SAR: Learning Cross-Language API Mappings with Little Knowledge

Nghi D. Q. Bui
School of Information Systems
Singapore Management University
Singapore
dqnbui.2016@phdis.smu.edu.sg

Yijun Yu
School of Computing & Communications
The Open University
UK
y.yu@open.ac.uk

Lingxiao Jiang
School of Information Systems
Singapore Management University
Singapore
lxjiang@smu.edu.sg

ABSTRACT

To save effort, developers often translate programs from one programming language to another, instead of implementing it from scratch. Translating application program interfaces (APIs) used in one language to functionally equivalent ones available in another language is an important aspect of program translation. Existing approaches facilitate the translation by automatically identifying the API mappings across programming languages. However, these approaches still require large amounts of *parallel* corpora, ranging from pairs of APIs or code fragments that are functionally equivalent, to similar code comments.

To minimize the need for parallel corpora, this paper aims at an automated approach that can map APIs across languages with much less *a priori* knowledge than other approaches. Our approach is based on a realization of the notion of *domain adaption*, combined with code embedding, to better align two vector spaces. Taking as input large sets of programs, our approach first generates numeric vector representations of the programs (including the APIs used in each language), and it adapts generative adversarial networks (GAN) to align the vectors in different spaces of two languages. For better alignment, we initialize the GAN with parameters derived from API mapping seeds that can be identified accurately with a simple automatic signature-based matching heuristic. Then the cross-language API mappings can be identified via nearest-neighbors queries in the aligned vector spaces. We have implemented the approach (**SAR**, named after three main technical components, Seeding, Adversarial training, and Refinement) in a prototype for mapping APIs across Java and C# programs. Our evaluation on about 2 million Java files and 1 million C# files shows that the approach can achieve 48% and 78% mapping accuracy in its top-1 and top-10 API mapping results respectively, with only 174 automatically identified seeds, which is more accurate than other approaches using the same or much more mapping seeds.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338924

CCS CONCEPTS

• **Computing methodologies** → *Artificial intelligence*; • **Software and its engineering** → *Language features.*

KEYWORDS

cross-language API mapping, Generative Adversarial Network, domain adaptation, word embedding, vector space alignment

ACM Reference Format:
Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: Learning Cross-Language API Mappings with Little Knowledge. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338924

1 INTRODUCTION

Migrating software projects from one language to another is a common and important task in software engineering. To support the process, various migration tools have been proposed. A fundamental challenge faced by such tools is to translate the library APIs of one language to functionally equivalent counterparts of another. Often, much manual effort is required to define the mappings between the respective APIs of two languages.

Several studies have addressed this API mapping problem, such as MAM [39], StaMiner [21], DeepAM [13], and Api2Api [26]. MAM [39] and StaMiner [21] require as input a large body of parallel program corpora, which contain functionally equivalent code that use APIs in both languages, in order to mine the mappings. Thus, they rely heavily on the availability of bilingual projects that implement the same functionality in two or more languages, which is not easy to find for any pair of languages. Although they rely on similar function names to reduce manual effort needed to identify parallel data, many functions with similar names may be actually functionally different, degrading the quality of training data and final mapping results. DeepAM [13] maps API sequences to sequences based on the text descriptions for the sequences. Its intuition is that two API sequences across languages may be mapped to each other if their text descriptions are similar. This approach does not need API mapping seeds, but requires many similar text descriptions across programs written in different programming languages whose availability can affect the mapping results. Api2Api [26] uses a vector space transformation method inspired by Mikolov et al. [17], but it still requires many API mapping seeds from an external source (Java2CSharp [5])) to map APIs across languages.

In this paper, we propose an approach that can map APIs across languages while alleviating the shortcoming of existing approaches. We realize that the underlying goal of state-of-the-art techniques is essentially to find a transformation that can align two different domains (in our context, the two vector spaces for APIs in two different languages). Api2Api [26] is also an instance of this idea to learn an optimal transformation matrix between two vector spaces while requiring much parallel training data. However, empirical evidence of existing approaches suggest that collecting the training data is an expensive process that requires either availability of manual inspection or high-quality documentations. This has led to the following research question we aim to answer in this paper: "*Can a model be built to minimize the need of parallel data to map APIs across languages?*".

We realize that the API mapping problem may be addressed by techniques based on generative adversarial training [10] with the assistance of a pre-trained model. Given large code bases in two languages, it is likely that certain similarities between the code bases can be exploited to discover APIs of similar functionality across languages, without manually specifying parallel corpora. Such knowledge of similar functionalities may not be big enough for a complete mapping model, but it is small enough to afford human validation. Once validated, the knowledge can be *transferred* through *adversarial training* techniques to maximize the alignment between the two languages which results in better API mappings.

Our approach for API mapping works in the following way: (1) it takes in a large number of programs in two languages, and generates a vector space representing code and APIs in each language via a word embedding technique adapted from previous studies [3, 13, 21, 26, 39]; (2) it adapts domain adaption techniques [6, 9, 10] to transform and align the two vector spaces for the two languages, with mainly three technical components: Seeding, Adversarial training, and Refinement; and (3) it utilizes nearest-neighbors queries in the aligned vector spaces to identify the mapping result of each API. We name our approach **SAR**, after the three main technical components in the domain adaption step.

We have implemented the approach in a prototype tailored for Java and C#, and evaluated and compared it with the state-of-the-art techniques, such as StaMiner [21], DeepAM [13] and Api2Api [26]. We have evaluated the prototype on a dataset of more than 14,800 Java projects containing approximately 2.1 million files and 7,800 C# projects containing approximately 958,000 files. Our evaluation results indicate that the approach can achieve 54% and 82% accuracy in its top-1 and top-10 API mapping results with only 174 automatically identified seeds, more accurate than other approaches using the same or much more mapping seeds. In addition, we also identify about 400 more API mappings between the Java and C# SDKs than other approaches.

The main contributions of this paper are as follows:

- We propose SAR, a new approach based on domain adaption techniques to transform and align different vector spaces across languages with the assistance of a seeding, adversarial learning, and refinement method. To the best of our knowledge, we are the first to apply the adversarial training techniques for the API mapping task.
- We adapt the adversarial training techniques in a number of ways to improve its alignment of the vector spaces: (1) we use nearest-neighbor queries to identify possible mapping candidates for better alignment; (2) we use a similarity-based model selection criteria and reduce the need of known API mappings during the training of our model; and (3) we use the Procrustes algorithm to find the exact solution of the mapping matrix.
- We have implemented the approach and evaluated it with a corpus containing millions of Java and C# source files; via an extensive empirical evaluation on different components of our approach, we demonstrate its advantages against other API mapping approaches in producing more accurate mappings with much fewer seeds that can be automatically identified.

The rest of the paper is organized as follows. Section 2 discusses studies in the literature closely related to this paper; Section 3 presents the background about vector space mapping and adversarial learning; Section 4 presents our approach in detail; Section 5 evaluates our approach to demonstrate its effectiveness and discuss its limitations; and Section 7 concludes with possible future work.

2 RELATED WORK

This section briefly reviews related work on cross-language program translation and relevant techniques.

Cross-Language Program Translation. For the problem of cross-language program translation, much work has utilized various statistical language models for tokens [23], phrases [15, 24, 25], or APIs [4, 8, 21, 22, 28, 38, 39]. A few studies also used word embedding for API mapping and migration (e.g., [12, 13, 26, 28, 35]), but our work does not need large number of manually specified parallel corpora or mapping seeds. Tools for translating code among specific languages in practice (e.g., Java2CSharp [5]) also often dependent on manually defined rules specific to the grammars of individual languages, while our approach alleviates the need of language-specific rules.

MAM [39] and StaMiner [21] rely on the availability of bilingual projects that implement the same functionality in two or more languages. DeepAM [13] requires many similar text descriptions across programs written in different programming languages whose availability can affect the mapping results. Api2Api [26] requires many API mapping seeds from Java2CSharp [5]) to map APIs. The idea of our approach is most similar to Api2Api, while we combine seed-based and unsupervised domain adaptation techniques to reduce the need of mapping seeds.

Relevant Techniques. For the techniques used to represent, model, learn source code, many studies exist for building various statistical language models of code for various purposes in recent years [1]. When it comes to what models to use for code, there is still much room for improvement. Hellendoorn et al. [14] showed that simpler code learning models (e.g., n-gram) with caches of code locality and hierarchy may outperform complex deep neural network models. While other studies (e.g., [13, 15, 22]) demonstrate that more grammatical and semantic code features at various levels of abstraction can be useful for more accurate models. These studies provoke us to perform code embedding with structural information, and in future to explore more semantic information for code embedding.

However, existing studies using domain adaption techniques for API mapping and translation still require the creation of mapping seeds [26, 28].

To transform vector spaces, studies in NLP on sentence comparison and translation involve variants of bilateral models to align the contents [33], but they require parallel corpora in two languages. Recent progresses in domain adaption alleviate the need of parallel corpora [6, 9, 10]. In an application to image learning, domain adaptation through GAN has shown benefit to transfer the models from other dataset as pre-training models when training on smaller dataset [32], which provides the technical foundation for our work.

3 BACKGROUND

The goal of domain adaptation is to produce a mapping matrix as an approximation of the similarities between vectors in the two spaces. This section gives a brief overview of two methods for domain adaptation: seed-based (supervised) or unsupervised. Apart from the two input vector spaces, the seed-based method also requires a set of seeds as the parallel training data to learn the matrix, while unsupervised method does not: the mapping matrix can be obtained through adversarial learning assuming that similarity exists between the distributions of vectors in the two spaces.

3.1 Seed-based Domain Adaptation

Given two sets of embeddings have been trained independently on monolingual data, seed-based domain adaptation is to learn a mapping using the seeds *s.t.* their translations are close in a shared vector space. Such an idea has been explored for word translation in NLP [17], and Api2Api [26] adapts it to learn API mappings.

Formally, given two vector spaces, $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, containing n and m embeddings for two languages L_1 and L_2, and a set S of seeds of API embedding pairs $\{(x_{s_i}, y_{s_i})\}_{s_i \in \{1,|S|\}}$, we want to learn a linear mapping W between the source and the target space, such that Wx_{s_i} approximates y_{s_i}. In theory, W can be learned by solving the following objective function:

$$W^* \triangleq \underset{W \in M \subset \mathcal{R}^{d \times d}}{argmin} ||WX_S - Y_S|| \quad (1)$$

where d is the dimension of the embeddings; $M \subset \mathcal{R}^{d \times d}$ is the space of $d \times d$ matrices of real numbers; $X_S \triangleq \{x_{s_i}\} \subset X$ and $Y_S \triangleq \{y_{s_i}\} \subset Y$ contain the embeddings of the APIs in the seeds, which are matrices of size $d \times |S|$.

Instead of approximating a solution using traditional stochastic gradient descent method used in Api2Api [26], there exists an analytical Procrustes problem [30] solved by Xing et al. [34], which has a closed form solution of the mapping matrix derived from the singular value decomposition (SVD) of YX^T:

$$W^* \triangleq argmin_W ||WX_s - Y_s|| = UV^T, with\ U\Sigma V^T = SVD(Y_s X_s^T) \quad (2)$$

The advantage of a closed form solution is that one can get the exact solution which is better than the approximate solution of gradient descent, and is faster in computation.

With the mapping matrix W, one can use $y_x = Wx$ to map a query vector x. The vector y_x is the mapping, or adaptation, of x in the target space.

3.2 Unsupervised Domain Adaptation

Adversarial learning has been successfully used for domain adaptation in an unsupervised manner. In particular, the Generative Adversarial Network [10] achieves this goal by a model which comprises a generator and a discriminator as two inter-playing components. A generator network that aims to learn real data distribution and produce fake data to fool the other component, so-called the discriminator; the discriminator network that acts as a classifier, which aims to distinguish the generated fake data from the real data. The two components are trained in a minimax fashion and would converge when the generator has maximized its ability to generate fake data so similar to the real data that the probability for the discriminator to make a mistake would be $\frac{1}{2}$.

Conneau et al. [6] use this idea as a variant for the machine translation task, which achieves significantly better results than other baselines of machine translation, which would require no parallel data to train the networks. The generator, in this case, is a mapping matrix W, which can simply be seen as a set of parameters that need to be learned, and the discriminator is a feed-forward neural network. We want to find a matrix W as an approximation of the mapping between the two vector spaces X and Y. In the adversarial learning setting, we aim to optimize two parameters: one is the discriminator's parameters, denoted as θ_D, the other is the mapping matrix W. Our goal is to find the optimal value of two sets of parameters, which results that we have two objective functions in the adversarial learning setting.

Discriminator objective. Given the mapping W, the discriminator (parameterized as θ_D) is optimized by this objective function:

$$L_D(\theta_D|W) = -\sum_{i=1}^{n} logP_{\theta_D}(source = 1|Wx_i) - \sum_{i=1}^{m} logP_{\theta_D}(source = 0|y_i) \quad (3)$$

where $P_{\theta_D}(source = 1|v)$ is the probability that a vector v originates from the source embedding space (as opposed to an embedding from the target space).

Mapping objective. Given the discriminator θ_D, the mapping W aims to fool the discriminator's ability of predicting the original domain of an embedding by minimizing this objective function:

$$L_W(W|\theta_D) = -\sum_{i=1}^{n} logP_{\theta_D}(source = 0|Wx_i) - \sum_{i=1}^{m} logP_{\theta_D}(source = 1|y_i) \quad (4)$$

Learning Algorithm. The discriminator θ_D and the mapping W are optimized iteratively to minimize L_D and L_W, respectively by following the training procedure of adversarial networks proposed by Goodfellow et al. [10]

4 OUR APPROACH

Combining the virtues of seed-based and unsupervised adversarial methods described in the background, our domain adaptation approach can approximate two spaces of vectors with minimal parallel corpora. Although unsupervised adversarial learning method does not require any seed as parallel data, the distributions of vectors (i.e., embeddings) in the two spaces may not be similar. Therefore, it is our hypothesis that the performance could be improved by initializing the unsupervised adversarial learning method with a

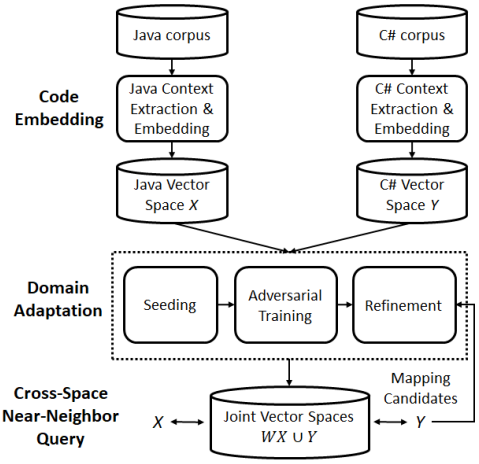

Figure 1: Approach Overview

small set of seeds taken from the seed-based domain adaptation, and by generating the rest of API mappings in the two steps below:

- From large code corpora in two different languages, we create two vector spaces for APIs by adapting word embedding technique for code. From such corpora, we derive a small set of mappings based on a simple text similarity heuristic (see Code Embedding in Figure 1);
- The two vector spaces, along with the mapping seeds, are transformed by a mapping matrix to get aligned with each other. This step comprises three sub-steps: Seeding, Adversarial Learning, and Refinement (see Domain Adaptation in Figure 1).

For any given API a in the source language and its continuous vector representation x, we can map it to the other domain space by computing $y_x = Wx$. Then, one can find the top-k nearest neighbors of y_x in the target vector space, using cosine similarity as the distance metric, and finally can retrieve the list of APIs in the target language that has the same embeddings as the top-k nearest neighbors. The list of APIs can then be used as the mapping results for a (see Cross-Space Near-Neighbor Query in Figure 1).

4.1 Code Embedding via Word Embedding

We first parse source code files into Abstract Syntax Trees (ASTs) using `fAST` [37] for both Java and C# projects. We convert each AST to a sequence of tokens by traversing the AST in its preorder. Through the traversal, we can identify the API token by checking the type of the node (e.g., function call nodes). We perform the normalization step to enrich the code sequence with structural information extracted from parsing, which constitutes two steps:

Filtering out unnecessary tokens: Once obtained the token sequence, we filter out tokens that are not necessary for our task, such as operators and primitive variable identifiers. Language keywords and AST node types are still kept for code embedding as they can enrich the structural information of the sequence.

Converting raw API tokens into signatures: This step reduces the variance of vocabulary existing in the source code. For example, one may extract the 'List.add' method from the 'java.util.List' class, or from the 'com.google.common.collect.List' in an external third-party library. Even though these two APIs have the same class and method names, their usages and semantics are different.

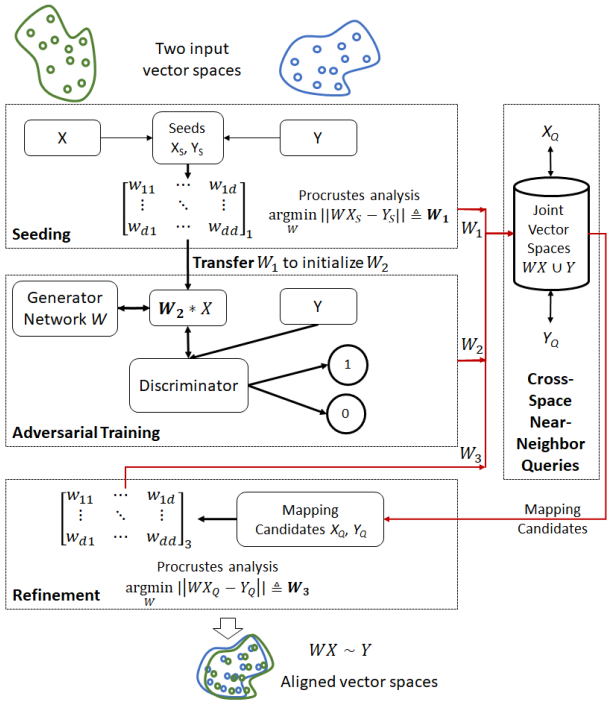

Figure 2: Domain adaptation steps to align two vector spaces

To handle such cases, we propose this additional step to convert a raw API token to its signature in qualified name format 'Package.Class.Method'. The 'Package' is identified by using the 'import' statements (Java) or 'using' statements (C#).

Below shows an example of the normalization for the code token sequence:

```
float f List.add List.add if List.addAll else HashMap.put return
==> float java.util.List.add java.util.List.add if java.List.addAll
else java.util.HashMap.put return
```

From the corpora of code sequences, we use the skip-gram model [16] to train the embedding of tokens. Given a large corpus as the training data, the tokens appearing in the same context would usually have their embeddings close by distance in the vector space.

4.2 Domain Adaptation

Our domain adaptation comprises three steps: seeding, adversarial training, and refinement (hence the abbreviation SAR of our approach). Seeing SAR from outside as a black-box, it receives two vector spaces and a set of seeds as input and generates a mapping matrix W as output. Internally, each step of SAR is a different way to improve the mapping matrix, which receives the matrix output from the previous step as input and produces the improved version of it as output. We assign W_1, W_2 and W_3 as the output matrix for the three steps, respectively. Figure 2 summaries the domain adaptation procedure. The rationale for each step is described as follows: (1) **The Seeding** step to initialize a mapping matrix between the two vectors spaces based on some prior knowledge (i.e., seeds) (2) **The Adversarial Learning** step to re-use the knowledge learned from the Seeding step as an initializer for adversarial training in order to maximize the similarity between the two vector spaces

(or two distributions); and (3) **The Refinement** step to make the mapping matrix reach its optimal state.

4.2.1 Seeding. After Code Embedding, two vector spaces are obtained to produce a mapping matrix that approximates the two vector spaces by using the knowledge from mapping seeds in a dictionary. Notice that by simple signature-based comparison to identify APIs having the same signature name [1], one can identify many high-quality mapping candidates to be used as the seeds without any human effort to verify because developers often use the same name for the same functionality even when they are in different languages.

Having the set of seeds obtained, in addition to the two vector spaces X and Y, the initial mapping matrix produced is W_1 by solving the Equation 1 in Section 3 (also see Seeding in Figure 2). This seeding step can be seen as a function A, which will receive these three inputs and produces a transforming matrix W_1 such that $W_1 = A(X, Y, D)$. Internally, A solves the optimization problem described in Section 3 given the three inputs.

4.2.2 Adversarial Learning. The quality of the matrix W_1 learned in the previous step is limited by the number of seeds one can provide, which results in an approximation between the source and target domains. In this case, the knowledge learned for W_1 can be seen as a pre-trained model and can be reused for the other model.

Formally, given the two original vector spaces, $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, containing n and m API embeddings obtained from the Code Embedding step, we want to find the matrix W_2 to *maximize* the approximation of the mapping between the two vector spaces. We use adversarial learning to achieve this goal, which comprises of two steps: the mapping matrix W_2 and a discriminator network as described in Section 3.2. Our goal is to find the optimal value of W_2 and θ_D (discriminator parameters) We achieve this by training the adversarial network with the objective functions as described in Section 3.2 to find W_2 and θ_D.

The key difference with the general adversarial setting described Section 3.2 is that we do not initialize W_2 randomly as one usually does when training a neural network. Instead, we use W_1 as a pre-trained model to initialize for the W_2 so that W_2 is initialized with some good knowledge, even if it is small (see Adversarial Learning in Figure 2). This step is essential to improve the performance of the API mapping results.

Model Selection Criteria. In short, this step is for choosing the optimal parameters for the Adversarial Learning step, although the heuristic used is similar to the Refinement step. To train the adversarial networks, like any other neural network architecture, we need a validation set to select the best model for the prediction step. The validation set is used to minimize over-fitting when training the neural network. Concretely, for each training epoch, one needs to evaluate against the validation dataset to pick the model that has the highest validation accuracy through training. Our goal is to use as little parallel data as possible to build the model. In practice, one only has a very small number of seeds inferred from the signature-based matching, or in the worst case, one cannot infer any seed to have data for validation. As such, it is impractical to use a parallel dataset as a validation set to train neural networks in the adversarial learning step, i.e., involving additional prior knowledge.

To address this issue, we perform a model selection using unsupervised criteria that quantify the closeness of the source and target embedding spaces. Specifically, we consider them as a set of K most frequent source APIs and multiply them with the mapping matrix W to generate a target mapping for each of them. After that, we get a set of mappings, then compute the average cosine similarity of these mappings and use the average as a validation metric.

4.2.3 Refinement for Better Alignment. The adversarial approach tries to align all words irrespective of their frequencies. However, rare tokens have embeddings that are less updated in the back-propagation step and are more likely to appear in different contexts in each corpus, which makes them harder to align [6]. To address this problem, we use the method proposed in [6] to infer a list of mapping candidates using only the most frequent tokens. Moreover, other heuristics are introduced to infer another candidate set of mapping based on the threshold of cosine similarity, which can be used as another synthetic dictionary that can combine with the top-K frequency mapping candidates.

Following the step shown in [6], it is possible to build a set of mapping candidates using W_2 just learned with adversarial training. Assume that one can induce a combined set of mapping candidates from different heuristics above, and the quality of the combined set is good, then this set of candidates should be used to learn a better mapping and, consequently, an even better set of candidates for the next iteration. The process can repeat iteratively to obtain a hopefully better mapping and candidates set each iteration until some convergence criteria are met. Formally, the refinement step receives W_2 from the previous adversarial learning step, along with the two original embeddings X and Y to produce the next W_3 iteratively (see Refinement in Figure 2).

Specifically, we produce the mapping candidates for refinement based on two heuristics:

Top-K Frequency: Conneau et al. [6] shows that by taking the top-k frequent words and their nearest neighbors in the transformed vector spaces, it can provide high-quality mapping candidates because the most frequently used words are likely to be the same across languages. Therefore, we can use the top-k frequent API names to induce the seeds for the refinement.

Cosine Similarity Threshold: Since finding API mappings in the aligned vector space is essential to finding APIs close enough in the vector space, all API pairs "similar enough" in the vector space aligned by Adversarial Learning can be good candidates for the refinement step. In this work, we use the cosine similarity as the metric to measure how similar two vectors are. We note that *not* all APIs in a language can have a mapping in another language. In the empirical case study, we show how a good threshold is found in Section 5.3.2.

Therefore, we can infer two sets of synthetic mapping candidates from the above heuristics. In fact, there are different ways to merge them into one single set as they can overlap as, e.g., (1) the union of the two sets, (2) the intersection of the two sets. In contrast to Conneau et al. [6], we use an additional Cosine Similarity Threshold heuristic to get a better set of mapping candidates.

[1] "Same signature" in our case means case-insensitive matches of the class and method names

Table 1: Example of Seeds from the Signature-based Matching Heuristic

Java	C#
java.lang.**String.equals**	System.**String.Equals**
java.util.**List.remove**	System.Collections.Generic.**List.Remove**
java.util.**Random.nextDouble**	System.**Random.NextDouble**
java.lang.**Math.round**	System.**Math.Round**
java.io.**File.Exists**	System.IO.**File.Exists**

The matrix W_3 in this step is the final output of the domain adaptation process. When it comes to the step to produce the mapping from the source query, the embeddings of the query will be multiplied with W_3 in order to obtain corresponding mappings in the target language.

5 EMPIRICAL EVALUATION

We have conducted extensive empirical evaluations on our approach in various settings to answer the following research questions:

RQ1 Compared to related methods, is our approach more effective in identifying API mappings?

RQ2 How well do different combinations of refinement heuristics improve the performance?

RQ3 What is the impact of each component in our approach on the performance?

5.1 Dataset

We use the Java Giga corpus data described by Allamanis et al. [2]. It involves approximately 14,807 Java projects from Github and contains approximately 2.1 millions of files. For C#, we clone the projects on Github that have at least 1 star and collect 7,841 C# projects with about 958,000 files. As the main advantage of our approach, there is no need to specify which code in Java is functionally equivalent to which code in C#. For each function in a file, we traverse the AST of the function to extract the API call sequences. For Java, we get a corpus containing 6.7 million code sequences; for C#, we get a corpus containing 5.1 million code sequences.

For evaluation, we take 860 method API mappings and 430 class API mappings defined in Java2CSharp [5] as the ground truth for evaluating our approach against the baselines.

5.2 Implementation

We adapt `Gensim` [29] in NLP to produce the embeddings of tokens for the Java and C# corpora. We use the same settings used by Mikolov et al. [18] during the training: stochastic gradient descent with a default learning rate of 0.025, negative sampling with 30 samples, skip-gram with a context window of size 10, and a subsampling rate of value $1e^{-4}$.

5.2.1 Evaluation Metrics. We use three metrics to measure the performance of our approach and the baselines.

Top-k Accuracy: The top-k accuracy is defined as follow: For a test JDK API j, SAR produces a resulting list. If the true mapping API in C# .NET for j is in the top-k resulting list, we count it a hit. If not, we count it a miss. Top-k accuracy is computed as the ratio between the number of hits and the total of hits and misses for a given ground-truth test set.

Mean Reciprocal Rank: For a test JDK API j as a query, SAR produces a resulting list, we calculate the Reciprocal Rank (RR) of that query. For all queries in our evaluation data, we calculate the Mean Reciprocal Rank (MRR) of the test set. MRR is the average of the reciprocal ranks of results for a sample of queries.

Formally, Reciprocal Rank can be defined as: $RR = \frac{1}{rank_i}$, where $rank_i$ refers to the rank position of the first relevant mapping for the i-th query. And the Mean Reciprocal Rank (MRR) can be defined as $MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} RR_i$, where RR_i refers to the Reciprocal Rank for the i-th query, $|Q|$ refers to the total number of queries.

F-score: The F-score is defined as $F = (2 * P * R)/(P + R)$, where Precision $P = TP/(TP + FP)$ and Recall $R = TP/(TP + FN)$. TP refers to the number of true positives, which is the number of API mappings that are in both result dataset and the ground truth set; TN refers to the number of true negatives, which is the number of API mappings that are neither in the returned results nor in the ground truth set; FP refers to the number of false positives which represents the number of result mappings that are not in the ground truth set; FN refers to the number of false negatives, which represents the number of mappings in the ground truth set but not in the results.

5.2.2 Code Embedding. From the two code corpora, we scan through all pairs of APIs in the two corpora to produce a set of seeds using the signature-based matching heuristic. We got 257 seeds for this step. Table 1 shows examples of the seeds. Among these 257 seeds, we found that 83 seeds overlap in the 860 ground truth mappings. Note that for a fair comparison with the baselines (Api2Api, DeepAM, StaMiner), we remove these 83 overlapping seeds from the 257 seeds and we get a set of **174 seeds for the Seeding step**. Then, we apply word embedding on the corpora to get the source embedding and target embedding. We use the embeddings, along with the seeds as the input for the domain adaptation process.

5.2.3 Domain Adaptation. For the seeding step, we find W_1 by using the Procrustes solution in Equation 2 with three inputs: source embedding X (Java), target embedding Y (C#), and 174 mapping seeds. This step gives us the mapping matrix W_1. We implement the adversarial learning by using `PyTorch` [27]. We use Momentum Gradient Descent method [31] to search for the optimal transformation matrix[2].

We use the unsupervised model selection criteria proposed in Section 4.2.2 to select the best model by choosing the top 1000 frequent API token pairs, e.g., top-1 frequent token in the source is aligned with top-1 frequent token in the target as the validation set, then we extract the W_2 from the model. Figure 3 shows three different lines: (1) the discriminator accuracy, which is the accuracy in classifying the samples from the source and target embeddings, (2) the API mapping accuracy, which is the accuracy when using the model to evaluate against the 1000 pairs validation set, and (3) the average cosine similarity of all the pairs. As shown, the criteria correlate well to the mapping accuracy. The high instability is because of over-fitting. Thus, we only selected the model of the best validation accuracy.

From W_2 resulting from the adversarial training, we obtain the final W_3 by performing the refinement step on the basis of two heuristics in Section 4.2.3. For the top-N frequency heuristics, we

[2]Our implementation (including source code and a docker image) can be accessed at the public repository: https://github.com/bdqnghi/SAR_API_mapping

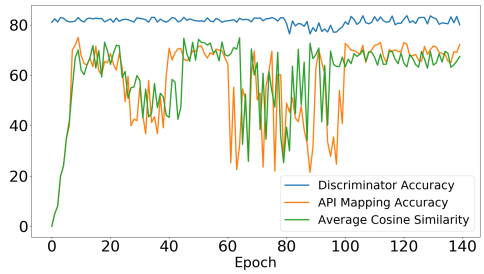

Figure 3: Unsupervised Model Selection Criteria

Table 2: API Mapping Results

Index	Baselines	K-folds	Seeds	Top-1	Top-5	Top-10
1	Random seeds: Api2Api	-	0	0.03	0.05	0.1
2		-	10	0.09	0.12	0.14
3		-	50	0.14	0.19	0.22
4		-	100	0.19	0.24	0.32
5	Random seeds: SAR	-	0	0.25	0.30	0.35
6		-	10	0.28	0.35	0.40
7		-	50	0.26	0.43	0.47
8		-	100	0.44	0.50	0.69
9	K-Fold: Api2Api	1-fold	172	0.24	0.35	0.41
10		2-folds	344	0.34	0.45	0.55
11		3-folds	516	0.37	0.51	0.67
12		4-folds	688	0.43	0.64	0.72
13	K-Fold: SAR	1-fold	172	0.36	0.39	0.48
14		2-folds	344	0.45	0.50	0.61
15		3-folds	516	0.54	0.66	0.71
16		**4-folds**	**688**	**0.59**	**0.77**	**0.84**
17	Signature-based: Api2Api	-	25	0.12	0.16	0.18
18		-	50	0.20	0.23	0.29
19		-	100	0.27	0.32	0.38
20		-	174	0.31	0.41	0.60
21	Signature-based: SAR	-	25	0.30	0.32	0.39
22		-	50	0.35	0.39	0.45
23		-	100	0.41	0.50	0.63
24		**-**	**174**	**0.48**	**0.71**	**0.78**

choose top-500 frequent tokens for the synthetic dictionary, as suggested in [6]. For the second similarity threshold rule, we use 0.7 as the threshold as shown in Section 5.3.2, we found that this number balances coverage and precision of API mappings well.

5.3 Evaluation

5.3.1 RQ1. Effectiveness of SAR in Mining API Mapping. The first question we want to answer is how effective our approach in identifying API mappings from the two vector spaces. We compare SAR with Api2Api, StaMiner, and DeepAM.

Result Summary. Index 24 in Table 2 uses 174 API mappings automatically selected by the signature-based matching heuristic and test against the 860 ground truth mappings. Index 16 uses 688 mappings selected randomly from the 860 ground truth set and test against the rest. The performance of SAR in terms of top-k accuracy is shown. As one can see in both cases, the top-1 accuracies are above 50%, and the top-10 accuracies are above 80%.

Compare to Api2Api. The method used in Api2Api is corresponding to the seeding step in our domain adaptation process, which finds a mapping matrix by solving the Equation 1 given a large set of seeds. We use the top-k accuracy as the evaluation metric.

Table 2 shows the top-k accuracy of our approach when comparing to Api2Api in various settings. First, we compare Api2Api with SAR using the seeds coming from two different sources: the 860 mappings defined by Java2CSharp and the 174 mappings inferred from the signature-based matching. Here we described the variances as results shown in Table 2, indicating that our approach can use *much* fewer number of seeds compared to Api2Api but still achieve better results.

Select randomly: we select a subset of mappings r randomly from 860 mappings in the ground truth, and test against the rest $860 - r$ mappings. Concretely, $r = 0$, 10, 50, and 100;

k-fold: we divide the 860 mappings into $k = 1, 2, 3, 4$ folds and perform the variants of five-fold cross-validation: while k folds are used as training data, the other $5 - k$ folds are used as testing data;

Select by signature: we use 174 mappings inferred by method signature, and select randomly a varying number of them as the training data and test against the remaining mappings in the ground truth.

The process repeats for using different folds as the training data for both Api2Api and we take the average accuracy are some observations from the results:

- Using the same number of seeds, either using the seeds from Random, K-fold or Signature-based, we get significantly better results than Api2Api for every setting.
- When using all of the 174 signature-based seeds, our approach gets significantly better results than Api2Api: top-1 improves 17%, top-5 improves 30%, and top-10 improves 18%.

We also compare with Api2Api by using MRR as the evaluation metric. For a JDK API in the 860 ground truth mappings, we use it as a query for SAR to produce a resulting list, then we calculate the RR for the first relevant mapping in the list. We do this for all of the JDK APIs in the 860 ground truth mappings and calculate the MRR of the test JDK APIs. We then do the same for Api2Api and get an MRR score. The MRRs for SAR and Api2Api are 0.67 and 0.43, respectively, which indicates that when given a query, SAR can retrieve the mapping results more accurately than Api2Api.

Compare to StaMiner and DeepAM. We follow the details described in StaMiner and DeepAM to measure how well SAR performs in mining API mappings for Class API and Method API [3]. In Java, an API element, by definition, can be a class, a method or a field in the class; and it must belong to a package (or the namespace in case of C#). As such, the goal in this task is to measure the performance the Class and Method API mapping task one by one for each API of each package, i.e., to see which package has the best performance for API mappings, so-called 1-to-1 mappings. For the method API mapping, we use the 860 method ground truth mapping described in Section 5.1 for evaluation. For the class API mapping, we use the 430 class ground truth mapping described in Section 5.1 for evaluation. We follow the details described in DeepAM to choose only the APIs under the packages as shown in Table 3, column 'Package', so that the total number of method API mapping left is 289 (remaining from 860 ground truth method API mappings), and the total number of class API Mapping 283 (remaining from 430 ground truth class API mappings).

Adapting SAR for class-level API mapping is relatively easy: one can remove the method part of a qualified API signature token so that only the package and class parts of the token are retained in

[3]Note that we still use the model without the overlapping seeds

Table 3: Accuracy of 1-1 Mapping when compares with StaMiner and MAM

Package	Class Migration									Method Migration								
	Precision			Recall			F-Score			Precision			Recall			F-score		
	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR	Sta	DeepA	SAR
java.io	70.0%	80.0%	80.0%	63.6%	75.0%	75.0%	66.6%	72.7%	72.7%	70.0%	66.7%	66.7%	64.0%	87.5%	82.9%	66.9%	75.2%	74.8%
java.lang	82.5%	80.0%	82.5%	76.7%	81.3%	80.2%	79.5%	80.7%	82.6%	86.7%	83.3%	81.5%	76.5%	87.2%	78.4%	81.3%	85.4%	84.4%
java.math	50.0%	66.7%	66.7%	50.0%	66.7%	66.7%	50.0%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%
java.net	100.0%	100.0%	100.0%	50.0%	100.0%	100.0%	66.7%	100.0%	94.5%	100.0%	100.0%	80.0%	33.3%	100.0%	66.7%	50.0%	100.0%	81.7%
java.sql	100.0%	100.0%	100.0%	50.0%	100.0%	90.0%	66.7%	100.0%	95.0%	100.0%	50.0%	70.0%	50.0%	66.7%	70.0%	66.7%	57.2%	70%
java.util	64.7%	69.6%	81.3%	71.0%	72.7%	71.0%	67.7%	71.1%	79.7%	63.0%	64.3%	64.8%	54.8%	85.7%	85.0%	58.6%	73.5%	76.9%
All	77.9%	82.7%	**85.0%**	60.2%	**82.6%**	80.5%	66.2%	81.9%	**83.1%**	**81.1%**	71.9%	71.7%	57.6%	**82.3%**	78.38%	65.0%	**76.3%**	75.5%

Table 4: Examples of newly found APIs in Java and C#

Java	C#
java.io.DataInputStream.readInt	System.Io.BinaryReader.ReadUInt16
java.lang.Byte.parseByte	System.sbyte.Parse
java.lang.Double.longBitsToDouble	System.BitConverter.Int64BitsToDouble
java.net.Datagramsocket.isConnected	System.Net.Sockets.Socket.Connect
java.awt.geom.AffineTransform.inverseTransform	System.Drawing.Drawing2d.Graphicspath.Transform
java.io.DataInputStream.readDouble	System.Io.BinaryReader.ReadDouble
java.net.Serversocket.accept	System.Net.Sockets.Socket.AcceptAsync

the code sequences. Then code embedding for the API sequence can be derived as the embedding of the class-level API, along with other keywords from the ASTs. We do this for both languages. To select mapping seeds by API signatures, we first infer the mappings from signatures at the class level, then follow a similar domain adaptation process from APIs at the method-level.

One could not run StaMiner and DeepAM directly because they require parallel data (aligned function body for StaMiner, and aligned code and text description for DeepAM) for training. Therefore, we had to compare to them by extracting the reported performance numbers from their papers. This is also how DeepAM compared itself to StaMiner. We use the F-score as the performance metric to measure accuracy in this evaluation

Table 3 shows the comparison results of our mined API mappings with StaMiner (Sta) and DeepAM (DeepA). Columns "Class Mapping" and "Method Mapping" list results of comparing API classes and methods, respectively. As one can see for the F-score, SAR produces better results than those of DeepAM and StaMiner at the class level. At method level, SAR produces better results than StaMiner, and is close to DeepAM in term of F-score. Note that while DeepAM needs to use millions of similar API sequence descriptions and StaMiner needs to use ten of thousands of pairs of parallel data, SAR only uses 174 pairs of mappings as a small set of parallel data for the Seeding step.

Newly found API mappings. More interestingly, we found more new API mappings than other studies in our actual code corpora. For each of the API in Java, we query the top-10 nearest neighbors in C# and manually verify the mappings. We enforce the threshold = 0.7 as mentioned in Section 5.3.2 for this task. We found 420 new SDK API mappings that can complement the tool Java2CSharp. Comparing to MAM (25 new mappings), StaMiner (125 new mappings), Api2Api (52 new mappings), we found a sufficiently larger number of mappings and our newly found APIs also overlap with the APIs in these baselines. In Table 4, we show some interesting examples of such newly found API mappings whose name do not match exactly using traditional approaches. Our list of newly found Java/C# APIs mappings can be accessed at our Github repository.[4]

5.3.2 RQ2. Effect of Different Refinement Approaches.

[4] https://github.com/bdqnghi/SAR_API_mapping/blob/master/new_found/new_found_apis.csv

Table 5: Accuracy of the filtered mappings using various similarity thresholds

Threshold	Coverage		Accuracy	
	Top-1	Top-5	Top-1	Top-5
0.6	0.66	0.90	0.42	0.59
0.7	0.45	0.68	0.51	0.73
0.8	0.12	0.22	0.65	0.80
0.9	0.08	0.15	0.78	0.89

Effects of Cosine Similarity Threshold. In this section, we measure the effect of different ways to combine the seeds for the refinement step. We want to measure the effects of cosine similarity threshold in order to choose a good one for the second heuristic in the refinement step. Since the threshold is a part of the refinement, the domain adaptation step only comprises of two steps: Seeding and Adversarial Learning. Once the threshold is found, we use it for the Refinement in the other experiments. Then we produce the mapping for each source query in the 860 ground truth mappings. For each mapping produce, we obtain the cosine similarity between the query and the result mapping. We choose a threshold to filter out the mapping that has the cosine similarity lower than the threshold, then we measure the accuracy of the filtered mappings.[5]

In Table 5, the column "Coverage" means the percentage of ground truth APIs that have mappings in the candidate selection results when choosing a specific cosine similarity threshold. The column "Accuracy" means the top-k accuracy in identifying the mapping given a cosine similarity threshold as a condition to identify. The results show that our approach in these experiments has higher mapping accuracy, but lower coverage with respect to the ground truth set when the similarity threshold increases. It is, therefore, a trade-off to have higher accuracy in the expense of coverage. For the other experiments that involve the cosine similarity threshold in the refinement, we choose 0.7 as the threshold as this number is balanced between the coverage and the accuracy.

Effects of Different Combinations of Refinement Heuristics. Obtained 0.7 as a good threshold to identify correct mappings, we use this number for the "Cosine Similarity Threshold " heuristic in the Refinement step. What we measure is the impact of the two refinement heuristics on the performance, either using only one of them or combine them together. The domain adaptation also comprises of Seeding and Adversarial Learning. After Adversarial Learning, we use different combinations of refinement heuristics to measure the effect of each heuristic. We use the 860 ground truth mappings from Java2CSsharp as the test set.

[5] Note that the number of filtered mappings can be different when using different cosine similarity threshold to filter out the mappings in the query results whose similarity is less than the threshold. For the whole SAR approach, we do not apply the filtering for more strict evaluation.

Table 6: Different ways to combine refinement heuristics

Refine Method	Top-1	Top-5	Top-10
Top-K	0.44	0.68	0.76
Cosine	0.25	0.31	0.36
Union Top-K + Cosine	0.36	0.42	0.50
Intersection Top-K + Cosine	0.48	0.71	0.78

Table 7: Ablation Study – effects of each component

Baselines	Seeds	Top-1	Top-5	Top-10
Seeding+Adv	25	0.30	0.39	0.45
	50	0.32	0.40	0.54
	100	0.39	0.53	0.71
	174	**0.43**	**0.67**	**0.75**
Seeding+Refine	25	0.13	0.14	0.20
	50	0.18	0.23	0.29
	100	0.22	0.29	0.43
	174	0.30	0.40	0.49
Seeding	25	0.11	0.15	0.18
	50	0.18	0.22	0.28
	100	0.20	0.27	0.36
	174	0.29	0.36	0.42
Refine	-	0.01	0.01	0.01
Adv+Refine	-	0.29	0.34	0.40
Adv	-	0.25	0.30	0.35

The results in Table 6 show that taking the Intersection between the Top-K Frequency and the Cosine Threshold heuristic results in the best performance. This implies that the Cosine Threshold has an effect to filter out poor Top-K Frequency synthetic seeds, thus making the refinement better in overall.

5.3.3 RQ3: Effect of Each Component. We performed an ablation study of domain adaptation to measure the performance of individual components as well as their combinations (Table 7). Note that for the Refinement component, since Section 5.3.2 shows that using the intersection of Top-K and cosine threshold leads to better results than union, we refer Refinement to those of "Intersection of Top-K and Cosine" performance.

Here are some observations from the results:

- The seeding step is an important step for the domain adaptation to works well, e.g., even with a small set of seeds (25), which is a very small knowledge, it sets up a basis for the adversarial learning to improve the performance significantly.
- Adversarial Learning is essential in improving the performance, e.g., comparing Seeding+Adv against Seeding, the top-1 accuracy is improved by 14% on average.
- Refinement alone does not achieve any good result because the initial input matrix was completely random that cannot be refined to anything better;
- Using the Adversarial Learning alone achieves some reasonable results, e.g., top-1 = 25%, top-10 = 35%. Further with the Refinement step, top-1 improves to 29%, top-10 becomes 40%. These can be seen as the results of unsupervised domain adaptation without any initial seeds.

5.4 Explainability Analysis of the Results

We performed various explainability analyses of our model in varying configurations to obtain some insights about our method. From the results, we show that our approach performs significantly better than Api2Api in every perspective. An interesting question one may ask is "*why does this approach perform better than Api2Api*?". Although theoretically, Adversarial Learning maximizes the similarity between two distributions, it is still useful to explain this phenomenon using analysis of the results.

Table 8: Effect of Refinement on Frequent vs. Rare Tokens

Baselines	% Ground truth	Eval size	Accuracy		
			Top-1	Top-5	Top-10
With Refine	Top 10%	86	0.65	0.78	0.85
	Bottom 10%	86	0.32	0.35	0.47
Without Refine	Top 10%	86	0.54	0.65	0.72
	Bottom 10%	86	0.30	0.34	0.45

5.4.1 Effect of Refinement on Frequent vs Rare tokens. We note that the frequency of an API token could affect the quality of the mapping result, i.e more frequent tokens could affect performance more than the less frequent ones. With this assumption, the Refinement of the mapping matrix tries to improve the mapping by using frequent tokens as the anchor. To measure the effect of the refinement on the frequent tokens and rare tokens, we ranked the 860 ground truth mappings in Java2CSharp by the frequency of the source APIs, i.e., the Java JDK APIs. Then we use our model to produce the mapping results against the top 10%, which is a subset of frequent tokens; and bottom 10%, which is a subset of rare tokens. Note that to ensure a fair comparison, we use the 174 non-overlapping seeds to train the domain adaptation procedure.

The results in Table 8 show the following observations:

- Mapping accuracy decreases while increasing top-k frequent tokens in the evaluation set, in either setting. This implies that token frequency does affect on the mapping result;
- The refinement step can improve the result of both the frequent tokens and rare tokens, although the impact is bigger on frequent tokens, e.g., improved by 10% for top-10% , and only 2% for bottom-10%.

5.4.2 Retrieved Results Comparison. To evaluate our approach qualitatively, we retrieved C# API methods from sample queries in Java SDK. Table 9 shows the resulting top-5 C# APIs for four queries: *java.util.Collection.add*, *java.io.File.exists* *javax.swing.Text.JTextComponent.setCaretPosition*, and *java.util.concurrent.atomic.AtomicInteger.getAndDecrement*. They are ordered by increasing difficulty in finding a mapping.

For the first query, we can see that both Api2Api and our approach can successfully select the correct top-1 mapping, the other results are also related. This case can be considered as easy for both approaches to performing well. For the second query, both approaches can achieve a good exact mapping, but for the other results, our approach can generalize all of the results under the '*System.IO.File*' class, while there are some less related results in the top-5 produced by Api2Api, e.g., '*System.Web.ErrorFormatter.ResolveHttpFileName*'. The third query token ranks the 11,204th in the embedding table[6]. As discussed earlier, embedding quality of rare tokens is not as good as those of frequent tokens. Therefore, it is more difficult to find an exact mapping for such a query. Even so, our approach can still rank a correct mapping at the third place ('*System.Windows.Controls.RichTextBox.CaretPosition*'), while

[6]The order of the token embedding provided by word2vec is proportional to the frequency of the token [18]

Table 9: Retrieved API mapping results from sample queries produced by SAR and Api2Api.

SAR	Api2Api
(1) java.util.Collection.add	
System.Collections.Objectmodel.Collection.Add	**System.Collections.Generic.List.Add**
System.Collections.Generic.List.Add	System.Collections.Generic.List.Get
System.Collections.ObjectModel.Collection.Clear	System.Collections.Generic.List.Remove
System.Collections.Generic.List.Contains	**System.Collections.Objectmodel.Collection.Add**
System.Collections.Generic.Dictionary.Add	System.Collections.IDictionary.GetEnumerator
(2) java.io.File.exists	
System.Io.File.Exists	**System.Io.File.Exists**
System.Io.File.AppendText	System.Web.Errorformatter.ResolveHttpFileName
System.Io.File.Delete	System.Io.File.OpenRead
System.Io.Fileinfo.LastWriteTime	System.Io.Compression.Zipfile.OpenRead
System.Io.File.GetAttributes	System.Io.Compression.ZipFile.ExtractToDirectory
(3) javax.swing.Text.JtextComponent.setCaretPosition	
System.Windows.Controls.RichTextBox.Clip	System.Drawing.Image.GetframeCount
System.Web.Ui.Webcontrols.DataGrid.PageSize	System.Media.SoundPlayer.PlaySync
System.Windows.Controls.RichTextBox.CaretPosition	System.Web.Ui.Webcontrols.Calendar.WeekendDayStyle
System.Windows.Forms.ContextMenuStrip.SuspendLayout	System.Configuration.Xmlutil.StrictSkipToNextElement
System.Windows.Controls.RichTextBox.CaretBrush	System.Media.SoundPlayer.PlayLooping
(4) java.util.concurrent.atomic.AtomicInteger.getAndDecrement	
System.Threading.Interlocked.Decrement	System.Directoryservices.SearchResultCollection.GetEnumerator
System.Threading.ReaderWriterLockSlim.EnterWriteLock	System.Directoryservices.SearchResultCollection.Dispose
System.Threading.Interlocked.Increment	System.Runtime.Serialization.ObjectIdGenerator.HasId
System.Threading.EventWaitHandle.OpenExisting	System.Collections.Generic.Queue.CopyTo

Api2Api produce totally unrelated results. For the last query, even though there has no mapping in C# by the ground truth, the retrieved results are still reasonably close. The query, in this case, is an API for an atomic operation, which is related to thread handling. Our approach can generalize the result mappings to the '*System.Threading*' APIs in C#, while the results from Api2Api are totally unrelated.

This experiment shows that Adversarial Learning can maximize the similarity between the two distributions so that similar APIs are clustered together.

6 THREATS TO VALIDITY AND LIMITATIONS

The goal of domain adaptation is to use as little knowledge as possible for any pair of languages. However, we only perform the experiments on Java and C# in this paper because it is not easy to find a good and large enough evaluation dataset for other pairs of languages. We leave this task in the future.

While unsupervised adversarial learning method does not require any seed as parallel data, there is a risk that the distributions of vectors (embeddings) in the two spaces are not so similar. Through our experiments, it is confirmed that the performance could be improved further by initializing the unsupervised adversarial learning method with a small set of seeds taken from the seed-based domain adaptation, and by generating the rest of API mappings.

Our approach can only generate single API mapping instead of an API sequence mapping. Both Api2Api and ours share such a limitation. In Api2Api, they use the new mappings mined from the tool as the input for an external machine translation tool [11], to generate the mapping for API sequences. In the future, we can also feed the newly found mapping APIs from our tool to [11] as inputs.

We mainly use a simplified top-k accuracy metric to measure our performance against the Api2Api. In real-world use cases, other information retrieval based metrics, such as MAP and MRR, may have less bias in evaluating the list of API mappings. We leave this for the future.

7 CONCLUSION & FUTURE WORK

We have proposed a domain adaptation approach, named SAR, to automatically transform and align the vector spaces of two different languages and APIs used therein. Before and after the adversarial learning step, we adapted the code embedding technique with a seeding and a refinement method respectively. SAR can identify API mappings across different programming languages. Our evaluation shows that the mappings between Java and C# APIs identified by SAR can be more accurate than other approaches with just 174 mapping seeds that can be easily identified by an automatic, simple signature-based heuristic, and that SAR helps to identify hundreds of more API mappings between Java and C# SDKs.

Domain adaptation methods are useful for other software engineering tasks that involve two different domains targeted by transferred learning [19, 20, 36], such as cross-language program classification, cross-language/project bug prediction. These tasks may benefit from the proposed approach when little curated data is available. Other SE tasks that are challenging due to lack of data, such as the out-of-vocabulary (OOV) problem [1, 7, 14] for learning and modeling fast-evolving software code, may also benefit from our domain adaptation approach, because the embeddings of OOV words may be approximated on-the-fly by adapting the known embeddings of their contextual or similar words in different languages. In the future, we will explore these variants of applications.

ACKNOWLEDGMENTS

This research is supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant from SIS at SMU, and EPSRC and EU at the Open University. We also thank the anonymous reviewers for their insightful comments and suggestions, and thank the authors of related work for sharing data.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys (CSUR)* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 207–216.

[3] Nghi D. Q. Bui and Lingxiao Jiang. 2018. Hierarchical Learning of Cross-Language Mappings through Distributed Vector Representations for Code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 33–36. https://doi.org/10.1145/3183399.3183427

[4] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019. Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* (2019).

[5] codejuicer. 2017. Java2CSharp: a maven plugin to convert java classes to c#. https://github.com/codejuicer/java2csharp Last commit in Sep 19, 2017.

[6] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. 2017. Word Translation Without Parallel Data. *CoRR* abs/1710.04087 (2017). arXiv:1710.04087 http://arxiv.org/abs/1710.04087

[7] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. 2018. Deep Learning On Code with an Unbounded Vocabulary. In *Machine Learning for Programming (ML4P) Workshop at Federated Logic Conference (FLoC)*.

[8] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 745–759.

[9] Yaroslav Ganin and Victor S. Lempitsky. 2015. Unsupervised Domain Adaptation by Backpropagation. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 1180–1189. http://jmlr.org/proceedings/papers/v37/ganin15.html

[10] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. *CoRR* abs/1406.2661 (2014). arXiv:1406.2661 http://arxiv.org/abs/1406.2661

[11] Spence Green, Daniel M. Cer, and Christopher D. Manning. 2014. Phrasal: A Toolkit for New Directions in Statistical Machine Translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation, WMT@ACL 2014, June 26-27, 2014, Baltimore, Maryland, USA*. 114–121. http://aclweb.org/anthology/W/W14/W14-3311.pdf

[12] 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)*. 631–642.

[13] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*. 3675–3681.

[14] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3106237.3106290

[15] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 173–184. https://doi.org/10.1145/2661136.2661148

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).

[17] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. 2013. Exploiting Similarities among Languages for Machine Translation. *CoRR* abs/1309.4168 (2013). arXiv:1309.4168 http://arxiv.org/abs/1309.4168

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NIPS)*. 3111–3119.

[19] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. 2018. Heterogeneous Defect Prediction. *IEEE Transactions on Software Engineering* 44, 9 (Sep. 2018), 874–896. https://doi.org/10.1109/TSE.2017.2720603

[20] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer Defect Learning. In *Proceedings of the 2013 International Conference on Software Engineering*. 382–391.

[21] 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)*. 457–468.

[22] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning of API mappings for language migration. In *36th International Conference on Software Engineering - Companion (ICSE)*. 618–619.

[23] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 651–654.

[24] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 585–596.

[25] Anh Tuan Nguyen, Zhaopeng Tu, and Tien N. Nguyen. 2016. Do Contexts Help in Phrase-Based, Statistical Source Code Migration?. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 155–165.

[26] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *39th International Conference on Software Engineering (ICSE)*. 438–449. https://doi.org/10.1109/ICSE.2017.47

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[28] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. 2017. Statistical migration of API usages. In *39th International Conference on Software Engineering - Companion Volume (ICSE)*. 47–50.

[29] 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.

[30] Peter H. Schönemann. 1966. A generalized solution of the orthogonal procrustes problem. *Psychometrika* 31, 1 (01 Mar 1966), 1–10. https://doi.org/10.1007/BF02289451

[31] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. 2013. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. 1139–1147. http://jmlr.org/proceedings/papers/v28/sutskever13.html

[32] Yaxing Wang, Chenshen Wu, Luis Herranz, Joost van de Weijer, Abel Gonzalez-Garcia, and Bogdan Raducanu. 2018. Transferring GANs: Generating Images from Limited Data. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VI (Lecture Notes in Computer Science)*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.), Vol. 11210. Springer, 220–236. https://doi.org/10.1007/978-3-030-01231-1_14

[33] Zhiguo Wang, Wael Hamza, and Radu Florian. 2017. Bilateral Multi-Perspective Matching for Natural Language Sentences. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*. 4144–4150.

[34] Chao Xing, Dong Wang, Chao Liu, and Yiye Lin. 2015. Normalized Word Embedding and Orthogonal Transform for Bilingual Word Translation. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*. 1006–1011. http://aclweb.org/anthology/N/N15/N15-1104.pdf

[35] 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*. IEEE Press, 335–346.

[36] S. Yan, B. Shen, W. Mo, and N. Li. 2017. Transfer Learning for Cross-Platform Software Crowdsourcing Recommendation. In *24th Asia-Pacific Software Engineering Conference (APSEC)*. 269–278. https://doi.org/10.1109/APSEC.2017.33

[37] Yijun Yu. 2019. fAST: Flattening Abstract Syntax Trees for Efficiency. In *Proceedings of the 41th International Conference on Software Engineering, ICSE*. 278–279.

[38] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing Behavioral Differences in Cross-Language API Mapping Relations. In *Proceedings of 16th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*. 130–145.

[39] 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)*. 195–204.