APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model

Yuning Kang[1], **Zan Wang**[1], **Hongyu Zhang**[2], **Junjie Chen**[1]*, **Hanmo You**[1]
[1]College of Intelligence of and Computing, Tianjin University, Tianjin, China
[2]The University of Newcastle, Callaghan, Australia
{kangyuning,wangzan,junjiechen,youhanmo}@tju.edu.cn
zhang.hongyu@newcastle.edu.au

Abstract

For programmers, learning the usage of APIs (Application Programming Interfaces) of a software library is important yet difficult. API recommendation tools can help developers use APIs by recommending which APIs to be used next given the APIs that have been written. Traditionally, language models such as N-gram are applied to API recommendation. However, because the software libraries keep changing and new libraries keep emerging, new APIs are common. These new APIs can be seen as OOV (out of vocabulary) words and cannot be handled well by existing API recommendation approaches due to the lack of training data. In this paper, we propose APIRecX, the first cross-library API recommendation approach, which uses BPE to split each API call in each API sequence and pre-trains a GPT-based language model. It then recommends APIs by fine-tuning the pre-trained model. APIRecX can migrate the knowledge of existing libraries to a new library, and can recommend APIs that are previously regarded as OOV. We evaluate APIRecX on six libraries and the results confirm its effectiveness by comparing with two typical API recommendation approaches.

1 Introduction

Application Programming Interface (API) is an integral part of software libraries. Being familiar with APIs could help improve programming productivity. However, a library tends to contain a large number of APIs and there could be complex dependencies among APIs, and thus understanding all APIs in a library is very challenging, especially for new developers. To facilitate correct and efficient usage of APIs during programming, many API recommendation approaches (Zhong et al., 2009; Nguyen et al., 2016; Xie et al., 2019; Bruch et al., 2009; Huang et al., 2018) have been proposed. More specifically, API recommendation aims to automatically recommend a correct API call at the current programming location based on its preceding part of code information.

As an example, Listing 1 shows a Java code snippet about opening a text file. Assuming a programmer forgets what to write in Line 6. API recommendation tool can help the programmer by prompting the most likely API call to be used next. In this case, $printStackTrace()$ will be returned. The API recommendation tools do so by learning API usage pattern from a large code corpus. Some tools (Nguyen et al., 2016; Nguyen and Nguyen, 2015) use probabilistic models to learn API usage pattern , while others (Zhong et al., 2009; Wang et al., 2013) use data mining methods to find API usage patterns . Recently, deep learning based language models are proposed to model the API sequences and have obtained promising results in recommending APIs (Raychev et al., 2014; Yan et al., 2018; White et al., 2015; Nguyen and Nguyen, 2015).

However, the existing API recommendation tools only focus on improving the performance of API recommendation when API usage data are sufficient (i.e., the usage data of the APIs to be recommended are sufficient in training data). That is, they mostly ignored the OOV (out of vocabulary) problem, which could have negative impact on the performance of API recommendation. More specifically, when some APIs are unseen in training data, these approaches cannot recommend them correctly. The OOV problem could be more serious for a new library, since it is very difficult to collect sufficient API usage data.

To conduct API recommendation for new libraries, cross-library API recommendation is a potentially feasible solution, which aims to recommend APIs in new libraries based on the usage data of APIs in other libraries, but it is still an open challenge due to the inherent OOV problem. For example, as shown in Listing 2, we may rarely (or even never) see *SQLException.printStackTrace()*

*Junjie Chen is the corresponding author.

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 3425–3436
November 7–11, 2021.

in the training set, but the usage of Exception is very common in the training set and the usage of *SQLException* and *Exception* are similar. So if we use a word segmentation algorithm to split *SQLException.printStackTrace()* into the sequence: *SQLException-.-print-StackTrace()*, we can use the *Exception* usage pattern learned during the training process to predict the *printStackTrace()* method and finally synthesize *SQLException.printStackTrace()* as the recommendation result.

```
public static void main(...) {
FileInputStream inputStream = null;
  try {
    File file = new File("tmp.txt")}
  catch (Exception e) {
    e.____); //To write a catch block
  }
}
---------------------------------
API Sequence: FileInputStream.new()-
    TRY-TryBlock-File.new(String)-
    CATCH-Exception._____()
```

Listing 1: An Example of API Recommendation

```
public static Connection
getConnection(){
Connection connection = null;
try {
connection = Drivermanager.get(URL,
username,password);
    } catch (SQLException e) {

    e.printStackTrace();
}
return connection;
}
```

Listing 2: An OOV Example in API Recommendation

To achieve the goal of cross-library API recommendation, we draw lessons from the area of text generation in relieving the OOV problem (Sennrich et al., 2016; Hermann et al., 2021). More specifically, we design a framework of cross-library API recommendation, called APIRecX, which consists of three main components, i.e., API segmentation, subword language model building, and API synthesis for recommendation. Since the OOV problem at the API level hampers cross-library API recommendation, APIRecX first incorporates BPE (Byte Pair Encoding) (Provilkov et al., 2020; Sennrich et al., 2016), one of the most widely-used word segmentation methods in text generation, to split each API call into a sequence of subwords. That is, the OOV problem at the API level could be largely relieved at the subword level. Based on a large number of subword data, APIRecX then adopts the "pre-training&fine-tuning" mechanism to build a GPT-based(Generative Pre-Training) pre-trained language model, which can recommend a subword in each prediction. Since the recommendation process is conducted at the subword level, it is necessary to compose a complete API call for recommendation based on predicted subwords. Here, APIRecX incorporates beam search for API synthesis.

To evaluate the performance of APIRecX, we conducted an extensive study based on 1,711 Java projects from GitHub involving six libraries in three domains as subjects for mimicking new libraries in the scenario of cross-library API recommendation, and over 14,000 GitHub Java projects that do not involve the former six libraries as training corpus. By comparing with two typical API recommendation approaches, i.e., LSTM-based language model(Yan et al., 2018; White et al., 2015) and N-gram-based language model (Raychev et al., 2014; Karampatsis and Sutton, 2019; Hindle et al., 2012), our experimental results demonstrate the effectiveness of APIRecX for cross-library API recommendation in terms of recommendation accuracy.

To sum up, this work makes the following major contributions:

- We propose the first framework for cross-library API recommendation, consisting of BPE-based API segmentation, subword language model building, and beam-search based API synthesis.
- We are the first to build a GPT-based language model in the area of API recommendation, which is more effective than the existing language models.
- We conduct an extensive study to evaluate our proposed approach, demonstrating its effectiveness in the scenario of cross-library API recommendation.

2 Approach

In the paper, we propose APIRecX, the first approach for cross-library API recommendation. With APIRecX, we can recommend APIs in some libraries (especially new libraries) by learning from a large amount of API usage data of some other libraries.

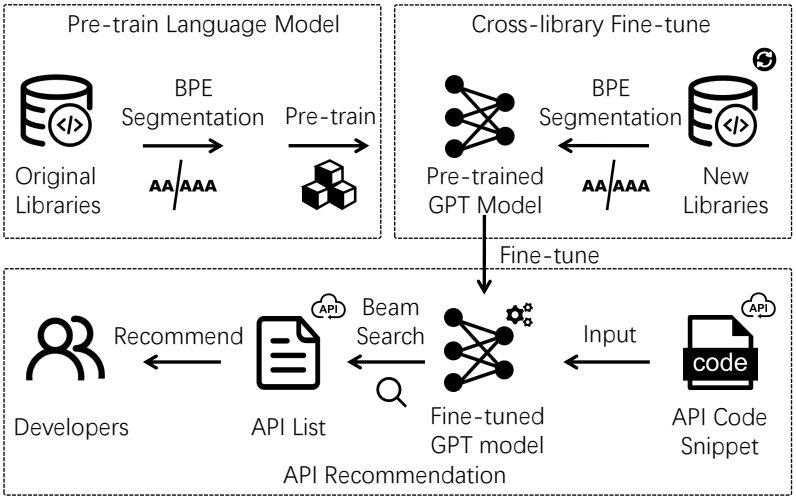

Figure 1: An Overview of APIRecX

2.1 Overview

Achieving the goal of cross-library API recommendation is challenging.

- First, different libraries tend to not contain APIs with the same names, and thus it is hard to adopt existing approaches to recommend APIs that are not seen in training data. That is, the first challenge is due to the OOV problem at the API level. To overcome it, APIRecX aims to recommend APIs at the subword level through API segmentation. The insight is that an API call usually consists of a set of relatively commonly-used subwords such as *Exception*, *print*, etc. Therefore, the OOV problem at the API level can be largely relieved at the subword level.

- Second, APIRecX recommends each subword in turn and then composes a complete API call for recommendation based on predicted subwords. That means that an API call can be correctly recommended only if all the subwords in the API call are recommended correctly, which largely aggravates the recommendation difficulty. To relieve the inaccuracy of API recommendation caused by inaccurate subword prediction, APIRecX incorporates beam search to enlarge the search space of API synthesis instead of directly recommending an API call composed by Top-1 subword in each prediction.

With the above two insights, we design a novel GPT-based method in APIRecX to build a subword language model. Here, APIRecX first pre-trains a subword language model based on a large number of API usage data of other libraries in an offline process. When new libraries are released, APIRecX then directly fine-tunes the pre-trained model after collecting a certain amount of API usage data of new libraries, which is much more efficient than retraining based on all API usage data (i.e., pre-training data and fine-tuning data). Also, to make APIRecX a light-weight approach, APIRecX does not build complex data-flow and control-flow graphs, but directly represents a method as an API sequence following the existing work (Gu et al., 2017; Yan et al., 2018; Nguyen et al., 2017). The overview of APIRecX is shown in Figure 1.

2.2 BPE-based API Segmentation

APIRecX extracts API sequences following the practice in the existing work (Gu et al., 2017), which extracts all API calls (identifier&arguments, e.g.DriverManager.getConnection(String)), and control statements with API call in a method to form an API sequence. Here, all variables in API sequences are replaced with their types. For example, for an API call `o.m()` where `o` is an instance of a class `C`, APIRecX adds `C.m` to the API sequence.

Although API names tend to be unique, they usually consists of a set of relatively commonly-used subwords. That is, different API names may include common subwords, and thus the OOV problem at the API level could be largely relieved at the subword level. With this insight, APIRecX splits an API call in an API sequence into a sequence of subwords and conducts follow-up learning and prediction at the subword level, and finally composes

a complete API call for recommendation based on predicted subwords. In this way, it is possible to compose an unseen API call in training data with subwords, which makes cross-library API recommendation become feasible.

Here, APIRecX adopts BPE (Provilkov et al., 2020; Sennrich et al., 2016; Devlin et al., 2018), one of the most widely-used word segmentation methods in text generation, for splitting an API call to subwords. The reason why choosing BPE is that it achieves a good balance between effectiveness and efficiency. More specifically, compared with character segmentation (Gao et al., 2020) ,whitespace segmentation (Tezcan et al., 2020; Mikolov et al., 2013),and CamelCase segmentation,BPE is more effective, since character segmentation is too fine-grained and thus leads to much semantic loss while whitespace segmentation is too coarse-grained for API calls and thus cannot effectively relieve the OOV problem.Although CamelCase segmentation can achieve a relatively appropriate segmentation granularity, compared with BPE,it has a larger granularity, which will cause more OOV words.By taking the domain of Swing as an example, there are 61.9% common subwords between training and test data achieved by BPE, while there are only 50.9% common subwords achieved by CamelCase.Compared with more advanced methods (e.g., WordPiece (Devlin et al., 2018) and ULM (Chen et al., 2005), BPE is more efficient but not much less effective, since these methods need to build language models during word segmentation while BPE is based on frequency. Besides, APIRecX adds a special subword `(/t)` to mark the end of each API call, which helps APIRecX determine the termination of subword recommendation for an API call. Through this step, APIRecX obtains a large amount of API usage data at the subword level. As an example, for the code in Listing 1, the API sequence after BPE-based segmentation is: *Connection–.–new()*$(/t)$*–TRY*$(/t)$*–Driver–Manager–.–get–Connection()*$(/t)$ *–CATCH*$(/t)$*–Exception–.–print–StackTrace()*$(/t)$.

2.3 Building a Subword Language Model

To build a subword language model, APIRecX adopts the "pre-training & fine-tuning" mechanism as presented above. That is, APIRecX first pre-trains a subword language model based on a large amount of subword data that do not involve APIs of new libraries, and then fine-tunes the pre-trained model by including a small amount of subword data involving the APIs of the library to be recommended. Besides the efficiency benefit presented above, fine-tuning has been demonstrated to be more effective than the strategy of direct training based on the mixed data of pre-training data and fine-tuning data (Mao et al., 2015), since the volume of API usage data of new libraries is significantly smaller than that of other libraries, leading to very difficult to learn usage patterns of the APIs of new libraries via the latter strategy.

In APIRecX, we design a GPT-based subword language modeling building method. GPT first maps an API subword sequence $S = a_1, ..., a_t$ into a vector matrix through the embedding layer Emb where t represents the total number of subword in the API subword sequence , and then we can get the embedding matrix H_0 of the API subword sequence after adding the position information through the position embedding matrix W_p.

$$H_x = \begin{cases} Emb(S) + W_p & x = 0 \\ Tblock(H_{x-1}) & 1 \leq x \leq n \end{cases} \quad (1)$$

Then, GPT inputs the obtained embedding matrix into the decoder block of the transformer for calculation. where x represents the order number of Transformer layers, and the vector matrix H_n outputted by the last layer of decoder block represents the attention weight for each subword in this sequence. Then, H_n is multiplied by the transpose of embedding layer matrix, and normalized by softmax to obtain $P(S)$ which represents the probabilities of all subwords in the vocabulary at each position in the sequence.

$$P(S) = Softmax(H_n * Emb^T) \quad (2)$$

In the training phase, we calculate the loss between ground truth and $P(S)$ through cross-entropy, and optimize GPT through the Adam optimization algorithm.

2.4 Beam-search based API Synthesis

With a subword language model, APIRecX recommends a subword in each prediction based on a sequence of subwords before the current position to be predicted. Given that an API call to be recommended is denoted as $A_m = \{s_m^1, s_m^2, \ldots, s_m^{n_m}\}$ where s_m^j refers to the j^{th} subword in A_m and n_m refers to the number of subwords in A_m, API calls before A_m are denoted as $\{A_1, A_2, \ldots, A_{m-1}\}$

where $A_i = \{s_i^1, s_i^2, \ldots, s_i^{n_i}\}$. When predicting at the position of s_m^1, APIRecX inputs $\{s_1^1, \ldots, s_1^{n_1}, \ldots, s_{m-1}^1, \ldots, s_{m-1}^{n_{m-1}}\}$ to the model, and when predicting at the position of s_m^j, APIRecX inputs $\{s_1^1, \ldots, s_1^{n_1}, s_{m-1}^1, \ldots, s_{m-1}^{n_{m-1}}, w_m^1, .., w_m^{j-1}\}$, where w_m^{j-1} is the predicted subword at the position of s_m^{j-1} and w_m^{j-1} is the same as s_m^{j-1} if the prediction is correct. That is, the currently predicted subword is used to predict subsequent subwords. When the prediction result ends with `(/t)`, APIRecX outputs the chain of predicted subwords as the API call for recommendation. For example, in listing 2, when the developer enters *e.* on line 6, APIRecX inputs $\{A_1, A_2, A_3, A_4, S_5^1, S_5^2\}$, where $A_1 = \{File, Input, Stream, ., new()(/t)\}$, $A_2 = \{TRY(/t)\}$, $A_3 = \{File, ., new(String)(/t)\}$, $A_4 = \{CATCH(/t)\}$, $S_5^1 = Exception$ and $S_5^2 = .$. Then APIRecX predicts the next subword based on the input. When APIRecX predicts a subword ending with (/t) such as $\{print, StackTrace()(/t)\}$, it will merge predicted subwords with S_5^1 and S_5^2 and return the result to the developer.

However, subword prediction aggregates the difficulty of API recommendation, since it is hard to guarantee the accurate prediction of each subword in an API call. Especially, when a wrong subword is predicted in a certain position, the predictions of all the subsequent subwords could be also affected, since the wrong subword will be used to predict subsequent subwords. Actually, each subword is assigned as a probability in each prediction. By considering all the subwords in each prediction and using each subword for subsequent predictions, the correct chain of subwords (used for composing a complete API call) cannot be missing, but exploring such enormous combination space is unaffordable. Therefore, it is still challenging to recommend a complete API call based on subword-level prediction.

To achieve the balance between the accuracy of API recommendation and the efficiency, APIRecX adopts widely-used beam search (Freitag and Al-Onaizan, 2017; Shu and Nakayama, 2018; Huang et al., 2017). More specifically, beam search considers Top-K subwords (K refers to beam size) in each prediction rather than only Top-1 subword or all the subwords. For each of Top-K subwords in a prediction, it then produces Top-K subwords and obtains K^2 chains of subwords, and then preserves Top-K chains according to their chain probabilities for the next prediction. Following the existing work (Shu and Nakayama, 2018; Huang et al., 2017; Freitag and Al-Onaizan, 2017; Karampatsis et al., 2020), we use Formula 3 to calculate the chain probability of a chain of subwords:

$$P(w_m^1, ..., w_m^i | s_1^1, ..., s_1^{n_1}, ..., s_{m-1}^{n_{m-1}}) = \prod_{j=1}^{i} p(w_m^j) \quad (3)$$

where, $p(w_m^j)$ (which is short for $p(w_m^j | s_1^1, \ldots, s_1^{n_1}, \ldots, s_{m-1}^{n_{m-1}}, w_m^1, \ldots, w_m^{j-1})$) is the probability of the j^{th} subword in the chain of $(w_m^1, \ldots, w_m^i)$ predicted by the subword model.

To relieve the effectiveness problem caused by the monotonicity of traditional beam search, APIRecX preserves the memory of poor-quality incomplete chains produced during the process of beam search following the existing work in text generation (Shu and Nakayama, 2018). More specifically, APIRecX constructs a candidate pool that stores the remaining incomplete chains except Top-K chains among k^2 chains produced in each prediction. When k^2 chains produced based on Top-K chains selected from the last prediction have smaller chain probabilities than those of chains in the candidate pool, APIRecX chooses Top-K chains among the k^2 chains produced in the current prediction and all the chains in the candidate pool rather than only the current k^2 chains. In this way, APIRecX has a chance to make up wrong choice in previous predictions. Besides, APIRecX improves the condition of terminating the beam search process following the existing work (Huang et al., 2017) in text generation, i.e., the searching stops until the smallest chain probability among all the produced complete chains is larger than the largest chain probability among all incomplete chains (including incomplete chains in both the candidate pool and current Top-K chains).

3 Evaluation

3.1 Experimental Setup

3.1.1 Datasets

We used six JDK libraries from three domains to mimic new libraries in the scenario of cross-library API recommendation. They are `java.sql` and `javax.sql` in the domain of JDBC (which is the domain about database operations), `java.awt` and `javax.swing` in the domain of Swing

Domain	#API	#Project	#Sequence
JDBC	909	784	42,298
Swing	10,622	722	63,249
IO	1,192	205	15,356

Table 1: Statistical information on three domains

#Projects	14,807
#LOC	352,312,696
#Methods	15,201,014
#Sequence	5,120,310

Table 2: Statistical information on pre-train corpus

(which is the domain about user interfaces), and `java.io` and `java.nio` in the domain of IO (which is the domain about stream-based inputs and outputs), respectively. Based on the three domains, we conducted three groups of experiments, each of which uses the two libraries in the corresponding domain as the new libraries for recommendation. Table 1 shows the information about the three experiments. where Column "#API" is the number of APIs in the corresponding domain libraries, Column "#Project" is the number of Java projects that are collected from GitHub and use the APIs in the domain libraries, and Column "#Sequence" is the number of API sequences that are extracted from the collected projects.

Besides, we adopted the corpus provided by the existing work (Allamanis and Sutton, 2013) for pre-training. The corpus has over 14,000 Java projects from GitHub after removing the projects involving the above three domains. From these projects, we extracted over 5,000,000 API sequences as pre-training data.Table 2 shows the information about the pre-train corpus. where Column "#Projects" is the number of Java projects in pre-train corpus, Column "#LOC" is the total number of lines of code, Column "#Methods" is the total number of java methods, and Column "#Sequence" is the number of API sequences that are extracted from this corpus.

3.1.2 Selecting test and fine-tune data

We used 10 projects (splitting domain projects into 10 groups and then selecting the one with the largest number of domain API calls in each group) as test projects, and extract API call sequences from them.For each sequence of API calls, we produced a set of API call sequences, each of which contain a "hole", as test data. Specifically, we produced them by digging a "hole" from the second API call in the sequence in turn respectively. Then, for each API call sequence with a "hole", we used the sequence of API calls before the "hole" as input for predicting the API call in the "hole". After selecting the test data, we sample a certain amount of data from the remained data at 5 different sampling ratios which are 0.2%, 1%, 10%, 50%, and 100% as fine-tune data.Then we use these sampled data to fine-tune the pre-trained model following the fine-tuning process presented in Section 2.3

3.1.3 Baselines

We adopted traditional LSTM-based API recommendation approach (Yan et al., 2018; White et al., 2015; Zhang et al., 2019; Chen et al., 2019) and N-gram based API recommendation approach (Raychev et al., 2014; Karampatsis and Sutton, 2019) for comparison in order to quantitatively investigate the superiority of APIRecX over traditional API recommendation approaches. We refer to the parameter settings in these two works(Yan et al., 2018; Raychev et al., 2014) to train baseline tools on the data we collected, and the specific parameter settings are shown in Table 6.

3.1.4 Parameters

The parameters comprise the model training parameters and the beam search parameters in the API recommendation process. Table 6 lists all the parameters of APIRecX and baselines. The structure of original GPT contains a 12-layer transformer decoder block with 12-head attention, containing nearly 100 million parameters, which requires an extremely huge amount of data to support training. However, compared with collecting text data, it is harder to collect such a huge amount of API usage data to support training such a complicated model, and thus we tailored the structure of the original GPT to match with the scale of our training data. Specifically, our tailored GPT uses a 6-layer transformer decoder block with 8-head attention. Besides, GPT handles fixed-length sequences, thus we set the subword-sequence length to be 512. In our context, the fixed-length sequence refers to the fixed-length subword sequence processed from an API call sequence. For the API call sequences in our dataset, the average length is 41, the largest length is 2,280, and the percentage of subword sequences that are longer than 512 is only 0.4%. Moreover, the longer the sequence is, the more difficult it is to model. Therefore, our setting (512) could reach a good trade-off following the existing study(Devlin et al., 2018). The baseline

model parameters were set according to the previous work(Yan et al., 2018; Raychev et al., 2014). We trained the APIRecX for 15 epochs in the pre-training stage, and then we adopted the early stop strategy to terminate the fine-tuning process in the fine-tuning stage. For baseline approaches, we adopted early stop strategy to terminate the training process according to the previous work.

Beam search process contains two parameters: beam size and max iteration. Beam size represents the width of beam search and max iteration represents the maximum search epoch. More details of parameters setting will be shown in the Appendix.

3.1.5 Evaluation Metric

To evaluate the performance of APIRecX, we adopted *Top-N accuracy* following the existing work on API recommendation (Xie et al., 2019; Nguyen et al., 2016; Nguyen and Nguyen, 2015). Each API recommendation approach can produce a ranking list of API calls for recommendation. Top-N accuracy measures the percentage of the cases that the correct API call is included in Top-N results among all the locations in the test set, and higher Top-N accuracy indicates better performance. Following the existing work (Nguyen et al., 2016; Nguyen and Nguyen, 2015; Xie et al., 2019; Yan et al., 2018), we set N to be 1, 5, and 10 respectively. Note that We focus on the recommendation of domain APIs, so we only report the accuracy of Top-N recommendation of domain APIs.

3.2 Results and Analysis

3.2.1 Overall effectiveness

Table 3 presents the comparison results between APIRecX and baselines under five sampling ratios in three domains, respectively.

From this table, APIRecX performs better than the two baselines under all the studied sampling ratios in all the three domains in terms of all the metrics. For example, under the sampling ratio of 0.2% in the domain of IO, APIRecX has achieved 52.9% Top-1 accuracy while the two baselines are only 30.6% and 16.5%. The improvements are 72.87% and 220.61%, respectively. We also performed a Wilcoxon rank sum test to investigate whether our approach can significantly outperform LSTM and N-gram across all the domains respectively. The results show that all the p-values are smaller than 0.004 (<0.05) regardless of Top-1/Top-5/Top-10 accuracy, demonstrating the effectiveness of our approach in statistics.

We then analyzed why APIRecX performs well as shown in Table 4. In this table, the fast three rows present the percentage that training data cover domain APIs in the test set, the percentage that training data cover subwords from domain APIs in the test set, percentage of unseen APIs in the correct recommendation result, and the last rows present the number of API call types that successfully recommended by APIRecX under the sampling ratio of 0.2%.

From Table 4, under the sampling ratio of 0.2%, the API coverage is small (10.9∼49.3%), only 25.5% APIs are covered by training data on average, but the subword coverage is large (61.9∼89.3%) and the average subword coverage rate reached 77.7%, indicating the power of API segmentation to handle the OOV problem. Indeed, APIRecX is able to recommend unseen APIs in both pre-training and fine-tuning data. For example, Among the APIs correctly recommended by the APIRecX, an average of 28.1%, 131.3 types is from the unseen APIs,demonstrating its ability for cross-library API recommendation.

3.2.2 Effectiveness of Beam Search

We compare our beam search strategy in APIRecX and the traditional beam search (Freitag and Al-Onaizan, 2017; Shu and Nakayama, 2018) under different beam sizes. Here, we use the JDBC domain with and the sampling ratio of 10% as the representative, whose comparison results are shown in Table 5. From this table, our used beam search performs better than traditional beam search under all the studied beam sizes in terms of all the metrics, demonstrating the contribution of the improved beam search strategy. In the meanwhile, its contribution becomes more obvious in Top-5 accuracy and Top-10 accuracy than Top-1 accuracy because the rescued chains of subwords by the improved beam search are difficult to have larger chain probabilities than Top-1 chain due to the small probability of certain subword prediction. More specifically, the probability of a complete API call (e.g., printStackTrace() in Line-6 of Listing-1) is the product of the probabilities of a chain of subwords (e.g., print, StackTrace, ()). Although the candidate pool storage of the improved beam search can relieve the effectiveness problem caused by the monotonicity of traditional beam search through preserving the memory of poor-quality incomplete chains produced during the beam-search process, the small probabilities of poor-quality incomplete

Sample	Approach	JDBC			Swing			IO		
		Top-1	Top-5	Top-10	Top-1	Top-5	Top-10	Top-1	Top-5	Top-10
0.2%	APIRecX	37.9	74.7	81.2	25.0	43.8	51.2	52.9	69.5	73.7
	LSTM	26.8	52.6	65.9	15.1	31.3	39.1	30.6	53.4	63.3
	N-gram	11.9	41.5	56.5	7.9	26.3	31.6	16.5	45.9	57.0
1%	APIRecX	42.8	77.7	83.7	25.3	46.9	54.5	56.4	75.5	79.8
	LSTM	31.6	67.4	74.8	17.2	34.3	44.0	36.7	56.5	66.4
	N-gram	16.0	40.6	58.5	10.2	28.2	36.7	16.7	45.9	57.8
10%	APIRecX	46.9	79.9	85.7	40.6	67.8	74.5	56.9	75.9	80.5
	LSTM	33.7	69.1	75.3	30.6	53.1	60.9	36.1	60.8	70.0
	N-gram	18.6	43.8	59.3	16.3	37.5	46.9	18.1	48.3	59.2
50%	APIRecX	56.6	86.3	93.0	48.7	79.0	80.9	60.6	81.9	85.3
	LSTM	41.8	73.8	84.7	32.8	56.7	65.0	39.1	64.1	70.9
	N-gram	25.4	55.1	63.7	16.3	39.4	48.7	18.6	48.5	62.8
100%	APIRecX	60.0	89.4	94.5	54.8	77.2	83.7	63.9	84.2	88.7
	LSTM	43.4	76.2	85.6	36.7	61.1	69.2	40.1	67.1	75.3
	N-gram	28.6	56.1	65.5	18.7	41.4	50.7	21.6	52.8	68.8

Table 3: Overall effectiveness of APIRecX

Criterion	Domain			Avg.
	JDBC	Swing	IO	
API Coverage	49.3%	10.9%	16.3%	25.5%
Subword Coverage	82.0%	61.9%	89.3%	77.7%
OOV Correct Rate	8.7%	26.4%	49.6%	28.1%
OOV Correct API type	14.7	317.6	61.5	131.3

Table 4: Analysis of the results

Beam size	Method	Top-1	Top-5	Top-10
10	Ours	45.2	79.4	84.3
	Traditional	44.1	69.7	76.1
15	Ours	46.6	78.6	84.6
	Traditional	44.5	69.1	75.8
20	Ours	46.9	79.9	85.7
	Traditional	44.1	70.1	77.1
25	Ours	46.6	83.1	85.7
	Traditional	44.0	72.7	77.2
30	Ours	45.3	80.1	86.7
	Traditional	44.3	71.9	78.0

Table 5: The results of different beam search methods on JDBC

chains could lead to the small probability of the corresponding complete API call, making it hard to be ranked as Top-1. Taking Line-6 in Listing-1 as an example, if "StackTrace" has a small probability, its small probability could make the probability of the complete API call small, causing it hard to be ranked as Top-1. Therefore, the improved beam search has less apparent improvement in terms of Top-1 accuracy. Also, APIRecX performs stably under different beam sizes.

4 Related work

4.1 API recommendation

In the literature, some statistical learning based (Nguyen and Nguyen, 2015; Liu et al., 2018; Raychev et al., 2014; Xie et al., 2019) and pattern mining based API recommendation approaches (Zhong et al., 2009; Wang et al., 2013; Fowkes and Sutton, 2016; Xie et al., 2019) have been proposed without dealing with the OOV problem, and thus all of them cannot be effective in the scenario of cross-library API recommendation. For example, Xie et al. (2019) proposed HiRec, which improves pattern-mining based approaches by utilizing the hidden information of project-specific code via call graph in mining API usage patterns. Nguyen and Nguyen (2015) designed a graph-based statistical language model by representing source code as graphs for API recommendation. Different from them, APIRecX is the first approach for cross-library API recommendation by handling the OOV problem via GPT-based pre-trained subword language model.

4.2 Pre-trained models across languages

Our approach is inspired by pre-training in the multilingual scenario (Chi et al., 2020; Huang et al., 2019; Yang et al., 2020a,b, 2019). For example, Lample and Conneau (2019) proposed the XLM model, which processes multiple languages via BPE so that all the languages can share subword dictionaries. Ren et al. (2019) proposed the cross-lingual masked language model, which uses more explicit cross-lingual information (such as translation table). More specifically, they used the monolingual corpus of two languages to train the monolingual N-gram vector through FastText (Bo-

janowski et al., 2017), and then used the unsupervised cross-lingual word vector method, VecMap, (Garneau et al., 2020) to obtain the cross-lingual N-gram vector. The translation table between the two languages is inferred from the similarity of the N-gram vectors of the two languages.

Different from them, our work targets the problem of API recommendation rather than cross-lingual problems, which have different characteristics, and APIRecX builds a GPT-based subword language model for API recommendation. CodeBERT(Feng et al., 2020) gets a general language model about programming language by pre-trained on six different programming languages, and can be applied to different downstream tasks,It seems that codebert can be our baseline but the reason why not use CodeBERT as the baseline for comparison is that it needs two-way information and we regard API recommendation as a one-way text generation task. When developers use API, they usually write API calls sequentially (forward) and the task of API recemmendation is to predict the future API calls, there is no reverse information (backward) in practice. Therefore, CodeBERT cannot be applied to our problem.

5 Conclusions

We propose the first approach APIRecX for cross-library API recommendation, which can automatically recommend API calls for new libraries. APIRecX first splits each API call into a sequence of subwords to relieve the OOV problem at the API level. It then pre-trains a GPT-based subword language model based on a large number of API usage data from other libraries. By fine-tuning the pre-trained model with a sample of API usage data of new libraries, APIRecX conducts subword prediction and incorporates beam search to compose a complete API call for recommendation. We conduct an extensive study based on six libraries of three domains for mimicking new libraries and 14,000 GitHub Java projects for pre-training, demonstrating the effectiveness of APIRecX. However, our work also has certain limitation, which is the generalization of our results and findings. Although we invested significant time and effort to prepare datasets, conducted experiments and analyzed results, our experiments involved only one program language with three domains. The performance of our neural architecture, and especially the findings on transfer learning, could be different with other programming languages or libraries. In the future, we will try to get rid of this limitation by applying our approach to more languages/libraries.

The source code of APIRecX and experimental data can be found in `https://github.com/yuningkang/APIRecX`.

6 Acknowledgements

This work was funded by National Natural Science Foundation of China (Nos. 61872263, 62002256, 20201180), Intelligent Manufacturing Special Fund of Tianjin. We are also very grateful to reviewers for their helpful comments.

References

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216. IEEE Computer Society.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5:135–146.

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 213–222, New York, NY, USA. Association for Computing Machinery.

Aitao Chen, Yiping Zhou, Anne Zhang, and Gordon Sun. 2005. Unigram language model for chinese word segmentation. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing, SIGHAN@IJCNLP 2005, Jeju Island, Korea, 14-15, 2005*. ACL.

Chi Chen, Xin Peng, Jun Sun, Zhenchang Xing, Xin Wang, Yifan Zhao, Hairui Zhang, and Wenyun Zhao. 2019. Generative API usage code recommendation with parameter concretization. *Sci. China Inf. Sci.*, 62(9):192103:1–192103:22.

Zewen Chi, Li Dong, Furu Wei, Wenhui Wang, Xian-Ling Mao, and Heyan Huang. 2020. Cross-lingual natural language generation via pre-training. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 7570–7577. AAAI Press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

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 *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 1536–1547. Association for Computational Linguistics.

Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic api mining across github. FSE 2016, page 254–265, New York, NY, USA. Association for Computing Machinery.

Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation, NMT@ACL 2017, Vancouver, Canada, August 4, 2017*, pages 56–60. Association for Computational Linguistics.

Yingqiang Gao, Nikola I. Nikolov, Yuhuang Hu, and Richard H. R. Hahnloser. 2020. Character-level translation with self-attention. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 1591–1604. Association for Computational Linguistics.

Nicolas Garneau, Mathieu Godbout, David Beauchemin, Audrey Durand, and Luc Lamontagne. 2020. A robust self-learning method for fully unsupervised cross-lingual mappings of word embeddings: Making the method robustly reproducible as well. In *Proceedings of The 12th Language Resources and Evaluation Conference, LREC 2020, Marseille, France, May 11-16, 2020*, pages 5546–5554. European Language Resources Association.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. Deep api learning.

Enno Hermann, Herman Kamper, and Sharon Goldwater. 2021. Multilingual and unsupervised subword modeling for zero-resource languages. *Comput. Speech Lang.*, 65:101098.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847. IEEE Computer Society.

Haoyang Huang, Yaobo Liang, Nan Duan, Ming Gong, Linjun Shou, Daxin Jiang, and Ming Zhou. 2019. Unicoder: A universal language encoder by pre-training with multiple cross-lingual tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 2485–2494. Association for Computational Linguistics.

Liang Huang, Kai Zhao, and Mingbo Ma. 2017. When to finish? optimal beam search for neural text generation (modulo beam size). In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2134–2139, Copenhagen, Denmark. Association for Computational Linguistics.

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, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 293–304. ACM.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: Open-vocabulary models for source code. *CoRR*, abs/2003.07914.

Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe deep neural networks are the best choice for modeling source code. *CoRR*, abs/1903.05734.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining.

Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective api recommendation without historical software repositories. ASE 2018, page 282–292, New York, NY, USA. Association for Computing Machinery.

Junhua Mao, Xu Wei, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan L. Yuille. 2015. Learning like a child: Fast novel visual concept learning from sentence descriptions of images. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 2533–2541. IEEE Computer Society.

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 511–522, New York, NY, USA. Association for Computing Machinery.

Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on*

Software Engineering - Volume 1, ICSE '15, page 858–868. IEEE Press.

Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 438–449. IEEE / ACM.

Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. 2020. Bpe-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 1882–1892. Association for Computational Linguistics.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. 49(6):419–428.

Shuo Ren, Yu Wu, Shujie Liu, Ming Zhou, and Shuai Ma. 2019. Explicit cross-lingual pre-training for unsupervised machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 770–779. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Raphael Shu and Hideki Nakayama. 2018. Improving beam search by removing monotonic constraint for neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 339–344, Melbourne, Australia. Association for Computational Linguistics.

Arda Tezcan, Véronique Hoste, and Lieve Macken. 2020. Estimating word-level quality of statistical machine translation output using monolingual information alone. *Nat. Lang. Eng.*, 26(1):73–94.

Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 319–328. IEEE Computer Society.

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 334–345. IEEE Press.

Rensong Xie, Xianglong Kong, Lulu Wang, Ying Zhou, and Bixin Li. 2019. Hirec: API recommendation using hierarchical context. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, pages 369–379. IEEE.

Jinpei Yan, Yong Qi, Qifan Rao, and Hui He. 2018. Learning API suggestion via single LSTM network with deterministic negative sampling. In *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018*, pages 137–136. KSI Research Inc. and Knowledge Systems Institute Graduate School.

Jian Yang, Shuming Ma, Dongdong Zhang, Shuangzhi Wu, Zhoujun Li, and Ming Zhou. 2020a. Alternating language modeling for cross-lingual pre-training. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 9386–9393. AAAI Press.

Jian Yang, Shuming Ma, Dongdong Zhang, Shuangzhi Wu, Zhoujun Li, and Ming Zhou. 2020b. Alternating language modeling for cross-lingual pre-training. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 9386–9393. AAAI Press.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237.

Haoyu Zhang, Jingjing Cai, Jianjun Xu, and Ji Wang. 2019. Pretraining-based natural language generation for text summarization. In *CoNLL*, pages 789–797. Association for Computational Linguistics.

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, page 318–343, Berlin, Heidelberg. Springer-Verlag.

Appendix

A Parameter settings

Section	Approach	Hyperparameter	Value
Model	GPT	ffn_hidden	512
		hidden	256
		num_head	8
		num_layer	6
		batch size	32
		sequence length	512
		learning rate	0.00015
		epoch(pre-train)	15
		epoch(fine-tune)	Early Stop
	LSTM	hidden	128
		num_layer	2
		batch size	128
		sequence length	60
		learning rate	0.005
		epoch	Early Stop
	N-gram	hidden	300
		context size	3
		batch size	40000
		learning rate	0.005
		epoch	Early Stop
Beam Search	-	beam size	20
		max iteration	10

Table 6: Parameters of APIRecX and baseline

B Retrain and pre-train

Our experiments shows that the "pre-train&fine-tune" mechanism is effective and efficient than the one-step training strategies. Table 8 lists the domain API recommendation accuracy of the model trained in three training strategies. "Pre-train&fine-tune" represents the strategy used in training APIRecX introduced in Section 2.3, "re-train" means training APIRecX from scratch using three different proportions of fine-tuning data combined with pre-training data in three domains.

Beam size	Sample	Top-1	Top-5	Top-10
10	0.2%	38.4	71.3	76.5
	10%	45.2	79.4	84.3
	100%	58.7	88.1	92.7
15	0.2%	38.2	73.1	79.3
	10%	46.6	78.6	84.6
	100%	58.8	88.1	93.7
20	0.2%	38.2	74.8	81.2
	10%	46.9	79.9	85.7
	100%	60.0	89.4	94.5
25	0.2%	38.5	74.1	81.5
	10%	46.6	83.1	85.7
	100%	59.6	88.7	93.1
30	0.2%	38.4	73.5	81.9
	10%	45.3	80.1	86.7
	100%	58.8	88.2	93.9

Table 7: Different beam size results in JDBC domain

Domain	Ratio	Strategy	Top-1	Top-5	Top-10
JDBC	100%	pre-train&fine-tune	60	89.4	94.5
		retrain	54.5	85.6	91.1
		scratch	52.9	85.4	91.9
	10%	pre-train&fine-tune	46.9	79.9	85.7
		retrain	42.1	71.5	79.4
		scratch	30.2	56.9	64.7
	0.2%	pre-train&fine-tune	37.6	75	81.2
		retrain	27.7	50.4	53.1
		scratch	13.2	33.2	33.3
Swing	100%	pre-train&fine-tune	54.8	77.2	83.7
		retrain	44.4	71.3	77.6
		scratch	48.8	75.3	80.8
	10%	pre-train&fine-tune	40.6	67.8	74.5
		retrain	33.6	57.9	64.2
		scratch	25.9	50.7	60.6
	0.2%	pre-train&fine-tune	25	43.8	51.2
		retrain	23.6	43.1	47.7
		scratch	3.2	4.9	8.8
IO	100%	pre-train&fine-tune	63.9	84.2	88.7
		retrain	62.7	81.9	87.2
		scratch	32.4	62.8	71.4
	10%	pre-train&fine-tune	56.9	75.9	80.5
		retrain	56.9	74.9	79.1
		scratch	0.7	10.2	20.8
	0.2%	pre-train&fine-tune	52.9	69.5	73.7
		retrain	51.7	68.4	71.3
		scratch	0.05	0.05	1.4

Table 8: Pre-train and retrain result

"Scratch" means training APIRecX from scratch using only three different proportions of fine-tuning data. As shown in Table 8, the "pre-train&fine-tune" mechanism is better than the other two one-step strategy at three sampling ratios, and proves superiority under low sampling ratios.

C Beam Size evaluation

We evaluate the effectiveness of different beam size under three different sampling ratios of JDBC domain to find the suitable beam size. Table 7 lists the average recommendation accuracy rates achieved in 5 different beam sizes under three different sampling ratios in JDBC domain. Table 7 shows that, as the beam size increases, the duration and the accuracy both increases. After the beam size reaches 20, the accuracy increases rather slowly and remains basically unchanged. To balance the performance and efficiency of APIRecX, we set beam size to be 20 as the parameter of other comparative experiments.