swift and Foundation library. As the Swift API documentation does not provide the list of all classes, we consider only the classes declared with keywords "struct|class|enum|protocol".

We select the APIs of 15 Java classes as our subjects. To make sure these are diverse and frequently used, we first crawl 20 Java projects with 500~1000 number of ".java" files based on their starrating on Github [15]. Second, we use the Eclipse JDT compiler [14] to count the Java APIs of each class present in the source code, and then select the top-100 classes, to ensure that the selected APIs are frequently used examples. Finally, we orderly remove interface classes and similar classes such as Integer and Long from the top-100 classes, and then random select 15 different classes, where these data classes involve different data structures, i.e., the utility, the I/O, the time, the system and the throwable, so as to ensure the diversity of APIs.

To make a convincing API mapping dataset, three people with Java and Swift programming knowledge separately use Java API documentation and code example websites such as GeeksforGeeks [39] and Stack Overflow [16] to examine each example of these 15 classes of Java API. Then, each person queries the Swift API documentation and Stack Overflow information to determine if there exists Swift API sequences that could map the Java API. If so, these Swift API sequences are used as much as possible, as well as the corresponding parameter mapping. We then write test cases to amend any inconsistent API mappings that are given by these three people. Next, we distribute the obtained API mapping pairs to 65 graduate students who have more than three years of programming experience to verify the correctness of the dataset once again. Finally, we obtain 259 one-to-one API mappings and 28 one-to-many API mappings.

For the Word2Vec embedding, we adapt Gensim [36] and use a skip-gram algorithm. The word dimension is 100, the window size is 5, the other parameters are the default, and the training corpus consists of all classes, methods, parameters and return descriptions.

Table 1: Result of one-to-one API mapping

#C	#T	top-1		top-5		top-10	
		#M	Acc	#M	Acc	#M	Acc
ArrayList	18	10	0.56	12	0.67	13	0.72
LinkedList	26	13	0.5	20	0.77	20	0.77
HashSet	7	3	0.43	5	0.71	5	0.71
HashMap	13	3	0.23	8	0.62	8	0.62
Calendar	22	8	0.46	12	0.55	15	0.68
Collections	15	5	0.33	10	0.67	13	0.87
Arrays	7	3	0.43	3	0.43	4	0.57
String	41	23	0.56	32	0.78	40	0.98
Integer	28	7	0.25	14	0.50	16	0.57
Throwable	5	1	0.2	4	0.80	4	0.80
Thread	11	5	0.45	7	0.64	8	0.73
Class	11	5	0.45	5	0.45	7	0.64
File	29	3	0.10	11	0.38	21	0.72
PrintStream	23	11	0.48	17	0.74	20	0.87
InputStream	3	2	0.67	3	1.00	3	1.00
Summary	259	102	0.39	163	0.63	197	0.76

3.2 RQ1: Effectiveness of API Mapping

3.2.1 Effectiveness of One-to-One API Mapping. Table 1 shows the result of the one-to-one API mapping. Column "#C" lists class names of Java classes. Column "#T" lists total numbers of Java methods that possess one-to-one API mapping. Subcolumn "#M" lists the total numbers of Java methods that our approach can successfully find in the top-k ranked API mappings. Subcolumn "Acc" lists the top-k accuracy [2]. The last row "Summary" lists the total numbers for columns "#T" and "#M", and gives the corresponding average top-k accuracy. From Table 1, it can be seen that the total number of one-to-one API mappings is 259, the top-1 accuracy is 0.39, the top-5 accuracy is 0.63 and the top-10 accuracy is 0.76. Therefore, our approach is proven to find most one-to-one API mappings.

We also conduct a statistical analysis for undiscovered one-to-one API mappings having "top-10" index rating. Figure 4 gives the proportion of one-to-one API mappings that our approach cannot find (for various reasons) to the total undiscovered number, where the chart label lists the specific reasons for the failure to find a given API mapping. Here, "behav_loss", "input_loss" and "behav&input_-loss" represent behavioral similarity, input information similarity as well as behavioral and input similarity, respectively, which are reasons for correct one-to-one API mappings being ranked lower than the wrong top-10 API mappings, and "one_to_many" represents our approach wrongly finding one-to-many API mappings for Java APIs that should map one Swift API. First, the "beha_loss" is >50%. We carefully inspect these wrong API mappings and find that the Word2Vec cannot distinguish synonyms in the programming

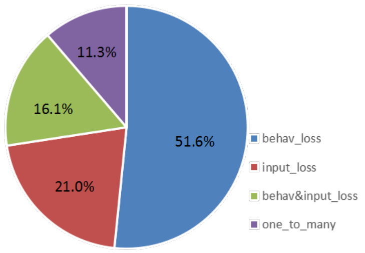

Figure 4: Reasons of undiscovered API mappings

language domain. For example, the Java API "`toString(int i)`" of Integer class means that "`Returns a String object representing the specified integer`", and this should map to the swift API "`description()`" of Int class, which means "`A textual representation of this value`". However, our approach cannot identify the semantic identity of "integer" with "value" and "string" with "textual", and this is the main reason for the accuracy of the Integer Java class being <60%. Second, the "input_loss" is 21.0%, mainly due to the gap between the input semantic information and the data type. Third, the "behav&input_loss" is 16.1%, mainly due to the integrated reasons of "beha_loss" and "input_loss". Fourth, our approach finds one-to-many API mappings for supposed one-to-one API mappings, because addition of a superfluous API leads to an increase in the similarity of input information. This is the main reason for the top-10 accuracy of arrays being <60%.

3.2.2 *Effectiveness of One-to-Many API Mapping.*

Table 2 shows the results of one-to-many API mappings. The column titled "EDR" metric means the difficulty of modifying the wrong API mapping, which is computed as

$$EDR = \frac{\min\limits_{S_T \in Top-k} EditDistance(S_R, S_T)}{length(S_T)}.$$

Here, $EditDistance(S_R, S_T)$ is the editing distance of each API mapping pair consisting of the correct Swift API sequence S_R and the one resulting API sequence S_T, which represents the minimum number of operations developers add/remove a Swift API to transform the wrong API mappings in top-k ranked API mappings into the correct API mappings. There are 28 one-to-many API mappings in 15 Java classes, with a top-1 accuracy of 0.21, top-five accuracy of 0.43 and a top-10 accuracy of 0.5, and thus this approach should be useful. The average "EDR" of top-1 being <0.4, average "EDR" of top-five and top-10 is approximately 0.2, meaning that wrong one-to-many API mappings can be corrected by use of minor operations. Next, we analyze the reasons that our approach fails to find some of the one-to-many API mappings. First, we find that six API mappings are lost because not all top-k APIs that are used to begin searches contain the correct API. Second, there are nine API mappings not found because an API is absent. In the nine incorrect API mappings, there are three Java APIs that have parameters shared by two objects. For example, Java API "`regionMatches(int toffset, String other, int ooffset, int len)`" of String class means "Tests if two string regions are equal", and the last

Table 2: Result of one-to-many API mapping

#C	#T	top-1			top-5			top-10		
		#M	Acc	EDR	#M	Acc	EDR	#M	Acc	EDR
ArrayList	2	0	0	0.489	0	0	0.489	0	0	0.489
LinkedList	4	2	0.5	0.145	3	0.75	0.132	3	0.75	0.132
Arrays	3	0	0	0.715	0	0	0.576	0	0	0.546
String	10	1	0.1	0.374	3	0.3	0.231	5	0.5	0.205
Throwable	2	0	0	0.446	0	0	0.446	0	0	0.446
PrintStream	6	2	0.33	0.394	5	0.83	0.051	5	0.83	0.039
InputStream	1	1	1	0	1	1	0	1	1	0
Summary	28	6	0.21	0.398	12	0.43	0.243	14	0.5	0.238

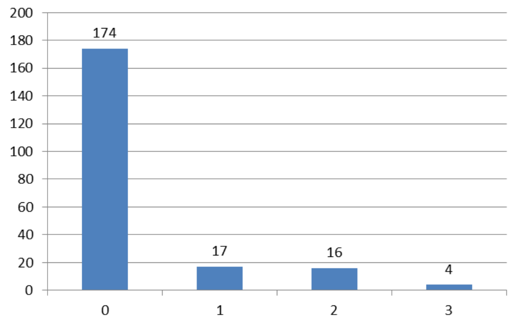

Figure 5: Effectiveness of parameter binding in API mapping

parameter "len" is shared by two strings, which means an API for obtaining a substring of a specified length should be used twice. The absence of the remaining six API mappings is due to the semantic gap in input information; for example, Java API "`valueOf (Array data, int offset, int count)`" of String class means "Returns the string representation of a specific subarray of the char array argument", and its correct API mapping needs a Swift API S "`suffix(Int from, Int start)`" of Array class. However, the low semantic vector similarity of "start" and "offset" leads to a S loss.

3.2.3 *Effectiveness of Parameter Binding.*

Figure 5 shows the top-10 result of parameter binding in all correct API mappings. The horizontal axis lists the numbers of wrong parameter bindings for an API mapping, and the vertical axis represents the corresponding number of APIs. In 211 correct API mappings, the number of totally correct parameter bindings is 174, equating to 82%. One and two wrong parameter bindings are present in 17 and 16 API mappings, respectively, equating to <10%. The maximum number of wrong parameter bindings is three, and occurs in only four instances. Therefore, our approach is effective for parameter binding.

3.2.4 *Characteristic of API Mapping.*

In this section, we explore the characteristics of one-to-one API mappings and one-to-many API mappings from Java to Swift.

Table 3 lists examples of different characteristic one-to-one API mappings. Column "api_attr" represents different characteristics: "dif_p_n" are S that have a different parameter number with T, "dif_m_n" represents the Levenstein distance of method names of S and T being <0.5, "constru" represents T as a Swift constructor API and "var" represents a Java API that should map to the Swift member variable. Column "#T" represents the total number of API mappings with corresponding characteristics and Column "#M" represents

Table 3: Characteristics of API mapping

2 api_attr	#T	#M	%	java_api	swift_api
dif_p_n	97	65	0.67	ArrayList boolean addAll(Collection<? extends E> c)	Array insert<C>(contentsOf newElements: C, at i: Int) where C : Collection, Self.Element == C.Element
dif_m_n	120	78	0.65	HashMap void clear()	NSMapTable removeAllObjects()
constru	28	18	0.64	String static String valueOf(double d)	String init<Subject>(describing instance: Subject)
var	73	56	0.77	File boolean isDirectory()	FileWrapper var isDirectory: Bool { get }
summary	190	134	0.71		

Table 4: Part of one-to-many API mapping

java_api	swift_api
String boolean **startsWith**(String prefix,int toffset) Tests if the substring of this string beginning at the specified index starts with the specified prefix	String **suffix**(from start: String.Index) -> Substring; String **starts**(Sequence with:possiblePrefix, by areEquivalent: (Character, PossiblePrefix.Element) throws -> Bool) rethrows -> Bool where PossiblePrefix : Sequence)-> Bool
String **replaceFirst**(String regex,String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement	NSRegularExpression **firstMatch**(in string: String, options: NSRegularExpression.MatchingOptions = [], range: NSRange) -> NSTextCheckingResult?; NSRegularExpression **replaceMatches**(in string: NSMutableString, options: NSRegularExpression.MatchingOptions = [], range: NSRange, withTemplate templ: String) -> Int
LinkedList boolean **remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present	Array **firstIndex**(of element: Element) -> Int?; Array **remove**(at index: Int) -> Element
FileInputStream int **read**(byte[] b, int off, int len) throws IOException Reads up to len bytes of data from this input stream into an array of bytes	Array **prefix**(through position: Int) -> ArraySlice<Element> ; InputStream **read**(_buffer: UnsafeMutablePointer<UInt8> , maxLength len: Int) -> Int

the number of API mappings that our approach can successfully find. Column "%" represents the percentage, and columns "java_api" and "swift_api" give an example of a corresponding characteristic API mapping. The final row, "summary", gives the sum of Column "#T" and Column "#M", and the average ratio of "%". In 259 one-to-one API mappings, it can be seen that there are 190 API mappings with these characteristics; this is a proportion of 73%, meaning that the API mapping from Java to Swift is complex and diverse. The most common characteristic is "dif_m_n" and "constru" is the least common, but the latter proportion >10%, and thus this is not a truly uncommon characteristic. The lowest and highest proportions of these characteristic API mappings that our approach successfully finds are 64% and 77%, respectively, showing that our approach can effectively perform one-to-one API mapping involving distinct characteristics.

Table 4 lists examples of different characteristic one-to-many API mappings. The first example needs an API for the getting substring, a situation that is widespread in one-to-many API mapping. The second example is about the operation of regular expressions, and is due to the class names of S and T exhibiting a semantic gap. The third example needs an API to obtain the index of an element to complete the remove action, and it thus exhibits a parameter semantic gap. The fourth needs two APIs from different classes to achieve the one-to-many API mapping.

RQ1 Summary: Our approach can effectively find one-to-one and one-to-many API mappings. The parameter binding it also generates is highly accurate. Our approach can also find diverse characteristic API mappings.

3.3 RQ2: Sensitivity Analysis

3.3.1 Learning Combination Weights. To synthesize total similarity based on the similarity of the behavior, input and output information, we need to learn a weight for the three dimensional information. We use the optimization function from [4], and randomly choose 50 API mapping pairs from 259 API mapping pairs to learn the weight.

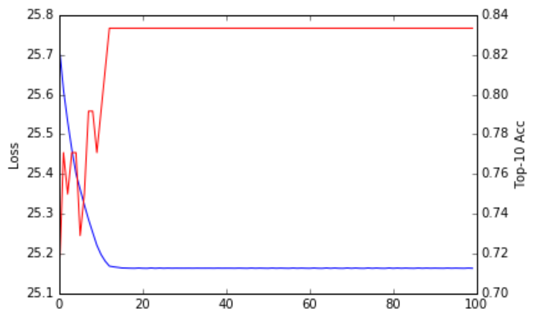

Figure 6: The impact of iteration index on loss and accuracy

Figure 6 shows the value of loss and accuracy as a function of iteration index. It can be seen that the accuracy increases and the loss decreases as the iterations increase. When the iteration index reaches approximately 20, the loss function tends to converge, and the accuracy is also at a steady state: therefore, at this point the optimization is correct. Finally, the weights of behavioral, input and output information are 1.4, 1.0, 0.3, respectively, and the top-10 accuracy of the 259 one-to-one API mappings is 79.5%.

3.3.2 Dimension Information Analysis . As our similarity computation involves the three dimensions of behavior, input and output information, we need to validate the effectiveness of the three-dimensional information.

Table 5 shows the one-to-one API mapping results of involving one dimensional information. The first row represents dimensional information names, the second row represents the corresponding top-10 accuracy, and the third row represents the change in the top-10 accuracy of individual information compared with that of three-dimensional information. From Table 5, we can see that the top-10 accuracy decreases by 14.7%, 45.5% and 54.8% if only behavioral, input and output information is supplied. This result shows that the behavioral information is most important, followed by input information, and then output information. It also shows that the learning weight is reasonable.

Table 5: Result of extracting one-dimensional information

	beh_info	input_info	output_info
Top-10 Acc	0.648	0.340	0.247
change	-0.147	-0.455	-0.548

Table 6: Result of dropping one-dimensional information

	beh_loss	input_loss	output_loss
Top-10 Acc	0.429	0.660	0.784
change	-0.366	-0.135	-0.011

Table 6 shows the result of one-to-one API mapping in the absence of one dimension information. From the table, we can see that the top-10 accuracy decreases by 36.6%, 13.5% and 1.1% if our approach ignores behavioral, input or output information, respectively. Thus, lacking output information has the smallest weight and thus the least effect on top-10 accuracy.

RQ2 Summary: Thus, the three-dimensional information and the learning weights we use are reasonable and effective.

3.4 RQ3: Comparison of One-to-One API Mapping

Previous works mainly focus on one-to-one API mapping, e.g., TMAP and SAR can only handle one-to-one API mapping. StaMiner can generate many-to-many API mapping, however, no enough bilingual projects written in Java and Swift are found to support such an approach. Therefore, following existing literature, we only compare the effectiveness of one-to-one API mapping.

3.4.1 Comparison with TMAP. To the best of our knowledge, we are the first to manage API mapping from Java to Swift. Other API mapping methods typically API-map Java to C#.

TMAP also uses a document-based approach to implement API mappings. We use the source code [33] based on the Apache Lucene [8] and modify part of this code to adapt it to the Swift Programming language. Owing to that TMAP only could handle one-to-one API mapping, our approach does not perform the one-to-many step here. The Table 7 shows the top-10 accuracy of the resulting one-to-one API mappings; overall, the accuracy of every class in our approach is greater than that of TMAP. For example, TMAP finds two more APIs in the Thread class than does our method. However, as TMAP contains more than 10 APIs with the same score and it ranks the top-10 results by documentID, our results are better than those of TMAP. More importantly, our top-10 accuracy is 80%, which is better than the 38% top-10 accuracy of TMAP.

3.4.2 Advantage of the Document Analysis. In recent years, code-based approaches have afforded good results in API mapping. Unfortunately, we find it is difficult to obtain sufficient high-quality datasets for applying these approaches to API mapping from Java to Swift, as briefly outlined below.

From the existing literatures, we found that in top-k accuracy, SAR [2] is better than API2API [28] and StaMiner [26]. For training, SAR uses 174 API mapping pairs with the same signature name; however there is only one API mapping pair for Java to Swift with this characteristic. Therefore, manual API datasets are required.

Table 7: Comparison of one-to-one API mappings between TMAP and our approach

#C	#T	TMAP		Our	
		#M	Acc	#M	Acc
ArrayList	18	12	0.67	15	0.83
LinkedList	26	10	0.38	23	0.88
HashSet	7	2	0.29	5	0.71
HashMap	13	5	0.38	8	0.62
Calendar	22	8	0.36	16	0.72
Collections	15	5	0.33	14	0.93
Arrays	7	1	0.14	6	0.86
String	41	19	0.46	41	1.00
Integer	28	6	0.21	16	0.57
Throwable	5	0	0	4	0.80
Thread	11	10	0.91	8	0.73
Class	11	0	0	7	0.64
File	29	3	0.10	20	0.69
PrintStream	23	16	0.70	20	0.87
InputStream	3	3	1.00	3	1.00
Summary	259	100	0.38	206	0.80

Meanwhile, the top-10 accuracy of our approach is >70% after it learns the weight of 50 API-mapping datasets. However, SAR needs 174 API mapping pairs to do this. Its best result of top-10 accuracy using 174 API mapping pairs is 76%, and our best results is 80%. Therefore, our approach performs slightly better than SAR in terms of top-10 accuracy. Moreover, our work can also manage the one-to-many API mapping but SAR cannot.

In P, R, F metrics, DeepAM [11] achieves the best result in R and F and StaMiner achieve the best result in P. Therefore, we only discuss these two techniques. Both extract API sequences by statistically analyzing source code, but as Swift is a dynamic language, we must dynamically analyze the source code to determine to which class the API belongs. Therefore, we write an automated script to compile Swift projects into type-checked ASTs, and then extract the API class type by parsing the AST.

For the StaMiner-based approach, we reimplement the source code based on the paper [26] and build Groum [29] for the Java and the Swift programming language, where this requires 9 bilingual projects and collects aligning methods according to the similarity of method names. We use the 11 bilingual projects offered by j2sINFERER [1]. The number of aligned methods is 1288, which is much smaller than the thousands of aligned methods in StaMiner. This is due to the relatively small scale of bilingual projects and the fact that some functional code is not yet implemented in Swift projects. After extracting API sequences based on Groum, only 384 aligned API sequences remain because of loss of API call sites or compilation failures. This is far fewer than the 34,628 API sequences of Java-to-C#. For the dataset, StaMiner could only find one correct one-to-one API mapping, i.e., that being the Java API `"valueof"` of String class and the Swift API `"init"` of String class.

DeepAM needs millions of <API sequences and descriptions> pairs. These API sequences are API call sequences of the codes of individual function, and the descriptions are function-level code comments. We crawl 1000 projects for Java and Swift separately based on their star-rating on Github, and find that the numbers of <API sequences, description> of Java projects and Swift projects are 321,469 and 1,674, respectively. We can see that the <API sequence and description> numbers of Java projects are hundreds of times more than for Swift projects. This is due to Swift projects having fewer methods and comments than Java projects, and to the fact that the Swift method lacks API call sites or has existing syntax errors. Therefore, the difference between Java and Swift is too large for processing: i.e., it is impossible to collect the millions of <API sequences and description> pairs to obtain a good result.

RQ3 Summary: Our document-based approach performs better and is simpler than previous document-based and code-based approaches.

4 LIMITATIONS

Although our document-based approach avoids different semantic spaces in different programming languages, our result may be influenced by the quality of API documentation. However, given the development of standardized API documents, we believe that our approach has great promise. Our approach also involves the setting of parameters, and thus even if the multi-dimensional information we use does improve the effectiveness of the API mapping, it faces the problem of weight-selection across three dimensions. However, as well as our approach being able to learn a weight through training, it can also adapt to the API mapping of other programming languages and only require a small dataset to fine-tune the weight to achieve an optimal result. We alleviate possible faults in our API mapping dataset by writing test cases, which are verified by 65 graduate students with more than three years of programming experience. In the future, we will write complete test cases and invite professional Java and Swift developers to check these to ensure correctness.

5 RELATED WORK

There are many related studies of API mapping. These can be divided into three approaches: code-based, documentation-based and code-comment-based approaches.

In terms of code-based works, MAM [44] collects bilingual projects from Java to C#. First, this method finds matching functional code fragments by comparing similarities of method names, and then builds ATG (API Transformation Graph) ATG for these. Second, it compares similarities of node names and types in two ATGs to realize many-to-many API mappings. However, MAM can only find API mappings with relatively high-similarity API method names. StaMiner [26] uses the same dataset as MAM, but, uniquely, uses statistical machine translation to achieve many-to-many API mapping. However, StaMiner usually produces coarse many-to-many API mappings. Rosetta [9] also collects bilingual projects from Java ME to Android and the same functional code fragments, and then uses probabilistic inference to extract likely one-to-two mappings. However, Rosetta only supports API mapping with up to two APIs. The API2API [28] does not need bilingual projects of Java and C#. It uses API embeddings for API usages and then learns a transformation matrix to realize the API mapping. However, it needs many API mapping pairs as the dataset. SAR [2] avoids the prepared API mapping datasets and collects API pairs based on the same API signatures. It uses GAN (generative adversarial networks) to further learn a better transformation matrix T, and then makes refinement to learn an optimal T. However, API2API and SAR can only find one-to-one API mapping and cannot manage parameter mapping. Finally, J2sINFERER [1] can find minor API mappings from Java to Swift because it focuses only on syntactic transformation. Our approach is better than all of the above approaches, as it does not need parallel or large numbers of projects, and it can manage parameter mapping.

TMAP [34] is the main approach similar to ours, in that it is based on API documentation. Thus, TMAP first extracts necessary information to build an Indexer for each API, and then extracts top-k keywords based on TF-IDF to build a Query. Finally, for any API I, the Searcher asks the Query for all APIs having the same keywords as I, and then ranks them according to cosine similarity. Our approach also uses API documentation to realize API mapping, but extracts three-dimensional information for an API, and achieves a better API mapping result than TMAP.

The last group of related works are DeepAM [11] and the approach of Chen et al. [3]. DeepAM first collects a large number of <API sequences and descriptions> as a dataset. Next, it uses a sequence-to-sequence framework to encode API sequences into semantic vectors and then calculates cosine similarity to implement alignment of API sequences. It uses the same algorithm as StaMiner to handle the many-to-many API mapping. However, its many-to-many API mappings are coarse, like those of StaMiner. The method of Chen et al. [3] collects <API sequences, descriptions and method names> as the dataset. It then determines the likelihood of analogical API mappings by combining API usage, name and document similarities. However, this approach can only manage one-to-one API mapping across third-party libraries.

6 CONCLUSION

In this paper, we conduct a deep-dive examination of API documentation to enable Java-to-Swift API mapping. We use a novel, in-depth approach to analyze API documentation to extract three-dimensional information, and use the Hungarian algorithm to implement one-to-one API mapping. Our approach can also find one-to-many API mappings by building API sequence graphs.

Compared to code-based approaches, our document-based approach requires less selective datasets (e.g., no need of bilingual projects or a large number of projects and API mapping pairs) than those required for code-based approaches. Meanwhile, it is also superior to other document-based approaches, as it can achieve one-to-many API mapping and implement parameter-binding, with the latter being ignored by the previous methods. The successful application of our approach to 15 diverse Java classes underscores its effectiveness.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program (Grant No. 2017YFB1001801), the National Natural Science Foundation (Nos. 61690204, 61972193), the Jiangsu Provincial Key R&D Program (No. BE2017004-4), and the Fundamental Research Funds for the Central Universities (Nos. 14380020, 14380022) of China.

REFERENCES

[1] Kijin An, Na Meng, and Eli Tilevich. 2018. Automatic inference of Java-to-swift translation rules for porting mobile applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018.* 180–190. https://doi.org/10.1145/3197231.3197240

[2] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: Learning Cross-language API Mappings with Little Knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, 796–806. https://doi.org/10.1145/3338906.3338924

[3] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong. 2019. Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2896123

[4] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 345–356. https://doi.org/10.1145/2884781.2884786

[5] Google Developers. 2019. Android API Documentation. https://developer.android.com/reference/. Accessed Nov. 4, 2019.

[6] Mohammad El-Ramly, Rihab Eltayeb, and Hisham Alla. 2006. An Experiment in Automatic Conversion of Legacy Java Programs to C#. *IEEE International Conference on Computer Systems and Applications, 2006* 2006, 1037–1045. https://doi.org/10.1109/AICCSA.2006.205215

[7] Alexandre FAU and Christian Mauceri. 2019. Java 2 CSharp Translator for Eclipse. http://sourceforge.net/projects/j2cstranslator/. Accessed Nov. 4, 2019.

[8] The Apache Software Foundation. 2019. Apache Lucene Core. http://lucene.apache.org/core/. Accessed Nov. 4, 2019.

[9] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. 2013. Inferring Likely Mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, 82–91.

[10] 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. https://doi.org/10.1145/2950290.2950334

[11] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-Modal Sequence to Sequence Learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17).* AAAI Press, 3675–3681.

[12] Ahmed E. Hassan and Richard C. Holt. 2005. A lightweight approach for migrating web frameworks. *Information & Software Technology* 47, 8 (2005), 521–532. https://doi.org/10.1016/j.infsof.2004.10.002

[13] Apple Inc. 2019. iOS API Documentation. https://developer.apple.com/documentation/. Accessed Nov. 4, 2019.

[14] Eclipse Foundation Inc. 2019. Eclipse JDT. http://www.eclipse.org/jdt/. Accessed Nov. 4, 2019.

[15] GitHub Inc. 2019. GitHub. https://github.com. Accessed Nov. 4, 2019.

[16] Stack Exchange Inc. 2019. Stack Overflow. https://stackoverflow.com/. Accessed Nov. 4, 2019.

[17] Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. 2017. Learning to Infer API Mappings from API Documents. In *Knowledge Science, Engineering and Management - 10th International Conference, KSEM 2017, Melbourne, VIC, Australia, August 19-20, 2017, Proceedings.* 237–248. https://doi.org/10.1007/978-3-319-63558-3_20

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press.

[19] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations.* 55–60. http://www.aclweb.org/anthology/P/P14/P14-5010

[20] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 353–363. https://doi.org/10.1109/ICSE.2012.6227179

[21] Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). 2013. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* ACM. http://dl.acm.org/citation.cfm?id=2491411

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of 27th Annual Conference on Neural Information Processing Systems. December 5-8, 2013, Lake Tahoe, Nevada, United States.* 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality

[23] Maxim Mossienko. 2003. Automated Cobol to Java Recycling. In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings.* 40. https://doi.org/10.1109/CSMR.2003.1192409

[24] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.

[25] Nguyen, Anh Tuan, Nguyen, Tung Thanh, and Tien N. Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013).* ACM, New York, NY, USA, 651–654. https://doi.org/10.1145/2491411.2494584

[26] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.* 457–468. https://doi.org/10.1145/2642937.2643010

[27] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Migrating code with statistical machine translation. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014.* 544–547. https://doi.org/10.1145/2591062.2591072

[28] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 438–449. https://doi.org/10.1109/ICSE.2017.47

[29] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-Based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09).* Association for Computing Machinery, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

[30] Patrick Niemeyer. 2019. j2swift. https://github.com/patniemeyer/j2swift. Accessed Nov. 4, 2019.

[31] Oracle. 2019. Javadoc Comments. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html#overviewcomment. Accessed Nov. 4, 2019.

[32] Chris D. Paice. 1990. Another Stemmer. *SIGIR Forum* 24, 3 (Nov. 1990), 56–61. https://doi.org/10.1145/101306.101310

[33] Rahul Pandita, Raoul Praful Jetley, Sithu D Sudarsan, and Laurie Williams. 2019. APISIM. https://sites.google.com/a/ncsu.edu/apisim/. Accessed Nov. 4, 2019.

[34] Rahul Pandita, Raoul Praful Jetley, Sithu D. Sudarsan, and Laurie A. Williams. 2015. Discovering likely mappings between APIs using text mining. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015.* 231–240. https://doi.org/10.1109/SCAM.2015.7335419

[35] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. 2017. Statistical migration of API usages. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume.* 47–50. https://doi.org/10.1109/ICSE-C.2017.17

[36] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[37] Harry M. Sneed. 2010. Migrating from COBOL to Java. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania.* 1–7. https://doi.org/10.1109/ICSM.2010.5609583

[38] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. 2012. C to O-O Translation: Beyond the Easy Stuff. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012.* 19–28. https://doi.org/10.1109/WCRE.2012.12

[39] PRAYAGRAJ Uttar Pradesh Public Service Commission(UPPSC). 2019. GeeksforGeeks. https://www.geeksforgeeks.org/. Accessed Nov. 4, 2019.

[40] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* 325–334. https://doi.org/10.1145/1806799.1806848

[41] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019.* 335–346. https://dl.acm.org/citation.cfm?id=3339128

[42] Kazuki Yasumatsu and Norihisa Doi. 1995. SPiCE: A System for Translating Smalltalk Programs Into a C Environment. *IEEE Trans. Software Eng.* 21, 11 (1995), 902–912. https://doi.org/10.1109/32.473219

[43] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing Behavioral Differences in Cross-Language API Mapping Relations. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Rome, Italy, March 16-24, 2013. Proceedings.* 130–145. https://doi.org/10.1007/978-3-642-37057-1_10

[44] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* 195–204. https://doi.org/10.1145/1806799.1806831