Adaptive Deep Code Search

Chunyang Ling*
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Peking University
Beijing, China
lingcy@pku.edu.cn

Zeqi Lin*
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Peking University
Beijing, China
linzeqi@pku.edu.cn

Yanzhen Zou†
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Peking University
Beijing, China
zouyz@pku.edu.cn

Bing Xie
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Peking University
Beijing, China
xiebing@pku.edu.cn

ABSTRACT

Searching code in a large-scale codebase using natural language queries is a common practice during software development. Deep learning-based code search methods demonstrate superior performance if models are trained with large amount of text-code pairs. However, few deep code search models can be easily transferred from one codebase to another. It can be very costly to prepare training data for a new codebase and re-train an appropriate deep learning model. In this paper, we propose AdaCS, an adaptive deep code search method that can be trained once and transferred to new codebases. AdaCS decomposes the learning process into embedding domain-specific words and matching general syntactic patterns. Firstly, an unsupervised word embedding technique is used to construct a matching matrix to represent the lexical similarities. Then, a recurrent neural network is used to capture latent syntactic patterns from these matching matrices in a supervised way. As the supervised task learns general syntactic patterns that exist across domains, AdaCS is transferable to new codebases. Experimental results show that: when extended to new software projects never seen in the training data, AdaCS is more robust and significantly outperforms state-of-the-art deep code search methods.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**.

KEYWORDS

code search, deep learning, domain adaption

*Both authors contributed equally.
†Corresponding author.

```java
public List<TrfvltBillInfoVo> getBillsByMonth(String accountInfo,
      String date) throws Exception {
    Map<String, Object> accountVo=TrfvltAppUtils.json2AccountMap(
          accountInfo);
    String carLicense=(String) accountVo.get("carLicense");
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("carLicense", carLicense);
    params.put("yearMonth", date);
    return billDao.getBillsByMonth(params);
}
```

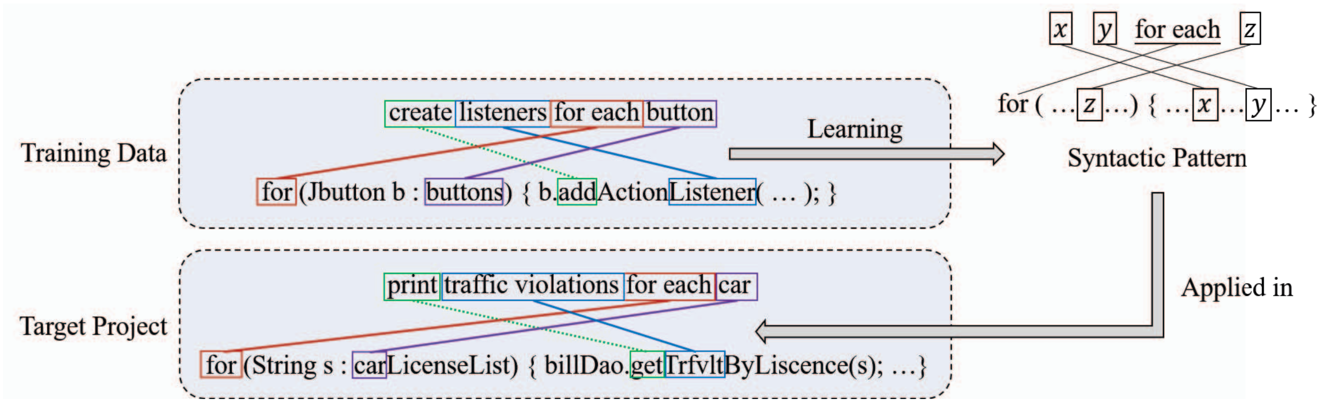
Figure 1: A code snippet in an industrial codebase, containing domain-specific words such as "*trfvlt*", and "*carLiscense*" that rarely appear in the Github training corpus

ACM Reference Format:
Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive Deep Code Search. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389278

1 INTRODUCTION

Code search is a common practice during software development [40, 48]. To implement a certain functionality, developers usually need to search and reuse previously written code by performing natural language queries over a large-scale codebase. Recent years, deep learning-based code search methods have been proposed and achieved superior performances. For example, Gu et al.[14] proposed DeepCS, a deep code search tool based on Recurrent Neural Networks (RNN). Chen et al.[8] proposed BVAE, a deep code search method based on Variational AutoEncoders (VAEs) [27, 45]. The basic idea is to train a deep model with a large number of text-code pairs, then use the model to search on the same codebase. The deep code search model can better learn the semantic meanings so that it improves the search accuracy of natrual language queries.

However, existing deep code search models perform badly when applied to a new codebase. In practice, we train a deep code search model with data collected from GitHub, then it can work well on searching GitHub. Nevertheless, when we apply it to an real-world industrial software codebase about smart city (here we do not



Training Data
create listeners for each button
for (Jbutton b : buttons) { b.addActionListener(…); }
Learning
x y for each z
for (… z …) { … x … y … }
Syntactic Pattern
Applied in
Target Project
print traffic violations for each car
for (String s : carLicenseList) { billDao.getTrfvltByLiscence(s); …}

Figure 2: An example of domain-specific words and syntactic patterns in code search

give detailed information about the project because of the double-blind review process), the model suffers from the out-of-vocabulary (OOV) problem and results in a low search accuracy. Figure 1 shows a code snippet in the target industrial codebase. This code snippet contains several domain-specific words that rarely appear in the training data, such as "*trfvlt*" (a.k.a., traffic violation), "*bill*" and "*carLiscense*". Thus, the deep code search model fails to represent the semantic meanings properly here.

For these deep code search methods, if we want to achieve superior performance on the real-world industrial codebase in a company, we have to prepare enough training data for the codebase and train a specific search model for it. However, preparing training data (i.e. text-code pairs) for deep models is very costly in regards to time and money. There may be only a few codes commented with proper natural language descriptions in the target codebase (e.g., some private industrial codebases), so that it is quite difficult to extract sufficient text-code pairs automatically.

In this paper, we propose AdaCS, an adaptive deep code search method, which can be trained with data from one codebase (e.g., Github) and transferred to many other codebases (e.g., real-world industrial codebases) easily. AdaCS learns domain-specific word meanings and general syntactic patterns respectively. Firstly, an unsupervised word embedding technique is utilized to construct a matching matrix for each text-code pair, in which entries represent the similarities between words (i.e., word meanings). Then, a recurrent neural network is utilized to capture the latent syntactic patterns from these lexical matching matrices in a supervised way. When it comes to a new software codebase, AdaCS learns the domain-specific words from its code corpus, constructs a matching matrix for each pair of natural language query and candidate code snippet, and then use the trained recurrent neural network to measure the degree of matching between the query and the code.

To evaluate the effectiveness of AdaCS, we perform the adaptive code search task on a target Java codebase collected from 3 well-known projects: *Apache Lucene*, *Apache POI* and *JFreeChart*. This test codebase contains 1,606 query-code pairs. To simulate the adaptive code search scenario, we first train the model with a large corpus from GitHub consisting of 77,920 text-code pairs, and then apply the model directly to search on the test codebase. Note that all training text-code pairs related to the 3 target projects are filtered out to ensure the adaptive experimental scene. Experimental results show that AdaCS improves the top-5 hit rate of the adaptive code search from 65.9% to 88.2% when compared against state-of-the-art methods.

Contributions of this paper include:

- The proposal of AdaCS, a new deep code search method that decomposes the learning process into embedding domain-specific words and matching general syntactic patterns.
- We apply AdaCS to adaptive code search tasks in which it can be trained once and transferred to new codebases easily. To our knowledge, we are the first to apply transfer learning more specifically unsupervised domain adaption to code search.
- We conduct experiments on adaptive code search tasks and the results demonstrate that AdaCS is more robust and outperforms the state-of-the-art deep code search methods.

The rest of this paper is organized as follows. Section 2 describes the motivation of this paper. Section 3 describes the details of AdaCS. Section 4 presents the evaluation setup, and Section 5 presents the evaluation results. Section 6 discusses related work. Section 7 concludes this paper.

2 MOTIVATION

This paper is motivated by a common observation that the matching patterns between a natural language text and a code snippet usually exist at different levels of granularity(e.g. from the lexical to syntactic levels). Figure 2 demonstrates both the lexical and syntactic matching patterns in text-code pairs and how a syntactic pattern can work across codebase.

Consider the query "*create listener for each button*" and its relevant code snippet in the training corpus. We denote this text-code pair as p_0. The matched words between text and code are linked via colored lines. Solid lines represent identical matches(e.g., "*listeners*" and "*Listener*"), and dotted lines represent similar matches(e.g., "*create*" and "*add*"). There is a syntactic pattern underlying this text-code pair: the "*x for each y*" syntactic structure in natural language text corresponds to the "*for (... : ... y ...) { ... x ... }*" syntactic structure in code.

We can see that the word-level similarities are quite diverse and domain-specific (e.g., "*traffic violation*"), while the syntax-level patterns are more general and abstract. The other text-code pair in the target codebase(i.e., "*print traffic violations for each car*"), denoted as p_1, contains some words that never appear in the training corpus. However, p_1 shares the similar syntactic patterns with p_0, so that the matching score of p_1 should be high. In fact, the word-level similarities are mostly independent of the sentence-level matching patterns in many cases.

We decompose the semantic similarity between a natural language text and a code snippet into two factors:

- **Domain-specific words meaning** refers to the semantic similarity between words. For example, "*trfvlt*" is a synonym of "*traffic violation*". Each codebase may contain lots of domain-specific words, which causes that words meaning can hardly work across codebase.
- **Syntactic pattern** refers to sequential patterns in code and natural language text. For example, "*print traffic violations for each car*" and "*print the car for each traffic violation*"share many common keywords, but their semantic meanings and corresponding code snippets are very different. Most syntactic patterns are universal across codebases so that we can reuse these patterns when applied to a new target codebase.

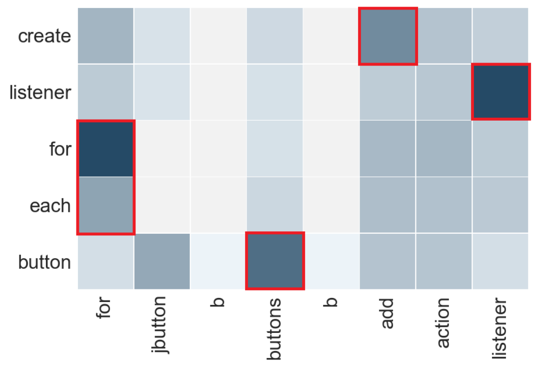
Our main idea is to **deal with domain-specific word meanings and domain-adaptable syntactic patterns respectively.** That is to learn the syntactic patterns from training corpus separately, and learn the domain-specific words meaning in an unsupervised way for each codebase. Therefore, we can combine the learned syntactic patterns with the domain words meaning and transfer them to a new target codebase.

However, for existing deep code search methods, they learn word meanings and syntactic patterns jointly during model training, so that the learned models cannot be transferred from the training corpus to a new type of codebase.

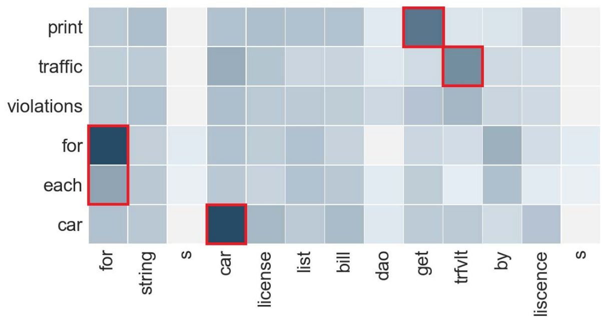
We formalize the code search problem as matching between a natural language query q and a code snippet c. Our goal is to define the matching function $f(q, c)$. Given a natural language query q, we compute the score $f(q, c)$ for each candidate code snippet c, then the score is used to rank all these candidate code snippets in descending order.

It has been widely recognized that **making a good matching decision requires of taking into account the rich interaction structures in the text matching process** [16, 42]. A large number of syntactic patterns can be mined from the interaction structures between the query and the document. We represent the interaction structure of each text-code pair as a lexical matching matrix. Figure 3 illustrates the lexical matching matrices of the aforementioned text-code pairs p_0 (Figure 3b) and p_1 (Figure 3a).

In the interaction matrix $\mathbf{M}$, each cell $\mathbf{M}_{i,j}$ represents the lexical similarity between the i-th word in q and the j-th word in c. The darker a cell is colored, the higher the corresponding lexical similarity is. Although the word-level similarity is domain-specific, it can be computed in an unsupervised way: for p_0, lexical similarity is computed through performing word embedding technique [60] on GitHub corpus; for p_1, lexical similarity is computed through performing word embedding technique on the target industrial corpus. Note that we do not need annotated text-code pairs for the target codebase. We denote the lexical matching matrix of p_0 and p_1 as $\mathbf{M}_0$ and $\mathbf{M}_1$, respectively.



(a) Interactions between "create listener for each button" and its relevant code snippet



(b) Interactions between "print traffic violations for each car" and its relevant code snippet

Figure 3: Syntactic patterns lie in the interaction structures. Each matrix stands for the interaction structure of a text-code pair. Each matrix cell stands for the similarity between two words.

As the interaction structure of each text-code pair is represented as a matrix, syntactic patterns are converted to matrix patterns. For example, in Figure 3, we highlight matrix cells in red borders corresponding to the aforementioned "*x for each y*" syntactic pattern. We can see that $\mathbf{M}_0$ and $\mathbf{M}_1$ contain a similar matrix matching pattern. Therefore, we can learn such syntactic patterns from training data like p_0, and then transfer these patterns to the target codebase like p_1. Now, $f(q, c)$ is defined as:

$$f(q, c) = score(\Psi(q, c)) \tag{1}$$

, where Ψ maps the text-code pair to an interaction matrix unsupervisedly, and then we use a supervised model to learn the matching patterns to calculate $f(q, c)$ score. When applied to a new codebase, we construct matrices with all candidate code snippets and the new natural language query, and rank the code snippets by the predicted scores.

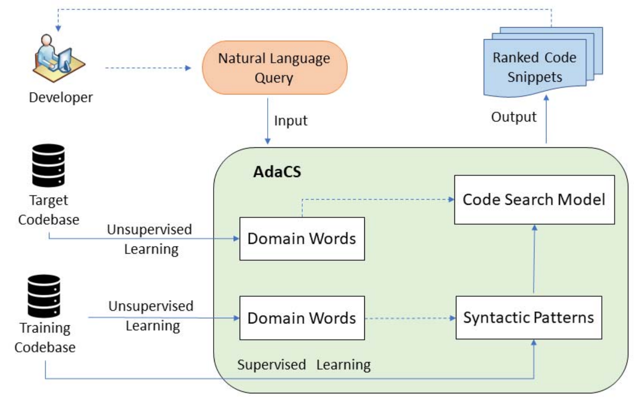


Figure 4: An overview of the code search process based on AdaCS

In our work, each word in code snippet c is not represented as a fixed vector, but represented dynamically by how it is similar to words in natural language query q. Since the word meanings can be learned in an unsupervised fashion, each word in the target codebase can be represented properly, even if it never appear in the training data. As a result, our approach can overcome the out-of-vocabulary problem and tranferred easily to new codebases.

3 APPROACH

In this section, we present our proposed approach namely AdaCS, which can be trained once and perform adaptive code search task on new codebases. To ensure the model transferability, AdaCS learns the abstract syntactic level patterns and the domain-specific lexical level meanings separately.

Figure 4 gives an overview of the adaptive code search workflow based on AdaCS. First, we use unsupervised learning based method to learn the domain-specific words meaning for both the training and target codebase. Next, we extract text-code pairs from the training codebase and use supervised learning based method to learn the general syntactic patterns. Finally, when the developer inputs a natural language query, we use the trained model to search on target codebase and then return the ranked code snippets as the final result.

Figure 5 gives the deep model architecture of AdaCS. It can be roughly divided into three parts: (1) learning domain words for each codebase to obtain the word embeddings; (2) constructing a *lexical* matching matrix for each text-code pair; (3) learning *syntactic* patterns from these matrices with deep neural networks. After that, AdaCS predicts the score $f(q, c)$ for each candidate code snippet c, then uses such scores to re-rank all candidate code snippets. The details of these three parts are introduced in the following sections.

3.1 Learning Domain Word Embeddings

For each codebase, we use unsupervised word embedding technique to represent its domain-specific semantic meaning. [2][37]. Specifically, we utilize *fastText*[3][1], a widely used word embedding tool

[1]https://fasttext.cc/

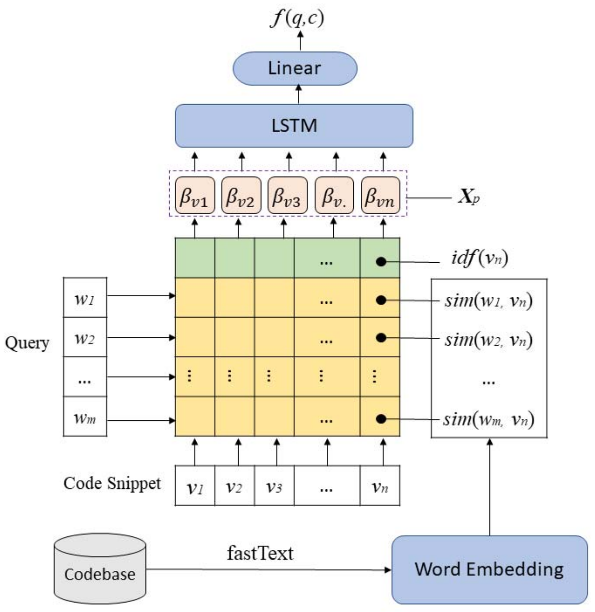


Figure 5: The model architecture of AdaCS

provided by *Facebook*, to encode each word w as a representation vector $\vec{\alpha}_w$.

Word embedding is an unsupervised learning-based method, which means that *fastText* requires a document corpus to train representation vectors for words. For each software project, we build the document corpus with code snippets (including API documentation and code comments of them) in its codebase, following Ye et al.'s work[60]. Moreover, if the project has more textual resources such as tutorials, issue reports and *StackOverflow* Q&A pairs, these textual resources can be added into the document corpus to learn better word representations.

We chose *fastText* to learn word representations, not classical word embedding algorithms like *word2vec*[36] or *GloVe*[43], since *fastText* can enrich word vectors with subword information. For example, in *fastText*, "*analyzer*" will share some parameters with "*analytics*" when training their word representations, since they are similar in morphology. Therefore, word representations trained by *fastText* are more reliable than those trained by classical word embedding algorithms.

3.2 Constructing Lexical Matching Matrices

As discussed in Section 2, AdaCS represents interaction structures of each text-code pair as a lexical matching matrix. Given a natural language text $q = (w_1, w_2, w_3, ..., w_m)$ and a code snippet $c = (v_1, v_2, v_3, ..., v_n)$, we construct a matrix $\mathbf{M}$ with m rows and n columns, in which each entry $\mathbf{M}_{i,j}$ represents the lexical similarity between word w_i and v_j:

$$\mathbf{M}_{i,j} = sim(w_i, v_j) \tag{2}$$

We use a simple tokenizer to split a natural language text or a code snippet into word sequences. It will decompose combined words (e.g., "*StandardAnalyzer*" will be decomposed to "*standard*" and "*analyzer*"), and it will filter out all punctuation such as commas, periods and braces. For example, as shown in Figure 3b, the code snippet: "*for (String s : carLiscenseList) { billDao.getTrfvltByLiscence(s); ... }*" will be tokenized into: ("*for*", "*string*", "*s*", "*car*", "*liscense*", "*list*", "*bill*", "*dao*", "*get*", "*trfvlt*", "*by*", "*liscence*", "*s*", ...)

The lexical similarity $sim(w_i, v_j)$ is scored based on the word embeddings obtained at previous stage. In particular, we define it as the cosine similarity between $\vec{\alpha}_{w_i}$ and $\vec{\alpha}_{v_j}$:

$$sim(w_i, v_j) = \frac{\vec{\alpha}_{w_i}^\top \vec{\alpha}_{v_j}}{\|\vec{\alpha}_{w_i}\| \cdot \|\vec{\alpha}_{v_j}\|} \tag{3}$$

, where $\|\cdot\|$ stands for the l_2 norm of a vector.

Based on the lexical matching matrix, we construct an interaction-focused representation vector $\vec{\beta}_{v_j}$ for each word v_j in code snippet c. This vector contains two parts of information:

(1) The j-th column in $\mathbf{M}$, expressing the **interaction structures** between word v_j and natural language query q. We assume that the maximum length of a natural language query will not exceed N (N is a hyper-parameter in AdaCS). Then, we make:

$$\vec{\beta}_{v_j,k} = \begin{cases} \mathbf{M}_{k,j} & 0 \le k < m \\ 0 & m \le k < N \end{cases} \tag{4}$$

(2) The **IDF** (Inverse Document Frequency) value of word v_j in the corpus, expressing the specificity of this word [51]. In the matching process, it is helpful to deal with words with different IDF values in different ways. Therefore, we add the IDF value of v_j as a dimension into the interaction-focused representation vector $\vec{\beta}_{v_j}$:

$$\vec{\beta}_{v_j,N} = idf(v_j) = \log \frac{|D|}{|\{d | d \in D \wedge v_j \in d\}|} \tag{5}$$

, where D stands for the aforementioned document corpus.

Through the above definitions, we can represent each text-code pair $p = (q, c)$ as $\mathbf{X}_p = (\vec{\beta}_{v_1}, \vec{\beta}_{v_2}, \vec{\beta}_{v_3}, ..., \vec{\beta}_{v_n})$, which is a vector sequence of length n. The i-th element in $\mathbf{X}_p$ is a $N+1$-dimensional vector, standing for the interaction-focused representation of the i-th word in code snippet c. In the next phase, $\mathbf{X}_p$ will be used as an input to learn cross-project syntactic patterns.

3.3 Learning Syntactic Patterns

Given a text-code pair $p = (q, c)$, we construct a vector sequence $\mathbf{X}_p$ for it (as presented in Section 3.2), then we develop an RNN model to predict the $f(q, c)$ value. This model should be trained with lots of text-code pairs, and cross-project syntactic patterns will be learned and encoded implicitly in model parameters. More specifically, our method works as follows:

3.3.1 *Text-code pair encoding.*

We first pass the vector sequence $\mathbf{X}_p = (\vec{\beta}_{v_1}, \vec{\beta}_{v_2}, \vec{\beta}_{v_3}, ..., \vec{\beta}_{v_n})$ as the input to a long short-term memory network (LSTM) [20] and thus obtain:

$$\{\vec{h}_1, \vec{h}_2, ..., \vec{h}_n\} = \text{LSTM}(\mathbf{X}_p) \tag{6}$$

, where $\vec{h}_i$ is expected to encode useful context information in subsequence $(\vec{\beta}_{v_1}, \vec{\beta}_{v_2}, \vec{\beta}_{v_3}, ..., \vec{\beta}_{v_i})$. Therefore, information in $\mathbf{X}_p$ is encoded in a fixed-length vector—$\vec{h}_n$.

3.3.2 *Prediction.*

Our goal is to predict the $f(q, c)$ value, which stands for the matching degree between natural language text q and code snippet c. To this end, we take $\vec{h}_n$ as input, and use a linear module $\vec{z}$ to convert it to $f(q, c)$:

$$f(q, c) = \vec{z}^\top \vec{h}_n \tag{7}$$

3.3.3 *Training.*

We train all model parameters including the LSTM module and $\vec{z}$ jointly. Our training data S consists of triples in the form of (q, c^+, c^-). In each triple, q is a natural language text; c^+ is a positive code snippet that relevant to q; c^- is a negative code snippet that is not relevant to q. The negative samples are randomly drawn from the pool of code snippets except c^+. For each training example $s = (q, c^+, c^-)$ in S, the training objective is to minimize the pair-wise hinge loss:

$$L_s = \max(0, 1 - f(q, c^+) + f(q, c^-)) \tag{8}$$

This loss encourages the model to produce a higher score of $f(q, c^+)$ than the score of $f(q, c^-)$. Based on it, we define the aggregated loss function L_S on dataset S as:

$$L_S = \sum_{s \in S} L_s \tag{9}$$

3.4 Transferring to Target Codebases

AdaCS is a domain-adaptive code search methods that can learn from a single code corpus and be applied to different codebases directly. When it comes to a new codebase, we do not need to re-train the model since the syntactic patterns can work across codebases but only substitute the domain-specific word embeddings part.

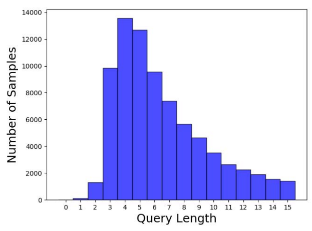
As described in Section 3.1, we learn the word embedding for the target code base in an unsupervised fashion. Then, we construct the lexical matching matrices for the natural language query and all candidate code snippets in the target codebase. In particular, given a natural language query q, and all candidate code snippets $C = \{c_1, c_2, \ldots, c_k\}$, we compute the matching score by $f(q, c\prime)$ for all $c\prime \in C$. Finally, we rank these code snippets in descending order, and return the ranked list as the final search result.

4 EVALUATION SETUP

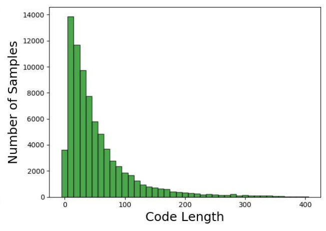
We evaluate AdaCS on the adaptive code search task: AdaCS is trained with text-code pairs collected from GitHub, then it is tested on a new codebase composed by three individual software projects—*Apache Lucene*[2], *Apache POI*[3] and *JFreeChart*[4]. In the training data, all text-code pairs about these three target projects are filtered out to ensure the adaptive nature of this evaluation.

We compare AdaCS with several state-of-the-art code search methods to validate its effectiveness. Details of the evaluation setup are presented as follows.

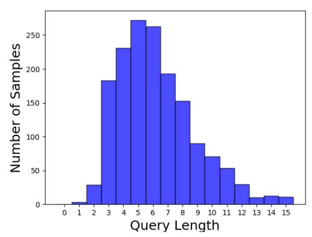
[2]http://lucene.apache.org/
[3]https://poi.apache.org/
[4]http://www.jfree.org/jfreechart/



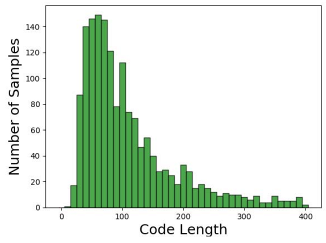
(a) Text length distribution of GitHub dataset



(b) Code length distribution of GitHub dataset



(c) Text length distribution of the test dataset



(d) Code length distribution of the test dataset

Figure 6: Text and code length distributions of our datasets

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1:** Does AdaCS outperform the state-of-the-art deep code search methods in adaptive code search?
- **RQ2:** Does AdaCS outperform the IR-based code search methods in adaptive code search?
- **RQ3:** How does each module in AdaCS (e.g., *fastText* word embeddings and the IDF value $idf(\cdot)$) affect its effectiveness?
- **RQ4:** How efficient is AdaCS? How long does it take to train the model, and how long does it take to make predictions?

4.2 Dataset

Our training data are collected from GitHub. Following Hu et al.'s work[22], we extract Java methods and their Javadoc comments from GitHub (methods without Javadoc comments are omitted in this evaluation). For each Java method c, we use the first sentence appeared in its Javadoc comment as the natural language query q since it typically describes the functionality of the Java method (according to Javadoc guidance[5]).

We further clean up these (q, c) pairs using the following rules:

- (q, c) will be filtered out if the code snippet c invokes any API in Apache Lucene, Apache POI or Jfreechart, which are target projects for testing in our evaluation.
- (q, c) will be filtered out if length of the query q is longer than N (we set $N = 15$), the code snippet c contains less than 3 statements, or the code snippet c contains more than 400 words.
- (q, c) will be filtered out if the code snippet c is a constructor, an overridden method, a getter/setter method or a test method.

[5]http://www.oracle.com/technetwork/articles/java/index- 137868.html

Table 1: Basic statistics of the test data

	Methods	NL Words	Code Words	Unique NL Words	Unique Code Words
Lucene	563	3,488	59,874	889	1,772
POI	707	4,549	82,259	878	2,083
JFreeChart	336	1,961	41,522	430	926
Total	1,606	9,998	1,83,655	1,457	3,075

Table 2: Some sampled queries from test dataset.

Project	Natural Language Queries
Lucene	creates a span query from the tokenstream
	load a stemmer table from an inputstream
POI	create a new comment at located cell address
	change the type of the text
JFreeChart	draws a section label on the right of the pie chart
	create a scatter plot with default settings

Finally, we get 77,920 text-code pairs which are positive samples $(q, c+)$. For each $(q, c+)$, we randomly select n negative samples from the pool of code snippets except $c+$, where n is set to be 20 empirically in this work. Thus, we obtain $77,920 \times 20$ triples in the form of $(q, c+, c-)$. This dataset contains 16,551 unique words in natural language queries and 25,459 unique words in code snippets. Figure 6a and Figure 6b gives the text length and code length distribution respectively. We randomly select 60% of the triples for training (S), 20% of the triples for validation (S'), and the rest 20% for testing in a secondary experiment (S'').

Our test dataset consists of 3 parts of code according to related projects, which are: Apache Lucene (version 7.0.0), Apache POI (version 4.1.0) and JFreeChart (version 1.0.19). To guarantee the quality of natural language queries, we use a few heuristic rules(e.g., should contain verb and noun phrases)to filter the inappropriate descriptions(e.g., "*This method is not thread-safe.*"). Finally, we get 1,606 (q, c) text-code pairs in total. For each natural language query q, we treat all these 1,606 code snippets as candidates and our goal is to rank the ground truth c as higher as possible.

Table 1 gives the basic statistics of the test dataset, and Table 2 shows some sampled queries from the test dataset. Figure 6c gives the text length distribution of this dataset, and Figure 6d gives the code length distribution of this dataset.

4.3 Comparison Methods

We compare the effectiveness of AdaCS with 5 existing code search methods, and these methods can be categorized into two groups.

4.3.1 IR-based Methods. For IR-based code search methods, the ordering of words usually has no effect on the search process. they can work across domains and do not need labelled training data. Since our dataset only contains text-code pairs, we do not compare with some IR-based methods requiring auxiliary information(e.g., posts from StackOverflow).

- **Baseline**: we take a simple information retrieval system as our baseline method, in which the matching degree between a natural language query and a code snippet is measured by the cosine similarity of their TF-IDF vectors. We use *Gensim*[6], a Python library for text processing to implement this baseline method.
- **Skip-gram**: Ye et al. [60] proposed a word embedding-based method to measure the matching degree between a natural language text and a code snippet. We apply it in the code search task and call this *Skip-gram*, as this method utilizes the classical skip-gram model [36] to learn word embeddings. As the authors didn't open source it, we reproduced this method based on the paper.
- **CodeHow**: CodeHow is a code search method proposed by Lv et al. [34]. It first identifies some potential APIs that may be related to a natural language query, then use the Extended Boolean Model to apply the information to the code search process. As the authors didn't open source it, we reproduced this method based on the paper.

4.3.2 Deep Learning-based Methods. For deep learning-based code search methods, they perform well if models are trained with a large amount of proper text-code pairs, but they will suffer from out-of-vocabulary(OOV) problem when applied to new codebases.

- **DeepCS**: DeepCS[7] is a deep learning-based code search method proposed by Gu et al. [14]. It is a typical representation focused deep code search method (while AdaCS is a interaction focused deep code search method): natural language queries and code snippets are encoded as real vectors in a unified space using deep neural networks, then the matching degree between a query and a code snippet is measured by the distance of the vectors.
- **BVAE**: BVAE [8] proposed by Chen et al. [8] is another representation based deep code search method. It utilizes two Variational AutoEncoders (VAEs) to better encode natural language queries and code snippets.

Moreover, to answer RQ3 (how each module in AdaCS affects its effectiveness), we compare AdaCS with some of its variants as follows:

- **AdaCS-IND**: in this variant, we use indicator function to calculate $sim(w_i, v_j)$, i.e., we produce either 1 or 0 to indicate whether two words are identical after they are stemmed:

$$sim(w_i, v_j) = \begin{cases} 1 & stem(w_i) = stem(v_j) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

  This variant is used to validate that it is important to capture the semantic matching between similar but not identical words using word embeddings from unsupervised learning in AdaCS.
- **AdaCS-noIDF**: In this variant, we remove the IDF value $idf(v_j)$ from from $\vec{\beta}_{v_j}$. This variant is used to validate that it is important to incorporate the specificity of words into AdaCS.

[6]https://pypi.org/project/gensim/
[7]https://github.com/guxd/deep-code-search
[8]https://github.com/betterenvi/ret-sum

Table 3: Comparison of Hit@K ($K = 1, 2, 3, 5$) and MRR scores on the adaptive code search task

	H@1	H@2	H@3	H@5	H@10	MRR
Baseline	0.279	0.405	0.467	0.541	0.661	0.407
Skip-gram	0.322	0.415	0.458	0.552	0.696	0.435
CodeHow	0.377	0.472	0.509	0.567	0.698	0.479
DeepCS	0.268	0.366	0.430	0.523	0.659	0.386
BVAE	0.268	0.366	0.430	0.523	0.709	0.397
AdaCS	**0.486**	**0.598**	**0.675**	**0.772**	**0.885**	**0.621**

4.4 Performance Metrics

Our evaluation task is a navigational search task. In other words, each test query corresponds to one and only one ground-truth relevant code snippet. Therefore, we use two common metrics for navigational search as the performance metrics in our evaluation:

- **Hit@K**, which considers whether the ground truth relevant code snippet of each query is ranked within the top K positions of the result list:

$$Hit@K = \frac{|\{q|q \in Q \land Rank(c_q|q) \leq K\}|}{|Q|} \tag{11}$$

  , where Q is the set of all test queries, and $Rank(c_q|q)$ stands for where the ground truth relevant code snippet c_q of q is ranked in the search result. We report the Hit@K results at $K = \{1, 2, 3, ..., 10\}$.
- **MRR** (Mean Reciprocal Rank), i.e., the average of the reciprocal ranks of the ground-truth relevant code snippets:

$$MRR = \frac{\sum_{q \in Q} \frac{1}{Rank(c_q|q)}}{|Q|} \tag{12}$$

4.5 Implementation Details

We implement AdaCS using PyTorch[9], an open-source deep learning framework, and the URL of our source code is masked now for double-blind review.

We make the LSTM module (presented in Equation 6) a 2-layer LSTM. The parameters are set as a result of experience. The hidden dimension of LSTM is set to 64, and the dropout rate is set to 5%. For training our AdaCS model, we use the ADAM[26] optimizer with learning rate 0.005 and batch size 64. Code, data, and experiments for this paper are available on GitHub. [10]

5 EVALUATION RESULTS

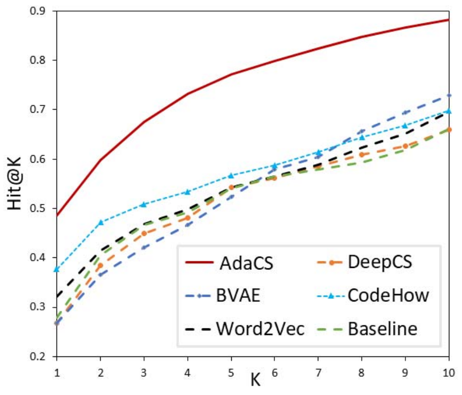
In this section, we show the evaluation results and answer the research questions presented in Section 4.1 empirically.

5.1 Effectiveness (RQ1 & RQ2)

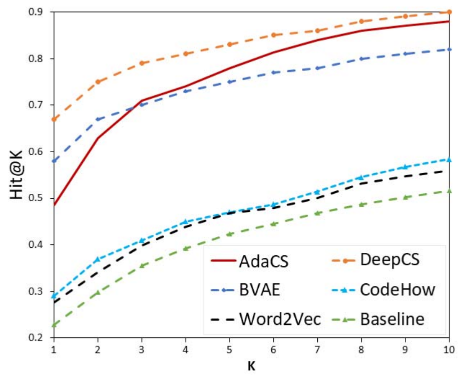
RQ1 and RQ2 aim to investigate whether AdaCS outperforms the state-of-the-art deep code search methods and IR-based code search methods in adaptive code search. We report our main results in Table 3.

AdaCS outperforms existing code search methods by a large margin. It obtains a test MRR score of 0.621, which is much better than

[9]https://pytorch.org/
[10]https://github.com/laurence-ling/drm_codesearch



(a) Comparison results on the adaptive code search task



(b) Comparison results on the non-adaptive code search task

Figure 7: Comparison of Hit@K ($K = 1, 2, ..., 10$) scores: (1) AdaCS outperforms the state-of-the-art methods in adaptive code search task, and achieves competitive results in non-adaptive search task; (2) Deep code search methods (DeepCS and BVAE) work well on the non-adaptive code search task, but their effectiveness drops a lot on the adaptive search task, since they suffer from the OOV problem caused by domain-specific words.

that of Baseline (0.407), Skip-gram (0.435), CodeHow(0.479), DeepCS (0.386) and BVAE (0.39). In terms of Hit@K, AdaCS improves the state-of-the-art Hit@5 score from 56.7% (obtained by CodeHow) to 77.2%. For 48.6%/67.5%/77.2%/88.5% of the test queries, the relevant code snippets can be found within the top 1/3/5/10 returned results. Figure 7a gives the Hit@K curves of AdaCS and those existing code search results. These results show that AdaCS leads to much better results than existing deep learning-based methods and IR-based methods on the adaptive code search task.

As Table 3 and Figure 7a shows, the state-of-the-art deep code search methods (i.e., DeepCS and BVAE) do not perform well on the adaptive code search task. As a comparison, we test AdaCS and those comparison methods on a non-adaptive code search task(i.e., the model is both trained and tested on the same dataset), and the results are shown in Figure 7b. In this secondary experiment, we use the remaining 20% of GitHub data for testing except the

Table 4: Comparison of Hit@K ($K = 1, 2, 3, 5$) and MRR scores of AdaCS and its variants

	H@1	H@2	H@3	H@5	H@10	MRR
AdaCS	**0.486**	**0.598**	**0.675**	**0.772**	**0.885**	**0.621**
AdaCS _IND	0.402	0.517	0.602	0.696	0.809	0.583
AdaCS _noIDF	0.435	0.549	0.630	0.732	0.848	0.546

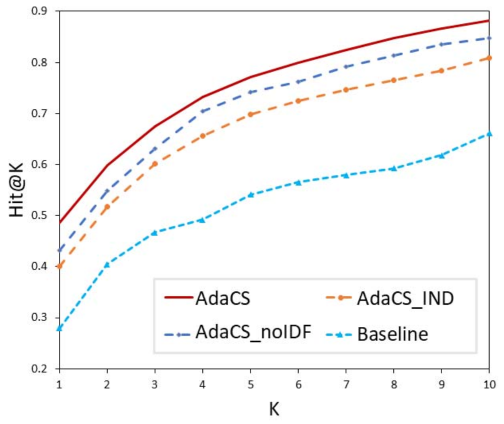


Figure 8: Comparison of Hit@K ($K = 1, 2, ..., 10$) scores of AdaCS and its variants

60% for training data and the 20% for validation data. We can find that AdaCS also achieves competitive results in this task. Note that the state-of-the-art deep code search methods learn each word's embedding as model parameters in a supervised fashion, while AdaCS only learns them unsupervisedly.

As is shown in Figure 7b, although the state-of-the-art deep code search methods work well on the non-adaptive code search task, their effectiveness drops dramatically in the adaptive code search scene(as shown in Figure 7a). This is mainly because that they suffer from the OOV problem caused by domain-specific words in target codebase (i.e., Apache Lucene, Apache POI and JFreeChart).

In summary, AdaCS, a novel interaction-based adaptive deep code search method, **outperforms the state-of-the art methods in adaptive code search task and also achieves competitive results in non-adaptive task.**

5.2 Module Utility (RQ3)

RQ3 aims to investigate how each module in AdaCS affects its effectiveness. To answer this research question, we compare AdaCS against some of its variants:

- AdaCS-IND, in which we use indicator function to calculate word similarity;
- AdaCS-noIDF, in which the IDF values are removed from $\vec{\beta}_{v_j}$;

The experimental results are shown in Table 4 , and Figure 8 gives the detailed Hit@K ($K = 1, 2, ..., 10$) curve of the results. We can find that AdaCS obtain the best results, whether in terms of Hit@K or MRR metrics. If the IDF values are removed out (i.e., AdaCS-noIDF), then the MRR drops from 0.621 to 0.583, which demostrates that the IDF values in $\vec{\beta_{v_j}}$ carry useful information for understanding the semantics. At same time, the MRR also drops if we do not use unsupervised learned word embeddings to calculate the similarity but use exact matches.

Therefore, we can conclude that: **all the modules including fastText-based word similarities and the IDF values are helpful to improve the effectiveness of AdaCS.**

5.3 Efficiency (RQ4)

RQ4 aims to investigate the time efficiency of AdaCS. More specifically, we focus on how long it takes to train the model and how long it takes to predict the $f(q, c)$ value of each (q, c) pair.

To answer this research question, we record the time spent on training and predicting during evaluation. Our experiments are all conducted on a workstation with a 2.9GHz 8 Core CPU and a NVIDIA GTX1080TI GPU.

5.3.1 ***Training efficiency.*** In our training process, each training iteration took about 30 minutes in average. The validation result reached its local optimal at the 9th iteration, and tended to stable afterwards.

5.3.2 ***Prediction efficiency.*** For each (q, c) pair, AdaCS took 1.29 milliseconds to predict the $f(q, c)$ value. Therefore, the response time of each query will be 1.29 seconds if we take 1,000 candidate code snippets in the code search engine. In order to reduce the number of candidates, we use a text retrieval system to filter out some code snippets before the model prediction. In the text retrieval step, we simply use the vector space model (VSM) [47] based cosine similarity between the code snippets and the natural language query q, and the top-k scored code snippets are kept as "candidate code snippets". Thus, we can adjust the value of k (e.g., set it to 1,000) so that the online computation time of AdaCS can be reduced. Therefore,**this time efficiency can basically meet the needs of real-time search**, though it is slower than existing code search methods.

5.4 Threats to Validity

The threat to *construct validity* is that the target projects (i.e., Apache Lucene, Apache POI and JFreeChart) in this evaluation are not actual standalone industrial software projects, but widely-used open source software projects. To reduce the threat, we isolate the target projects from the training data, and simulate the adaptive code search scene. As described in section 4, for any (q, c) pair from GitHub, it will not be added to the training data if the code snippet c invokes any API in any of the three target projects. Once data from a completely different domain is available, we will further validate our method.

The threat to *internal validity* is that our evaluation uses Javadoc descriptions as code search queries for testing. It is likely that such test queries are different from queries introduced by developers in real code search scenarios. To reduce the threat, we cleaned up these test queries carefully. We use the heuristics rules to filtered out a lot of inappropriate queries from the test data to ensure that our test queries are as close as possible to the real-world code search queries. As is shown in Table 2, the final test queries are good for evaluating the effectiveness of code search methods. We will evaluate our model in other datasets that contain actual natural language search queries in the future.

The threat to *external validity* is that AdaCS is currently tested on Java programs. However, AdaCS makes few assumptions on the underlying language and only requires the training text-code pairs to learn the syntactic patterns. Generalizing AdaCS to other languages will be our future work.

6 RELATED WORK

In this section, we present and discuss related work of this paper mainly from three aspects: deep learning-based code search, IR-based code search and transfer learning.

6.1 Deep Learning-based Code Search

Code search has been widely studied in literature[5, 25, 49]. Recently, deep learning-based methods are also applied to code search [14, 15]. Typically, DeepCS [14] uses CODEnn that learns a unified vector representation of both source code and natural language queries so that code snippets semantically related to a query can be retrieved according to their vectors. Chen et al. [8] proposed BVAE that is composed by two Variational AutoEncoders (VAEs) to model source code and natural language respectively. Both VAEs are trained jointly to capture the closeness between the latent variables of the code and the description. Jiang et al. [24] proposed ROSF that uses supervised learning to re-rank the candidate results. Richardson et al. [46] proposed Assistant that learnes a translation model from amount of code-text pairs. Word embedding technique is also used in some research work [10, 18]. Ye et al. [60] proposed a word emmbedding-based method to measure the matching degree between natural language text and source code. API2Vec [39] used the CBOW model to learn API embeddings from API sequences extracted from source code, and verified that the vectors can represent similar semantics of APIs. BIKER [23] extracted similar questions and APIs from StackOverflow and used word embeddings to calculate the text similarity.

Different from existing deep code search methods, AdaCS learns domain-specific word embeddings and general syntactic patterns respectively. Leveraging the interaction matrix to learn the syntactic patterns has been used in text matching methods, which is called *interaction-focused* deep text matching[16].

The *interaction-focused* methods first build local interactions between the query and the document, and then use neural networks to learn hierarchical matching patterns. For example, ARC-II[21] and MatchPyramid[42] build hierarchical CNNs on the similarity matrix of two texts' word embeddings. They can successfully identify salient signals such as n-gram and n-term matchings. DRMM[16] uses histogram mapping, a feed forward matching network, and a term gating network to summarize relevance matching factors. K-NRM[57] uses a new kernel-pooling technique to extract multi-level soft-match features from the translation matrix, and a learning-to-rank layer that combines those features into the final ranking score. Conv-KNRM[9] adds convolution layers over K-NRM and outperforms prior neural methods and feature-based methods.

Different form deep text matching methods which focuses on two texts, AdaCS measure the similarity between texts and code snippets, and utilizes the unsupervised word embedding to represent the meanings of so many domain-specific words in code snippets.

6.2 IR-based Code Search

Many IR-based code search methods were proposed [38], which are mainly based on the text similarity between natural language queries and code snippets. Here are some typical IR-based code search work: Bajracharya et al. [1] proposed Sourcerer, an internet-scale code search engine that uses Lucene to build index on the parsed source code and provides keyword search service for developers. Chatterjee et al. [7] proposed SNIFF, a code search tool that annotates each API with its related documentation first, and then perform the natural language based code search on the annotated code snippets.

To deal with the lexical-gap between natural language and source code, different approaches were proposed such as query reformulation [17] and query expansion [19, 28, 53, 58]. Lu et al. [33] proposed to extend the query with synonyms from WordNet. Lv et al. [34] proposed CodeHow, a code search engine that first identifies some potential APIs that may be related to the question, and then use the Extended Boolean Model to incorporate the APIs' information during the code search process. Wang et al. [56] proposed to use feedback information from users to reformulate the query. Recently, Sivaraman et al. [50] proposed ALICE, a interactive code search tool that uses active inductive logic programming to interact with users and infers a new logic query that separates positive examples from negative examples.

Meanwhile, some relationship aware methods were proposed to leverage the structural information of source code. For example, Mcmillan et al. [35] proposed Portfolio that organizes source code as directed graph and uses PageRank and Spread Activation Network(SAN) to return the top related nodes in the graph as answers. Chan et al. [6] returns a connected subgraph so that it can present the relationships between these API nodes clearly. Li et al. [29] proposed RACS also proposed a relationship aware code search approach for JavaScript frameworks.

IR-based methods are unsupervised but the accuracy is not satisifying, while DL-based methods improve the accuracy but cannot be adopted to different codebases. Our approach aims to perform adaptive code search and keep high accuracy at the same time.

6.3 Transfer Learning

Transfer learning aimed at transferring knowledge from a source domain to a target domain has been extensively studied in machine learning (e.g., [41] for an overview). With the surge of deep learning, various neural network based transfer learning methods[52] have been proposed for different fields including computer vision[31], speech recognition[55] and NLP[12, 59]. Based on the data requirements, existing transfer learning methods can be generally categorized into two groups. The first group of work assumes that we have labeled data from the source domain and also a little labeled data from the target domain. A representative and widely used framework is the *fine-tuning* approaches, which initialize the model parameters from a well-trained model on the source domain and then fine tune the parameters using the labeled data in the target domain. The second group of work assumes that we only have labeled data from the source domain but may also have some unlabeled data from the target domain. Since we do not have labeled text-code pairs for the new codebase, our work falls to the latter case, which is also called unsupervised domain adaptation.

For unsupervised domain adaption, *instance-based* transfer learning methods re-weight instances from the source domain against the distribution of the target domain[13]. However, it is hard to compute the importance weight without labeled data in the target domain and may cause negative results[44]. Another group of work belong to *feature-based* transfer learning, which map instances of the source and target domains to a shared feature space and reduce the cross-domain discrepancy in the new space[11]. The basic idea is to learn domain-invariant representations, rather than domain-specific ones. Different deep neural networks are used in these work to learn intermediate feature representations, such as residual transfer networks[32] and generative adversarial networks[4]. A typical framework is to use a shared neural network to learn the invariant feature space[59], while another representative framework is to use a shared network to learn the shared feature space and two domain-specific network for the specific feature space[30].

Our work belongs to the category of feature-based transfer learning, but there is no out-of-shelf methods that can be used in the code search scenario directly. To our knowledge, AdaCS is the first work to apply unsupervised domain adaption to code search. AdaCS firstly use unsupervised word embedding technique to construct a matching matrix between queries and code snippets, and then use a RNN to learn the cross-domain syntactic patterns. Thus, AdaCS can learn from data in the source domain and perform well in the target domain.

7 CONCLUSION

In this paper, we propose AdaCS, an adaptive deep code search method. It can be trained in one codebase and applied to another one easily, thus solving the problem that existing deep code search methods require a lot of time and effort on collecting domain-specific training data. To ensure the transferability, AdaCS learns word meanings and syntactic patterns respectively in the learning process. Firstly, an unsupervised word embedding technique is utilized to construct a matching matrix for each text-code pair. Then, a recurrent neural network is utilized to capture latent syntactic patterns from these matching matrices in a supervised way. Experimental results show that AdaCS outperforms the state-of-the-art methods when adapting to a new codebase. For future work, we will further improve the time efficiency of AdaCS by using parallel deep sequence encoders (e.g., Transformer [54]) instead of RNN to learn syntactic patterns. Moreover, we will further apply AdaCS to more programming languages and wider test datasets.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Fund for Distinguished Young Scholars of China under Grant No. 61525201, and the General Program of National Natural Science Foundation of China under Grant No. 61972006.

REFERENCES

[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 681–682.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[4] Konstantinos Bousmalis, Nathan Silberman, David Dohan, Dumitru Erhan, and Dilip Krishnan. 2017. Unsupervised pixel-level domain adaptation with generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3722–3731.

[5] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

[6] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 10.

[7] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 385–400.

[8] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 826–831.

[9] Zhuyun Dai, Chenyan Xiong, Jamie Callan, and Zhiyuan Liu. 2018. Convolutional neural networks for soft-matching n-grams in ad-hoc search. In *Proceedings of the eleventh ACM international conference on web search and data mining*. ACM, 126–134.

[10] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to specification mining. *arXiv preprint arXiv:1802.07779* (2018).

[11] Yaroslav Ganin and Victor Lempitsky. 2014. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495* (2014).

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 513–520.

[13] Boqing Gong, Kristen Grauman, and Fei Sha. 2013. Connecting the dots with landmarks: Discriminatively learning domain-invariant features for unsupervised domain adaptation. In *International Conference on Machine Learning*. 222–230.

[14] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[16] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W.Bruce Croft. 2016. A Deep Relevance Matching Model for Ad-hoc Retrieval. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 55–64.

[17] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 842–851.

[18] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 163–174.

[19] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 524–527.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. 2014. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*. 2042–2050.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[23] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 293–304.

[24] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. 2016. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing* (2016).

[25] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[27] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[28] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 212–221.

[29] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 690–701.

[30] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2017. Adversarial multi-task learning for text classification. *arXiv preprint arXiv:1704.05742* (2017).

[31] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I Jordan. 2015. Learning transferable features with deep adaptation networks. *arXiv preprint arXiv:1502.02791* (2015).

[32] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I Jordan. 2016. Unsupervised domain adaptation with residual transfer networks. In *Advances in Neural Information Processing Systems*. 136–144.

[33] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[34] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[35] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

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

[38] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 880–890.

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

[40] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing* 9, 5 (2016), 771–783.

[41] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[42] Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Shengxian Wan, and Xueqi Cheng. 2016. Text matching as image recognition. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI, 2793–2799.

[43] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[44] Barbara Plank, Anders Johannsen, and Anders Søgaard. 2014. Importance weighting and unsupervised domain adaptation of POS taggers: a negative result. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 968–973.

[45] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082* (2014).

[46] Kyle Richardson and Jonas Kuhn. 2017. Function Assistant: A Tool for NL Querying of APIs. *arXiv preprint arXiv:1706.00468* (2017).

[47] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[48] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. IBM Corp., 174–188.

[49] Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries

to support efficient free-form code search. *Empirical Software Engineering* 23, 5 (2018), 2622–2654.

[50] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2018. Active Inductive Logic Programming for Code Search. *arXiv preprint arXiv:1812.05265* (2018).

[51] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[52] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. In *International Conference on Artificial Neural Networks*. Springer, 270–279.

[53] Yuan Tian, David Lo, and Julia Lawall. 2014. Automated construction of a software-specific word similarity database. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 44–53.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[55] Dong Wang and Thomas Fang Zheng. 2015. Transfer learning for speech and language processing. In *2015 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*. IEEE, 1225–1237.

[56] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 677–682.

[57] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017. End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 55–64.

[58] Jinqiu Yang and Lin Tan. 2014. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering* 19, 6 (2014), 1856–1886.

[59] Zhilin Yang, Ruslan Salakhutdinov, and William W Cohen. 2017. Transfer learning for sequence tagging with hierarchical recurrent networks. *arXiv preprint arXiv:1703.06345* (2017).

[60] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*. ACM, 404–415.