**When Language Model Meets Private Library**

Daoguang Zan\(^1,2\)∗, Bei Chen\(^3\), Zeqi Lin\(^3\), Bei Guan\(^2,4\), Yongji Wang\(^2,4,5\), Jian-Guang Lou\(^3\)

\(^1\)Cooperative Innovation Center, Institute of Software, Chinese Academy of Sciences
\(^2\)University of Chinese Academy of Sciences; \(^3\)Microsoft Research Asia
\(^4\)Integrative Innovation Center, Institute of Software, Chinese Academy of Sciences
\(^5\)State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

{daoguang@, guanbei@, ywang@itechs.}iscas.ac.cn
{beichen, zeqi.lin, jlou}@microsoft.com

**Abstract**

With the rapid development of pre-training techniques, a number of language models have been pre-trained on large-scale code corpora and perform well in code generation. In this paper, we investigate how to equip pre-trained language models with the ability of code generation for private libraries. In practice, it is common for programmers to write code using private libraries. However, this is a challenge for language models since they have never seen private APIs during training. Motivated by the fact that private libraries usually come with elaborate API documentation, we propose a novel framework with two modules: the APIRetriever finds useful APIs, and then the APICoder generates code using these APIs. For APIRetriever, we present a dense retrieval system and also design a friendly interaction to involve users. For APICoder, we can directly use off-the-shelf language models, or continually pre-train the base model on a code corpus containing API information. Both modules are trained with data from public libraries and can be generalized to private ones. Furthermore, we craft three benchmarks for private libraries, named TorchDataEval, MonkeyEval, and BeatNumEval. Experimental results demonstrate the impressive performance of our framework.\(^1\).

1. Introduction

Code generation, automatically generating code snippets based on user descriptions, is one of the long-standing challenges in the software engineering and artificial intelligence communities. With the rapid development of pre-training techniques, a number of language models are pre-trained on large-scale code corpora and able to generate decent code snippets, for example, Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), CODEGEN (Nijkamp et al., 2022), and InCoder (Fried et al., 2022). They bring fresh energy to code generation and improve coding efficiency (Vaihtilingam et al., 2022).

Although making remarkable progress, these models may be biased towards generating code that is similar to the training distribution (Chen et al., 2021). What if one wants to generate code beyond the training distribution? A real-world scenario for programmers is to write code using a private library, which is very common in practice. For example, for security and functionality reasons, companies often build private libraries for internal use only. Private libraries provide a number of private APIs that have not been seen by the language models and are also not publicly available on any code hosting platform like GitHub. Therefore, it is worth exploring whether and how pre-trained language models can generate code using private libraries.

It is challenging for existing language models to generate code that uses a private library directly. A practical evidence is shown in Figure 1. We built a pseudo private library named Monkey based on a

---

\(^*\)Work done as an intern at Microsoft Research Asia.

\(^1\)Our work is available at https://github.com/microsoft/PyCodeGPT/tree/main/apicoder.

---

Figure 1: A practical example of converting PandasEval (public) to MonkeyEval (private). The changed parts are highlighted in yellow. The performance of Codex 12B and CODEGEN-Mono 350M is shown at the bottom.
public one named Pandas. PandasEval (Zan et al., 2022) is a benchmark consisting of 101 Pandas programming problems. We convert all Pandas-related keywords in PandasEval into the new version and construct MonkeyEval (details in Section 4). As seen in Figure 1, Codex 12B and CODEGEN-MONO 350M show a significant drop in performance on the private MonkeyEval compared to their performance on the public PandasEval. For example, Codex 12B drops from 18.88% to 1.47% on pass@1, showing the inadequacy of the language models in code generation for private libraries.

To meet the challenge, we propose a framework to equip pre-trained language models with the ability to generate code that uses private libraries. As is known, private libraries usually come with elaborate API documentation, which motivates our main idea to mimic the process of a programmer learning to write code using private libraries. This process is also known as API practices in the software engineering field (Snodgrass and Winnie, 2019): first learning the private API documentation and then invoking the APIs to implement the needed functionalities. Analogically, there are two modules in our framework: an APIRetriever first retrieves the useful APIs based on the programming problem and the API documentation, and then an APICoder uses these APIs to generate code. For APIRetriever, we train a dense retriever and also design a friendly interaction to involve users in the loop optionally. For APICoder, we can directly use existing language models of code generation, such as CODEGEN, to invoke the private APIs; furthermore, to better teach a language model how to invoke APIs, we also continually pre-train the base model on a code corpus containing API information from public libraries, and obtain our reinforced model called CODEGENAPI. Since we only have access to the data of public libraries during training, we expect that APIRetriever and APICoder can be generalized to private libraries via learning.

To evaluate the code generation for private libraries, we craft three benchmarks, named TorchDataEval, MonkeyEval, and BeatNumEval. TorchDataEval includes 50 programming problems using the TorchData library. The last two are adapted from PandasEval and NumpyEval (Zan et al., 2022), respectively, each consisting of 101 programming problems. Extensive experiments on the three benchmarks have revealed that our framework effectively improves the performance of pre-trained language models on code generation for private libraries. We also provide a thorough analysis to facilitate progress in this direction.

2 Framework

First, we would like to define the task of code generation formally. Given context, the task aims to generate target code. In Figure 1, context and target code are shown in white and grey backgrounds, respectively. Context consists of a comment, which is a natural language description of the programming problem, and a code snippet including import statements, function header, etc. Target code solves the programming problem in context. We denote the context by x. Code generation model M outputs target code y based on x. For the task of code generation for private library, the context x contains the instruction for using a private library, such as an import statement. The target code y contains the calls of the corresponding private library APIs.

As mentioned in Section 1, private libraries are usually equipped with elaborate API documentation. As a technical reference manual outlining how to use the library, API documentation typically includes a quick start guide, tutorials, and an instruction for each API (e.g., API name, signature, description, parameters, and examples). To take advantage of the API documentation, we propose to mimic the generic process of a programmer coding with private APIs, and design a framework to generate code that can invoke private APIs. The framework consists of APIRetriever and APICoder with the overview shown in Figure 2. Given the context, APIRetriever M_R aims to retrieve possible used APIs from the API documentation; and APICoder M_C is dedicated to generating code using the retrieved APIs. The process can be formalized as A = M_R(x) and y = M_C(A; x), where A represents the set of information of all proper APIs, and each a ∈ A is the information of an API. In our implementation, we design the API information to include the API name, signature and description. Note that we only use the first sentence of the API
description since it is sufficient to summarize.

3 Methodology

We have introduced our framework that provides pre-trained models a fantastic way to deal with private libraries. In this section, we present the data collection, followed by the detailed design of our APIRetriever and APICoder.

3.1 Data Collection

We collect API information and code files of public libraries due to the fact that we can only access data from public libraries. Then we train the models based on the public data with the expectation that the model can be generalized to private libraries. For API information, we consider the 31 most popular public libraries in Python (e.g., Pandas, NumPy, and scikit-learn) according to the popularity ranking on StackOverFlow2. For each of the libraries, we crawled its API documentation and extracted detailed information about each API, including the API name, signature, description, parameters, usage examples, and so on. Please refer to Appendix A for the details of the 31 public libraries. For code files, we first collected a 330GB corpus from GitHub containing 60.6M python files and then extracted those files that involved one or more API calls from the 31 public libraries. After a bunch of pre-processing strategies, for example, de-duplicating, cleaning, and formatting, we finally obtained 4.54M python files, denoted by $D$.

3.2 APIRetriever

APIRetriever aims to find the proper APIs based on the description of a programming problem. We regard it as a dense retrieval task (Qu et al., 2020; Xiong et al., 2020; Santhanam et al., 2021; Formal et al., 2022) and design a simple dual-encoder model (Karpukhin et al., 2020) to retrieve the possible used APIs for each programming problem. To further boost the retrieval performance, a friendly interaction approach is designed to involve users.

Training. To train APIRetriever, we need a large amount of pairwise data, natural language description and API information. We first segment each python file $d \in D$ into $K$ code blocks $(d_1, d_2, \cdots, d_K)$ using the pip-tools, i.e., redbaron, autopep8, and docformatter, where each code block is a relatively well-rounded code fragment, such as a function or a class. For each code block $d_i$, we extract all API names and obtain the corresponding API signatures and descriptions by searching our collected 31 API documentations3. The information of an API includes its name, signature, and description, denoted by $a \in A$. Each $a$ and the natural language description $p$ extracted from the same code block $d_i$ are regarded as a positive training sample. For the negative sampling, we randomly sample an API $a$ that is unrelated to $d_i$ from the same library. In total, we obtained 40.3M $(p, a, a_1, a_2, \ldots)$ sets as training samples. As in Figure 3, the left part shows the training process of APIRetriever. Our APIRetriever is a dual-encoder model. The two dense encoder, $E_p(\cdot)$ and $E_a(\cdot)$, map $p$ and $a$ to $z$-dimensional vectors, respectively. Then, we use the dot product of their vectors to calculate the similarity score formalized as $E_p(p)^\top E_a(a)$, where $E_p(\cdot)$ and $E_a(\cdot)$ are implemented by two independent BERT (Devlin et al.,

---

2https://stackoverflow.com/tags?tab=popular

3If an API name matches more than one candidate API, we randomly pick one.
Programming Problem

```python
from torchdata.datapipes.iter import IterableWrapper
datapipe = IterableWrapper([1,2,3])
# How to augment the datapipe by repeating it six times.
new_datapipe = ...
```

**Choices:** (choice: API Name: API Description)

1. **flatmap**: Applies a function over each item from ... 0
2. **cycle**: Cycles the specified input in perpetuity by default, or for the specified number of times. 0
3. **map**: Yields one element at a time from each of ... 0
4. **header**: Yields elements from the source DataPipe ... 0
5. **concat**: Concatenates multiple Iterable DataPipes ... 0
6. **None of the above.** 0
7. **Not sure.** 0

Which APIs would you like to use?

Your Choices: [ ]

Figure 4: Friendly interaction interface for users.

in Figure 3. Each API information is in the form of “name(signature):description”. This is to mimic programmers learning the APIs properly before writing code using them.

Technically speaking, the off-the-shelf code generation models, such as CodeT5, CodeGPT, Code-Clippy, CodeParrot, CODEGEN, and Codex, can be applied directly to land APICoder. Although these base models can achieve gains in correctly invoking APIs, they have not learned how to use them as an explicit training task. To better use the APIs, we devised a fantastic idea of continually pre-training the base models using code files with API information inserted.

In practice, we use CODEGEN-MONO 350M (Nijkamp et al., 2022) as our base model, based on which we continually pre-train and obtain our reinforced model called CODEGENAPI. CODEGEN is a GPT-based model skilled at generating code. We choose it because it is by far the most popular and publicly available model. As for the training corpus, we use the collected python files \( D \) mentioned in Section 3.1. Firstly, as done for APIRetriever, each file \( d \in D \) is split into \( K \) code blocks \( (d_1, d_2, \ldots, d_K) \). For each code block \( d_i \), we obtain the set of API information \( A_i \). Then, the \( K \) code blocks and sets of API information are cross-merged to output a new file \( d = (A_1, A_2, d_2, \ldots, A_K, d_K) \). This mimics the process of API information as a prompt for each block. Then, we continually pre-train the base model on the new code files, teaching the model to write code based on the prompted APIs. In addition, as shown in Figure 3, to make APICoder more robust, we shuffle the APIs in each set \( A_i \) and also add noise APIs, since APIRetriever does not know the order of APIs in the target code and often finds incorrect APIs.

During the training phase of CODEGENAPI, unlike the previous settings that force all files to have the same priority, we design a resampling strategy to enable high-quality python files to appear more frequently and vice versa. The strategy considers the star number of the repository, the unit test function rate of the code file, and the API rate of the code file. More details can be found in Appendix B.

### 4 Benchmark Construction

Private libraries are commonly used in practice, but few attempts have been made to evaluate the performance of generating code invoking private APIs.
To fill this gap, we craft three benchmarks, called TorchDataEval, MonkeyEval, and BeatNumEval. Each programming problem consists of context, target code, and the corresponding test cases.

To create a realistic benchmark for evaluating code generation for private library, we use TorchData, a Python library released just recently. We carefully learnt the official API documentation of TorchData and made sure we were proficient in all APIs. Then, we manually created 50 programming problems based on the API usage examples in the documentation. Two volunteers with extensive experience in Python were invited to check the correctness of each problem. We control the difficulty of the programming problems by the number of APIs in the target code. The percentage of programming problems containing 1 API, 2 APIs, and more APIs is set to 6:3:1.

We also construct two benchmarks using pseudo private libraries, named MonkeyEval and BeatNumEval, each containing 101 programming problems. They are modified from PandasEval and NumpyEval, which were proposed for the public libraries Pandas and Numpy (Zan et al., 2022). In detail, we manually modified all library-related keywords in PandasEval and NumpyEval. For example, as shown in Figure 1, pandas is converted to monkey, dataframe is converted to knowledgeframe, and the API name isin is converted to iscontain. For more details on keyword conversion, please refer to Appendix C. To craft the API documentation for Monkey and BeatNum, we manually paraphrased the descriptions of all the new APIs to ensure that the pre-trained language models have never seen them.

5 Experiments

In this section, we conduct experiments to illustrate the superiority of our proposed framework.

5.1 Experimental Setup

API Information. As shown in the second column (APIs) in Table 1, there are four settings for prompting API information before the context:

- **No API**: there is nothing to be prompted;
- **Perfect**: the information of golden APIs in the target code is prompted;
- **Top-N**: the information of top N APIs retrieved by APIRetriever is prompted, where \( N \in \{1, 2, 3, 5\} \);
- **Human**: the information of the APIs chosen by users is prompted. In our experiments, we invited three volunteers who are programmers familiar with Python but without any background in our three benchmarks. As in Figure 4, they interacted with the APIRetriever and provided their choices for all programming problems. The final APIs are determined by voting on their choices.

Baselines. Our contributions can be reviewed in terms of both APIRetriever and APICoder. For APIRetriever, all models in the No API setting are our baseline, while we propose the Perfect, Top-N, and Human settings. For APICoder, the main baseline is our base model, CODEGEN-MONO 350M (Nijkamp et al., 2022), in the same API information setting. We use CODEGEN for short in the following. In addition, we include advanced pre-trained code generation models that are comparable in parameter size: CodeT5 (Wang et al., 2021), CodeOPT (Lu et al., 2021), CodeClippy\(^5\) and CodeParrot\(^6\). Codex 12B (Chen et al., 2021) is also used to show the performance of giant models.

Evaluation Metrics. Followed by Chen et al. (2021), we regard pass@k as our metric. For each programming problem, we sample \( n \geq k \) code snippets, and then count the number of correct ones \( c \), where passing all test cases is considered as correct. If \( n - c < k \), then pass@k equals 1; otherwise, equals \( 1 - \prod_{i=n-c+1}^{n}(1 - \frac{k}{i}) \). In our experiments, \( k \) is set to one of \([1, 10, 100]\) and \( n \) is set to 200.

Implementation Details. We implement our approach based on PyTorch (Paszke et al., 2019) and Huggingface’s transformers (Wolf et al., 2019). We use a dense retrieval toolkit\(^7\) to train APIRetriever by setting the batch size to 10 per device, the learning rate to 1e-5, the ratio of positive vs. negative samples to 1:8, and the vector dimensions \( z \) of \( p \) and \( a \) to 768. The model uses cross-entropy as the loss function and Adam (Kingma and Ba, 2014) as the parameters optimizer. It is trained for 100K steps on a cluster of 8 NVIDIA V100 GPUs with 32GB memory. The training time is about 3 days.

---

\(^5\)https://github.com/CodedotAl/gpt-code-clippy
\(^6\)https://huggingface.co/lvwerra/codeparrot
\(^7\)https://github.com/luyug/Dense
For pre-training CodeGenAPI, we set the code block size to 1,024, the batch size to 4, the learning rate to 5e-4, the gradient accumulation steps to 4, the weight decay to 0, the gradient accumulation steps to 1, and the warm up steps to 1,000. Noise APIs are added at a rate of 0.05. It is trained for 100K steps about 1.6 days on 32GB NVIDIA V100 GPUs. In all of our training phases, we use mixed precision FP16 to speed up. When generating code snippets using pre-trained models, we conduct various temperatures ranging from 0.1 to 1.0 with the interval of 0.1. All results are reported with the best values across these hyper-parameters.

### 5.2 Main Results

Table 1 summarizes the performance of our framework and all baselines on TorchDataEval, MonkeyEval, and BeatNumEval. Based on numerous experimental results, we derived plausible observations and valuable insights to answer the following research questions.

"Is API information useful for private library oriented code generation?" As we can see in Table 1, all models without prompting any APIs (the No API setting) achieve relatively poor performance on all benchmarks. Especially, Codex 12B, a powerful code generation model with large parameters, can only achieve similar performance to CodeGen and CodeGenAPI 350M in the No API setting. This indicates that even with gigantic models, the task of code generation with private libraries is extremely challenging. Encouragingly, with prompted API information (the Perfect, Top-N, Human settings), both the off-the-shelf models (e.g., CodeParrot, CodeGen, and Codex) and our continually pre-trained CodeGenAPI achieve consistent performance gains compared to those in the No API setting. Moreover, the more powerful the model itself in code generation (i.e., Codex 12B > CodeGen 350M > CodeParrot 110M), the more benefits that API information can bring. For example, on TorchDataEval in the Perfect setting, Codex 12B brings pass@10 an absolute improvement of 36.89%, while CodeParrot 110M only improves 1.77%. This observation also suggests that prompting API information can unleash the potential of gigantic models towards invoking private APIs. All the above results prove the usefulness of API information for code generation for private libraries.

"Is the APIRetriever effective in finding useful
Recall Rate
40%
50%
60%
70%
80%
90%
Top-1 Top-2 Top-3 Top-5 Top-10
PandasEval NumpyEval ... CODE GENAPI can boost the perfor-
mance of generating code snippets using private
library on varying difficulty.

API information? ” All models in the Top- N setting outperform the same models in the No API setting, suggesting that APIRetriever is able to find useful APIs. For a certain model, we observe that the Top-1/Top-2 settings usually perform better than the Top-3/Top-5 settings due to the fact that the latter introduces more noise APIs to the APICoder. In addition, involving humans (the Human setting) in the selection of APIs can further improve performance, suggesting the effectiveness of the human interaction we designed. Note that the Top- N and Human settings are occasionally superior to the Perfect setting, which is reasonable because the noise APIs exist when training the model.

“ Is the APICoder effective in invoking private APIs? ” As shown in Table 1, off-the-shelf models like CODEGEN are capable of handling private library invocations. To seek more extraordinary performance, we continually pre-train CODEGEN and obtain a new model CODEGENAPI. We can observe that CODEGENAPI consistently outperforms its base model CODEGEN on TorchDataEval and MonkeyEval, which proves the effectiveness of CODEGENAPI. However, on BeatNumEval, CODEGENAPI is inferior to CODEGEN. After careful troubleshooting, we reveal that the process of continual pre-training aims to essentially learn how to invoke the correct APIs with maximum like-

5.3 Closer Analysis
We have demonstrated the effectiveness of our framework. In this subsection, we provide sev-

Figure 5: The recall rates of retrieved APIs.

Figure 6: Accuracy of retrieved APIs.

Figure 7: Accuracy of CODEGENAPI and CODEGEN with respect to the number of APIs. The problem is solved if one of 200 samples passes all test cases.

direction.

Quality of Retrieved APIs. Retrieving the correct APIs as prompts can enhance the code generation performance for private libraries, so we would like to evaluate the effectiveness of APIRetriever. Figure 5 shows the recall rate of APIRetriever on five benchmarks. We can see that the recall rates of top-5 are already high, demonstrating that it is reasonable to provide 5 API candidates for users to choose from. Furthermore, as shown in Figure 6, we analyze the accuracy of APIs chosen by users. We observe that it dramatically exceeds the accuracy of top 1, 2 or 3 APIs retrieved by APIRetriever. This suggests that it is feasible to involve humans in the retrieval of APIs.

Different Difficulty. We would like to explore the performance of CODEGENAPI on varying difficulty problems. So we calculate its accuracy across various numbers of APIs in target code y. Each benchmark is divided into 3 parts, according to the number of APIs. Figure 7 shows that CODEGENAPI outperforms CODEGEN by a large margin on the problems containing only one API. The trend still holds as the number of APIs increases. It demonstrates CODEGENAPI can boost the performance of generating code snippets using private library on varying difficulty.
Thanks to the recent development of pre-training techniques, a lot of pre-trained language models have been proposed for code-related tasks. For example, CuBERT (Kanade et al., 2020), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), CodeT5 (Wang et al., 2021), CodeGPT (Lu et al., 2021), PLBART (Ahmad et al., 2021), PyCodeGPT (Zan et al., 2022), CODEGEN (Nijkamp et al., 2022), Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), and InCoder (Fried et al., 2022). Almost all of them focus on standalone code, while JigSaw (Jain et al., 2021) and CERT (Zan et al., 2022) are presented for generating code using public libraries. In this paper, we aim to generate code invoking private APIs, which is a common scenario in practice. It is more challenging because pre-trained language models have never seen any information about private libraries. As for benchmarks, HumanEval (Chen et al., 2021), APPs (Hendrycks et al., 2021), P3 (Schuster et al., 2021), MBPP (Austin et al., 2021), BIG-bench (Srivastava et al., 2022), and CodeContests (Li et al., 2022) were proposed to evaluate the performance of generating standalone code. GSM8K-Python (Cobbe et al., 2021) and MathQA-Python (Austin et al., 2021) were engaged in evaluating the capability of solving mathematical problems. PandasEval and NumpyEval (Zan et al., 2022) were released to evaluate the code generation for public library. We propose

### Table 2: Ablation studies for CODEGENAPI in the Top-2 setting (top 2 APIs provided by APIRetriever are prompted). The default setting of CODEGENAPI is to use the resampling strategy and a noise rate of 5%.

<table>
<thead>
<tr>
<th>APICoder</th>
<th>TorchDataEval</th>
<th>MonkeyEval</th>
<th>BeatNumEval</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEGENAPI</td>
<td>pass@1</td>
<td>pass@10</td>
<td>pass@100</td>
</tr>
<tr>
<td>-w/ noise rate 0%</td>
<td>10.41</td>
<td>23.50</td>
<td>31.98</td>
</tr>
<tr>
<td>-w/ noise rate 10%</td>
<td>9.19</td>
<td>22.87</td>
<td>30.98</td>
</tr>
<tr>
<td>-w/ noise rate 20%</td>
<td>8.92</td>
<td>23.04</td>
<td>30.57</td>
</tr>
<tr>
<td>-w/o resampling</td>
<td>8.65</td>
<td>21.00</td>
<td>29.71</td>
</tr>
</tbody>
</table>

### Table 3: Results of CODEGEN and CODEGENAPI on PandasEval and NumpyEval.

<table>
<thead>
<tr>
<th>APICoder</th>
<th>APIs</th>
<th>PandasEval</th>
<th>NumpyEval</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEGEN</td>
<td>No API</td>
<td>14.24</td>
<td>30.71</td>
</tr>
<tr>
<td>Perfect</td>
<td>11.21</td>
<td>33.59</td>
<td>48.47</td>
</tr>
<tr>
<td>Top-2</td>
<td>9.54</td>
<td>29.02</td>
<td>40.56</td>
</tr>
<tr>
<td>CODEGENAPI</td>
<td>No API</td>
<td>13.58</td>
<td>34.95</td>
</tr>
<tr>
<td>Perfect</td>
<td>19.96</td>
<td>42.36</td>
<td>53.43</td>
</tr>
<tr>
<td>Top-2</td>
<td>11.25</td>
<td>28.61</td>
<td>39.48</td>
</tr>
<tr>
<td>CODEGENAPI</td>
<td>No API</td>
<td>19.31</td>
<td>40.89</td>
</tr>
<tr>
<td>Perfect</td>
<td>21.41</td>
<td>41.08</td>
<td>56.38</td>
</tr>
<tr>
<td>Top-2</td>
<td>18.30</td>
<td>35.12</td>
<td>48.46</td>
</tr>
<tr>
<td>CODEGENAPI</td>
<td>No API</td>
<td>16.55</td>
<td>29.48</td>
</tr>
<tr>
<td>Perfect</td>
<td>24.83</td>
<td>41.47</td>
<td>54.41</td>
</tr>
<tr>
<td>Top-2</td>
<td>12.67</td>
<td>27.32</td>
<td>35.62</td>
</tr>
</tbody>
</table>

**Noise Rate.** A well-chosen noise rate can improve the robustness of CODEGENAPI against a variety of APIs. If we set the noise rate too large, it may change the original distribution of the code corpus, while too small will lose the capability to deal with noise APIs. The default noise rate is 5%, and we also try 0%, 10%, and 20%. As shown in Table 2, both too large and too small noise rates can degrade the performance.

**Resampling Strategy.** Making high-quality python files high-priority, and vice versa, is in line with our intuition. To demonstrate it, we remove the resampling strategy as mentioned in Section 3.3. As shown in Table 2, we observe a steady decline in performance on the three benchmarks. Such an observation demonstrates the effectiveness of the sampling strategy.

**CODEGENAPI for Public Library.** Technically speaking, CODEGENAPI also can be employed for generating code for public libraries. So, we do experiments on PandasEval and NumpyEval and show the results in Table 3. We find that the performance improvement of CODEGENAPI over the base model on public libraries is not as significant as on private libraries. One major reason is that the models have seen the public libraries during pre-training, so prompting API information yields limited benefit. We can see CODEGENAPI excels over CODEGEN when prompting perfect APIs. But when prompting top-2 APIs, the advantages of CODEGENAPI are not exhibited. This means that CODEGENAPI can also work on third-party public libraries, but it depends heavily on the performance of APIRetriever.

### 6 Related Work

#### 6.1 Code Generation

Thanks to the recent development of pre-training techniques, a lot of pre-trained language models have been proposed for code-related tasks.

---

284
three benchmarks, called TorchDataEval, MonkeyEval, and BeatNumEval, aiming to evaluate the performance of code generation for private library.

6.2 Retrieval-Based Generation

In the natural language field, retrieval-based generation is a hot topic. A lot of works (Izacard and Grave, 2020; Karpukhin et al., 2020; Qu et al., 2020; Xiong et al., 2020; Santhanam et al., 2021; Formal et al., 2022) have emerged under this topic. Therefore, we refer to the above methods and design our APIRetriever for private API retrieval. In the programming language field, there are also several attempts to use retrieval techniques, such as DEEPAPI (Gu et al., 2016), REDCODER (Parvez et al., 2021), ReACC (Lu et al., 2022), and DocCoder (Zhou et al., 2022). Our work is fundamentally different from them. They all aim to retrieve public code snippets or other resources on GitHub/StackOverFlow based on the user query, while our goal is to retrieve APIs from the API documentation of private library based on code comments. Besides, we design retrieval because we focus on private APIs, which have not been seen by the pre-trained generative language models.

7 Conclusion

In this paper, we propose a novel framework for code generation for private library. There are two modules: for a specific programming problem, APIRetriever first finds out the useful private APIs from API documentation, and then APICoder leverages these APIs to generate the code. We craft three benchmarks, including TorchDataEval, MonkeyEval, and BeatNumEval, for better evaluating private library oriented code generation. The experimental results and thorough analysis demonstrate the reasonableness and effectiveness of our framework. In future work, we would like to explore how to make better use of API documentation for code generation and improve the approach for real use when programming with private libraries.

Limitations

While our proposed approach exhibits many advantages, it also has a few limitations. (1) As stated in Section 5.2, our approach that prompts APIs for programming problem relies heavily upon the code generation capacity of the language model itself. The more powerful the model itself, the more benefits the prompting APIs bring. Likewise, we also find that if a model itself shows very poor performance, prompting APIs will not bring any benefit to it or even bring negative effects. (2) As the first navigator to explore code generation with private library, we have built three private libraries, but they all include a relatively small number of APIs (<200). With these APIs, our APIRetriever can exhibit decent performance. But we surmise that it may become more challenging for APIRetriever as the number of APIs increases. (3) It is extremely challenging to find a real private library and craft a benchmark like TorchDataEval. To evaluate our idea quickly and cost-effectively, besides TorchDataEval, we also crafted two pseudo private libraries that are modified from the existing public ones as mentioned in Section 4. Although we have done our best to preserve the two pseudo private libraries in line with the real private library, it may still pose some threats to the fair evaluation of code generation for private library. (4) We can see from Table 1 that most models with the Top-N setting fall behind the same model with the Perfect setting. Such observation demonstrates that APIRetriever we designed has a big room for improvement. (5) Our experiments show that our framework can enhance the quality of private library oriented code generation on Python. Limitations may exist when we generalize it to other programming languages such as Java, C, and C++ since the characteristics of libraries for different programming languages are slightly different.

References


Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher


A Collection of API Documentation

We aim to use data from public libraries for training and generalize the models to private libraries. Thus, we crawled the API documentation of the 31 most popular public libraries in Python. Table 4 summarizes the number of APIs we extracted for each library.

B Resampling Strategy

The resampling strategy allows high-quality python files to be more frequently sampled, and vice versa. So the resampling weight \((w)\) of each python file is defined in the following aspects: the star number of the corresponding repository \((N_{\text{star}})\), the unit test function rate \((R_{\text{ut}})\) that is the number of unit test functions divided by the number of all functions, the number of API name \((N_{\text{api}})\) in the file, and the number of APIs \((M_{\text{api}})\) considering one API name may match multiple APIs. Formally, the strategy can be formulated as follows:

\[
\begin{align*}
    w_{\text{star}} &= 1.0 + \log(N_{\text{star}} + 1).\text{clip}(5) \times 0.2, \\
    w_{\text{ut}} &= (0.5 + (1 - R_{\text{ut}})).\text{clip}(1), \\
    w_{\text{api}} &= 5.0 - \log(M_{\text{api}}).\text{clip}(5) \times 0.2, \\
    w &= w_{\text{star}} \times w_{\text{ut}} \times w_{\text{api}},
\end{align*}
\]  

where \(\text{clip}(\frac{y}{x})\) limits the value to \([x, y]\).

C Keywords Conversion from Public Library to Private Library.

As mentioned in Section 4, we convert the public library benchmarks (PandasEval and NumpyEval) to the private library benchmarks (MonkeyEval and BeatNumEval) by manually modifying all public library-related keywords. In Table 5, we list all the keywords before and after the conversion.
**Table 4:** The number of APIs in the 31 public libraries we crawled.

<table>
<thead>
<tr>
<th>Pandas</th>
<th>NumPy</th>
<th>sklearn</th>
<th>PyTorch</th>
<th>TensorFlow</th>
<th>Django</th>
<th>selenium</th>
<th>Matplotlib</th>
<th>Flask</th>
<th>SciPy</th>
<th>Seaborn</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,394</td>
<td>29,452</td>
<td>53,166</td>
<td>32,176</td>
<td>24,975</td>
<td>43,642</td>
<td>35,915</td>
<td>25,367</td>
<td>33,396</td>
<td>161,277</td>
<td></td>
</tr>
<tr>
<td>NLTK</td>
<td>BeautifulSoup</td>
<td>Pygame</td>
<td>PIL</td>
<td>jieba</td>
<td>Gensim</td>
<td>spaCy</td>
<td>transformers</td>
<td>faireseq</td>
<td>SQLAlchemy</td>
<td>Scrapy</td>
</tr>
<tr>
<td>25,810</td>
<td>37,121</td>
<td>26,820</td>
<td>37,131</td>
<td>23,945</td>
<td>45,021</td>
<td>25,721</td>
<td>34,763</td>
<td>5,579</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllenNLP</td>
<td>datasets</td>
<td>tokenizers</td>
<td>MXNet</td>
<td>imageio</td>
<td>pytest</td>
<td>MetPy</td>
<td>ansible</td>
<td>requests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>276,088</td>
<td>1,36,847</td>
<td>-195</td>
<td>1,42,370</td>
<td>1,124</td>
<td>1,242</td>
<td>1,40,839</td>
<td>39,333</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5:** The keywords of converting PandasEval to MonkeyEval, and NumpyEval to BeatNumEval. The grey background means the original keywords, and the white background means the converted ones.

<table>
<thead>
<tr>
<th>PandasEval-MonkeyEval</th>
<th>NumpyEval-BeatNumEval</th>
</tr>
</thead>
<tbody>
<tr>
<td>isnull</td>
<td>mean</td>
</tr>
<tr>
<td>isnull</td>
<td>average</td>
</tr>
<tr>
<td>tolist</td>
<td>apply</td>
</tr>
<tr>
<td>convert_list</td>
<td>employ</td>
</tr>
<tr>
<td>innull</td>
<td>astype</td>
</tr>
<tr>
<td>isnone</td>
<td>totype</td>
</tr>
<tr>
<td>last</td>
<td>shift</td>
</tr>
<tr>
<td>final_item</td>
<td>shifting</td>
</tr>
<tr>
<td>concat</td>
<td>to_dict</td>
</tr>
<tr>
<td>concatenating</td>
<td>convert_dict</td>
</tr>
<tr>
<td>round</td>
<td>format</td>
</tr>
<tr>
<td>value_round</td>
<td>formatting</td>
</tr>
<tr>
<td>drop</td>
<td>Series</td>
</tr>
<tr>
<td>sip</td>
<td>Collections</td>
</tr>
<tr>
<td>reindex</td>
<td>get</td>
</tr>
<tr>
<td>reindexing</td>
<td>getting</td>
</tr>
<tr>
<td>applymap</td>
<td>sort_values</td>
</tr>
<tr>
<td>conduct_map</td>
<td>sort_the_values</td>
</tr>
<tr>
<td>head</td>
<td>series</td>
</tr>
<tr>
<td>header_num</td>
<td>collections</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NumpyEval-BeatNumEval</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>to_numpy</td>
<td>ndarray</td>
</tr>
<tr>
<td>to_beatnum</td>
<td>ndnanset</td>
</tr>
<tr>
<td>column_stack</td>
<td>concatenate</td>
</tr>
<tr>
<td>stack_col</td>
<td>connect</td>
</tr>
<tr>
<td>fill_diagonal</td>
<td>all</td>
</tr>
<tr>
<td>pad_diagonal</td>
<td>total</td>
</tr>
<tr>
<td>add</td>
<td>histogram</td>
</tr>
<tr>
<td>add_concat</td>
<td>last_operation</td>
</tr>
<tr>
<td>astype</td>
<td>arrange</td>
</tr>
<tr>
<td>convert_type</td>
<td>arr_range</td>
</tr>
<tr>
<td>vstack</td>
<td>squeeze</td>
</tr>
<tr>
<td>vertical_stack</td>
<td>szip</td>
</tr>
<tr>
<td>diff</td>
<td>unique</td>
</tr>
<tr>
<td>difference</td>
<td>uniq</td>
</tr>
<tr>
<td>append</td>
<td>any</td>
</tr>
<tr>
<td>apd</td>
<td>any_condition</td>
</tr>
<tr>
<td>array_split</td>
<td>inv</td>
</tr>
<tr>
<td>split_array</td>
<td>inverse</td>
</tr>
</tbody>
</table>