CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation

Daoguang Zan[1,2*] , **Bei Chen**[3] , **Dejian Yang**[3] , **Zeqi Lin**[3] , **Minsu Kim**[4*] ,
Bei Guan[2,5] , **Yongji Wang**[2,5,6] , **Weizhu Chen**[7] , **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]Korea University
[5]Integrative Innovation Center, Institute of Software, Chinese Academy of Sciences
[6]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[7]Microsoft Azure AI
{daoguang@, guanbei@, ywang@itechs.}iscas.ac.cn; minsu@korea.ac.kr
{beichen, deyang, zeqi.lin, wzchen, jlou}@microsoft.com

Abstract

Code generation is a longstanding challenge, aiming to generate a code snippet based on a natural language description. Usually, expensive text-code paired data is essential for training a code generation model. Recently, thanks to the success of pre-training techniques, large language models are trained on large-scale unlabelled code corpora and perform well in code generation. In this paper, we investigate how to leverage an unlabelled code corpus to train a model for library-oriented code generation. Since it is a common practice for programmers to reuse third-party libraries, in which case the text-code paired data are harder to obtain due to the huge number of libraries. We observe that library-oriented code snippets are more likely to share similar code sketches. Hence, we present CERT with two steps: a sketcher generates the sketch, then a generator fills the details in the sketch. Both the sketcher and the generator are continually pre-trained upon a base model using unlabelled data. Furthermore, we craft two benchmarks named PandasEval and NumpyEval to evaluate library-oriented code generation. Experimental results demonstrate the impressive performance of CERT. For example, it surpasses the base model by an absolute 15.67% improvement in terms of pass@1 on PandasEval. Our work is available at https://github.com/microsoft/PyCodeGPT.

Code snippet 1
```python
import pandas as pd
'''Read in the csv using pandas'''
tweets_data = pd.read_csv('./cookbook.csv')
# Count value on the column 'label'
tweets_data["label"].value_counts(normalize=False)
# randomly sample data and get its 'tweet' value
tweets_data["tweet"].sample().values[0]
```

Code snippet n ...
```python
import pandas as pd
# Using pandas to read the file anonyijcai22.csv
df_ijcai = pd.read_csv('./anonyijcai22.csv')
# In the column title,count the number of values
# Note that the argument normalize is set to True
df_ijcai["title"].value_counts(normalize=True)
# Randomly select a row, return 'abstract' value
df_ijcai["abstract"].sample().values[0]
```

The same sketch
```python
import pandas as pd
variable = pd.read_csv(string)
variable[string].value_counts(normalize=bool)
variable[string].sample().values[number]
```

Figure 1: An example in Python: multiple code snippets using Pandas may have the same sketch after anonymizing user-defined terms.

1 Introduction

Code generation, aiming to generate a code snippet for a given natural language description, is a longstanding challenge in the artificial intelligence community. Usually, to train a code generation model with good performance, the massive amount of code snippets paired with natural language descriptions are indispensable [Sun *et al.*, 2019; Lu *et al.*, 2021]. However, it is costly and time-consuming to annotate such a dataset. To alleviate this problem, inspired by GPT-3's powerful zero-shot natural language generation ability [Brown *et al.*, 2021], recent years have witnessed a trend to train large language models using large-scale code corpora (e.g., GitHub), and expect these models to work well directly on code generation tasks, without fine-tuning on expensive text-code pairs. For example, Codex shows that a 12B parameters language model can solve 28.8% of standalone Python programming problems[1].

In this paper, we focus on investigating whether and how

*Work done during the internship at Microsoft Research Asia.

[1]It is measured on HumanEval [Chen *et al.*, 2021a] with pass@1.

language models pre-trained on code corpora (without fine-tuned on pairwise labelled data) can generate library-oriented code snippets rather than standalone ones. During software development, it is a common practice for programmers to reuse third-party libraries (e.g., Pandas and NumPy) to implement needed functionalities. It is not easy for programmers to learn how to use these libraries properly. For example, according to our statistics, more than 40% of StackOverflow questions with "Python" tag also have at least one library tag. Moreover, for library-oriented code generation, the necessity of training the model without pairwise labelled data is raised, as programmers usually need to reuse different libraries in different scenarios, and it is extremely costly to label sufficient text-code pairs that cover most of these libraries.

Compared to standalone code snippets, library-oriented code snippets are more likely to share similar sketches. *Sketch* is the code structure after anonymizing the user-defined terms in the code, such as variable names, method names, constants, etc., which has also been identified as an API usage pattern in previous research litterateurs on software data mining [Zhong *et al.*, 2009; Wang *et al.*, 2013; Niu *et al.*, 2017]. An example is shown in Figure 1. After anonymizing variables and constants, multiple code snippets using the Pandas APIs may have the same (or similar) sketch. Based on this observation, a natural idea to improve library-oriented code generation is to decompose this task into two subtasks: generating the sketch and then filling in the details. Many methods based on this idea have been proposed in different code generation tasks (e.g., Coarse-to-Fine [Dong and Lapata, 2018] and PLOTCODER [Chen *et al.*, 2021b]) and have shown that this idea can effectively improve the quality of generated code snippets. However, these methods are proposed for the fine-tuning process, in which high-quality text-code pairs are required to derive supervision signals for the two-step generation. Therefore, in our scenario that no pairwise labelled data is provided, a research question arises: how to leverage the insight of sketching to enhance the language model pre-training on unlabelled code corpora, thus improving the quality of generated library-oriented code snippets?

To meet the challenge, we propose CERT (for sket**C**her and g**E**ne**R**a**T**or), a continual pre-training approach on sketches for library-oriented code generation. In CERT, a sketcher firstly focuses on predicting a sketch, which omits user-defined details; then, a generator uses the sketch as a prompt to generate the complete code. Both the sketcher and the generator are continually pre-trained based on a base language model for code, using unlabelled code corpora rather than pairwise labelled data. In addition, we craft two evaluation benchmarks for Python libraries, called PandasEval and NumpyEval, each including 101 programming problems using Pandas and NumPy, respectively. We perform extensive experiments on CERT. Results indicate that CERT has superior performance on library-oriented code generation. We further draw several insights via thorough analysis.

2 Task Formulation

Before diving into the details of our proposed approach, we start with a formal description of the task. Code generation is to solve a programming problem: generate *target code* based on *context*. Context contains natural language problem description in the form of code comments, and a code snippet that includes statements such as import, function header and variable definition; target code is a code snippet that solves the programming problem described in the context. Formally, let $\mathbf{x} = (x_1, x_2, \cdots, x_N)$ denote the context, where each x_n can be either a code token or a natural language token. Given $\mathbf{x}$, the code generation model can be formulated as $\mathbf{y} = \mathcal{M}(\mathbf{x})$, where $\mathbf{y} = (y_1, y_2, \cdots, y_M)$ denotes the target code and each y_m is a code token.

```python
import numpy as np
x = np.array([[1], [2], [3]])
# Numpy Vector (N,1) dimension -> (N,) dimension conversion
out = x.reshape(3,)
```

```python
import pandas as pd
def normalize(df):
    # Normalization using pandas
    # We simply subtract the mean and divide by standard deviation
      on df.iloc[:,0,-1] obj with axis is zero.
    # Return the normalized dataframe
    func_ = lambda x: (x-x.mean()) / x.std()
    df.iloc[:,0:-1] = df.iloc[:,0:-1].apply(func_, axis=0)
    return df
```

Figure 2: Two examples of programming problems from the PandasEval and NumpyEval benchmarks. Context is shown with a white background and the target code with a gray background.

For standalone code generation, the programming problem is expected to be solved by a code snippet without using third-party libraries; conversely, for library-oriented code generation, the target code $\mathbf{y}$ contains library API calls. Two examples of library-oriented programming problems can be found in Figure 2. Note that carefully labelled context and target code pairs are indispensable for model fine-tuning, while our proposed approach only requires continual pre-training on unlabelled code corpora.

3 Methodology

In this section, we introduce our base models, followed by the details of our proposed approach CERT.

3.1 Base Models

Codex [Chen *et al.*, 2021a] is a milestone pre-trained model that can generate decent code, but it is not publicly available. Several attempts have been made to reproduce Codex's powerful code generation capability, e.g., CodeClippy[2] and CodeParrot[3], but their performance in Python are not satisfactory. To this end, we present PYCODEGPT, a pre-trained language model, which has the ability to generate pretty good standalone Python code, for example, achieving 8.33% pass@1 on HumanEval [Chen *et al.*, 2021a]. Specially, PYCODEGPT is a 110M parameters model based on GPT-Neo [Black *et al.*, 2021]. We collected 60.6M raw python files with a total size of 330GB. After a series of data pre-processing strategies, such as de-duplicating python files, cleaning and formatting the contents, etc., the final pre-training corpus contains about 13.0M high-quality python files with the size of 96GB. PYCODEGPT is pre-trained for 200K steps and 100B

[2] https://github.com/CodedotAl/gpt-code-clippy

[3] https://huggingface.co/transformersbook/codeparrot

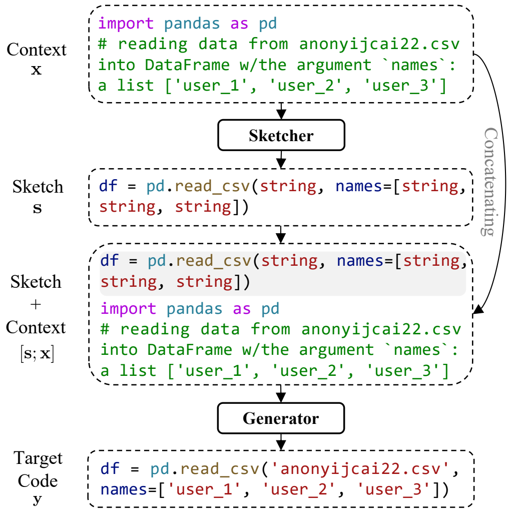

Figure 3: Overview of CERT: a sketcher and a generator.

tokens on a cluster of 16 NVIDIA V100 GPUs with 32GB memory. The pre-training time is about 2 days. We summarize the three key points that make PyCodeGPT powerful: 1) a large amount of carefully cleaned data for pre-training; 2) a newly trained tokenizer, which is specialized in python; and 3) a resampling strategy that prioritizes high-quality data. Besides PyCodeGPT, we also regard CodeGen (Mono 350M) [Nijkamp *et al.*, 2022] as one of our base models, which is by far the best performing publicly available model on HumanEval[4].

3.2 CERT

As mentioned in Section 2, code generation is to generate target code $\mathbf{y}$ based on context $\mathbf{x}$. Since we observe that library-oriented code snippets are more likely to share similar sketches, we present a novel approach CERT and decompose the code generation model $\mathcal{M}$ into two modules: a sketcher $\mathcal{M}_{\mathrm{S}}$ and a generator $\mathcal{M}_{\mathrm{G}}$. Figure 3 shows the overview of CERT with a concrete example in Pandas. Given $\mathbf{x}$ as the input, the sketcher predicts $\mathbf{s}$, which is the sketch of the target code $\mathbf{y}$. The sketcher generates multiple candidate sketches (200 in our experiments) and we choose the one that appears the most. Then, the input of the generator is the concatenation of $\mathbf{s}$ and $\mathbf{x}$. Formally, the process of CERT can be written as $\mathbf{s} = \mathcal{M}_{\mathrm{S}}(\mathbf{x})$ and $\mathbf{y} = \mathcal{M}_{\mathrm{G}}([\mathbf{s}; \mathbf{x}])$. Note that if the sketch $\mathbf{s}$ is already a complete code snippet without anonymous symbols, we directly take it as the final prediction instead of using the generator; and if the sketch $\mathbf{s}$ is an empty sequence, we directly feed $\mathbf{x}$ into the generator.

We build the sketcher and generator on the top of the base model (PyCodeGPT or CodeGen) by continual pre-training. At first, we extract the python files that use a specific library (e.g., Pandas) from the whole pre-training corpus (13.0M files mentioned in Section 3.1), and obtain the sub-corpus denoted by $\mathcal{D}$. Then, we will detail the continual pre-training process of the sketcher and generator for this library.

[4]CodeGen was released during the review period of this paper.

Type/Pre-defined symbol *(user-defined constants)*
int/number; string/string; float/float; boolean/bool; complex/complex
interpolated_raw_string/interrawstring; float_exponent/floatexponant
binary_string/binarystring; interpolated_string/interstring
raw_string/rawstring; unicode_string/unicodestring
Type/Pre-defined symbol *(user-defined names)*
def/func; class/AnClass; variable/variable

Figure 4: The pre-defined symbols in sketcher.

Sketcher. Given the library-oriented sub-corpus $\mathcal{D}$, we perform the sketching operation on each file $\mathbf{d} \in \mathcal{D}$. An example is shown in the upper part of Figure 5. The sketching operation is used to anonymize the user-defined terms in the code file with our pre-defined symbols. The file after sketching is denoted as $\bar{\mathbf{d}}$. We design three different types of sketching operations: 1) only anonymizing the user-defined constants (Default CERT); 2) only anonymizing the user-defined names, including function names, class names, and variable names (CERT-N); and 3) anonymizing both the user-defined constants and names (CERT-NC). For example, in Figure 5, the constant '`user_1`' is anonymized with the pre-defined symbol '`string`'. The details of pre-defined symbols are shown in Figure 4. Then, we continually pre-train the base model on the library-oriented corpus after sketching, and we obtain the sketcher model. The pre-training objective is the same as that of the base model. We pre-train the model for 100K steps on a cluster of 8 NVIDIA V100 GPUs with 32GB memory.

Generator. In order to prepare the pre-trained corpus for the generator, we firstly split the original file $\mathbf{d}$ and the sketching file $\bar{\mathbf{d}}$ into K blocks[5], and obtain $\mathbf{d} = (d_1, d_2, \cdots, d_K)$ and $\bar{\mathbf{d}} = (\bar{d}_1, \bar{d}_2, \cdots, \bar{d}_K)$. Each block is a relatively complete code snippet, such as a function or a class. Note that before splitting, we remove the natural language code comments from the sketching file $\bar{\mathbf{d}}$. Then, the two files are cross-merged to give a merged file $\hat{\mathbf{d}} = (\bar{d}_1, d_1, \bar{d}_2, d_2, \cdots, \bar{d}_K, d_K)$. This is to mimic the process of having a sketch as a prompt for each block. An example is shown in the lower part of Figure 5. Then, the base model is continually pre-trained on all the merged files $\hat{\mathbf{d}}$ and we obtain the generator model. As with the sketcher model, we continually pre-train for 100K steps.

4 Benchmark Construction

Third-party libraries are widely used in reality, while little work has been done to evaluate library-oriented code generation. To meet this challenge, we craft PandasEval and NumpyEval, two benchmarks for library-oriented code generation in Python. Each sample in the benchmarks is a programming problem consisting of context and target code. The programming problems are solved using libraries, where Pandas is for PandasEval, and NumPy is for NumpyEval. The benchmarks are expected to be diverse, authentic, high quality, moderately difficult, and unseen during pre-training.

In order to craft programming problems using libraries, we

[5]We use pip-tools: autopep8, docformatter and redbaron.

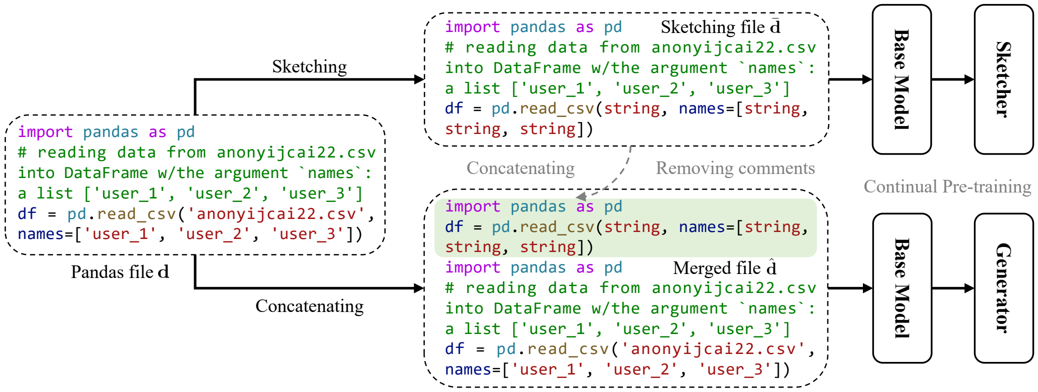

Figure 5: Training data preparation for sketcher and generator with an example in Pandas.

refer to StackOverflow[6], a Q&A website for programmers. There are plenty of real-world programming problems posted by real users, which helps us to improve the authenticity of our data. Specifically, we search for posts using the library tag on StackOverflow, and select those with high votes. To ensure quality, we only refer to posts with accepted answers. We go through a post's question and its accepted answer, then manually organize them into the form needed for our benchmarks, containing both context and target code. We also polish all programming problems so that the problem descriptions are clear and the codes are correct. Note that we keep the intentions and the descriptions of the programming problems consistent with the posts to the maximum extent. Finally, two programmers with more than three years of coding experience in the library are invited to act as code generation models and check the quality of the data.

As a result, we craft 101 programming problems for PandasEval and NumpyEval, respectively. Each programming problem is equipped with test cases for evaluation. For the programming problems in the form of a function, such as the bottom one in Figure 2, we create 20 test cases for each of them. For the others that contain no functions, such as the top one in the Figure 2, we provide 1 test case to check the correctness of predicted variable (e.g., `out` in Figure 2). In total, 64% programming problems in PandasEval and 30% in NumpyEval are equipped with 20 test cases. In addition, we craft programming problems that refer to StackOverflow rather than GitHub, and also carefully organize and polish the problems, so that we can ensure they are unseen by the pre-trained models.

5 Experiments

In this section, we evaluate CERT on PandasEval and NumpyEval to verify its effectiveness.

Evaluation Metrics. We use pass@k as the metrics. When k code samples are generated per problem, pass@k indicates the fraction of correct ones. But computing pass@k in this way may have high variance. Hence, we follow Chen *et al.* [2021a] to generate $n \geq k$ code samples per problem

[6]https://stackoverflow.com

Model	pass@1	Model	pass@1
GPT-Neo 125M	0.75	GPT-Neo 1.3B	4.79
AlphaCode 89M	4.30	CodeParrot 110M	3.80
Codex 42M	5.06	Codex 85M	8.22
Codex 2.5B	21.36	Codex 12B	28.81
PyCodeGPT 110M	8.33	CodeGen-Mono 350M	12.76

Table 1: The pass@1 (%) results on HumanEval benchmark. We omit CodeT5 (220M), CodeGPT-Adapted (124M), and CodeClippy (125M) as their pass@1 $= 0$.

($n = 200$ in our experiments) and count the number of correct samples c. If $n - c < k$, then pass@$k = 1$; otherwise, pass@$k = 1 - \prod_{i=n-c+1}^{n}(1 - k/i)$. Note that a predicted code is correct if it can pass all the test cases.

Implementation Details. We implement our approach using PyTorch [Paszke *et al.*, 2019], HuggingFace's transformers library [Wolf *et al.*, 2019], and DeepSpeed[7]. In the training phase of PyCodeGPT, we set the batch size to 10, the window size to 1024, the learning rate to 5e-4, the gradient accumulation steps to 4 and the weight decay to 0.1. The settings of sketcher and generator are the same as PyCodeGPT. We use the mixed-precision of FP16 to accelerate the pre-training. In inference phase, we set the temperature to one of $[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$. The best performance is reported across the above hyper-parameters.

5.1 Main Results

Before evaluating CERT, we would like to evaluate our base model PyCodeGPT on HumanEval [Chen *et al.*, 2021a] compared to several advanced pre-trained models. As shown in Table 1, PyCodeGPT (110M) achieves competitive 8.33% pass@1. It largely exceeds other models with comparable parameters, e.g., AlphaCode (89M) [Li *et al.*, 2022], CodeClippy (125M), and CodeParrot (110M), and also is better than the larger model GPT-Neo (1.3B).

Then, our proposed CERT is evaluated on PandasEval and NumpyEval. We train CERT on two base models, including PyCodeGPT and CodeGen, named PyCodeGPT-CERT and CodeGen-CERT, respectively. For each benchmark, we extract corresponding library-oriented files to train

[7]https://github.com/microsoft/DeepSpeed

Benchmark	Model	pass@1	pass@10	pass@100
Pandas Eval	CodeT5 (220M)	0.00	0.00	0.00
	CodeGPT-Adapted (124M)	0.62	2.65	4.95
	CodeClippy (125M)	0.14	0.92	1.92
	CodeParrot (110M)	3.21	13.62	33.27
	CodeGen (350M)	14.24	30.71	46.04
	CodeGen-XL	21.07	37.67	49.07
	CodeGen-CERT	**26.40**▲12.16	**46.49**▲15.78	**58.16**▲12.12
	PyCodeGPT (110M)	12.75	37.80	59.65
	PyCodeGPT-XL	19.80	46.80	60.04
	PyCodeGPT-CERT	**28.42**▲15.67	**48.04**▲10.24	**60.96**▲1.31
	- PyCodeGPT-CERT-N	23.66	41.73	55.08
	- PyCodeGPT-CERT-NC	19.07	41.50	54.82
	- PyCodeGPT-CERTg	20.58	42.61	56.00
Numpy Eval	CodeT5 (220M)	0.00	0.10	0.74
	CodeGPT-Adapted (124M)	1.59	4.17	8.54
	CodeClippy (125M)	0.08	0.59	1.24
	CodeParrot (110M)	8.42	21.46	45.94
	CodeGen (350M)	19.31	40.89	60.58
	CodeGen-XL	27.33	44.75	63.39
	CodeGen-CERT	**32.00**▲12.69	**49.45**▲8.56	**67.82**▲7.24
	PyCodeGPT (110M)	18.04	38.13	63.37
	PyCodeGPT-XL	20.50	43.40	56.06
	PyCodeGPT-CERT	**31.47**▲13.43	**46.42**▲8.29	**66.41**▲3.04
	- PyCodeGPT-CERT-N	24.91	42.88	54.02
	- PyCodeGPT-CERT-NC	19.88	41.64	55.82
	- PyCodeGPT-CERTg	16.55	44.07	56.80

Table 2: The pass@k (%) results on PandasEval and NumpyEval. The absolute improvements of CERT over the base model are highlighted in red. Also, we report the performance of different sketching operations (CERT-N and CERT-NC) and the performance of CERTg trained for general code generation.

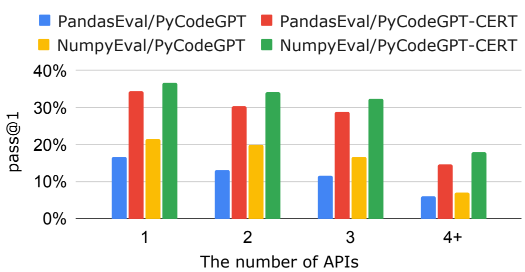

Figure 6: The pass@1 result with respect to the number of APIs.

CERT. The file numbers are about 0.61M for Pandas and 2.62M for NumPy. Baselines include our base models PyCodeGPT and CodeGen; PyCodeGPT-XL and CodeGen-XL, which are continual pre-trained PyCodeGPT and CodeGen on the extracted library-oriented files; and advanced pre-trained models for code, like CodeT5 [Wang *et al.*, 2021], CodeGPT [Lu *et al.*, 2021], CodeClippy and CodeParrot. Table 2 summarizes the performance. CERT consistently outperforms all the baselines by a large margin. The absolute improvements over PyCodeGPT and CodeGen are shown in red, which are significant, for example, 12.69% pass@1 for CodeGen-CERT and 13.43% pass@1 for PyCodeGPT-CERT on NumpyEval. The results demonstrate the effectiveness of CERT with the idea of leveraging sketches for library-oriented code generation.

Additionally, we would like to investigate the performance of CERT with respect to the number of API calls involved in the target code. We divided the programming problems in each benchmark into four parts based on the number of APIs. As shown in Figure 6, compared to PyCodeGPT, PyCodeGPT-CERT has a steady improvement on each part. It indicates that CERT can improve the performance of library-oriented code generation of varying difficulties.

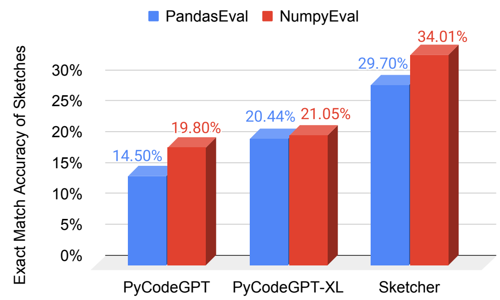

Figure 7: The exact match accuracy of sketches. The sketcher refers to the one in PyCodeGPT-CERT.

Model	pass@1	pass@10	pass@100
PyCodeGPT	**8.33**	13.36	19.13
PyCodeGPT-CERTg	8.25	**14.12**	**20.41**

Table 3: The pass@k (%) results of PyCodeGPT and PyCodeGPT-CERTg on HumanEval.

5.2 Closer Analysis

We conduct some closer analyses to provide more insights.

Different Types of Sketching. As mentioned in Section 3.2, we propose three types of sketching operations. By default, CERT only anonymizes user-defined constants. The other two types include CERT-N, which anonymizes only user-defined names, and CERT-NC, which anonymizes both user-defined constants and names. As shown in Table 2, CERT with default setting achieves the best performance. This observation may be related to the inherent characteristics of Pandas and NumPy. They are commonly used in data statistics and analysis, often involving manipulation of the data constants. Thus, it is necessary to anonymize user-defined constants. Anonymizing both user-defined constants and names would probably make the sketches too abstract.

Quality of Generated Sketches. Intuitively, it is easier to generate a sketch than a complete code. Thus, we would like to evaluate the quality of sketches generated by the sketcher of CERT. We use exact match accuracy as the metric and include PyCodeGPT and PyCodeGPT-XL for comparison. For PyCodeGPT and PyCodeGPT-XL, we anonymize the user-defined constants in the predicted code to obtain the sketch. As shown in Figure 7, our sketcher surpasses baselines by 15.20% and 14.21% on PandasEval and NumpyEval, respectively. It indicates that the sketcher can generate high-quality sketches, and such sketches further benefit the generator. Additionally, the generator does not necessarily require an exactly correct sketch, as the sketch is just a prompt (A case will be discussed in Section 5.3).

CERT for General Code Generation. Technically speaking, CERT can also be used for general code generation tasks, not just the library-oriented ones. Concretely, following the procedure in Figure 5, we can continually pre-train PyCodeGPT using the whole 13.0M python corpus instead of the extracted library-oriented files, and obtain the model we called CERTg. We evaluate CERTg for general code generation on HumanEval compared to the base

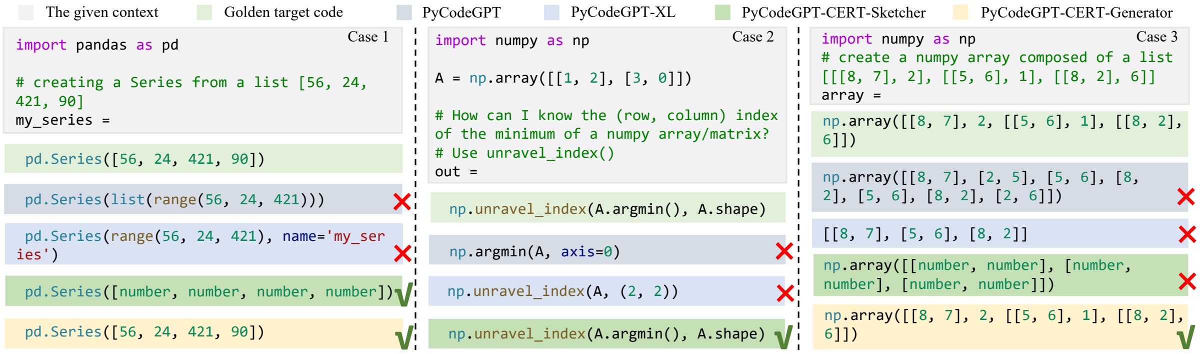

Figure 8: Three library-oriented code generation cases.

Benchmark	Model	Size	pass@1	pass@10	pass@100
Pandas Eval	PyCodeGPT-CERT	110M	**28.42**	**48.04**	60.96
	CodeGen-CERT	350M	26.40	46.49	58.16
	GPT-3	175B	12.97	20.54	25.43
	Codex	12B	18.88	43.05	**64.37**
Numpy Eval	PyCodeGPT-CERT	110M	31.47	46.42	66.41
	CodeGen-CERT	350M	32.00	49.45	67.82
	GPT-3	175B	16.25	22.15	27.38
	Codex	12B	**34.42**	**55.75**	**71.74**

Table 4: GPT-3 and Codex on PandasEval and NumpyEval.

model PyCodeGPT. As shown in Table 3, they have similar pass@k results. This observation verifies our assumption that library-oriented code snippets are more likely to share similar sketches, so it is beneficial to use sketches as prompts in this situation. But in the general case, it is not useful. Meanwhile the results of CERTg on PandasEval and NumpyEval are in Table 2. CERTg is inferior to CERT, suggesting that extracting library-oriented files is essential for CERT to learn the knowledge of library-oriented sketches.

Evaluation of GPT-3 and Codex. We evaluate GPT-3 and Codex to see how these extremely large models perform on PandasEval and NumpyEval. As shown in Table 4, CERT is competitive with only 110M parameters. Such observation proves CERT's powerful code generation capability in library-oriented programming problems.

5.3 Case Study

For a more comprehensive comparison, we show three cases in Figure 8. We show in turn the context, the golden target code, the predicted code of PyCodeGPT and PyCodeGPT-XL, the sketch generated by PyCodeGPT-CERT and the predicted code of PyCodeGPT-CERT. Case 1 is from PandasEval, both PyCodeGPT-CERT's sketcher and generator reach the correct results, while the baselines do not. It reveals that sketcher and generator can work well together. Case 2 is from NumpyEval, the sketcher predicts the correct sketch, which has no anonymous symbols, then this sketch is the final predicted code. It indicates that the sketcher has the ability to predict code without user-defined constants. At last, in Case 3, the sketcher makes a wrong prediction `pd.Series([[number*2]*3])`, while the correct sketch is `pd.Series([[number*2], number, [[number*2], number]*2])`. But PyCodeGPT-CERT's generator rectifies it and finally generates the correct code. Since the sketch acts only as a prompt, it is not necessarily to be perfectly correct, which endows the generator with solid robustness.

6 Related Work

The most related work is the line of large pre-trained models for code. As for the encoder-style pre-trained models, they cannot be employed directly to generate code, such as CuBERT [Kanade *et al.*, 2020], CodeBERT [Feng *et al.*, 2020], and GraphCodeBERT [Guo *et al.*, 2020]. As for the decoder-style or encoder-decoder-style ones, they are trained on large unlabelled code corpora and can work directly on code generation task, such as CodeT5 [Wang *et al.*, 2021], CodeGPT [Lu *et al.*, 2021], PLBART [Ahmad *et al.*, 2021], PolyCoder [Xu *et al.*, 2022], CodeGen [Nijkamp *et al.*, 2022], AlphaCode [Li *et al.*, 2022], and Codex [Chen *et al.*, 2021a]. All of them focus on generating standalone code, while we investigate library-oriented code generation. Also, similar to our idea, there are several works leveraging code sketches, for example, Coarse-to-Fine [Dong and Lapata, 2018], Bayou [Murali *et al.*, 2018], SketchAdapt [Nye *et al.*, 2019], and PlotCoder [Chen *et al.*, 2021b]. However, they require labelled text-code paired data for fine-tuning, while our models continually pre-train on unlabelled code corpora. For code generation benchmarks, there are few works, including APPS [Hendrycks *et al.*, 2021], HumanEval [Chen *et al.*, 2021a], and PlotCoder's dataset [Chen *et al.*, 2021b]. The former two ones focus on evaluating the capability of generating standalone code, and the last one is primarily devoted to generating plotting APIs and visualization code. PandasEval and NumpyEval are dedicated to evaluating the performance of library-oriented code generation.

7 Conclusion

In this paper, we propose a novel approach CERT for library-oriented code generation. It leverages the code sketches and consists of a sketcher and a generator. The sketcher and generator are continually pre-trained upon a base model using unlabelled code corpora. Also, we carefully craft two benchmarks to evaluate library-oriented code generation, namely PandasEval and NumpyEval. Experimental results and thorough analysis show the effectiveness of CERT. In future work, we are interested in code generation for private libraries with fewer data.

References

[Ahmad *et al.*, 2021] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *North American Chapter of the Association for Computational Linguistics*, pages 2655–2668, 2021.

[Black *et al.*, 2021] Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. In *Zenodo*, March 2021.

[Brown *et al.*, 2021] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Neural Information Processing Systems*, 2021.

[Chen *et al.*, 2021a] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[Chen *et al.*, 2021b] Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Association for Computational Linguistics*, pages 2169–2181, 2021.

[Dong and Lapata, 2018] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Association for Computational Linguistics*, pages 731–742, 2018.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Conference on Empirical Methods in Natural Language Processing*, pages 1536–1547, 2020.

[Guo *et al.*, 2020] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, et al. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2020.

[Hendrycks *et al.*, 2021] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, et al. Measuring coding challenge competence with apps. In *Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

[Kanade *et al.*, 2020] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121, 2020.

[Li *et al.*, 2022] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[Lu *et al.*, 2021] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[Murali *et al.*, 2018] Vijayaraghavan Murali, Letao Qi, et al. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[Nijkamp *et al.*, 2022] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, et al. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[Niu *et al.*, 2017] Haoran Niu, Iman Keivanloo, and Ying Zou. API usage pattern recommendation for software development. *Journal of Systems and Software*, 129:127–139, 2017.

[Nye *et al.*, 2019] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870, 2019.

[Paszke *et al.*, 2019] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. PyTorch: An imperative style, high-performance deep learning library. In *Neural Information Processing Systems*. 2019.

[Sun *et al.*, 2019] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural cnn decoder for code generation. In *Association for the Advancement of Artificial Intelligence*, number 01, pages 7055–7062, 2019.

[Wang *et al.*, 2013] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Mining Software Repositories*, pages 319–328, 2013.

[Wang *et al.*, 2021] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[Wolf *et al.*, 2019] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[Xu *et al.*, 2022] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Deep Learning for Code Workshop*, 2022.

[Zhong *et al.*, 2009] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.