2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Mapping API Elements for Code Migration with Vector Representations

Trong Duc Nguyen
ECpE Department
Iowa State University, USA
trong@iastate.edu

Anh Tuan Nguyen
ECpE Department
Iowa State University, USA
anhnt@iastate.edu

Tien N. Nguyen
ECpE Department
Iowa State University, USA
tien@iastate.edu

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, Enhancement**]: Portability

Keywords

Code Migration, Language Migration, API Mappings, Word2Vec

1. INTRODUCTION

Problem. Code migration between languages is challenging partly because different languages require developers to use different software libraries and frameworks. For example, in Java, Java Development Kit library (JDK) is a popular toolkit while .NET is the main framework used in C# software development. Code migration requires not only the mappings between the language constructs (*e.g.,* statements, expressions) but also the mappings among the APIs of the libraries/frameworks used in two languages. For example, in Java, to write to a file, one can use FileWriter.write of FileWriter, and in C#, one can achieve the same function with StreamWriter.Write of StreamWriter. Such mapping is called *API mapping*.

Due to a large number of API mappings, a manual process of defining those mappings is error-prone and incomplete [22]. To reduce manual effort, several approaches have been introduced to automatically *mine API mappings* from the corpus of the libraries' client code that had two respective versions in two languages [15, 17, 22]. The mined API mappings are useful for both automated migration tools [4, 6, 7, 8, 18, 19, 21] and developers in their code migration. Despite their successes, many mining tools are limited to discover the API mappings with *textually similar APIs' names*. Notwithstanding, in general, the names could be different. For example, to traverse a data list structure, developers use the JDK APIs ArrayList.iterator(), Iterator.hasNext(), and Iterator.next(). The same functions can be achieved in .NET with the APIs List.GetEnumerator(), IEnumerator.MoveNext(), and IEnumerator.Current, respectively. As a consequence, automated migration tools have low accuracy because API mappings are insufficiently defined for them [22].

Related Work. To mine API mappings, MAM [22] uses API Transformation Graphs, which describe the inputs/outputs of API methods and help compare the APIs via their names and calling structures. Twinning approach [17] allows users to specify migration changes to use new APIs. HiMa [10] aggregates the revision-level rules to obtain framework-evolution rules. Aura [20] uses call dependency and text similarity analysis to identify change rules for one-replaced-by-many and many-replaced-by-one methods. Both Aura and HiMa use textual and structural matching. Rosetta [5] uses machine learning but depends on run-time information of pairs of functionally-equivalent applications. It is limited to mapping only graphic APIs. StaMiner [15] uses IBM Model [2] to mine API mappings via an expectation-maximization algorithm. The model requires a training dataset of pairs of corresponding client code of the APIs in two languages. However, building such dataset with parallel implementations in general requires much manual effort.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX,USA

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4205-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2889160.2892661

2. APPROACH AND UNIQUENESS

We introduce a statistical approach with vector representations to mine single API mappings between Java JDK and C# .NET. The idea in our approach has two folds. First, we characterize an API element by how it has been used, *i.e.,* by its usage(s) in the context(s) of surrounding, co-occurring APIs (rather than by its names). Let us use the word *usage relations* to denote such relations among APIs in API usages. For example, each of three APIs ArrayList.iterator(), Iterator.hasNext() and Iterator.next() has its role in an API usage involving a list traversal: 1) one is used to obtain the Iterator, 2) one is for checking if there is more element(s), and 3) one is for getting the next element in the list. We do not aim to detect the role of each API but learn the frequent usage relations with a model that maximizes the likelihood of observing a certain API given its contexts, consisting of the surrounding API elements in usages.

Second, despite that the respective APIs in C# might have different names, each of them would have the same/similar role in the corresponding C# code since they are used to achieve the same functionality. For example, List.GetEnumerator() is for obtaining an iterator; IEnumerator.MoveNext() is for checking; and IEnumerator.Current is for retrieving the next element. Thus, we rely on such similar structures in the roles of APIs to derive API mappings. For example, the usage relation between Iterator.hasNext() and Iterator.next() is the same as the respective APIs in C# IEnumerator.MoveNext() and IEnumerator.Current. If we can learn the usage relations among APIs (*i.e.,* characterizing an API via its surrounding APIs), based on some known API mappings in two languages (*e.g.,*Iterator.hasNext $\leftrightarrow$ IEnumerator.MoveNext), we could train a model to derive other mappings based on the relations of those APIs with others.

Word2Vec Vector Representation. We propose to use Word2Vec vector representation [13] to characterize an API by its surrounding context consisting of other APIs that are used with it in API usage scenarios. Word2Vec vector representation has been shown to be able to capture regularities in natural-language texts. It can char-

acterize a word via its surrounding words. The words being used in similar contexts tend to be mapped into nearby locations along some dimension(s) in the projected continuous space [3]. Several semantic relations among words can be captured via simple vector operations [13]. Specifically, the regularities are observed as similar vector offsets between pairs of words sharing a particular relationship. The syntactic relations that are captured include base/comparative, base/superlative, singular/plural, base/past tense, etc [14]. For example, $V(\text{“}better\text{”})-V(\text{“}good\text{”}) \approx V(\text{“}faster\text{”})-V(\text{“}fast\text{”})$, where V is Word2Vec and the minus sign denotes vector substraction operation. The following types of semantic relationships are also observed: country-capital city, city-state, well-known person's name-profession, company's name-famous product, etc. [11]

We expect that Word2Vec would help characterize an API via its usages and capture its relations with other surrounding APIs. The rationale is that APIs are repeatedly used in API usages. That is, APIs in API usages have high regularities (*i.e.,* repetitive) as shown in API pattern mining research [16, 23]. Moreover, the semantic relations (*e.g.,* data and control dependencies) among those APIs regularly occur, thus, the similar vector offsets between pairs of APIs with some dependency are expected to exist in the vector space. This phenomenon would be observed in both vector spaces for the JDK APIs and .NET APIs. Thus, the vectors for the corresponding APIs in Java and C# are expected to have similar geometric arrangements (Figure 1), which represent the similar structures in the roles of APIs in usages. This similar geometric arrangement phenomenon has been observed for the words for numbers and animals in English and Spanish [12]. Thus, if we learn the transformation (*e.g.,* rotating and/or scaling) between two vector spaces from some API mappings, we can use the learned transformation to locate the vectors for other APIs having relations to those APIs in the known mappings.

Implementation. We use Word2Vec [13] to build vector representations in continuous spaces for JDK and .NET APIs by learning from large code corpora. Specifically, for a method in a training project, we parse and collect the API elements (classes, method calls, and fields), control units (while, for, if), and parameters' types. All those sequences for Java and C# methods are used to train the respective Word2Vec models to build the vectors for Java and C# APIs.

We then use a transformation method [12] to learn the projection between two vector spaces for the APIs in Java and C#. Let us have a set of API pairs and their associated vector representations $\{j_i, c_i\}, i = 1..n$ where j_i is a vector in Java continuous space with d_1 dimensions and c_i is a vector in C# space with d_2 dimensions. We need to find a transformation matrix T such that $T \times j_i$ approximates c_i. We learn T by treating it as an optimization of $\min_W \sum_{i=1}^{n} ||T \times j_i - c_i||^2$. For training, we used stochastic gradient descent on a small set of human-written mappings between APIs in Java and C#. For prediction, for a given API in Java j, we compute $c = T \times j$. The C# API whose vector is closest to c via cosine similarity is the top result. Candidates are ranked via cosine similarity.

3. EMPIRICAL EVALUATION

The first training dataset is for the Word2Vec model for JDK APIs (we chose CBOW architecture in this work [13]). We used the dataset in the work by Allamanis *et al.* [1]. The second dataset is for training the Word2Vec model for the APIs in C# .NET (Table 1). For this, we chose 7,724 C# projects with the ratings of +10 stars in GitHub. For all methods in the training projects, we parsed the code, and built API sequences to train the Word2Vec models.

3.1 Transformation between Vector Spaces

We conducted an experiment in which we picked 3 groups of APIs in Java JDK and the corresponding in C# .NET. Each group consists

Table 1: Datasets for Building Word2Vec Vectors

	#Projects	#Classes	#LOCs	Vocabulary's size
Dataset in Java	14,807	2.1M	352M	103K
Dataset in C#	7,724	900K	292M	130K

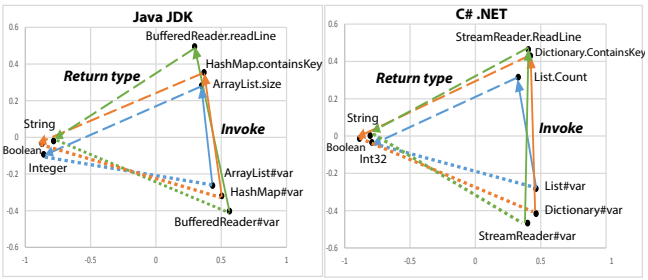

Figure 1: Distributed vector representations for JDK and .NET APIs

the APIs with two relations *method invocation* and *return type*. For example, ArrayList#var (variable declaration) would invoke a method call ArrayList.size and get the return type Integer. The vectors for the corresponding APIs in JDK and .NET in each group were projected into two dimensions using PCA [9] (Figure 1).

From Figure 1, we can observe that Word2Vec captures the regularities of usage relations (method invocation, return type) between those pairs of APIs within one language. This suggests a potential use in API suggestion if we know their types. Moreover, these APIs are projected into the locations with similar geometric arrangements, *i.e.,* Word2Vec captures the similar structures of APIs' roles.

3.2 Deriving API Mappings

Table 2: Top-ranked Accuracy in API Mappings

top-1	top-2	top-3	top-4	top-5
42.8%	59.8%	67.0%	70.7%	73.2 %

In our second experiment, we used a set of 276 pairs of corresponding APIs between Java JDK and C# .NET provided by the rule-based migration tool, Java2CSharp [7]. We used 10-fold cross-validation to evaluate the accuracy of our tool. Nine folds of the pairs of mappings were used to train to determine the projection matrix T. Finally, the matrix was used to derive the ranked lists of the corresponding APIs in C# for the JDK APIs in the testing fold.

We count a result correct if the true API in C# .NET for a JDK API is in the top-k ranked list of resulting APIs for that JDK API. Top-k accuracy is accumulatively computed as the number of correct mining results over the total number of results. As seen in Table 2, for just one suggestion, we are able to correctly derive the API in C# in almost 43% of the cases. With 5 suggestions, we can correctly suggest the C# API in almost 3 out of 4 cases (73.2%).

Our tool is able to detect many pairs of APIs with different names, *e.g.*, java.lang.StringBuffer.delete ↔ System.Text.StringBuilder.Remove, java.lang.System.getenv ↔ System.Environment.GetEnvironmentVariable, java.util.Map ↔ System.Collections.Generic.IDictionary.

In conclusion, we introduce a new approach to mine API mappings via vector projection without requiring a large parallel corpus.

4. ACKNOWLEDGMENTS

This work was supported in part by the US NSF grants CCF-1518897, CNS-1513263, CCF-1413927, CCF-1320578, CCF-1349-153, TWC-1223828, CCF-1018600, and CCLI-0737029.

5. REFERENCES

[1] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of 10th Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013.

[2] P. F. Brown, V. J. Della Pietra, S. A. Della Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311, June 1993.

[3] L. Deng and D. Yu. *Deep Learning Methods and Applications – Foundations and trends in signal processing*, 7:197–387. June 2014.

[4] DMS. http://www.semdesigns.com/Products/DMS/DMSToolkit.html.

[5] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 82–91. IEEE CS, 2013.

[6] Java to C# Converter. http://www.tangiblesoftwaresolutions.com/Product_Details/Java_to_CSharp_Converter.html.

[7] Java2CSharp. http://sourceforge.net/projects/j2cstranslator/.

[8] Microsoft Java Language Conversion Assistant. http://support.microsoft.com/kb/819018.

[9] I. Jolliffe. *Principal Component Analysis*. Springer-Verlag, USA, 1986.

[10] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 353–363. IEEE, 2012.

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[12] T. Mikolov, Q. V. Le, and I. Sutskever. Exploiting similarities among languages for machine translation. *CoRR*, abs/1309.4168, 2013.

[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13)*, pages 3111–3119, 2013.

[14] T. Mikolov, W.-T. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[15] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 457–468, New York, NY, USA. ACM, 2014.

[16] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of Conference on the Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009.

[17] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 205–214, New York, NY, USA. ACM, 2010.

[18] Octopus.Net Translator. http://www.remotesoft.com/octopus/.

[19] Sharpen. https://github.com/mono/sharpen.

[20] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: A hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 325–334, New York, NY, USA. ACM, 2010.

[21] XES. http://www.euclideanspace.com/software/language/xes/userGuide/convert/javaToCSharp/.

[22] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204. ACM Press, 2010.

[23] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.