Mapping APIs in Dynamic-typed Programs by Leveraging Transfer Learning

ZHENFEI HUANG, Tianjin University, China
JUNJIE CHEN, Tianjin University, China
JIAJUN JIANG*, Tianjin University, China
YIHUA LIANG, Tianjin University, China
HANMO YOU, Tianjin University, China
FENGJIE LI, Tianjin University, China

Application Programming Interface (API) migration is a common task for adapting software across different programming languages and platforms, where manually constructing the mapping relations between APIs is indeed time-consuming and error-prone. To facilitate this process, many automated API mapping approaches have been proposed. However, existing approaches were mainly designed and evaluated for mapping APIs of statically-typed languages, while their performance on dynamically-typed languages remains unexplored.

In this paper, we conduct the first extensive study to explore existing API mapping approaches' performance for mapping APIs in dynamically-typed languages, for which we have manually constructed a high-quality dataset. According to the empirical results, we have summarized several insights. In particular, the source code implementations of APIs can significantly improve the effectiveness of API mapping. However, due to the confidentiality policy, they may not be available in practice. To overcome this, we propose a novel API mapping approach, named MATL, which leverages the transfer learning technique to learn the semantic embeddings of source code implementations from large-scale open-source repositories and then transfers the learned model to facilitate the mapping of APIs. In this way, MATL can produce more accurate API embedding of its functionality for more effective mapping without knowing the source code of the APIs. To evaluate the performance of MATL, we have conducted an extensive study by comparing MATL with state-of-the-art approaches. The results demonstrate that MATL is indeed effective as it improves the state-of-the-art approach by at least 18.36% for mapping APIs of dynamically-typed language and by 30.77% for mapping APIs of the statically-typed language.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Software maintenance tools**.

Additional Key Words and Phrases: API mapping, Program transformation, Transfer learning

*Corresponding author.

Authors' addresses: Zhenfei Huang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, jeff_huang@tju.edu.cn; Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, junjiechen@tju.edu.cn; Jiajun Jiang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, jiangjiajun@tju.edu.cn; Yihua Liang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, liangyihua@tju.edu.cn; Hanmo You, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, youhanmo@tju.edu.cn; Fengjie Li, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, lfj_legend1@tju.edu.cn.

1049-331X/2024/1-ART1 $15.00
https://doi.org/10.1145/3641848

ACM Reference Format:
Zhenfei Huang, Junjie Chen, Jiajun Jiang, Yihua Liang, Hanmo You, and Fengjie Li. 2024. Mapping APIs in Dynamic-typed Programs by Leveraging Transfer Learning. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2024), 29 pages. https://doi.org/10.1145/3641848

1 INTRODUCTION

Modern software usually depends on third-party library APIs (Application Program Interfaces) to facilitate the development of software by reusing existing code. However, when the APIs change, such as platform changing or library upgrading, the original software that depends on them has to be updated accordingly to work normally. They usually happen along with the evolution of software or libraries [9, 22, 23, 31, 32, 38, 43, 64, 68, 82, 86, 87]. In particular, deep learning techniques have attracted much attention in recent years and have been widely adopted in diverse applications [30, 63, 86, 87]. Based on the feedback from our industrial partner (an international IT company), it is a common task for developers to transform an implementation of a deep learning model from one platform to another, such as from Tensorflow [7] to Pytorch [6], where the APIs in the program need to be updated so as to adapt to a certain execution environment. To facilitate the API mapping and updating task, a set of automated program transformation tools [31, 76, 88] has been proposed. For example, Java2Csharp [88] is dedicated to automatically transforming Java programs into C# programs of the same functionality, a tool named 2to3 script [68] can assist developers in migrating programs from Python 2 to Python 3, and Patl4J [76] can transform Java programs using different libraries. However, these tools typically require external API mapping relations as input, which play a central role in the program transformation process.

However, providing the API mapping relations is indeed challenging and labor-intensive. The reason is that constructing the mapping relations usually requires high expertise. The developers have to be familiar with the functionality of different APIs; otherwise, incorrect API mappings may be provided. Moreover, the number of API mapping relations is usually large in practice, e.g., more than 1,000 APIs in TensorFlow[7]. In the face of this challenge, researchers have proposed several API mapping approaches that are devoted to constructing the mapping relations automatically, such as mapping APIs according to either their usage patterns [57, 60] or the associate documents [85]. Particularly, all existing approaches were designed for mapping APIs in statically-typed languages, such as Java and C#, where the type information can be taken as a critical feature to guide the construction of API mappings. According to the latest programming language popularity report in June 2023 from TIOBE Index[1], dynamically-typed programming languages have become the most popular ones, such as Python and JavaScript. Therefore, mapping APIs in these programs is urgently important. Especially with the rapid growth of deep learning techniques as aforementioned, deploying deep learning models that are usually implemented in Python on different platforms has become a common task [30]. However, whether existing API mapping approaches can still work well remains unexplored.

To fill this gap, in this paper, we conducted the first study to investigate the performance of existing API mapping approaches over dynamically-typed programs. In particular, we mainly focus on the APIs across different deep learning libraries in Python programs as subjects in this study due to their wide application in recent practice. Specifically, we first curated two high-quality datasets of Python programs for evaluating the performance of API mapping approaches, based on which we empirically compared the performance of existing approaches involving both categories of API mapping approaches, i.e., usage-based (i.e. StaMiner and API2API) and document-based (i.e., CODEnn and GraphCodeBert). The details of these approaches will be introduced in

[1]https://www.tiobe.com/tiobe-index/

Section 3. The empirical results show that the usage-based approaches are likely to be affected by the type information generally. Specifically, type information contributes less to the performance of StaMiner, which, however, requires parallel implementations of the same functionality. On the contrary, the other usage-based approach, API2API, is sensitive to the type information, making it perform relatively poorly for the dynamically-typed language, i.e., Python. Particularly, the excellent performance of CODEnn and GraphCodeBert in the document-based category indicates that the source code implementations and the document descriptions can effectively contribute to more accurate API mapping by providing deep semantic features of the APIs' functionalities, which inspires new research in this direction for further promoting the effectiveness of API mapping.

However, although the source code implementations of the APIs can significantly promote mapping effectiveness, they are not always available in real practice due to confidentiality policies in commercial companies. To overcome the unavailability while still taking the superiority of source code features for promoting API mapping, we propose a novel API mapping approach, named MATL, by leveraging transfer learning techniques. Specifically, the basic idea of our approach is to enforce the API signatures and documents, both of which are public information that will be provided by API vendors, to encode the rich semantics of source code implementations by learning from a large corpus of open-source programs. Then, the learned model that takes the API signature and document as inputs ideally can embed the unavailable source code semantics. Finally, we map APIs based on the similarity between their associated embeddings by joint learning. In this way, our approach is expected to inherit the superiority of utilizing source code for API mapping without accessing the source code.

To evaluate the performance of our approach, we have also conducted an extensive study on two datasets for mapping APIs of both dynamically-typed (i.e., Python) and statically-typed languages (i.e., Java and Swift) and compared the performance of our approach with state-of-the-art existing approaches. The results demonstrate that our approach significantly outperforms the baselines. Specifically, our approach improves the state-of-the-art CODEnn and GraphCodeBert on average by 56.75% and 18.36% in terms of Top-1 accuracy on the dataset of the dynamically-typed language, i.e., Python. In particular, our approach does not depend on the source code implementations of the APIs, making it more feasible and general for practical use. Besides, our approach also outperforms the latest and state-of-the-art API mapping approach Deep-Dive and API2API by up to 30.77% and 183.33% on the statically-typed API mapping dataset. To sum up, the experimental results demonstrate the effectiveness and practicability of our approach.

In summary, this paper makes the following major contributions:

- We conduct the first empirical study to investigate the performance of existing API mapping approaches on dynamically-typed languages.
- We construct a high-quality benchmark for API mapping and summarize a set of findings from our study, which motivate our approach and promote future studies in this research area.
- We propose a novel API mapping approach by leveraging transfer learning techniques, which can effectively map APIs without knowing the type information and source code implementations.
- All our experimental data and implementations have been released to promote future research. **https://github.com/TJUISTAJeff/MATL**

Paper Organization. The remaining sections of the paper are organized as follows: Section 2 briefly introduces the background of API mapping tasks. Section 3 elaborates on our empirical study and the findings that motivate the design of our approach. Section 4 describes our automated API mapping approach, MATL, which is not constrained to statically-typed languages. Section 5 presents

the details of the experiment setup and the evaluation results of MATL. Section 6 introduces the implications from our study. Sections 7-9 describe the threats to the validity, the related work, and the conclusion, respectively.

2 PRELIMINARY

In this section, we will define some notations that will be referred to throughout the article and formalize the problem of API mapping. In software engineering, the Application Programming Interface (API) is the bridge to connect different program implementations of diverse functionalities and provides a way for source code reusing. In practice, developers must be familiar with the semantics of the APIs in order to achieve certain functionalities. Besides, the APIs must be used by obeying certain syntactic or semantic constraints, such as the number of arguments and their types. In this paper, we define an API entity e as a ternary tuple of $\langle sig, src, doc \rangle$. Specifically, sig denotes the signature of the API, including the name, argument list, and return type, restricting the syntactical usage of the API. src represents its source code implementation, and doc represents the associated documents (including the description in natural language and code examples if available) that usually introduce the functionality of the API. They depict the API semantics by explicit code logic and the informal description in natural language, respectively. In particular, constructors are also regarded as APIs in our definition and their return types are their associated classes, such as `torch.nn.Linear()` and `tensorflow.keras.layers.Dense()` from PyTorch and Tensorflow frameworks, respectively. In addition, the source code of certain APIs, i.e., src, may be unavailable in real practice due to confidentiality properties, where src will be empty. Please note that this definition can apply to most of the widely-used programming languages, e.g., Java and Python. However, there may still be some specialized features from certain languages. For example, the signature in Java programs may also involve exception handling. We do not include such specialized features in our definition as our approach does not need them, and thus, we mainly focus on the features that we will use. Listing 1 presents an example of the ternary tuple of API `torch.nn.Linear()`. According to this definition, we define the relation of *API mapping* as follows.

```
Signature:
- - - - - - - -
    Name:
        torch.nn.Linear()
    Arg List:
        in_features(int)-size of each input sample
        out_features(int)-size of each output sample
        ...
Document:
- - - - - - - -
    Applies a linear transformation to the incoming data
Source Code:
- - - - - - - -
class Linear(Module):
    __constants__ = ['in_features', 'out_features']
    ...
    def __init__(self, in_features: int, out_features... :
        super(Linear, self).__init__()
        ...
```

Listing 1. An Example of the Ternary Tuple in PyTorch

Definition 2.1. (*API Mapping*) We define a sequence of APIs $e = [e_s^1, e_s^2, \ldots, e_s^m]$ from the source library L map another sequence of APIs $e' = [e_t^1, e_t^2, \ldots, e_t^n]$ from the target library L' *iff* e and e' implement the same functionality on the basis of the corresponding libraries. We use $\mathscr{R}(e) = e'$ (and $\mathscr{R}(e') = e$ vice versa) to represent the mapping relation.

According to the value of m and n, the mapping relations are generally classified into two types according to the existing studies [76, 85], i.e., one-to-one mappings if $m = n = 1$ and otherwise many-to-many mappings. In particular, one-to-many and many-to-one mappings fall into the many-to-many category by following existing studies [76]. Therefore, the API mapping problem can be defined as building such a function $\mathscr{R}$ that can correctly map the given APIs to those in the targeted library accordingly. In this paper, we target the one-to-one API mapping problem. The reasons are twofold: (1) Most existing approaches target this category [14, 19, 27, 44, 57, 60, 67], and thus it is possible for us to investigate their performance on dynamic-typed programs in our extensive empirical study; (2) One-to-one API mapping is the dominant category in practice, which are prevalent in many application scenarios, such as API upgrading [31] and platform migration [47].

3 EMPIRICAL STUDY

As aforementioned, existing API mapping approaches were designed and evaluated for APIs of static-typed languages, e.g., Java and C#, while their performance for mapping APIs of dynamic-typed programming languages is still unknown to us. To fill this gap, in this paper, we conduct the first empirical study to investigate their performance without knowing the static type information related to the APIs, such as the types of arguments and the return value. Specifically, existing API mapping approaches can be classified into two categories:

- **Usage-based** that constructs the mapping relations according to the usage patterns of APIs in client programs;
- **Document-based** that depends on both API documents and signature features.

In our study, we investigate the performance of both these two categories of API mapping approaches, where we focus on the following research question:

> ***How effective are existing approaches in mapping APIs of dynamically-typed languages?***

3.1 Studied Approaches and Adaptation

Our study includes both categories of API mapping approaches, i.e., usage-based and document-based. In this section, we briefly introduce the processes corresponding to the studied approaches in each category and explain their adaptation to the dynamic-typed programming language, i.e., Python, in our study. Please note that we do not try to survey all the existing approaches in this field but provide an overview of the corresponding approaches.

3.1.1 Usage-based API mapping. As aforementioned, this category of approaches depends on the usage patterns of APIs in the client programs to construct the mapping relations. In particular, this category includes two types of state-of-the-art approaches, which are the most representative.

The first type of approach requires the existence of parallel code fragments that implement the same functionality for building API mapping relations. The basic idea of these approaches is intuitive — The APIs implementing the same functionality have greater possibilities to map each other. For example, it is a common case in programs for file reading, where `new InputStreamReader()`, `InputStreamReader.read()`, and `InputStreamReader.close()` in Java are usually successively invoked while the APIs used in C# are usually `new StreamReader()`, `StreamReader.Read()` and

```
...
if not torch.is_tensor(item):
   item=torch.from_numpy(item)
elif torch.is_nonzero(item):
   item=torch.zero_like(item)
else:
   item=torch.squeeze(item)
...
```

```
...
if not tf.is_tensor(item):
   item=tf.convert_to_tensor(item)
elif tf.math.not_equal(item, 0):
   item=tf.zero_like(item)
else:
   item=tf.squeeze(item)
...
```

Fig. 1. Code fragments of parallel implementations using PyTorch (left) and Tensorflow (right) frameworks.

`SteamReader.Close()` as the correspondence. Therefore, the mapping relations can be constructed by aligning the APIs in parallel implementations. However, the limitation of the usage-based techniques is also straightforward. The required parallel implementations of the API mapping approaches are typically well-structured and fine-grained code fragments (i.e., within a single method) of the same functionality, such as the code fragments shown in Figure 1. In fact, such parallel implementations are indeed rare in practice since different projects tend to vary in code structures even though the complete projects are implementing the same functionality, making the usage-based API mapping techniques inapplicable to most cases. In this category, the most notable approach is StaMiner [57], which extracts the usage graph of APIs, named *Groum graph* that consists of both program syntactic (e.g., method calls and conditional structures) and semantic features (e.g., control and data dependencies), and then maps APIs according to the Groum graph by leveraging Expectation-Maximization (EM) Algorithm [40]. The latest research also proposed deep-learning-based API mapping approaches on the basis of parallel implementations [13, 88]. However, due to the requirement of massive training data, which is hard to construct since they are not open-source, we do not include them in our study and take StaMiner as the representative.

Similar to the approaches of the first type that map APIs according to the commonality of their usages for achieving the same functionality, the second type of approaches are also on the basis of the naturalness of API usage sequences, but in a different way, such as API2API [60] and SimilarAPI [16]. As introduced, the former tries to align the APIs directly according to the parallel implementations, while the latter endeavor to encode API's high-level semantic vector representations. For example, the most notable API2API [60] and SimilarAPI [16] both endeavor to learn high-level semantic representations of the APIs and then map them according to the similarity of the representations. In our study, we take API2API as a baseline since it is the state-of-the-art approach that considers more API usage logic. API2API assumes that APIs appearing in a similar context in the program should share the commonality and thus have a higher possibility to implement the same functionality, which is analogous to measuring the similarity of words in Natural Language. Specifically, API2API depends on an API2Vec model by referring word2vec (i.e., CBOW [50]) to learn the representations of APIs, and then maps APIs via learning a transformation mechanism from one vector space of the source APIs to the other of the target APIs. Formally, suppose that the vector representations of the source and target APIs are $\vec{a}$ and $\vec{a'}$, respectively; a transformation matrix T will be learned when providing a set of API mapping examples between the source and target APIs, where $\vec{a} \times T$ can approximate $\vec{a'}$. In this way, API2API maps APIs according to the cosine similarity between $\vec{a} \times T$ and $\vec{a'}$, and higher similarity indicates a higher possibility of mapping each other. Taking the code examples shown in Figure 1 as an example, the APIs in the two code fragments can be visualized in Figure 2 after obtaining their corresponding vector representations. From the figure, we can observe that API representations shown on the left

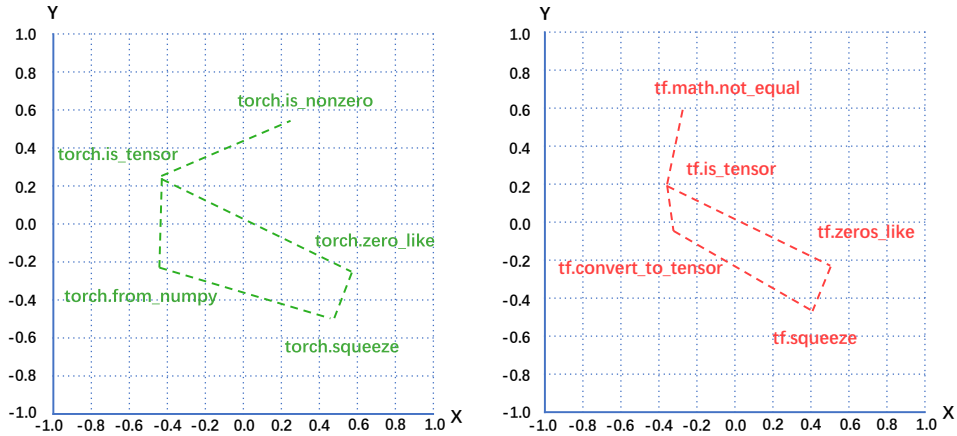

Fig. 2. Visualization of APIs' vector representations by using API2API.

Table 1. Rules for constructing the *Groum* graph based on the code templates in Python programs.

Code Structure	Code Template	PyGroum	Code Structure	Code Template	PyGroum
Method Invocation	m()	m	**While Statement**	while(X): Y	X⇒while⇒Y
Parameters	m(X, Y, Z)	(X∨Y∨Z)⇒ m	**For Statement**	for...in X: Y	X⇒for⇒Y
Expression	X∘Y	X∨Y	**Block**	{X Y Z}	X⇒Y⇒Z
If Statement	if(X):Y else:Z	X⇒if⇒(Y∨Z)	**Try-except Statement**	try:X except:Y	X∨Y

[1] X, Y, Z denote the structures (in the "Code Template" column) and their corresponding Groums (in the "PyGroum" column), m denote the API.
[2] X⇒Y denotes there is a edge in Groum from X to Y, while X ∨ Y denotes there is no edge in Groum between X and Y.

can be approximately transformed into those on the right by a uniform transformation matrix. In other words, after applying the transformation matrix T, the mapped APIs are expected to be close to each other in the vector space, e.g., `torch.is_tensor` and `tf.is_tensor`.

Adaptation: StaMiner and API2API were both designed for mapping APIs between Java and C#. Besides, since their implementation is not publicly available, we re-implemented it by referring to the corresponding papers [57] [60]. For StaMiner, we re-implemented the *Groum graph* representation for Python programs according to the same construction rules, including the node types and their connections (e.g., control flow and data dependencies). Following the rules of Groum graph construction defined by StaMiner, we adapted them for building Groum graphs for Python programs and developed PyGroum, which is defined in Table 1. In the table, we present the code structures, the templates of the structures, and the corresponding relations that will be constructed in the Groum graph. Compared with the original rules, there is no type information in the PyGroum. For example, the Groum of a method invocation $m()$ in statically-typed programs should be $C.m$ (C is the type), while it is m in our PyGroum. Then, we construct the corresponding Groum graphs based on the rules defined in the table when feeding Python code fragments. For example, Figure 3 shows the constructed Groum graph of the example code. After building Groum graph representations, the API sequences used in the two parallel implementations will be mapped according to the EM Algorithm [40] by taking the sequences as the inputs. Specifically, given two API sequences Seq_s and Seq_t from the source and target projects (i.e., the parallel implementations), for each API a in Seq_s, a list of APIs in Seq_t will be returned with the associated probabilities, where the larger the probability is, the more confidence we will have to deem that the corresponding API is the desired one mapping to a.

```
...
if not torch.is_tensor(item):
    item = torch.from_numpy(item)
elif torch.is_nonzero(item):
    item = torch.zero_like(item)
else:
    item = torch.squeeze(item)
...
```

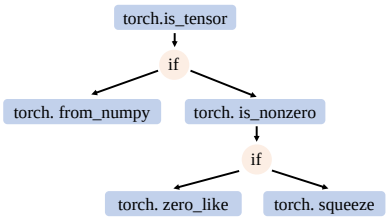

Fig. 3. A simple example of code fragment using PyTorch APIs (left) and its associated Groum graph (right).

For API2API, compared with the original algorithm, our adaptation has two updates: (1) API2API replaces variable tokens in Java/C# programs with the corresponding type names, while we instead use the fully-qualified *package* names of the APIs since the type information is unavailable in dynamic-typed programs, such as `torch.nn.Linear()` in Python. (2) We further perform tokenization for API names using a commonly-used tool, wordninja [8], and then take the sum of the sub-tokens as the API embedding, which was found to be much more effective in our study than embedding the complete API name directly. Besides, the parameters will be ignored by following the original paper [60]. In this way, the same APIs under different contexts are ensured to have the same representation. In general, given the trained API2Vec model and the learned transformation matrix T, we first obtain the corresponding representations of all the APIs from both source and target frameworks through API2Vec. Then, for each source API, we transform its representation to a new one by leveraging the transformation matrix T and compute its similarities with every target API by using the default cosine similarity in API2API. Finally, the target APIs will be sorted by descending order of the similarity.

3.1.2 Document-based API mapping. As introduced in Section 2, the document provides a high-level description of the APIs' functionalities. Therefore, the basic idea of document-based API mapping approaches is to capture the semantic/textual similarity between API documents. For example, the document in JDK of API `InputStreamReader.read()` is described as "*Reads a single character*", while the document of API `StreamReader.Read()` in .NET of C# is "*Reads the next character from the input stream and advances the character position by one character*". The high textual similarity between them indicates that the two APIs have a higher possibility of implementing the same functionality and thus mapping each other. Based on this, the latest research incorporates document information to improve the performance of automated API mapping. For instance, Zhang et al. [85] proposed to combine the semantic similarities of both documents and signatures to facilitate the ranking of the most probable mapping APIs, and the evaluation results on mapping Java to Swift APIs demonstrate the effectiveness of utilizing documents. Actually, the core task of such approaches is to embed the semantics of documents. The latest studies have proved that deep learning models can better embed the semantics of documents [26, 28, 29]. Since there is no document API mapping approaches that are open-source, we adapted two existing deep learning models to the API mapping task as the representative: CODEnn [26] and GraphCodeBert [29]. The reasons for selecting these two approaches are manifold: (1) They are designed for very close tasks, i.e., code search, and evaluated to be effective. (2) They take the source code implementations of APIs as inputs, which encode richer semantic (or functionality) features than the signatures and thus may improve the performance of document-based approaches. (3) CODEnn takes source code as plain text for embedding, while GraphCodeBert incorporates the structure (i.e., Abstract Syntax

Tree) and semantic features (i.e., program dependencies) for code embedding. Therefore, they can complement each other in our study.

Adaptation: As introduced above, the CODEnn and GraphCodeBert were originally designed for close tasks, e.g., code search. To adapt them to API mapping, we transform the mapping task to code search since each API associates with the unique document and source code implementation, as explained in Section 2. In particular, in order to preserve the effectiveness of these two approaches, we adapt them to our API mapping task by aligning them with their original design. That is, we take the document of the source API as input and the code implementations of the target API as query results. Formally, given two APIs $e_s = \langle sig_s, src_s, doc_s \rangle$ and $e_t = \langle sig_t, src_t, doc_t \rangle$, we say e_s maps e_t if src_t is the result of the highest similarity with doc_s according to CODEnn and GraphCodeBert.

3.2 Metrics

To evaluate the effectiveness of each method in our study, we follow the existing works [14, 85] and apply *Top-K* accuracy [73] and Mean Reciprocal Rank (*MRR*)[75] as the evaluation metrics throughout the experimental result analysis. Particularly, the former evaluates the performance under the condition where the number of candidate APIs that will be checked by users is restricted, while the latter evaluates the performance without such a constraint. Both metrics reflect the satisfaction of users at the ranking list [11, 15]. We use m to denote the number of APIs in the source library for mapping and use $rank_i$ ($1 \leq i \leq m$) to denote the rank of desired target API for the i^{th} source API in the candidate list. The *Top-K* accuracy is defined by Formula 1.

$$Top\text{-}K = \sum_{i}^{m} \frac{\mathcal{I}(K - rank_i)}{m} \tag{1}$$

In the formula, the function $\mathcal{I}(n)$ returns 1 if $n \geq 0$; otherwise returns 0, while K represents an integer number. Particularly, we set $K \in \{1, 5, 10\}$ by following previous studies [14, 85]. Similarly, the metric of *MRR* is defined by Formula 2, which is a commonly-adopted metric for evaluating the performance of API recommendation[16, 34, 69, 84]. It represents the multiplicative inverse of the rank of the first correct API in the returned candidate list; the larger the *MRR* is, the better the performance will be.

$$MRR = \frac{1}{m} \sum_{i}^{m} \frac{1}{rank_i} \tag{2}$$

Intuitively, suppose there are 5 APIs from the source framework (i.e., $m = 5$); for each API, a list of candidate APIs from the targeted framework can be obtained by a certain API mapping approach. Assuming that the ranks of the desired target APIs are 1, 3, 3, 4, and 6, respectively, in the corresponding candidate list. Then, we can compute *Top-1*= $\frac{1+0+0+0+0}{5} = 0.2$ and *Top-5*= $\frac{1+1+1+1+0}{5} = 0.8$. Similarly, *MMR*= $\frac{1}{5}(\frac{1}{1} + \frac{1}{3} + \frac{1}{3} + \frac{1}{4} + \frac{1}{6}) = 0.417$.

3.3 Datasets

Existing approaches have been evaluated over different datasets of static-type programming language, and none of them can be accessed publicly. Therefore, to conduct our empirical study, we have manually constructed two distinct and high-quality datasets for evaluating the performance of existing approaches and meeting their diverse requirements. Also, the datasets will be used to evaluate the performance of our own method (to be introduced in Section 5) for a fair comparison. In particular, as introduced in Section 3.1, the usage-based approach StaMiner requires parallel implementations of the same functionality for API mapping, while the other approaches require the typical API information, including API signatures, associated documents and source code

Table 2. Details of API mappings in APIxDLF.

Frameworks	TensorFlow	PyTorch	MXNet	CNTK
TensorFlow	-	175	152	123
PyTorch	113	-	92	67

The rows present the frameworks of source APIs, while the columns present the frameworks of the target APIs, e.g., 175 API mappings are from Tensorflow to PyTorch.

implementations. In particular, we choose Python as our target dynamic-typed language since it has been widely used in real practice, especially with the rapid growth and wide adoption of deep learning techniques on different platforms. The construction details of the datasets are presented as follows.

BiDL: This dataset includes the parallel implementations for 41 deep learning projects involving many widely-used models, e.g., Bert [21], where one implementation is based on the TensorFlow v2.11 [7] framework (i.e., APIs are from Tensorflow) while the other is based on the PyTorch v1.13 [6] framework. All projects are implemented in Python. Particularly, this dataset is provided by our industry partner, an international technology company (we hide the company name due to the confidentiality policy), which needs to deploy the same deep learning models on different running environments, i.e., using different deep learning frameworks. All the implementations and the mapping relations have been double-checked by both the authors and the developers from the company. Finally, we have obtained 38 pairs of API mappings among the ones used in the projects.

APIxDLF: This dataset is constructed based on four widely-used deep learning frameworks (i.e., TensorFlow v2.11 [7], PyTorch v1.13 [6], MXNet v1.9.1 [5], CNTK v2.7 [2]), the APIs in which play as the backbone for building deep learning models. According to the requirements, we collected the API information (including signatures and documents) from their official websites and the associated code implementations from the open-source repositories. However, since the manual analysis requires expertise and is indeed time-consuming, we select the most widely-used 200 APIs from Tensorflow and PyTorch, respectively, as the basis and manually construct the mapping relations between frameworks within these APIs. Specifically, for each framework, we first collected the top 200 projects with the most number of Stars on GitHub [3]. Then, we collected and ranked all the APIs used in those projects by descending order of their occurrences and selected the most widely-used 200 APIs whose source code is available as existing baseline approaches require access to the source code. To ensure the correctness of the mappings, the three authors have manually checked the documents and associated implementations and searched the websites (e.g., StackOverflow[2] and StackExchange[3]) independently. After two-round discussion regarding inconsistencies, we finally achieved a Cohen's Kappa coefficient over 90%, which leaves us a high confidence to ensure the quality of the constructed dataset, i.e., the mapped APIs indeed implement the same functionality. Finally, all remaining inconsistencies were further analyzed and discussed to reach a consensus. As a result, we obtained 722 mapping instances, which is more than the number of mappings used for evaluation in previous studies [16] [27] [44] [85]. The details of the obtained dataset are listed in Table 2. Each cell presents the number of unique source APIs. Please note that the mapping instances of mapping A to B and mapping B to A may be different, e.g., TensorFlow and PyTorch. The reason is that PyTorch APIs are usually high-level implementations, where one API may map to multiple TensorFlow APIs by feeding

[2]https://stackoverflow.com
[3]https://stackexchange.com

different parameters. For example, `torch.nn.init.constant_` in PyTorch corresponds to both `tensorflow.constant_initializer` and `tensorflow.keras.initializers.Constant` in TensorFlow. In this case, we consider the mapping correct when recommending any one of them. This phenomenon is more common for MXNet and CNTK. As a result, only a small number of unique mapping instances for them. To reduce the evaluation bias due to insufficient training/testing data, we only consider the mappings that take TensorFlow and PyTorch as the source framework. We also make this dataset open-source to facilitate the replication of our study and also aim at providing a high-quality benchmark for API mapping and thus promoting future research.

In summary, the API mappings in BiDL have the associated parallel code implementations of the same functionalities, while the API mappings in APIxDLF do not have such restrictions. Therefore, APIxDLF consists of more mapping instances than BiDL. However, for methods that require parallel implementations, only the BiDL dataset can be used.

3.4 Experiment Configuration

Usage-based approach (i.e., StaMiner and API2API): As aforementioned, StaMiner requires parallel implementations, and we evaluated its performance on the dataset of BiDL. In particular, according to the EM algorithm, each pair of APIs between the source and target frameworks will be assigned a mapping score, according to which we compute the *Top-K* and *MRR*.

On the other hand, as for the data configuration of API2API, it requires a large-scale codebase for training the vector representation of APIs like Word2Vec. To achieve this, we have collected the source code of the top 1,000 repositories using TensorFlow and PyTorch, respectively, according to the number of *stars* on GitHub, which in total include more than 42M lines of Python code involving 19.0K TensorFlow APIs and 9.0K PyTorch APIs. However, the projects using the other frameworks (i.e., MXNet and CNTK) are indeed rare. For example, the numbers of GitHub projects using Tensorflow and Pytorch with over 200 stars are 1.2k and 2.8k, respectively, while the numbers of CNTK and MXNet projects are only 5 and 49. Even if considering all projects that have at least 100 stars, there are still only 8 and 81 projects. In other words, the training data for API2API tends to be insufficient and less representative when working with CNTK and MXNet. Therefore, we evaluate the performance of API2API on BiDL and APIxDLF for mapping TensorFlow and PyTorch only.

Document-based approach (i.e., CODEnn and GraphCodeBert): Since these two approaches require API document and source code implementation, we conduct the experiment only on the dataset APIxDLF. Particularly, both CODEnn and GraphCodeBert were firstly pre-trained on the original dataset for the task of code search [4, 26], which were regarded as the pre-trained models. Then, we further fine-tuned these two models on our dataset APIxDLF. In our study, we will report the results of both the original and the fine-tuned models.

Implementation and Environment: As introduced in Section 3, we re-implemented the approaches of StaMiner and API2API based on the corresponding papers in Python while adapting the open-source implementations of CODEnn [26] and GraphCodeBert [29]. We also make all our implementations open-source to facilitate replication and comparison for future research. All our experiments were conducted on a server with Ubuntu 18.04, equipped with 128GB RAM and a processor of Intel(R) Xeon(R) E5-2640 that has 10 cores of 2.40GHz. The website of our project is at **https://github.com/TJUISTAJeff/MATL**.

3.5 Empirical Result Analysis

In this section, we will display and analyze our empirical results of existing API mapping approaches introduced in Section 3.1. By analyzing the results, we will provide several findings that can

Table 3. Experimental results of usage-based API mapping methods on the dataset of BiDL.

Mapping	Approach	Top-1	Top-5	Top-10	MRR
TF ⇒ PT	StaMiner	0.66	1.00	1.00	0.77
	API2API	0.11	0.19	0.38	0.21
PT ⇒ TF	StaMiner	0.71	0.97	0.97	0.81
	API2API	0.08	0.18	0.36	0.19

TF: TensorFlow; PT: PyTorch. $A \Rightarrow B$ denotes mapping A to B.

provide guidelines for future research as well as motivate our transfer-learning-based API mapping approach.

3.5.1 Results on dataset of BiDL. As introduced in Section 3.4, we evaluate StaMiner and API2API on BiDL. Please note that StaMiner does not require training data and computes the API mapping relations directly according to the parallel implementations. Regarding API2API, we employ the manually constructed mapping relations in APIxDLF as the training data by removing the ones included by BiDL, which includes 136 training instances. Finally, we report the evaluation results of both approaches on BiDL. The experimental results are presented in Table 3, where the first two columns show the API mapping directions and the studied approaches, while the following columns present the results regarding different metrics.

According to the results, we can see that StaMiner performs significantly better than API2API in terms of all the metrics. Specifically, StaMiner outperforms API2API by 55% for TF ⇒ PT, and by 63% for PT ⇒ TF regarding the Top-1 accuracy, respectively, and the average improvement is about 59%. Regarding the metric of MRR, StaMiner outperforms API2API by 64% on average. By further comparing with the results for mapping static-typed APIs, the performance of StaMiner slightly drops (77.5% for Top-1 in the original paper [26]), while the Top-1 accuracy of API2API suffers from a large decline (43.3% in [60]). The results demonstrate that the type information contributes to the usage-based approaches: StaMiner is less likely affected by the lack of type features for API mapping, while API2API relies more on type information. The reason is also clear. Since StaMiner maps APIs according to the *Groum graph* of parallel implementations, the rich context information (e.g., dependencies) can facilitate the accurate mapping between APIs. However, without the type information, diverse contexts of source code may not better distinguish different APIs or even mislead the embedding of APIs in API2API and finally affect the mapping accuracy. For example, the APIs for different layers (e.g., convolution and fully connected) are both commonly used during the model construction process, where the contexts are very close. In contrast, in static-typed programs, the type information (i.e., the classes to which the APIs belong) can effectively confine the mapping results.

However, despite the superior performance of StaMiner, it still suffers from severe usability issues since it highly depends on the parallel implementations of the same functionality, which are hard and even impossible to obtain, especially for some languages that are used in a niche market or for libraries that are newly released. Therefore, the application scope of StaMiner is strictly restricted, which hinders its practical usage. On the contrary, API2API can be easily adapted to map APIs of different programming languages and thus should be more practical. But the sensibility to the type features makes it not work well for dynamic-typed languages.

Table 4. Experimental results of usage-based and document-based methods on the dataset of APIxDLF.

Mapping	Approach	Top-1	Top-5	Top-10	MRR	Mapping	Approach	Top-1	Top-5	Top-10	MRR
TF ⇒ PT	API2API	0.11	0.23	0.41	0.24	**PT ⇒ TF**	API2API	0.11	0.19	0.36	0.23
	CODEnn$_{pt}$	0.06	0.17	0.37	0.15		CODEnn$_{pt}$	0.03	0.21	0.27	0.12
	CODEnn$_{ft}$	0.37	0.60	0.63	0.46		CODEnn$_{ft}$	0.32	0.55	0.55	0.37
	GraphCodeBert$_{pt}$	0.36	0.64	0.82	0.50		GraphCodeBert$_{pt}$	0.55	0.86	0.89	0.68
	GraphCodeBert$_{ft}$	0.39	0.88	0.91	0.59		GraphCodeBert$_{ft}$	0.60	0.91	0.97	0.75
TF ⇒ MX	CODEnn$_{pt}$	0.03	0.22	0.41	0.15	**PT ⇒ MX**	CODEnn$_{pt}$	0.03	0.15	0.24	0.12
	CODEnn$_{ft}$	0.60	0.76	0.79	0.67		CODEnn$_{ft}$	0.25	0.63	0.78	0.40
	GraphCodeBert$_{pt}$	0.32	0.76	0.89	0.51		GraphCodeBert$_{pt}$	0.59	0.89	0.91	0.74
	GraphCodeBert$_{ft}$	0.34	0.81	0.97	0.51		GraphCodeBert$_{ft}$	0.64	0.95	0.99	0.76
TF ⇒ CT	CODEnn$_{pt}$	0.12	0.30	0.61	0.25	**PT ⇒ CT**	CODEnn$_{pt}$	0.15	0.30	0.39	0.24
	CODEnn$_{ft}$	0.42	0.76	0.85	0.58		CODEnn$_{ft}$	0.27	0.48	0.70	0.38
	GraphCodeBert$_{pt}$	0.39	0.64	0.90	0.56		GraphCodeBert$_{pt}$	0.53	0.84	0.88	0.65
	GraphCodeBert$_{ft}$	0.44	0.90	0.96	0.61		GraphCodeBert$_{ft}$	0.53	0.88	0.98	0.69

[1] TF: TensorFlow; PT: PyTorch; CT: CNTK; MX: MXNet.
[2] *pt* stands for the pre-trained model, while *ft* for the fine-tuned model.

Observation 1: Type information in static-typed programs contributes to the effectiveness of usage-based approaches. In particular, the missing type information may cause a large decline in API mapping accuracy, e.g., API2API.

3.5.2 Results on dataset of APIxDLF. As introduced in Section 3.4, we evaluate API2API, CODEnn, and GraphCodeBert on this dataset. Particularly, we only evaluate the performance of API2API on the frameworks of TensorFlow and PyTorch due to the training data constraint as aforementioned. For the other two approaches, we employ both the pre-trained models and the fine-tuned models on our dataset APIxDLF and compare their performance. Please note that the numbers of testing data are consistent throughout this experiment in order to make the result comparison meaningful. Specifically, we set the testing size the same across different tasks and approaches (i.e., half of the least mapping instances in the tasks) for a fair comparison.

Table 4 presents the mapping results of the three methods. From the table, we can observe that although CODEnn and GraphCodeBert are not designed for API mapping, their performance is indeed excellent. In particular, GraphCodeBert significantly outperforms API2API even without fine-tuning on our API mapping dataset (i.e., GraphCodeBert$_{pt}$). Compared with API2API, the improvement of GraphCodeBert are on average 227.27% and 400% in terms of Top-1 accuracy on the tasks of TF ⇒ PT and PT ⇒ TF. Moreover, the improvement increases to 254.55% and 445.45% after further fine-tuning on the API mapping dataset. The results demonstrate the effectiveness of the document-based API mapping approaches.

From the table, we can also see that the fine-tuned models perform consistently better than the corresponding pre-trained models. On average, CODEnn$_{ft}$ improves CODEnn$_{pt}$ by 428.57% regarding Top-1 accuracy, and GraphCodeBert$_{ft}$ improves GraphCodeBert$_{pt}$ by 6.52%. The reason is that the source code implementations and the document descriptions for the framework APIs in our dataset are slightly different from those in the original training data for model pre-training, most of which are from open-source repositories whose quality is hard to control. When comparing the results of different networks, GraphCodeBert significantly outperforms CODEnn (i.e., $p\text{-}value < 0.05$) by performing a paired sample Wilcoxon signed-rank test [70] at the significance level of 0.05. One major reason should be the incorporation of deep semantic features for code embedding in GraphCodeBert since it considers the program dependency features for source code embedding while CODEnn does not. To sum up, GraphCodeBert$_{ft}$ achieves the best results and significantly outperforms all the other approaches over almost all the mapping tasks (i.e., 5 out of 6 tasks).

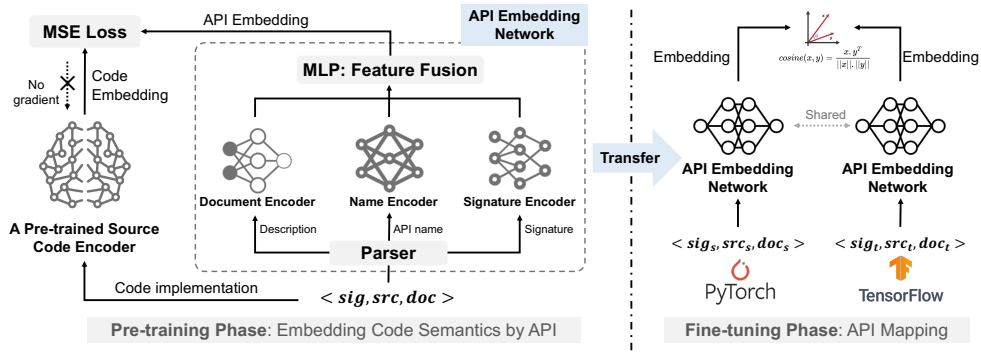

Fig. 4. Overview of our approach MATL that leverages transfer learning for API mapping.

Specifically, the average results of GraphCodeBert$_{ft}$ in terms of Top-1, Top-5, and Top-10 accuracy across all the mapping tasks are 49%, 89% and 96%, respectively. In particular, this result is also better than the latest document-based API mapping approach [85] that depends on the textual similarity of the API signature and the association documents, indicating that the source code is indeed effective in facilitating the API mapping task.

> **Observation 2:** The deep semantics of the associated documents and source code implementations of APIs play an essential role in accurate API mapping, and it is feasible and promising to leverage the idea of code search for API mapping.

4 OUR APPROACH

As concluded in our empirical study, the source code implementations are effective in promoting API mapping. However, they may be unavailable in practice due to confidentiality properties (e.g., commercial competitions or the protection of intellectual property). To address this issue, we propose a new API mapping approach, named MATL, by leveraging transfer learning techniques. The basic idea of our approach is that ***leveraging the source code semantics to improve the performance of API mapping without accessing the source code***.

Specifically, MATL employs the transfer learning technique to embed the source code semantics through publicly accessible information of APIs, i.e., signatures and documents. Then, the learned model can be adopted to facilitate the mapping of APIs by providing more accurate embedding of the API functionality. Transfer learning has the superiority of requiring a small number of training data of the targeted task, and thus has been widely used in many applications and evaluated to be effective, e.g., vulnerability prediction [42], clone detection [74], and code search [77]. Figure 4 shows the overview of our approach, which consists of two training phases: (1) **Pre-training phase** that aims at learning the accurate code semantic embedding by jointly learning the embeddings of source code and the embeddings of API signature and the associated document, where the large-scale open-source programs can be used as the training data. (2) **Fine-tuning phase** that optimizes the learned semantic embedding model for the task of API mapping by providing a small number of labeled mapping instances. As seen, MATL is a general API mapping approach that neither depends on the hard-to-obtain parallel implementations nor depends on the features that are specific to certain programming languages or libraries. Moreover, MATL is also free from a large number of labeled API mapping instances for model training. In the following, we will introduce the details of our approach.

4.1 API Embedding Network

As aforementioned that the source code implementations of the commonly-used APIs may not be publicly accessible. In order to take the superiority of source code semantics for effective API mapping, we design a novel deep learning model in the pre-training phase that aims at learning the code semantics using the API signature and the associated document, both of which are public information that can be referred by end-users while using the corresponding API. As presented in Figure 4, this model comprises two sub-networks, i.e., a pre-trained source code encoder network and the API embedding network. The former is a well-trained source code encoder that is devoted to providing the deep semantic embeddings of the source code functionality, while the latter is the one for embedding documents and signatures. By joint learning the embeddings of the two networks, the trained API embedding network is expected to gain the ability to well encode source code semantics of the APIs by providing the API signature and the associated document only.

In particular, we employ the GraphCodeBert model to provide the source code embedding since it has been proven effective in our empirical study due to the incorporation of rich program semantic features, e.g., program dependencies. Please note that this model can be replaced by any other pre-trained models, and the adaptation is also straightforward. As for the API embedding network, we design a two-level neural network architecture by carefully considering the characteristics of different input features. At the first level, we encode the API signatures and document descriptions with independent encoders (which will be introduced in the following) and then fuse the respective embeddings with a multi-level perceptron (i.e., fully connected layers) at the second level.

Formally, suppose the embeddings of the document descriptions, API name and API signature are represented by $\mathbf{H}_{doc}$, $\mathbf{H}_{name}$ and $\mathbf{H}_{sig}$, respectively, then the API embedding $\mathbf{H}$ based on the two-level architecture is defined by Formula 3.

$$\begin{aligned} \mathbf{H}' &= < \mathbf{H}_{doc}, \mathbf{H}_{name}, \mathbf{H}_{sig} > \\ \mathbf{H} &= \mathbf{W}_1 * tanh(\mathbf{W}_0 * \mathbf{H}' + \mathbf{b}_0) + \mathbf{b}_1 \end{aligned} \tag{3}$$

Finally, given the expected source code embedding from the pre-trained model as $\mathbf{Y}$, the pre-training phase in MATL aims at optimizing the API embedding $\mathbf{H}$ by joint learning through minimizing the loss defined by Formula 4.

$$MSELoss(\mathbf{H}, \mathbf{Y}) = [(\mathbf{H} - \mathbf{Y})]^2 \tag{4}$$

Next, we will introduce the details of the designed encoders for documents, API names and signatures, respectively.

4.1.1 Document and Name Encoder. As shown in Listing 1, the document description of an API is typically a summary of its functionality in natural language, while the (qualified) name of the API is composed of the *package* name and *method* name, implicitly reflecting the general functionality of the API as well. For example, according to the name of the API shown in Listing 1, it possibly performs *linear* transformations. In order to capture the rich semantics in the short descriptions, we adopt the bidirectional Recurrent Neural Network (BRNN) for both document embedding and API name embedding. It has been widely used and proved to be effective in Natural Language Processing (NLP). Specifically, BRNN improves upon RNN by allowing the model to attend to the context information of certain time steps in the sequence (beyond information from prior token words), which can facilitate the semantic embedding by eliminating ambiguity due to the incomplete context. Besides, we pre-process the API names and the document descriptions before feeding them to the network by following the conventional NLP pre-processing pipeline (e.g., word segmentation stemming, etc.). In this process, we also adopted the commonly-used tokenization tool wordninja [8] for word segmentation.

Formally, suppose the vector representation of input tokens from the document or the name of an API is $[\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_i, ..., \boldsymbol{x}_t]$, where t denotes the length of the input sequences and each $\boldsymbol{x}_i$ (i.e., $\boldsymbol{x}_i \in \mathbb{R}^d$, where d stands for the embedding size) represents the embedding of the corresponding token. For any step i, let the hidden layer activation function be ϕ and the forward and backward states of BRNN be $\mathbf{H}_i^{\rightarrow}$ (i.e., $\mathbf{H}_i^{\rightarrow} \in \mathbb{R}^h$ where h is the number of hidden units) and $\mathbf{H}_i^{\leftarrow}$ (i.e., $\mathbf{H}_i^{\rightarrow} \in \mathbb{R}^h$), respectively. The hidden states are forward and backward updated according to Formulas 5 and 6, respectively. In the formulas, $\mathbf{U} \in \mathbb{R}^{d \times h}$ and $\mathbf{W} \in \mathbb{R}^{h \times h}$ denote the trainable weights, while $\mathbf{b} \in \mathbb{R}^h$ denotes the bias of the model.

$$\mathbf{H}_i^{\rightarrow} = \phi(\boldsymbol{x}_i \mathbf{U}^{(f)} + \mathbf{H}_{i-1}^{\rightarrow} \mathbf{W}^{(f)} + \mathbf{b}^{(f)}) \tag{5}$$

$$\mathbf{H}_i^{\leftarrow} = \phi(\boldsymbol{x}_i \mathbf{U}^{(b)} + \mathbf{H}_{i+1}^{\leftarrow} \mathbf{W}^{(b)} + \mathbf{b}^{(b)}) \tag{6}$$

Finally, the embedding of the fed document or the API name is the combination of the hidden states $\mathbf{H}_t^{\rightarrow}$ and $\mathbf{H}_t^{\leftarrow}$ from both directions. We use $\mathbf{H}_t$ to denote the final output of the encoder, then it is defined by Formula 7, which is expected to capture the semantics of API names (i.e., $\mathbf{H}_{name} = \mathbf{H}_t$ when feeding API name as the input) and documents (i.e., $\mathbf{H}_{doc} = \mathbf{H}_t$ when feeding documents as the input).

$$\mathbf{H}_t = \mathbf{H}_t^{\rightarrow} + \mathbf{H}_t^{\leftarrow} \tag{7}$$

4.1.2 Signature Encoder. As presented in Listing 1, the signature of an API in dynamically-typed languages (i.e., Python) comprises two parts, the API name, and the arguments. In particular, the arguments restrict what inputs can be processed by the API and thus implicitly reflect its functionality. Besides, each argument has its associated explanation, for instance, the argument `in_features` represents "*the size of each input sample*" as shown in Listing 1. Therefore, we design our signature encoder to capture such semantic features in arguments. Specifically, the input of the signature encoder is a list of arguments with the associated explanations, while the output is the embedding of argument semantics. To achieve that, we leverage N parallel BRNNs, where each BRNN extracts the semantics of one parameter. In particular, according to our statistical analysis of all 67147 Java APIs and 48439 Python APIs in our dataset, the median number of parameters is 2 and 1, respectively, in Java and Python. Additionally, about 91.8% APIs have no more than 5 parameters in Python, while the number is about 98.8% in Java. Therefore, we empirically set N as 5 in order to avoid severe information loss, and also balance effectiveness and efficiency.

However, although different arguments may restrict the usage from different perspectives, the importance of them can be significantly different. The reasons are twofold. First, some arguments are the core inputs to achieve the functionality, while the others may be simple configurations that even can be omitted while using the API (e.g., *name* for layers in TensorFlow). Second, some arguments are commonly used by different APIs (e.g., *size*), and thus it cannot well discriminate the functionality difference between them. Therefore, in order to emphasize the importance of certain arguments, we further incorporate the self-attention mechanism for importance re-weighting.

Formally, suppose the arguments of an API are represented by a list of tuples as $[\langle p_1, q_1 \rangle, \langle p_2, q_2 \rangle, \ldots, \langle p_z, q_z \rangle]$, where z is the number of arguments, p_i denotes the name of the i^{th} argument and q_i denotes the associate explanation of the argument. Then, we take each argument tuple $\langle p_i, q_i \rangle$ as plain text and feed the token sequences to a BRNN network (please refer to Formulas 5-7) to produce the corresponding embedding $\mathbf{H}_{p_i}$. Finally, the output of the signature embedding is calculated by Formulas 8-10, where $\mathbf{V} \in \mathbb{R}^h$ denotes the trainable weight parameters of the model and h denotes the number of hidden units. In particular, $a_i \in [0, 1]$ indicates the normalized weight by *softmax* function for parameter p_i. As a result, the final output embedding $\mathbf{H}_{sig}$ of the signature encoder is expected to encode the semantics in arguments by emphasizing the most important ones.

$$s_i = tanh(\mathbf{H}_{p_i}\mathbf{V}^\top) \tag{8}$$

$$a_i = \frac{e^{s_i}}{\sum_{i=1}^{z} e^{s_i}} \tag{9}$$

$$\mathbf{H}_{sig} = \sum_{i=1}^{n} a_i \mathbf{H}_{p_i} \tag{10}$$

4.2 API Mapping by Transfer Learning

In the pre-training phase, an API embedding model will be trained, which can output the API embeddings when providing the API signature and the associate document as the only inputs. In particular, the output embeddings are expected to reflect the source code semantics (or functionality). In the transfer learning phase, MATL further optimizes the learned embedding model on the task of API mapping since the pre-training phase is not specially designed for this task and thus may not perform the best. Actually, this optimizing process is indeed a typical model fine-tuning, where only a small number of labeled API mappings are required.

Specifically, when given the source API and the target API, their embeddings $\mathbf{H}_{src}$ and $\mathbf{H}_{trg}$ can be obtained by the learned API embedding model. Then, MATL optimizes the model by joint learning via minimizing the loss defined by Formula 11, which enforces the API embedding model to produce close embeddings for mapped APIs. As a consequence, when given the embeddings of both source and target APIs, the API mapping will be decided by their similarities. The larger the similarity is, the higher mapping possibility will be.

$$cos\_similarity(\mathbf{H}_{src}, \mathbf{H}_{trg}) = \frac{\mathbf{H}_{src} \cdot \mathbf{H}_{trg}^\top}{||\mathbf{H}_{src}||\ ||\mathbf{H}_{trg}||} \tag{11}$$

5 EVALUATION

5.1 Research Questions

In order to evaluate the effectiveness of our approach, we have conducted an extensive experiment, where we aim to answer the following research questions.

- **RQ1: How effective is MATL in mapping APIs of dynamic-typed programming languages?** This RQ explores the overall effectiveness of MATL for mapping APIs of dynamic-typed programming languages. Particularly, we use the Python language as the representative since it has been widely used in real practice due to the wide adoption of deep learning techniques as aforementioned. In the experiment, we compare its results with the best approaches studied in our empirical study.
- **RQ2: What is the contribution of each component in MATL?** This RQ analyzes the contribution of each component in MATL by an ablation study, including the two encoders for API embedding. In particular, to investigate the effectiveness of our document encoder for API mapping, we further compare it with the widely-used Bert [21] model. Please note that the Bert model should be the same as GraphCodeBert when there is no source code. By comparing with Bert, we can present the performance distinction of our approach from GraphCodeBert under the condition where source code is unavailable.
- **RQ3: How effective is MATL in mapping APIs of static-typed programming languages?** Since the type feature may largely affect the performance of existing approaches according to our empirical study, to evaluate the effectiveness and generalizability of our approach, we further compare its performance with the latest state-of-the-art document-based approach and the API2API for mapping APIs of static-typed programming languages, i.e., Java and Swift, from

the study [85]. As for the other approaches, i.e., StaMiner, GraphCodeBert, and CODEnn, we do not include them in this experiment due to either we do not have the parallel implementations of Java and Swift or not all the source code implementations of Swift APIs (included in the studied dataset) are available, making them inapplicable in this experiment.

5.2 Experiment Setup

5.2.1 Dataset. To answer the first two research questions, we conducted our experiment on the dataset of APIxDLF, which has been introduced in Section 3.3. To answer the last research question, we compared the performance of our approach with the latest document-based API mapping approach proposed by Zhang et al. [85] (named **Deep-Dive** in this paper) and API2API. Therefore, we adopted the same dataset provided by the authors of the original paper [85] for evaluating Deep-Dive, which consists of 259 one-to-one API mappings between Java and Swift.

5.2.2 Metric. We adopted the same metrics as those used in our empirical study, i.e., *Top-K* and *MRR* as introduced in Section 3.2. Higher values denote better effectiveness.

5.2.3 Implementation and Configuration. We have implemented our approach in a tool named Matl in Python. Specifically, we set the API embedding size as 128, the learning rate as 0.01, and the maximum lengths of document descriptions and source code implementations as 50 and 300, respectively. Besides, in order to ensure the comparison is fair among different approaches across different mapping tasks in RQ1 and RQ2, we set the testing size the same as that in our empirical study. Besides, we employ $\text{BERT}_{\text{Base}}$[1, 21] (Bert for short) in RQ2 since it is much more efficient without seriously sacrificing its performance [21]. Specifically, we adapt it to the API mapping task by replacing the API embedding network in our approach with Bert, which takes the documents of the source and target APIs as the inputs. It has been evaluated to be effective in natural language processing tasks [21] that align with our task of document embedding. Similarly, we have also fine-tuned it over the API mapping task like our approach. For RQ3, we employ different numbers of training instances to study its effects (will be introduced in Section 5.3.3). Please note that when mapping Java to Swift, we make all the Swift APIs (i.e., 259) in the search space for mapping, making the comparison between Matl and Deep-Dive fair since they share the same search space. Finally, as mentioned in Section 3, a large corpus of client code involving mapped APIs of both languages is required by API2API for training API embeddings. Therefore, we have collected the top 1,000 open-source repositories of Java and Swift, respectively, according to the number of *stars* on GitHub [3], which in total include 556,518 source files. Moreover, we have further manually checked the collected source code and ensured all the APIs involved in the Deep-Dive dataset are included. In other words, we first trained the API embedding model in API2API over the collected dataset and then performed the API mapping experiment on the Deep-Dive dataset.

5.3 Result Analysis

5.3.1 RQ1: Overall effectiveness of Matl. Table 5 shows the experimental results of Matl on the dataset of APIxDLF, where we compare it with CODEnn and GraphCodeBert since they achieved the best results on this dataset. As seen, Matl significantly outperforms CODEnn regarding both Top-K and MRR. In particular, Matl improves CODEnn on average by 56.75% and 43.75% regarding Top-1 and MRR, respectively. To investigate whether the improvement is significant or not, we further performed a Wilcoxon signed-rank test like that in our empirical study with the significance level of 0.05. The results demonstrate that Matl significantly outperforms CODEnn with all *p-values* less than 0.05 regarding all metrics. When comparing Matl with GraphCodeBert, it achieves comparable effectiveness. Particularly, compared with GraphCodeBert, Matl achieved a bit higher Top-1 accuracy (i.e., 18.36%), which is regarded as the most important metric since it denotes the

Table 5. Results of MATL and the baselines on APIxDLF

Mapping	Approach	Top-1	Top-5	Top-10	MRR	Mapping	Approach	Top-1	Top-5	Top-10	MRR
	MATL	0.61	0.89	0.93	0.72		MATL	0.60	0.81	0.86	0.7
TF ⇒ PT	CODEnn	0.37	0.60	0.63	0.46	**PT ⇒ TF**	CODEnn	0.32	0.55	0.55	0.37
	GraphCodeBert	0.39	0.89	0.91	0.59		GraphCodeBert	0.60	0.91	0.97	0.75
	MATL	0.61	0.85	0.88	0.73		MATL	0.50	0.79	0.86	0.62
TF ⇒ MX	CODEnn	0.60	0.76	0.79	0.67	**PT ⇒ MX**	CODEnn	0.25	0.63	0.78	0.40
	GraphCodeBert	0.33	0.81	0.97	0.52		GraphCodeBert	0.64	0.95	0.99	0.76
	MATL	0.67	0.83	0.88	0.75		MATL	0.50	0.76	0.86	0.59
TF ⇒ CT	CODEnn	0.42	0.76	0.85	0.58	**PT ⇒ CT**	CODEnn	0.27	0.48	0.70	0.38
	GraphCodeBert	0.44	0.90	0.96	0.61		GraphCodeBert	0.53	0.88	0.98	0.69

[1] We present the results of fine-tuned models for CODEnn and GraphCodeBert in this table since they achieved better results.

first returned result is the desired one and thus can effectively reduce human efforts. Nevertheless, neither one significantly outperforms the other between MATL and GraphCodeBert according to the Wilcoxon signed-rank test (i.e., *p-values* > 0.05). Considering the performance over different mapping tasks, MATL also performs consistently well, with an average Top-1 accuracy of 58%. The close effectiveness between MATL and GraphCodeBert is also due to the transfer learning process, where MATL takes the embedding of GraphCodeBert as the reference. In other words, the effectiveness of MATL is closely related to the performance of GraphCodeBert. Therefore, MATL can possibly be further improved by incorporating more effective source code embedding techniques. In particular, please note that although GraphCodeBert also performs well as MATL, it requires source code implementations, which may restrict its usability in practice. On the contrary, as introduced in Section 4, although MATL depends on GraphCodeBert and requires source code implementations of a large-scale of open-source (easy-to-obtain) projects for model pre-training, it requires neither GraphCodeBert nor the source code implementations for API mapping on the targeted APIs, which attributes to the adoption of the transfer learning in our approach. Therefore, our approach can be used for mapping diverse APIs in real practice. However, since we conducted the experiment on the Python framework APIs only, its generality to a wider range of application scenarios remains to be evaluated. We plan to perform a more comprehensive study to demonstrate its effectiveness over different applications by building high-quality and diverse benchmarks in the near future.

In addition, MATL tends to perform better on TensorFlow than on PyTorch. For example, MATL achieves on average 28% higher Top-1 accuracy for TensorFlow when the target framework is the same, e.g., TF ⇒ MX *vs* PT ⇒ MX. One possible reason is that the descriptions of documents and arguments in TensorFlow usually have higher quality. For example, the description of the argument kernel_size for API `tensorflow.keras.layers.Conv1D` is "*An integer or tuple/list of a single integer, specifying the length of the 1D convolution window*", while the corresponding description in Pytorch is simply "*Size of the convolving kernel*". The results demonstrate that high-quality documents are urgently important for document-based API mapping. Furthermore, the size of training data may also affect the mapping results, as shown in Table 2, the mapping instances of TF ⇒ MX are more than those of PT ⇒ MX, which results in more instances of TF ⇒ MX for model fine-tuning since we keep their testing instances the same in the experiment for a fair comparison (i.e., the number of testing instances affects Top-K and MRR directly). We will further investigate the effect of fine-tuning data in Section 5.3.3.

5.3.2 Contribution of each component. To explore the contribution of each component, we have developed two variants of MATL, i.e., MATL_{doc} and MATL_{sig}, which remove the document encoder and signature encoder respectively. In addition, we further compare our approach with Bert for investigating the performance of our document encoder network in the API mapping task as

Table 6. Ablation study results of MATL

Mapping	Approach	Top-1	Top-5	Top-10	MRR	Mapping	Approach	Top-1	Top-5	Top-10	MRR
TF ⇒ PT	MATL	0.61	0.89	0.93	0.72	**PT ⇒ TF**	MATL	0.60	0.81	0.86	0.7
	MATL$_{sig}$	0.55	0.76	0.85	0.64		MATL$_{sig}$	0.30	0.64	0.70	0.43
	MATL$_{doc}$	0.52	0.82	0.88	0.65		MATL$_{doc}$	0.45	0.73	0.82	0.59
	Bert	0.20	0.66	0.91	0.40		Bert	0.18	0.45	0.88	0.33
TF ⇒ MX	MATL	0.61	0.85	0.88	0.73	**PT ⇒ MX**	MATL	0.50	0.79	0.86	0.62
	MATL$_{sig}$	0.53	0.72	0.84	0.61		MATL$_{sig}$	0.38	0.59	0.72	0.48
	MATL$_{doc}$	0.47	0.88	0.91	0.60		MATL$_{doc}$	0.38	0.69	0.78	0.49
	Bert	0.31	0.81	0.94	0.50		Bert	0.28	0.81	0.91	0.52
TF ⇒ CT	MATL	0.67	0.83	0.88	0.75	**PT ⇒ CT**	MATL	0.50	0.76	0.86	0.59
	MATL$_{sig}$	0.55	0.79	0.88	0.67		MATL$_{sig}$	0.33	0.64	0.70	0.42
	MATL$_{doc}$	0.48	0.70	0.85	0.59		MATL$_{doc}$	0.36	0.61	0.79	0.46
	Bert	0.36	0.81	0.87	0.57		Bert	0.17	0.49	0.74	0.34

[1] MATL$_{sig}$ (MATL$_{doc}$) removes signature (document) encoder in MATL.

explained in Section 5.1. Table 6 presents the comparison results. As seen, compared with MATL, removing either component of the two (i.e., document and signature encoders) will incur a large drop in performance. Specifically, compared with MATL$_{sig}$, the signature embedding model contributes to on average 51.52% higher Top-1 accuracy, while the document embedding model contributes to on average 38.89% higher Top-1 accuracy by comparing with MATL$_{doc}$. The result demonstrates that both these two models significantly contribute to the effectiveness of MATL (i.e., MATL outperforms these two variants with *p-values*<0.05). However, there is an exception that MATL$_{doc}$ achieves slightly higher Top-10 accuracy than MATL in the task of mapping APIs from TensorFlow to MXNet. By a manual inspection, we found the major reason is that the quality of documents in TensorFlow and MXNet is inconsistent. This inconsistency may sacrifice the performance of MATL since such documents can produce noise. On the contrary, MATL$_{sig}$ was not affected.

When comparing MATL with Bert, MATL significantly outperforms Bert on average by 132% and 56.82% regarding Top-1 and MRR (*p-values*<0.05), respectively. In particular, without the signature encoder, MATL$_{sig}$ still outperforms Bert, and the improvement ranges from 35.71% to 175% in terms of Top-1, and the average improvement is 76%, which is significant (*p-value*<0.05). In fact, in most cases, MATL$_{sig}$ outperforms Bert regarding different metrics. The results further demonstrate that the document encoder in MATL is indeed effective in API mapping. This result can also show that our approach should be more effective than GraphCodeBert when the source code implementation is unavailable. The reason is that GraphCodeBert was built on the basis of Bert. They share the same model structure for document embedding.

5.3.3 Performance over static-typed language. According to our empirical study, the missing type information may largely affect the performance of existing API mapping approaches. To evaluate whether MATL is still effective and outperforms the state-of-the-art API mapping approaches when the type features can be used, we further compare MATL with the state-of-the-art Deep-Dive and API2API for mapping APIs between Java and Swift. Deep-Dive is also a document-based approach that depends on the semantic similarity of documents and signatures for API mapping. Since Deep-Dive is not open-source, we take its results from the original paper directly, where the metric of MRR was not reported. Table 7 shows the results of both MATL and the baselines. Particularly, since transfer learning is usually to overcome the insufficiency of training data, we also take different sizes of training data (i.e., 100~200) in our experiment for comparison. For each configuration, we have performed a 5-fold cross evaluation and report the average number. The results show that our approach can still outperform the state-of-the-art Deep-Dive and API2API for mapping APIs of

Table 7. Results of mapping APIs between Java and Swift

Approach	Training	Top-1	Top-5	Top-10	MRR
MATL	100	0.43	0.70	0.79	0.54
	125	0.45	0.75	0.80	0.57
	150	0.47	0.75	0.81	0.59
	175	0.50	0.74	0.81	0.60
	200	0.51	0.75	0.83	0.60
Deep-Dive	-	0.39	0.63	0.76	-
API2API	-	0.18	0.36	0.54	0.33

static-typed programming languages. In particular, MATL improves Deep-Dive and API2API by up to 30.77% and 183.33% in terms of Top-1. In particular, the evaluation results also demonstrate that more training data may contribute to higher effectiveness. Nevertheless, our approach can always outperform the baseline methods with only a very small set of data (i.e., 100) for model fine-tuning regarding all the metrics. In summary, the results further confirm the effectiveness and generalizability of our approach.

6 IMPLICATIONS AND FUTURE WORK

According to the empirical study and our experimental results, we have summarized a set of implications that may facilitate future research in this area.

Constructing open-source and high-quality benchmarks. Benchmark plays a critical role in scientific research as they are usually the basis for constructing or evaluating the performance of techniques. As aforementioned in this article, most of the existing studies related to API mapping do not publish their experimental data and/or implementations, which indeed hinders the fair comparison among different approaches and the health development of this research area. In fact, constructing a high-quality benchmark for API mapping is not easy as it requires relatively high expertise to make the constructed mappings reliable. In particular, it can be infeasible for some scenarios, e.g., constructing parallel implementations of the same functionalities. In our study, we have tried our best to collect such parallel implementations. However, although the deep learning APIs are widely used among diverse projects, the desired implementations are still rare because the code structures from different projects can vary greatly (existing approach, e.g., StaMiner, requires parallel implementations within a single method.) even though they are implementing the same neural networks. As a result, existing approaches cannot be applied. In this study, the parallel implementations are provided by our industrial partner and thus cannot be published due to confidentiality policies. On the contrary, we have published our manually-constructed dataset APIxDLF to facilitate future research and comparison. Nevertheless, high-quality benchmarks in the API-mapping research area are still in urgent need, such as constructing API mappings among other programming languages.

Developing high-quality documents of APIs. According to our experimental results, the quality of documents is important for better API mapping. A good document should sufficiently introduce the functionality of the API, including the usage and the constraints, and even provide examples for better understanding. Incomplete documents (e.g., keywords or phrases) are usually inadequate or even misleading, especially for novice developers. Therefore, it would be of great value if the API vendors could provide high-quality documents since they can benefit both end-users of the APIs and the tool vendors for performing downstream tasks. In fact, with the recent advance

of deep learning techniques, leveraging large language models to generate high-quality documents automatically can also be a potential solution.

Integrating source code implementations. According to the results of our empirical study, the source code implementations are effective in promoting API mapping as they contain the most accurate semantics of the API functionality. Although MATL is effective for mapping APIs in Python programs without the need to access the source code implementations, its performance still has much room for improvement. Actually, if the source code implementations of the APIs are available, MATL potentially can be further enhanced from two different perspectives. First, the API embedding network in MATL can be specially fine-tuned for the targeted mapping tasks, which can further optimize the performance of MATL as it may produce more accurate embeddings of the APIs and thus produce better mapping results. Second, the source code implementations can also be independently embedded, such as using the GraphCodeBert model, whose results (i.e., embedding vectors) can be incorporated into MATL through vector concatenation. We leave the empirical exploration of them to our future work. Nevertheless, MATL can be a good choice when the targeted APIs are close-sourced since it does not rely on their source code implementations.

Achieving complete API translation. To the best of our knowledge, there is currently no work to map the parameters of the APIs. This means that after developers get the API mappings, they may still need to consult the relevant documents to understand the usage of the corresponding parameters, which requires extra human effort. In fact, if the APIs are well documented, it is possible to achieve the parameter mapping as well, along with the API mapping. In the future, we plan to make full use of the API description of parameters for more accurate API mapping results, including mapping corresponding parameters. Actually, the program translation techniques (will be introduced in Section 8.3) can be incorporated in this application.

7 THREATS TO VALIDITY

External Threats to Validity mainly lie in the selection of studied approaches and the datasets for evaluation. To mitigate these threats, we have included existing approaches from all two categories, which can increase our confidence in the reliability and significance of our experimental results and conclusions in the paper. Besides, we employed the APIs from the deep learning frameworks as the main subjects since they are widely used in practice, which may also not be representative of a broader range. The reasons are twofold: (1) The mapping APIs from different frameworks may be similar to each other. Therefore, when mapping APIs whose names are largely different, the effectiveness of our approach remains to be evaluated. (2) Deep learning APIs may have similar naming conventions across different frameworks. For example, they all use the *snake_case* naming convention and similar abbreviations (e.g., "conv" for "convolutional"), which may also affect the results of MATL as well. To evaluate the generality of our approach, we also adopted an extra dataset for Java and Swift from existing work [85] and compared MATL with the state-of-the-art method. The results further confirmed the effectiveness of our approach. In addition, some approaches were originally designed and evaluated for transforming programs across different programming languages, e.g., StaMiner. In contrast, we conducted our empirical study within a single programming language, i.e., Python, which may affect the generality of our empirical results. However, since we lack extra datasets designed for the API mapping task, it is currently infeasible for us to conduct more comprehensive experiments over different mapping tasks. In the future, we plan to construct high-quality benchmarks that can enable more experiments and facilitate future research.

Internal Threats to Validity mainly lie in the tool implementations and dataset constructions. However, although API mapping is an important task and many approaches have been proposed, there is still no public dataset that can be used, and almost all approaches are not open-source, which

is indeed a great barrier that hinders the healthy development of this research area. In our study, we have tried our best to re-implement the corresponding approaches by following the instructions in the corresponding papers. Two experienced authors have carefully checked the correctness of all the implementations and scripts to ensure they are correct. Furthermore, to ensure the quality of our datasets, three authors have carefully labeled the mapping relations independently by querying the websites and discussing inconsistencies. Nevertheless, we conducted the first extensive study on API mapping approaches, and all our data and implementations are open-source to facilitate the replication and comparison experiments in future studies.

Construct Threats to Validity lie in the measurements and randomness in the experiments. To mitigate the threats in measurement selection, we have adopted two different metrics that are widely used in existing studies [14, 85]. To reduce the effect of randomness, we repeated our experiments 10 times and calculated the average results in our study.

8 RELATED WORK

8.1 API Mapping

In this section, we introduce the most related work to our approach in API mapping. According to Section 3, API mapping approaches can be categorized as usage-based and document-based. Usage-based approaches usually depend on the behavior characteristics of APIs, such as the studied StaMiner [57]. Besides, Zhong et al. [88] proposed MAM, which can mine the API relations based on the API transformation graph. Wu et al. [81] proposed Aura, which considers calling dependency and text similarity to facilitate mining, while Meng et al. [55] improves it based on historical information. Lamothe et al. [44] proposed A3, which learns API migration patterns from code examples. Bui et al. proposed SAR [14], which leverages the domain adaption technique to transform and align different vector spaces across languages based on adversarial training. Nguyen et al. [60] proposed API2API, which maps APIs according to the similarity of their vectorial representations. Similarly, Chen et al. [16] proposed SImilarAPI that takes comment sequences and API names into account to reinforce API behavior characteristics. To further enhance the representation of API, several approaches leverage information from the document to facilitate API mapping. Gu et al. [27] proposed DeepAM to build the relationship between code and its description so that API sequences with embeddings tend to have similar natural language descriptions. Pandita et al. [66] proposed TMAP to compare similarities between API documents for mapping. These techniques concentrate more on static-typed languages such as Java and C#, etc., and rely on source code information of APIs. Unlike these works, MATL performs API mapping by learning the deep semantics of APIs by leveraging transfer learning. Besides, we conducted the first extensive study to explore the performance of existing approaches.

8.2 Transfer Learning

Transfer learning aims to improve the performance of a learner on the target domain by transferring the knowledge it learned from the different but related source domain. In this way, transfer learning can reduce the dependence of training on the large-scale dataset of the target domain, which is suitable for training scenarios with limited training data [80]. A common usage of transfer learning is to pre-train a deep neural network on the source domain, which is usually large-scale multi-category datasets with complex semantics, and then fine-tune it based on the training set from the target domain, which usually has limited data scale with domain-specific knowledge [89]. It has been widely used in Computer Vision (e.g., image recognition [52]) and Natural Language Processing domains (e.g., text generation [18]).

In recent years, transfer learning has also been applied to software engineering domains [53, 63]. The proposal of code representation models boosts the usage of transfer learning. Several pre-trained code representation models, such as CodeBERT [25], GraphCodeBERT [29], and UniXcoder [28], are demonstrated to be effective on multiple downstream tasks (e.g., vulnerability prediction [42], clone detection [74], and code search [77]) through transfer learning. In addition to these models and tasks, Kang et al. [36] proposed APIRecX, which leverages transfer learning to utilize knowledge learned from the source library to facilitate API recommendation for the target library. Liu et al. [48] proposed CugLM according to pre-trained transformer-based architecture for code completion tasks. Kandade et al. [35] learned to predict precise exception types according to context code information through transfer learning from CuBERT trained on the large-scale dataset. Different from the works above, we designed the pre-training phase of MATL to train the encoder to learn code semantics embeddings and leverage transfer learning to further optimize the semantics for API mapping tasks without source code.

8.3 Program Translation

Program translation, as a downstream task of API mapping, is also related to our approach and has been widely studied [24, 49, 61, 78]. Typically, the program translation methods can be classified into two categories, one of which takes the API mapping results as inputs and then transform the programs according to the mapping relations (named rule-based methods) [12, 20, 47, 62, 65], while the other one transforms the source programs into the target programs directly without the aid of API mapping result (named learning-based methods) [9, 10, 17, 37, 58, 59, 71].

Rule-based methods generally depend on a set of API mapping rules for program transformation, which can be viewed as a pattern that matches a piece of code with certain contexts, and then the target code will be generated to replace the matched code. For example, Cordy et al. [20] proposed TXL, a source-to-source transformation language that uses first-order functional programming and term rewriting rules to support flexible program translation. Padioleau et al. [65] proposed SmPL, a domain-specific language used for describing the contexts of the source code to be replaced and the target code to be generated. Similarly, Bravenboer et al. [12] proposed Stratego, a language for implementing transformations based on the paradigm of programmable rewriting strategies. Furthermore, Li et al. [47] proposed SWIN by adding extra conditions to Twinning [62] to address the type-safe problem in program translation. It allows programmers to specify a class of program changes.

While rule-based approaches are typically more accurate, they usually require considerable expert knowledge and effort for mining the API mapping relations. In contrast, learning-based methods take this process as a program translation task between different programming languages, which does not require the API mappings as inputs. For example, Nguyen et al.[58, 59] proposed lpSMT and mppSMT to map or migrate source codes between two programming languages. LpSMT directly applied phrase-based Statistical Machine Translation on lexical tokens to migrate Java to C#, producing much semantically incorrect code. More precisely, mppSMT treated a program as a sequence of syntactic units and mapped the tokens within the syntactic units, which could achieve a higher syntax accuracy. Karaivanov et al. [37] improved the legitimacy of the translated code by inspecting the prefix grammar and adding rules to the translation process. In addition, Statistical Machine Translation was also leveraged to convert Python 2 to Python 3 code [9]. Recently, Chen et al. [17] proposed a tree-to-tree neural network to align the sequence with the grammar. Though existing phrase-based SMT approaches achieved good performance, the reliance on parallel code in different programming languages still constraints their usability in practice.

In order to eliminate the dependency on parallel code corpus, unsupervised training approaches for programming languages were proposed. Roziere et al.[71] proposed Transcoder to train a

fully unsupervised neural transcompiler. They leveraged a large amount of monolingual source code to train the model and translate among three popular languages, i.e., C++, Java, and Python. Furthermore, Ahmad et al. [10] presented a summarize-and-generate-based approach to enable unsupervised program translation training via back-translation. Specifically, they performed the program translation task by first performing the code summarization and then the code generation process. As a consequence, the training process would be independent of parallel implementations.

Compared with these program translation techniques, the output of our approach can be taken as the input of them, especially of the rule-based methods. In addition, MATL may also be combined with those learning-based program translation methods and potentially improve their performance by performing a post-refinement according to the API mappings. We leave it as our future work.

8.4 API Evolution

API evolution, usually changing APIs' signatures or their functionalities, may also require updating the APIs accordingly, and thus is also related to our work. Specifically, due to the increasing complexity [43, 54] and continuous changes [23, 43, 46], APIs often evolve in software systems [38, 39, 45, 64, 72], or even frameworks and platforms [22, 32, 33, 41, 51, 56, 82, 83, 86, 87]. Earlier studies of API evolution focused more on statically-typed languages, such as Java [22] and Android [51, 64, 79, 82]. Recently, with the growing popularity of dynamically-typed languages, more researchers have paid attention to API evolution of dynamically-typed languages, and a series of studies have been conducted [33, 56, 86, 87]. For example, Zhang et al. [87] reported that breaking API changes (which are backward-incompatible and would cause compilation or runtime problems) in Python frameworks happen more frequently and tend to have larger impacts than those in Java. In particular, API evolution in Python frameworks may cause crashes or unexpected behaviors in client applications. Based on the findings, they proposed PYCOMPAT to detect incompatibility issues caused by misusing evolved APIs. Similarly, Zhang et al. [86] analyzed the API evolution trends in the Tensorflow framework. Moreover, they also emphasized the value of API parameter descriptions for the proper use of APIs. On the contrary, Nascimento et al. [56] conducted an empirical study to analyze the API deprecation in the evolution of JavaScript. They found that the deprecated APIs were often not removed from the projects in time although the new APIs have been adopted. Furthermore, Lamothe et al. [43] conducted a more comprehensive and systematic review of existing research on API evolution, including empirical studies, novel techniques, and datasets. Based on that, they summarized existing challenges and a set of suggestions for future research. Different from the above studies, Hora et al. [32] explored the impact of API evolution from the developers' perspective. Their study shows that developers tend to have a lag in the API evolution process. In other words, the evolved APIs are usually updated with delay by end-users. Nevertheless, automated API mapping and updating are common tasks in real practice, such as in the scenario of API evolution. These studies further demonstrate the importance of our work.

9 CONCLUSION

In this paper, we have conducted the first extensive study to explore the performance of existing approaches for mapping APIs in dynamic-typed language, which shows that the source code implementations can significantly improve the performance of API mapping. Inspired by this, we propose a novel API mapping approach that leverages the transfer learning technique to embed APIs' functionality semantics without knowing the source code implementations. The evaluation results for mapping APIs of both dynamic-typed and static-typed languages demonstrate that our approach can significantly outperform the state-of-the-art approaches.

ACKNOWLEDGMENTS

We thank the editors and anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62202324, 62322208 and 62232001.

REFERENCES

[1] Accessed: May 2023. Bert. https://storage.googleapis.com/bert_models/2020_02_20/uncased_L-12_H-768_A-12.zip.
[2] Accessed: May 2023. CNTK. https://cntk.azurewebsites.net/pythondocs/.
[3] Accessed: May 2023. GitHub. https://github.com/.
[4] Accessed: May 2023. GraphCodeBert. https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch/.
[5] Accessed: May 2023. MXNet. https://mxnet.apache.org/.
[6] Accessed: May 2023. PyTorch. https://pytorch.org/tutorials/.
[7] Accessed: May 2023. TensorFlow. https://tensorflow.google.cn/.
[8] Accessed: May 2023. Wordninja. https://github.com/keredson/wordninja/.
[9] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. Using machine translation for converting *Python 2* to *Python 3* code. *PeerJ Prepr.* 3 (2015), e1459.
[10] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. In *EACL*. Association for Computational Linguistics, 1520–1534.
[11] Haya Brama, Lihi Dery, and Tal Grinshpoun. 2022. Evaluation of Neural Networks Defenses and Attacks using NDCG and Reciprocal Rank Metrics. *CoRR* abs/2201.05071 (2022).
[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* 72, 1-2 (2008), 52–70.
[13] Nghi D. Q. Bui and Lingxiao Jiang. 2018. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *ICSE (NIER)*. ACM, 33–36.
[14] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *ESEC/SIGSOFT FSE*. ACM, 796–806.
[15] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. 2009. Expected reciprocal rank for graded relevance. In *CIKM*. ACM, 621–630.
[16] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Trans. Software Eng.* 47, 3 (2021), 432–447.
[17] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *ICLR (Workshop)*. OpenReview.net.
[18] Yen-Chun Chen, Zhe Gan, Yu Cheng, Jingzhou Liu, and Jingjing Liu. 2020. Distilling Knowledge Learned in BERT for Text Generation. In *ACL*. Association for Computational Linguistics, 7893–7905.
[19] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F. P. O'Boyle. 2020. M3: Semantic API Migrations. *CoRR* abs/2008.12118 (2020). arXiv:2008.12118 https://arxiv.org/abs/2008.12118
[20] James R. Cordy. 2006. The TXL source transformation language. *Sci. Comput. Program.* 61, 3 (2006), 190–210.
[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.
[22] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *CSMR-WCRE*. IEEE Computer Society, 64–73.
[23] Danny Dig and Ralph E. Johnson. 2005. The Role of Refactorings in API Evolution. In *ICSM*. IEEE Computer Society, 389–398.
[24] Joost Engelfriet. 1972. Translation of Simple Program Schemes. In *ICALP*. North-Holland, Amsterdam, 215–223.
[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings) (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.
[26] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. ACM, 933–944.
[27] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *IJCAI*. ijcai.org, 3675–3681.
[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *ACL (1)*. Association for Computational Linguistics, 7212–7225.

[29] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net.
[30] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms. In *ASE*. IEEE, 810–822.
[31] Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2022. AndroEvolve: automated Android API update with data flow analysis and variable denormalization. *Empir. Softw. Eng.* 27, 3 (2022), 73.
[32] André C. Hora, Romain Robbes, Marco Túlio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. 2018. How do developers react to API evolution? A large-scale empirical study. *Softw. Qual. J.* 26, 1 (2018), 161–191.
[33] Mingzhe Hu and Yu Zhang. 2023. An empirical study of the Python/C API on evolution and bug patterns. *J. Softw. Evol. Process.* 35, 2 (2023).
[34] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *ASE*. ACM, 293–304.
[35] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *ICML (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121.
[36] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model. In *EMNLP (1)*. Association for Computational Linguistics, 3425–3436.
[37] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Onward!* ACM, 173–184.
[38] Holger Knoche and Wilhelm Hasselbring. 2022. Continuous API Evolution in Heterogenous Enterprise Software Systems. In *Software Engineering (LNI, Vol. P-320)*. Gesellschaft für Informatik e.V., 49–50.
[39] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. 2023. Web API evolution patterns: A usage-driven approach. *J. Syst. Softw.* 198 (2023), 111609.
[40] Philipp Koehn. 2010. *Statistical Machine Translation*. Cambridge University Press.
[41] Hobum Kwon, Juwon Ahn, Sunggyu Choi, Jakub Siewierski, Piotr Kosko, and Piotr Szydelko. 2018. An Experience Report of the API Evolution and Maintenance for Software Platforms. In *ICSME*. IEEE Computer Society, 587–590.
[42] Sunjae Kwon, Jong-In Jang, Sungu Lee, Duksan Ryu, and Jongmoon Baik. 2022. CodeBERT Based Software Defect Prediction for Edge-Cloud Systems. In *ICWE Workshops (Communications in Computer and Information Science, Vol. 1668)*. Springer, 11–21.
[43] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2022. A Systematic Review of API Evolution Literature. *ACM Comput. Surv.* 54, 8 (2022), 171:1–171:36.
[44] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. A3: Assisting Android API Migrations Using Code Examples. *IEEE Trans. Software Eng.* 48, 2 (2022), 417–431.
[45] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. 2022. A Large-scale Empirical Assessment of Web API Size Evolution. *J. Web Eng.* 21, 6 (2022), 1937–1980.
[46] M. M. Lehman. 1996. Laws of Software Evolution Revisited. In *EWSPT (Lecture Notes in Computer Science, Vol. 1149)*. Springer, 108–124.
[47] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation between APIs. In *PEPM*. ACM, 91–102.
[48] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *ASE*. IEEE, 473–485.
[49] X. Liu and P. Gontey. 1987. Program Translation by Manipulating Abstract Syntax Trees. In *C++ Workshop*. USENIX Association, 345–360.
[50] Qun Luo, Weiran Xu, and Jun Guo. 2014. A Study on the CBOW Model's Overfitting and Stability. In *Web-KRM@CIKM*. ACM, 9–12.
[51] Tarek Mahmud, Meiru Che, and Guowei Yang. 2022. Android API Field Evolution and Its Induced Compatibility Issues. In *ESEM*. ACM, 34–44.
[52] Muazzam Maqsood, Faria Nazir, Umair Khan, Farhan Aadil, Habibullah Jamal, Irfan Mehmood, and Oh-Young Song. 2019. Transfer Learning Assisted Classification and Detection of Alzheimer's Disease Stages Using 3D MRI Scans. *Sensors* 19, 11 (2019), 2645.
[53] Antonio Mastropaolo, Nathan Cooper, David Nader-Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *CoRR* abs/2206.08574 (2022).
[54] Anderson S. Matos, João Bosco Ferreira Filho, and Lincoln S. Rocha. 2019. Splitting APIs: an exploratory study of software unbundling. In *MSR*. IEEE / ACM, 360–370.

[55] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *ICSE*. IEEE Computer Society, 353–363.
[56] Romulo Nascimento, André C. Hora, and Eduardo Figueiredo. 2022. Exploring API Deprecation Evolution in JavaScript. In *SANER*. IEEE, 169–173.
[57] 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 *ASE*. ACM, 457–468.
[58] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *ESEC/SIGSOFT FSE*. ACM, 651–654.
[59] 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 *ASE*. IEEE Computer Society, 585–596.
[60] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *ICSE*. IEEE / ACM, 438–449.
[61] Douglas Niehaus. 1991. Program representation and translation for predictable real-time systems. In *RTSS*. IEEE Computer Society, 53–63.
[62] Marius Nita and David Notkin. 2010. Using twinning to adapt programs to alternative APIs. In *ICSE (1)*. ACM, 205–214.
[63] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *IJCAI*. ijcai.org, 5546–5555.
[64] Koki Ogasawara, Tetsuya Kanda, and Katsuro Inoue. 2020. On the Variations and Evolutions of API Usage Patterns: Case Study on Android Applications. In *ICSE (Workshops)*. ACM, 746–753.
[65] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*. ACM, 247–260.
[66] Rahul Pandita, Raoul Praful Jetley, Sithu D. Sudarsan, and Laurie A. Williams. 2015. Discovering likely mappings between APIs using text mining. In *SCAM*. IEEE Computer Society, 231–240.
[67] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. 2017. Statistical migration of API usages. In *ICSE (Companion Volume)*. IEEE Computer Society, 47–50.
[68] Mark Pilgrim and Simon Willison. 2009. *Dive into python 3*. Vol. 2. Springer.
[69] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. 2018. RACK: Automatic API Recommendation using Crowdsourced Knowledge. *CoRR* abs/1807.02953 (2018).
[70] Denise Rey and Markus Neuhäuser. 2011. Wilcoxon-Signed-Rank Test. (2011), 1658–1659.
[71] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NeurIPS*.
[72] Paul Schmiedmayer, Andreas Bauer, and Bernd Bruegge. 2023. Reducing the Impact of Breaking Changes to Web Service Clients During Web API Evolution. In *MOBILESoft*. IEEE, 1–11.
[73] Yang Song, Ziming Zhuang, Huajing Li, Qiankun Zhao, Jia Li, Wang-Chien Lee, and C. Lee Giles. 2008. Real-time automatic tag recommendation. In *SIGIR*. ACM, 515–522.
[74] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. 2022. Generalizability of Code Clone Detection on CodeBERT. In *ASE*. ACM, 143:1–143:3.
[75] Ellen M. Voorhees and Dawn M. Tice. 1999. The TREC-8 Question Answering Track Evaluation. In *TREC (NIST Special Publication, Vol. 500-246)*. National Institute of Standards and Technology (NIST).
[76] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. 2016. Transforming Programs between APIs with Many-to-Many Mappings. In *ECOOP (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:26.
[77] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging Pre-trained Models and Downstream Tasks for Source Code Understanding. In *ICSE*. ACM, 287–298.
[78] Richard C. Waters. 1988. Program Translation via Abstraction and Reimplementation. *IEEE Trans. Software Eng.* 14, 8 (1988), 1207–1228.
[79] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps. In *ASE*. ACM, 226–237.
[80] Karl R. Weiss, Taghi M. Khoshgoftaar, and Dingding Wang. 2016. A survey of transfer learning. *J. Big Data* 3 (2016), 9.
[81] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *ICSE (1)*. ACM, 325–334.
[82] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *ICSE*. ACM, 886–898.
[83] Ping Yu, Fei Yang, Chun Cao, Hao Hu, and Xiaoxing Ma. 2018. Mining API usage change rules for software framework evolution. *Sci. China Inf. Sci.* 61, 5 (2018), 050108:1–050108:3.

[84] Jingxuan Zhang, He Jiang, Zhilei Ren, and Xin Chen. 2018. Recommending APIs for API Related Questions in Stack Overflow. *IEEE Access* 6 (2018), 6205–6219.
[85] Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. 2020. Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping. In *ICPC*. ACM, 106–116.
[86] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John C. Grundy. 2021. Unveiling the Mystery of API Evolution in Deep Learning Frameworks: A Case Study of Tensorflow 2. In *ICSE (SEIP)*. IEEE, 238–247.
[87] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *SANER*. IEEE, 81–92.
[88] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *ICSE (1)*. ACM, 195–204.
[89] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. A Comprehensive Survey on Transfer Learning. *Proc. IEEE* 109, 1 (2021), 43–76.