Learning to Infer API Mappings from API Documents

Yangyang Lu[1,2], Ge Li[1,2(✉)], Zelong Zhao[1,2], Linfeng Wen[1,2], and Zhi Jin[1,2(✉)]

[1] Key Lab of High-Confidence Software Technology, Ministry of Education, Peking University, Beijing 100871, China
{luyy,lige,zhaozl,wenlf,zhijin}@pku.edu.cn
[2] School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

Abstract. To satisfy business requirements of various platforms and devices, developers often need to migrate software code from one platform to another. During this process, a key task is to figure out API mappings between API libraries of the source and target platforms. Since doing it manually is time-consuming and error-prone, several code-based approaches have been proposed. However, they often have the issues of availability on parallel code bases and time expense caused by static or dynamic code analysis.

In this paper, we present a document-based approach to infer API mappings. We first learn to understand the semantics of API names and descriptions in API documents by a word embedding model. Then we combine the word embeddings with a text similarity algorithm to compute semantic similarities between APIs of the source and target API libraries. Finally, we infer API mappings from the ranking results of API similarities. Our approach is evaluated on API documents of *JavaSE* and *.NET*. The results outperform the baseline model at precision@k by 41.51% averagely. Compared with code-based work, our approach avoids their issues and leverages easily acquired API documents to infer API mappings effectively.

Keywords: API mappings · API similarity · API documents

1 Introduction

To support business requirements of different platforms or devices, software developers often need to release several corresponding versions of their software projects or products. Open source projects, like Lucene and JUnit, often support different versions for Linux, Windows and Mac OS X. Then with the development of mobile devices, application products also release versions corresponding to iOS, Android and Windows Phone. Usually, developers often write code under one platform first, then migrate them from the current platform

© Springer International Publishing AG 2017
G. Li et al. (Eds.): KSEM 2017, LNAI 10412, pp. 237–248, 2017.
DOI: 10.1007/978-3-319-63558-3_20

(annotated as "the source platform") to another (annotated as "the target platform"). Compared with developing different versions independently, it is usually more economical to make the migration shown in Fig. 1. Since the programming languages used in the source and target platforms may be different, this migration process is often called language migration or code migration.

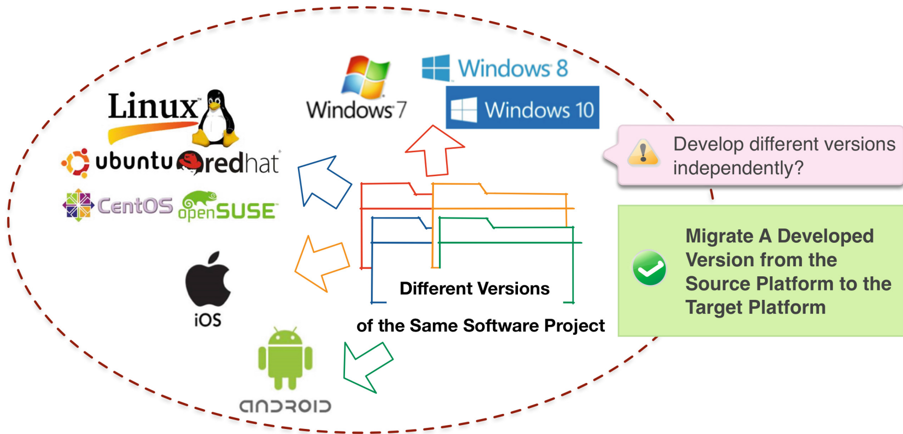

Fig. 1. Migrating software projects from one platform to another platform

During the process of code migration, a key task is to figure out API mappings between API libraries of the source and target platforms [1,4,10]. Here API mappings are defined as the mappings of APIs that belong to different libraries but implement the same or similar functionality. Since modern software development heavily relies on API libraries [9], code migration usually need to replace APIs according to the knowledge of API mappings. For example, *JavaSE* is the basic library for Java projects and *.NET* for C# projects. Then API mappings between the *JavaSE* library and the *.NET* library are important for the migration process of software code from platforms supporting Java to platforms supporting C#.

Since it is time-consuming and error-prone to discover API mappings manually [10], researchers proposed several code-based approaches to mine API mappings automatically. These approaches mostly built parallel code bases from software projects' different versions and took API calling information parsed by static [4,7,8,10] or dynamic [1] code analysis to figure out API mappings. However, they have the issues that there may be no available corresponding versions of software projects and it is usually slow to parse API calling information with the static and dynamic code analysis.

In this paper, we propose a document-based approach to infer API mappings. We find that API documents provide functional information of APIs in API names and descriptions, which can be leveraged to infer APIs of the same or similar functionality. Our assumption is that two APIs have the mapping relation if their

names or descriptions express similar semantics in functionality. So our approach tries to understand the semantics of API names and descriptions first by learning embeddings of words in them. On this basis, we use a text similarity algorithm to compute the semantic similarity of APIs and infer API mappings. Our experiments on *JavaSE* and *.NET* libraries outperform the baseline model at precision@k $(1, 5, 10, 20)$ by 41.5% averagely. Compared with current code-based work, it is easy to access API documents. Then the process of word embedding learning and similarity computation is fast and effective to infer API mappings.

The rest of this paper is organized as follows: Sect. 2 introduces related work; Sect. 3 illustrates our approach to infer API mappings from API documents; Sect. 4 presents details of the dataset and the experiment results; Sect. 5 gives the conclusion and discusses future work.

2 Related Work

The state-of-art work of discovering API mappings mostly takes code-based approaches.

Zhong et al. [10] proposed the MAM model to mine API mappings from different versions of software client code. They first constructed API transformation graphs (ATGs) based on aligned client code snippets, then leveraged heuristic rules to infer API mappings between *JavaSE* and *.NET* libraries.

Gokhale et al. [1] developed a prototype tool named Rosetta to infer API mappings between *JavaME* and *Android* libraries. They crawled mobile graphic applications using the above two libraries and recorded the information parsed by dynamic code analysis to mine API mappings.

Nguyen et al. [4,5] proposed the StaMiner model which takes the task of discovering API mappings as a machine translation task. They also built parallel code bases from different version of client code. Then they extracted API usage sequences from source code with static analysis and applied the IBM translation model to align API usages and infer API mappings.

After the work of StaMiner, Nguyen et al. [6] applied word2vec model on API sequences extracted from Java and C# source code and proposed the API2VEC model to learn APIs' vector representations. Then based on API2VEC's vectors and a large amount of known API mappings between *JavaSE* and *.NET*, they trained a transformation classifier (implemented as a multilayer perceptron) to find API mappings.

In the above code-based approaches, MAM, Rosetta and StaMiner need the alignment relations between method-level code snippets. These relations acquire parallel code bases of different client versions which may be unavailable for parts of projects. Then API2VEC needs a large amount of known API mappings. These approaches also have the issue of high time expense caused by static or dynamic code analysis.

Our work presents a document-based approach to infer API mappings on the basis of understanding words in API documents. API documents are available from official websites of API libraries. Compared with code analysis, it is fast

to processing documents, learning word embeddings and ranking potential API mappings via similarity in our approach. Also, our approach is unsupervised. It can avoid issues of the above code-based approaches and be effective for any two API libraries without available code bases.

3 Approach

3.1 Overview

In API documents, API names and their descriptions provide useful semantic information of APIs' functionality. Because API documents are originally published to help developers understand what functions APIs have implemented. If we could understand the semantics of API names and descriptions, it seems to be feasible to infer API mappings based on their similarity.

Table 1 gives examples of API mappings between *JavaSE* and *.NET* from the literature [4]. We can find that mapped APIs share words in their names and descriptions, such as "io", "exception", "length" and "xml". Besides the same words in API names and descriptions, there are also words of relevant semantics in them, such as "length" and "number", "sequence" and "string". So we proposed the following approach to capture semantic relevance of API names and descriptions based on word embeddings and evaluate API similarity to infer API mappings.

Table 1. Examples of API mappings (left: *JavaSE*, right: *.NET*)

java.io.IOException	*System.IO.IOException*
Exception	Class
Signals that an I/O exception of some sort has occurred	The exception that is thrown when an I/O error occurs
java.lang.CharSequence.length	*System.String.Length*
Methods	Properties
Returns the length of this character sequence	Gets the number of characters in the current String object
org.w3c.dom	*System.Xml*
Package	Namespace
Provides the interfaces for the Document Object Model (DOM) which is a component API of the Java API for XML processing	The System.Xml namespaces contain types for processing XML. Child namespaces support serialization of XML documents or streams, XSD schemas, XQuery 1.0 and XPath 2.0, and LINQ to XML, which is an in-memory XML programming interface that enables easy modification of XML documents

Figure 2 shows the overview of our approach. We crawl raw documents of the source and target libraries from their official website and use HTML parser to extract names, descriptions and types of APIs as the preliminary corpus. Then our approach uses the following four steps to infer API mappings:

- We clean the preliminary corpus with text processing operations, which mainly contain removing analphabetic characters, removing stopwords, splitting words and lowercasing. Specifically, we add a decomposition operation of CamelCase words and package prefixes before lowercasing. This is to dense the semantic space and capture semantic relevance implicated in shared or similar words of API names and descriptions.
- Then we learn word embeddings so that we can capture semantic relevance between words. Here we train a CBOW model [3] on the processed corpus of the source and target libraries. We concatenate the name and description as training sentences, then merge vocabularies of the source and target libraries as one vocabulary.
- Before matching APIs based on the names and descriptions, we need to group APIs based on their types. Because the API in the source library should find candidates with the same type from the target library to avoid wrong mappings and reduce time expense of computation. However, API types in the source library are usually different from the ones in the target library (e.g. "Package" and "Namespace", "Methods" and "Properties" in Table 1). So in this step, we transfer their API types into a unified list with a predefined transfer map. Details are introduced in Sect. 4.1.
- Finally, for a given API in the source library, we take APIs of the same (unified) type as its candidates. Then we compute and rank APIs' semantic similarity based on word embeddings and a text similarity algorithm. Based on the ranking results, the top ones are inferred as potential API mappings.

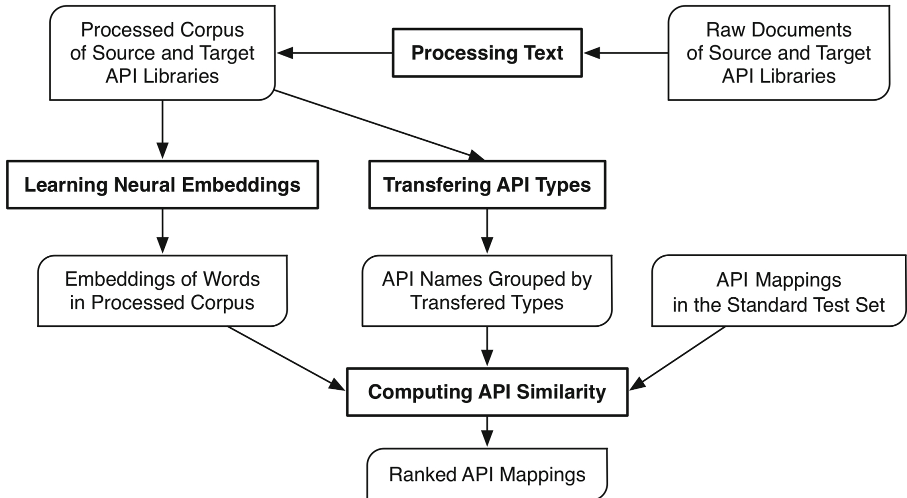

Fig. 2. Overview of our approach

3.2 Understanding API Documents

In this paper, we leverage the CBOW model to learn and represent the semantics of words in API documents. The CBOW model is proposed by Mikolov [3] to learn the language model from the natural language corpus. As shown in Fig. 3, its basic idea is predicting the intermedia word w_t through the previous k words $\{w_{t-k}, \ldots, w_{t-1}\}$ and posterior k words $\{w_{t+1}, \ldots, w_{t+k}\}$ to capture semantic relevance implicated in co-occurrence of words. Its training objective is to maximize the following log-likelihood function:

$$\frac{1}{T}\sum_{t=k}^{T-k} \log p(w_t|w_{t-k}, \ldots, w_{t-1}, w_{t+1}, w_{t+k}) \tag{1}$$

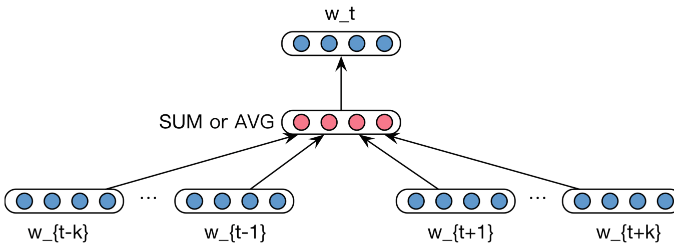

Fig. 3. CBOW model of learning word embeddings

During the training process, we concatenate the name and description of each API to sentences, and merge all the sentences of the source and target libraries together as the training corpus. We expect that the learnt word embeddings by the CBOW model can capture the shared semantics between words in documents of the source and target libraries.

3.3 Computing Similarity Between APIs

On the basis of learnt word embeddings, we take a text similarity algorithm proposed in the literature [2] to compute the similarity between APIs. Here one concatenated sentence of the API name and description is treated as one bag-of-words. The similarity of two APIs A_s and A_t is computed as the similarity of two bag-of-words based on the word-to-word similarity, that is:

$$sim(A_s, A_t) = sim(A_s \rightarrow A_t) + sim(A_t \rightarrow A_s) \tag{2}$$

The directed similarity $sim(A_s \rightarrow A_t)$ is computed by the weighted summation of the similarity between words in A_s and the text A_t. The weights here use IDF (Inverse Document Frequency,) values.

$$sim(A_s \rightarrow A_t) = \frac{\sum_{w_s \in A_s} sim(w_s, A_t) * IDF(w_s)}{\sum_{w_s \in A_s} IDF(w_s)} \tag{3}$$

As for the similarity $sim(w_s, A_t)$ between A_s's word w_s and the text A_t, it is defined as the maximal similarity between word w_s and all the words in A_t.

$$sim(w_s, A_t) = \max_{w_t \in A_t} sim(w_s, w_t) \tag{4}$$

Then the key point of this algorithm is a reasonable measure of the similarity between words. Here we use the cosine similarity between learnt embeddings of words from Sect. 3.2, which is used widely in natural language processing tasks.

4 Evaluation

4.1 Dataset

We use the documents of *JavaSE* and *.NET* libraries to evaluate our approach. The text processing operations have been introduced in Sect. 3.1. Here we present details of transferring API types. Ideally, given an API A_s in the source library, its matching candidate A_t in the target library should have the same type so that mappings with mis-matched types will not be inferred. For example, APIs of the "Package" type should not be matched with APIs of the "Class" type. However, API types of different libraries are usually different, especially when they support different programming languages. Thus we do statistics on API types of *JavaSE* and *.NET* (the "Original" column in Table 3) and predefine a transfer map of API types based on their definitions (Table 2). Finally we get 8 unified API types for these two libraries (the "Transfered" column in Table 3).

4.2 Experimental Settings

We use the Word2Vec module in gensim 0.13.3[1] to implement the CBOW model. Its basic parameters are set as: embedding dimension = 128, the context window = 10 words, training epoch = 200. The training process of CBOW is fast

Table 2. The transfer map of API types

JavaSE		*.NET*	
Original types	Transfered types	Original types	Transfered types
Annotation type	Class	Attached properties	Method
Error	Class	Enumeration	Enum
Exception	Class	Events	Other
Nested classes	Class	Namespace	Package
Enum constants	Other	Properties	Method
Optional elements	Other	Attached events	Other
Required elements	Other	Delegate	Other
		Operators	Other
		Structure	Other

[1] http://radimrehurek.com/gensim.

Table 3. Dataset information of *JavaSE* and *.NET* API documents

	# Sample	Original		Filtered	
		49,150		44,762	
JavaSE	Grouped by API type	Original		Transfered	
		Annotation type	78	Class	3,370
		Class	2,261	Constructor	4,088
		Constructors	4,088	Enum	71
		Enum	71	Field	5,096
		Enum constants	430	Interface	1,026
		Error	33	Method	30,350
		Exception	506	Package	197
		Fields	5,096	Other	564
		Interface	1,026		
		Methods	30,350		
		Nested classes	492		
		Optional elements	113		
		Package	197		
		Required elements	21		
	# Sample	Original		Filtered	
		366,772		366,609	
.NET	Grouped by API type	Original		Transfered	
		Attached events	21	Class	9,005
		Attached properties	80	Constructor	12,464
		Class	9,005	Enum	1,645
		Constructors	12,464	Field	5,332
		Delegate	592	Interface	925
		Enumeration	1,645	Method	306,887
		Events	28,220	Package	95
		Fields	5,332	Other	30,256
		Interface	925		
		Methods	206,815		
		Namespace	95		
		Operators	1,127		
		Properties	99,992		
		Structure	296		

which takes less than 30 min on the dataset of Table 3 in the Core i7 CPU. Then the process of computing API similarity is also quick which outputs results averagely for 5–10 given APIs per minute of the source libraries. The speed is related to the number of candidate APIs with the same type in the target library.

We use the one-hot model as the baseline. Each API is represented as a vector, for which the dimension is the same as the vocabulary size. If the name or description of the API contains the i_{th} word of the vocabulary, then the value at the i_{th} position of the one-hot vector is 1, otherwise 0. API similarity in the baseline is computed by the cosine similarity of one-hot vectors.

The standard test set comes from the literature [4]. After filtering meaningless mappings like "@1", we finally get 145 standard mappings from $JavaSE$ to $.NET$. The inferred API mappings is evaluated by precision@k. Given an API A_s from the source library, if the standard mapped API $A_{s \rightarrow t}$ of the target library is at the top-k results of inferred matched A_t, then we record it as one hit at top-k. The final precision@k is the ratio of the hitting count at top-k to the total number of test mappings.

4.3 Results

Table 4 shows precision@k (k = 1, 5, 10, 20) results of the baseline and our work (embedding dimension = 128, the context window = 10 words, training epoch = 200). It shows that our work outperforms the baseline at all the k-values, which improves precision@k (1, 5, 10, 20) by 61.95%, 47.60%, 34.63%, 21.86% respectively. The average promotion is 41.51%.

Furthermore, we make groups of experiments to analysis the effect brought by training parameters of word embeddings. First, we compare results with different embedding dimensions (Table 5). We find that precision@1 and precision@5 decrease distinctly with the growth of dimension from 128 to 256 then to 512. It shows that 128 dimension is enough for the current corpus to avoid over-fitting. Then we analyze results with different training epochs (Table 6).

Table 4. precision@k(%) of API migration

Model	One-hot	Our work
precision@1	14.48	23.45
precision@5	28.97	42.76
precision@10	35.86	48.28
precision@20	44.14	53.79

Table 5. precision@k(%) with different embedding dimensions

Dimension	128	256	512
precision@1	23.45	23.45	21.38
precision@5	42.76	40.00	40.00
precision@10	48.28	50.34	48.28
precision@20	53.79	53.79	53.79

Table 6. precision@k(%) with different training epochs

Train iteration	200	300	400	500
precision@1	23.45	24.83	25.51	24.83
precision@5	42.76	43.45	42.76	41.38
precision@10	48.28	48.28	48.90	50.34
precision@20	53.79	52.41	53.10	53.79

These experiments use 128 dimension of embeddings. We find that the growth of training epochs after 200 has little effect on the precision@k results.

Finally, we want to figure out the contribution of API names and descriptions in the process of inferring API mappings. We concatenated the name and description of an API as a sample in the above experiments. Here we try to divide these two parts from the corpus. Table 7 shows the statistical information

Table 7. Corpus information of different types

API library	*JavaSE*			*.NET*		
	NAME	DESC	FULL	NAME	DESC	FULL
MinLength	1	1	4	1	1	5
MaxLength	21	84	88	36	80	88
AvgLength	7.14	7.19	14.32	8.82	10.89	19.71
VocabSize	4,250	6,489	7,372	4,960	7,457	8,232
MergedVocabSize	NAME: 6,882;		DESC: 9,588;		FULL: 11,019	

Table 8. precision@k(%) with different kinds of corpus

Corpus	Iteration	precision@1	precision@5	precision@10	precision@20
NAME	200	20.69	32.41	38.62	44.83
	300	20.00	35.17	40.00	45.52
	400	20.69	31.72	41.37	45.52
	500	21.38	34.48	39.21	45.52
DESC	200	7.59	16.55	20.69	28.97
	300	7.59	14.49	19.31	28.28
	400	7.59	15.86	20.69	26.80
	500	7.59	15.17	17.24	28.97
FULL	200	23.45	42.76	48.28	53.79
	300	24.83	43.45	48.28	52.41
	400	25.51	42.76	48.90	53.10
	500	24.83	41.38	40.34	53.79

about different kinds of training corpus. "NAME" and "DESC" means use only API names and only API descriptions respectively. "FULL" means the concatenated corpus of names and descriptions. Table 8 presents precision@k results with different kinds of corpus. We can find that the results only using names is better then the results only using descriptions, but both of them can not achieve the results of concatenated corpus. It shows that the name part contributes most semantics in the process of inferring API mappings and the description part provides supplementary information for this process.

5 Conclusion

Software developers often need to release different versions of projects or products to support different platforms or devices. They usually migrate developed code from one platform to another. API mappings provide the key knowledge acquired in the process.

In this paper, we propose a document-based approach of understanding words in API documents and computing API similarity to infer API mappings. Our approach achieves averagely 41.51% improvement of precision@k ($\mathrm{k} = 1, 5, 10, 20$) to the baseline model. While existed code-based approaches may come across the unavailability of parallel code and high time expense of code analysis, API documents is easy to access and our approach can infer API mappings more quickly by learning word embedding and computing API similarity. Then our approach only leverages weak-supervised knowledge of API types rather than strong supervision of method-level alignment on code or many known API mappings.

As for future work, we hope to measure similarity better with other algorithms or other networks, such as recurrent neural networks or convolutional neural networks. Then the available number of known API mappings for us is too small to provide supervision knowledge. We expect to collect more available API mappings and extend our approach by training supervised modules in the future.

Acknowledgement. This research is supported by the National Basic Research Program of China (the 973 Program) under Grant No. 2015CB352201 and the National Natural Science Foundation of China under Grant Nos. 61421091, 61232015 and 61502014.

References

1. Gokhale, A., Ganapathy, V., Padmanaban, Y.: Inferring likely mappings between APIs. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 82–91. IEEE (2013)
2. Mihalcea, R., Corley, C., Strapparava, C., et al.: Corpus-based and knowledge-based measures of text semantic similarity. In: AAAI, vol. 6, pp. 775–780 (2006)
3. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119 (2013)

4. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: Statistical learning approach for mining API usage mappings for code migration. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 457–468. ACM (2014)
5. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: Statistical learning of API mappings for language migration. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 618–619. ACM (2014)
6. Nguyen, A.T., Nguyen, T.T., Nguyen, T.N.: Migrating code with statistical machine translation. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 544–547. ACM (2014)
7. Nguyen, T.D., Nguyen, A.T., Nguyen, T.N.: Mapping API elements for code migration with vector representations. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 756–758. ACM (2016)
8. Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N.: Exploring API embedding for API usages and applications. In: Proceedings of the 39th International Conference on Software Engineering. ACM (2017)
9. Thung, F., David, L., Lawall, J.: Automated library recommendation. In: Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE 2013), Koblenz, Germany, 14–17 October 2013, pp. 182–191 (2013)
10. Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., Wang, Q.: Mining API mapping for language migration. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 195–204. ACM (2010)