SKCODER: A Sketch-based Approach for Automatic Code Generation

Jia Allen Li
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lijia@stu.pku.edu.cn

Yongmin Li
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
liyongmin@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Zhi Jin*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

Yiyang Hao
aiXcoder
Beijing, China
haoyiyang@aixcoder.com

Xing Hu
Zhejiang University
Ningbo, China
xinghu@zju.edu.cn

***Abstract*—Recently, deep learning techniques have shown great success in automatic code generation. Inspired by the code reuse, some researchers propose copy-based approaches that can copy the content from similar code snippets to obtain better performance. Practically, human developers recognize the content in the similar code that is relevant to their needs, which can be viewed as a *code sketch*. The sketch is further edited to the desired code. However, existing copy-based approaches ignore the code sketches and tend to repeat the similar code without necessary modifications, which leads to generating wrong results.**

In this paper, we propose a sketch-based code generation approach named SKCODER to mimic developers' code reuse behavior. Given a natural language requirement, SKCODER retrieves a similar code snippet, extracts relevant parts as a code sketch, and edits the sketch into the desired code. Our motivations are that the extracted sketch provides a well-formed pattern for telling models "how to write". The post-editing further adds requirement-specific details into the sketch and outputs the complete code. We conduct experiments on two public datasets and a new dataset collected by this work. We compare our approach to 20 baselines using 5 widely used metrics. Experimental results show that (1) SKCODER can generate more correct programs, and outperforms the state-of-the-art – CodeT5-base by 30.30%, 35.39%, and 29.62% on three datasets. (2) Our approach is effective to multiple code generation models and improves them by up to 120.1% in Pass@1. (3) We investigate three plausible code sketches and discuss the importance of sketches. (4) We manually evaluate the generated code and prove the superiority of our SKCODER in three aspects.

***Index Terms*—Code Generation, Deep Learning**

I. INTRODUCTION

As the complexity and scale of the software continue to grow, developers cost lots of effort to write the source code by hand. Code generation aims to automate this coding process and generate the source code that satisfies a given natural language requirement. Nowadays, deep learning (DL) techniques have been successfully applied to automatic code generation [1], [2], [3]. DL-based models take a natural language (NL) description as the input and output the corresponding source code. The models are trained with a corpus of real NL-code pairs. During the inference, trained models can automatically generate the desired code for a new NL description.

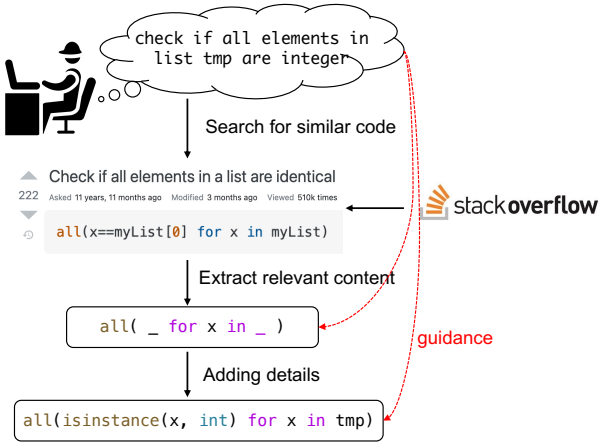

Fig. 1. The process of reusing the similar code by developers.

Recently, inspired by the code reuse [4], some researchers [5], [6], [7] introduce the information retrieval techniques into code generation. They retrieve the similar code and provide it as a supplement to code generation models. The models are trained to copy some content from the similar code and obtain a better performance. In this paper, we refer to these studies as *copy-based* code generation models.

Practically, human developers often make necessary modifications in the similar code instead of simply copying, during the code reuse process [8]. As shown in Figure 1, developers search for a similar code snippet in open-source communities (*e.g.,* Stack Overflow [9]) and further analyze the relevance of similar code to their requirements. Then, developers recognize the parts (*i.e.,* `all( _ for x in _ )`) that are relevant to their needs and ignore the irrelevant parts

* Corresponding authors

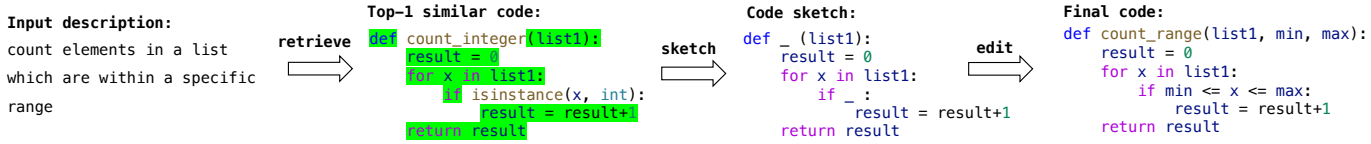

Fig. 2. The illustration of how developers reuse the similar code. The relevant content in the similar code is highlighted.

(*i.e.,* `x==myList[0]` and `myList`). The relevant content can be viewed as a *code sketch*, which specifies a viable code pattern (*e.g.,* API usage patterns [10], [11]) to guide developers on how to write their code. Next, developers understand the current requirement (*i.e.,* check integer) and edit the sketch into the desired code by adding some details (*i.e.,* `isinstance(x,int)`). In the above pipeline, code sketches play a key role in the code reuse. The sketches denote the knowledge that developers extract from the similar code, and are further reused in the newly-written code. However, previous copy-based models [5], [7] ignore the importance of sketches. Experimental results show that copy-based models tend to repeat the similar code without necessary modifications and even copy the irrelevant content.

To mimic the above developers' code reuse behavior, we propose a novel *sketch-based* code generation approach, named SKCODER. Different from simply copying in previous copy-based approaches, SKCODER can identify the content in similar code that is relevant to current requirements and further modify those relevant content. Our motivations are that code sketches denote the guidance from the similar code that tells models "how to write", and NL descriptions express requirements that tell models "what to write". Specifically, SKCODER generates the source code in three steps:

- **Retrieve**. Given an NL description, we use a *retriever* to choose a similar code snippet from a retrieval corpus.
- **Sketch**. Based on the NL description, we use a *sketcher* to extract a code sketch from the similar code.
- **Edit**. We employ an *editor* to edit the sketch based on the NL description and obtain the target code.

We conduct extensive experiments to evaluate our SKCODER. (1) We evaluate SKCODER on two public datasets [12], including HearthStone and Magic. We employ three widely used evaluation metrics (exact match (EM), BLEU [13], and CodeBLEU [14]). Results demonstrate the impressive performance of our SKCODER. In terms of the EM, SKCODER outperforms state-of-the-art (SOTA) baselines by up to 22.41% and SOTA copy-based baselines by up to 42.86%. (2) We collect a new code generation dataset named AixBench-L that consists of 200k real NL-code pairs. Each test sample is equipped with a set of unit tests. We use Pass@1 and AvgPassRatio to verify the correctness of the generated code. Results show that SKCODER outperforms SOTA baselines 12.9% in Pass@1 and 8.49% in AvgPassRatio. (3) We conduct an ablation study of our approach on multiple code generation models by gradually adding the retriever and sketcher to these models. Results prove the contributions of different modules and our SKCODER can substantially improve different models by up to 120.1% in Pass@1. (4) We investigate three plausible design choices for code sketches. Results demonstrate the importance of the sketch and our used sketch has a better performance. We also discuss the importance of code sketches through real examples. (5) We conduct a human evaluation to evaluate the generated code in three aspects, including correctness, code quality, and maintainability. Results show that SKCODER outperforms baselines in all three aspects.

We summarize our contributions in this paper as follows.

- To mimic developers' code reuse behavior, we propose a sketch-based code generation approach named SKCODER. It extracts a code sketch from the retrieved similar code and further edits the sketch into the target code based on the input description.
- We collect a new code generation dataset named AixBench-L that consists of 200k real NL-code pairs. Each test sample is equipped with a set of unit tests to evaluate the correctness of functions.
- We conduct extensive experiments on three datasets. Qualitative and quantitative analysis shows the effectiveness of our SKCODER. We further investigate different design choices of code sketches and discuss the importance of code sketches.

Data Availability. We open source our replication package [15], including the datasets and the source code of SKCODER, to facilitate other researchers and practitioners to repeat our work and verify their studies.

II. MOTIVATING EXAMPLES

In Figure 2, we show an example to analyze how developers reuse the similar code and explain our motivations.

(1) For an input requirement, the retrieved similar code contains the relevant content and irrelevant parts. Given an NL description, developers first retrieve a similar code snippet. Figure 2 shows the Top-1 similar code snippet that is retrieved based on the similarity of NL descriptions. Then, developers understand the implementation details of the similar code and determine which parts are relevant to their requirements. We can see that the similar code contains lots of relevant content (*i.e.,* highlight in Figure 2), *e.g.,* parameters (`list1`), control flow statements (`for x in list1:`), and data flow statements (`result=result+1`). Meanwhile, the similar code also contains irrelevant parts, such as the if condition statement (`if instance(x,int):`).

Thus, simply copying from the similar code is inappropriate, which probably causes the generated code contains some irrel-

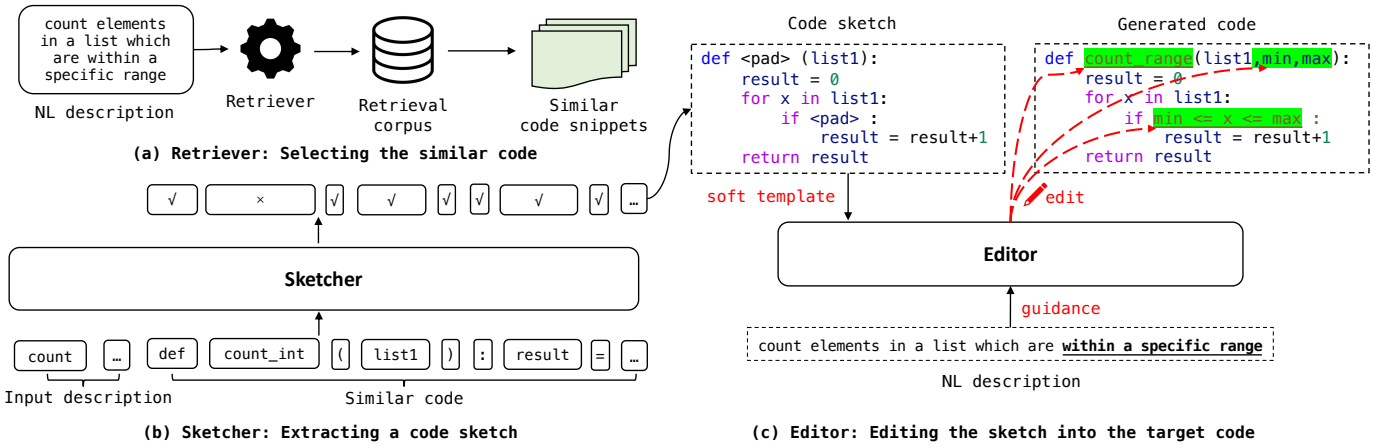

Fig. 3. The overview of our approach.

evant parts. We show a wrong output of the SOTA copy-based approach named REDCODER [7] in Figure 6. REDCODER directly copies an incorrect statement from the similar code without necessary modifications.

(2) We should extract the relevant content from the similar code as a code sketch. Practically, developers will recognize the relevant content from the similar code, ignoring irrelevant parts. The relevant content can be viewed as a code sketch, which specifies a code pattern to guide developers on how to write the source code. Figure 2 shows a sketch extracted from the similar code. The token "_" is a placeholder. We can see that the sketch provides a high-level code structure for developers, *i.e.,* initializing a counting variable → iterating the list and counting → returning the counting variable. Some details are replaced by placeholders and elaborated by developers.

Thus, we argue that code sketches are the core of a code reuse process, which denote the valuable knowledge from the similar code and are further reused in the new code.

(3) The sketch needs to be edited based on the input description to obtain the target code. Code sketches provide code patterns that tell developers "how to write", and the NL descriptions express requirements that tell developers "what to write". Thus, developers will edit sketches based on their requirements and obtain the final code. Figure 2 shows the final code. Developers understand requirements (*i.e.,* counting elements within a specific range) from the input description and fill in sketches with implementation details, *e.g.,* function name (`count_range`), if condition statements (`if min<=x<=max:`).

Based on the above observations, we propose a sketch-based code generation approach to mimic the developers' code reuse behavior. Different from previous copy-based code generation models, our approach contains a sketcher module that can extract the relevant content from the similar code and output a code sketch. Then, we utilize an editor module to edit the sketch into the target code. Through the above pipeline, our approach effectively mines the knowledge from existing high-quality code corpus and transfers the knowledge into newly-written programs.

III. Approach

In this section, we present a sketch-based code generation approach, named SkCoder. We formally define the overview of our SkCoder and describe the details in the following sections, including three modules and the training details.

A. Overview

The goal of code generation is to train a model $G(Y|X)$ that predicts a code snippet Y based on an input natural language (NL) description X. In this work, we decompose this model into three modules, including a retriever, a sketcher, and an editor. The three modules work in a pipeline as shown in Figure 3:

- **Retrieve**. Given an NL description X, a retriever selects a similar code snippet Y' from a retrieval corpus.
- **Sketch**. Based on the NL description X, a sketcher extracts a code sketch S from the similar code Y'.
- **Edit**. An editor edits the sketch S into the target code Y based on the NL description X.

B. Retriever

As shown in Figure 3 (a), the retriever aims to select similar code snippets from a retrieval corpus based on the input NL description. Inspired by previous studies [5], [6], we think that similar code snippets are likely to have similar NL descriptions. Therefore, we take the input description as a query to search for similar descriptions in a retrieval corpus. Then, the corresponding code of similar descriptions is viewed as the similar code.

Specifically, we employ the BM25 score [16] as our retrieval metric, which is widely used in previous studies [17], [18], [19]. BM25 is a bag-of-words retrieval function to estimate the lexical-level similarity of two sentences. The more similar two sentences are, the higher the value of BM25 scores. We leverage the open-source search engine Lucene [20] to build our retriever and use the training set as our retrieval corpus.

Our motivation is that BM25 and Lucene can bring a nice retrieval accuracy and have low complexity. Considering that the retrieval corpus is often large-scale, a fast retriever is closer to practical applications. We also notice that there are some more advanced code search approaches [21], [22], and they can be applied to our approach in a plug-and-play fashion. Because these approaches have higher complexity, we leave them for further work.

C. Sketcher

The goal of our sketcher is to extract a code sketch from the similar code based on the input description. In other words, the sketcher should extract the content that is relevant to the input description and ignore irrelevant parts. We consider this procedure as a series of token-level classification actions. We first split the similar code into a token sequence. Then, we utilize a neural network to capture relations between the input description and the similar code tokens. For more relevant tokens, the neural network assigns higher weights. Based on the outputs of the neural network, we further decide whether each token in the similar code is extracted or ignored. The ignored tokens are replaced by placeholders. Figure 3 (b) shows the workflow of our sketcher.

Specifically, we concatenate the NL description X and the similar code Y' into an input sequence and tokenize it. Then, we use a neural encoder $\mathrm{Encoder}(\cdot)$ to convert the input sequence into vector representations $[H; H']$.

$$\begin{aligned} X &= (x_1, x_2, \ldots, x_n) \\ Y' &= (y'_1, y'_2, \ldots, y'_m) \\ [H; H'] &= \mathrm{Encoder}([X; Y']) \end{aligned} \tag{1}$$

where x_i and y'_i are the i-th token in the NL description and the similar code; n and m are the maximum lengths of the NL description and the similar code.

We further extract vector representations of the similar code and feed them into a linear classification layer. The classification layer will output a probability p_i for each token in the similar code. If the probability is greater than a threshold t, the token is extracted; otherwise, it is replaced with a placeholder (`<pad>`).

$$\begin{aligned} H' &= (h'_1, h'_2, \ldots, h'_m) \\ p_i &= \mathrm{softmax}(W_s h'_i + b_s) \end{aligned} \tag{2}$$

$$s_i = \begin{cases} y'_i & \text{if} \quad p_i > t \\ \texttt{<pad>} & \text{otherwise} \end{cases} \tag{3}$$

$$S = (s_1, s_2, \ldots, s_m) \tag{4}$$

where h'_i denotes the vector representation of i-th token in the similar code. W_s and b_s are trainable parameters in the classification layer. S is the predicted sketch and s_i is the i-th token in the sketch. We further merge consecutive placeholders in the sketch into one placeholder.

Input NL description:
```
print script 's directory
```
Target code:
```
print(os.path.dirname(os.path.realpath(__file__)))
```
Retrieved similar code:
```
return os.path.dirname(os.path.realpath(sys.argv[0]))
```
Code Sketch (LCS):
```
<pad> os.path.dirname(os.path.realpath( <pad> ))
```

Fig. 4. An illustration of our sketch.

D. Editor

As shown in Figure 3 (c), our editor treats the sketch as a soft template and generates the target code with the guidance of the input description. The editor is trained to follow code structures provided by the sketch and add details to some placeholders (*e.g.,* `count_range`, `min<=x<=max`). The editor also can generate some necessary components that are not in the sketch, *e.g.,* additional parameters (`min, max`).

In this paper, we employ an encoder-decoder neural network to implement our editor, which has been widely used in code generation [1], [2], [7], [3]. Specifically, we concatenate the NL description and the sketch into an input sequence. The input sequence is transformed into vector representations by an encoder, and a decoder generates the target code based on vector representations.

E. Training and Testing

Our SkCoder contains three modules: retriever, sketcher, and editor. We employ a deterministic retriever that does not contain trainable parameters. Besides, considering that the sketcher performs non-differentiable hard classifications, the overall approach cannot be trained in an end-to-end fashion. Thus, we employ a two-stage training strategy (*i.e.,* firstly training the sketcher and then training the editor), which is widely used in other fields like code completion [19] and code summarization [18].

1) Training the sketcher: The sketcher takes an NL description X and a similar code snippet Y' as inputs and outputs a code sketch S. But existing code generation datasets only contain NL-code pairs (X, Y) without explicit sketches. Thus, we propose an approach to construct sketches for facilitating the training. We first pick a dataset and use our retriever to make lots of triples (X, Y, Y'). Then, we treat the longest common subsequence (LCS) [23] between the similar code Y' and the target code Y as a code sketch S. Figure 4 shows an illustration of our sketch. We can see that the LCS effectively keeps reusable parts in the similar code (*e.g.,* API and code structures). In Section V, we experimentally investigate other design choices of sketches and prove the superiority of our used sketch.

Based on the above setting, we can build lots of training triples (X, Y', S). Then, we train our sketcher by minimizing the following loss function:

$$\mathcal{L}_s = -\sum_{i=1}^{m} [\mathbb{I}_i \cdot \log(p_i) + (1 - \mathbb{I}_i) \cdot \log(1 - p_i)] \tag{5}$$

TABLE I
STATISTICS OF THE DATASETS IN OUR EXPERIMENTS.

Statistics	Hearthstone	Magic	AixBench-L
# Train	533	11,969	190,000
# Dev	66	664	10,000
# Test	66	664	175
Avg. tokens in description	27.92	59.54	27.55
Max. tokens in description	44	174	3752
Avg. tokens in code	87.14	302.44	170.74
Max. tokens in code	407	2395	25237

where p_i is a predicted probability that i-th token of similar code y'_i is kept in the sketch S. $\mathbb{I}_i$ is an indicator function that outputs 1 when y'_i is in S and outputs 0 when y'_i is not in S.

2) Training the editor: The inputs of our editor contain an NL description X and a code sketch S, and the output is the target code Y. We utilize a retriever to make triples (X, Y, Y') and further use a trained sketcher to predict the code sketches, obtaining lots of training triples (X, S, Y). We train our editor by minimizing the following loss function:

$$\mathcal{L}_e = -\sum_{i=1}^{m} \log P(y_i|X, S, y_{<i}) \tag{6}$$

where y_i denotes i-th token in the target code and $y_{<i}$ is the part of the target code before y_i.

3) Testing: After training the sketcher and editor, our SKCODER can be applied to online inference. Given a new NL description, we use a retriever to search for a similar code snippet from a retrieval corpus. Then, our sketcher extracts a code sketch from the similar code and our editor generates the desired code snippet based on the sketch.

IV. STUDY DESIGN

To assess the effectiveness of our approach, we perform a large-scale study to answer three research questions. In this section, we describe the details of our study, including datasets, metrics, and baselines.

A. Research Questions

Our study aims to answer three research questions (RQ). In RQ1, we compare our SKCODER to SOTA code generation models on three representative datasets. In RQ2, we conduct an ablation study to prove the contributions of different modules. In RQ3, we investigate different design choices of code sketches and validate the effectiveness of our design.

RQ1: How does SKCODER perform compared to SOTA baselines? We train our SKCODER with three representative datasets. Then, we use multiple metrics to evaluate the SKCODER and compare it to existing SOTA code generation baselines.

RQ2: What are the contributions of different modules in our approach? Our SKCODER consists of three modules: a retriever, a sketcher, and an editor. We assess the contributions of different modules by gradually adding them to a base model. We select multiple neural networks as the base models and aim to verify that our approach is effective to different network architectures.

RQ3: What is the better design choice of the sketcher? In Section III-E, we treat the longest common subsequence (LCS) as the code sketch. In this RQ, we provide other design choices of the sketch and compare them to our design.

B. Datasets

We conduct experiments on two public datasets (*i.e.,* HearthStone in Python and Magic in Java) collected by Ling et al. [12] and a new Java dataset named AixBench-L collected by this work.

HearthStone and Magic datasets are proposed for the automatic code generation for cards in games. Each sample is composed of a semi-structural description and a human-written program. The description comes with several attributes such as card name, and card type, as well as a natural language description for the effect of the card. We follow previous work [1], [2] to pre-process the two datasets, and the statistic is listed in Table I.

AixBench-L is a function-level code generation benchmark and is an augmented version of the public AixBench benchmark [24]. We treat the original AixBench as the test data and collect lots of NL-code pairs from Github [25] as the train and dev data. Specifically, we mined Java open-source projects with at least 30 stars from GitHub, and avoid projects containing test data. From mined projects, we remove auto-generated functions and extract functions (i) having an English docstring; (ii) having $<$1024 tokens and $>$1 lines. Finally, we select 200k samples from mined projects and randomly split them into train data and valid data. We consider all mined projects as the retrieval corpus. The statistic is shown in Table I. Each test sample contains a functionally independent and well-described natural language description, a signature of the target function, and a set of unit tests that verify the correctness of the function. Following previous work [24], we take the natural language description and the function signature as models' inputs.

C. Metrics

On HearthStone and Magic datasets, we view human-written programs as the ground-truth, and employ three widely used metrics to evaluate the similarity of the generated code and the ground-truth [1], [2], [3].

- **Exact match (EM)** is the percentage of the generated code that has the same token sequence as the ground-truth.
- The **BLEU** score [13] is used to measure the token-level similarity between the generated code and the ground-truth. Specifically, it calculates the n-gram similarity and can be computed as:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{7}$$

where p_n is the n-gram matching precision scores, N is set to 4 in our experiments. BP is a brevity penalty to prevent very short generated code.

- The **CodeBLEU** score [14] is a variant of the BLEU score. It specializes in the source code and considers syntactic and semantic matches based on the code structure in addition to the n-gram match.

The test data in AixBench-L does not contain human-written programs. We have to omit metrics (e.g., EM, BLEU) requiring ground-truths. Following previous work [24], we use unit tests to evaluate the correctness of generated programs. Specifically, we employ the following metrics:

- **Pass@1** is the percentage of the generated code that passes all unit tests. It has been widely used in previous studies [26], [27], [28].
- **AvgPassRatio** denotes the average test cases pass ratio and can be calculated like this:

$$\begin{aligned} \text{AvgPassRatio} &= \frac{1}{T}\sum_{i}^{T}\text{PassRatio}_i \\ \text{PassRatio}_i &= \frac{\text{Count}_{i,\text{pass}}}{\text{Count}_{i,\text{total}}} \end{aligned} \tag{8}$$

  where $\text{Count}_{i,\text{pass}}$ and $\text{Count}_{i,\text{total}}$ are the number of passed test cases and the total number of test cases in i-th test sample, respectively. T is the size of test data.

D. Baselines

We select 20 recently proposed code generation models as baselines. They can be divided into three categories: sequence-based baselines, tree-based baselines, and pre-trained baselines.

The sequence-based baselines treat the source code as plain text and directly generate a code token sequence:

- **RNN** [29] is a classic neural network in source code processing. We utilize the RNN to implement a vanilla encoder-decoder code generation model as the baseline.
- **Transformer** [30] is a popular encoder-decoder model and has obtained promising results in code generation and code completion tasks [19].
- **LPN** [12] and **ReEdit** [6] are RNN-based code generation models. LPN proposes a structured attention mechanism to handle the semi-structural inputs. ReEdit introduces a retrieved similar program as an additional input.

The tree-based baselines directly generate a parsed tree (*e.g.,* abstract syntax tree) of the source code. The generated tree is further converted to the source code.

- **Seq2Tree** [31] is a pioneer tree-based work that proposes a attention-enhanced code generation model.
- **TRANX** [1] is a representative tree-based code generation model that can map an NL description into a tree using a series of tree construction actions.
- **ASN** [32] utilizes a dynamically-determined decoder to efficiently generate a tree.
- **TreeGen** [2] incorporates grammar rules and tree structures into the Transformer. It significantly outperforms previous RNN-based code generation models.
- **ReCode** [5] is a variant of the TRANX, which can copy n-gram actions from the tree of a similar program.

The pre-trained baselines are first pre-trained with a large-scale code corpus and then fine-tuned with code generation datasets. Nowadays, pre-trained code generation models have achieved SOTA results on many code generation datasets.

- **CodeBERT** [33] and **GraphCodeBERT** [34] are two encoder-only pre-trained models. They mainly apply the pre-training techniques for natural languages to the source code. We add a six-layer transformer decoder along with the two models, to support code generation. Both models contain 175 million parameters.
- **CodeGPT** [35] and **CodeParrot** [27] are two decoder-only pre-trained models. They are derived from the GPT-2 [36] and are continually pre-trained with the code. Both models contain 124 million parameters.
- **PyCodeGPT** [26] and **GPT-CC** [28] are two decoder-only pre-trained models. They are initialized with the GPT-Neo [37] and are continually pre-trained with a large-scale code corpus in Python. Both models contain 110 million parameters.
- **CERT-PyCodeGPT** [26] is a variant of the PyCodeGPT. It first predicts a sketch based on the NL description and further generates the complete code based on the sketch. We follow instructions in the original paper and train a CERT-PyCodeGPT (220M) in our experimental datasets.
- **CodeGen** [38] is a decoder-only pre-trained model. It casts code generation as a multi-turn conversation between a user and a system. In this paper, we use the CodeGen-Mono-350M version.
- **REDCODER** [7] is a encoder-decoder pre-trained model. It provides multiple similar code snippets as a supplement to a pre-trained code generation model. We use GraphCodeBERT to initialize the retriever and employ PLBART-base [39] to initialize the generator. The full REDCODER contains 315 million parameters.
- **CodeT5-small** and **CodeT5-base** [3] are two encoder-decoder pre-trained models. They propose an identifier-aware pre-training task and have achieved SOTA results on many code generation datasets. CodeT5-small contains 60 million parameters and CodeT5-base consists of 220 million parameters.

E. Implementation Details

The implementation details of our SKCODER are as follows:

- **Retriever.** We use the open-source search engine - Lucene [20] to build the retriever. The retrieval metric is the BM25 score. For HearthStone and Magic, the retrieval corpus is its training data. Note that we exclude the ground truths from the outputs of our retriever.
- **Sketcher.** We implement the sketcher with a 12-layer Transformer encoder. Its network architecture follows previous studies [33], [34]. We initialize the sketcher using pre-trained weights of GraphCodeBERT [34].
- **Editor.** The editor is an encoder-decoder Transformer, and the encoder and decoder both contain 12 Transformer layers. The editor follows the network architecture in the

TABLE II
RESULTS ON THE HEARTHSTONE DATASET (PYTHON). "*" REPRESENTS THE COPY-BASED BASELINES.

Type	Approach	EM	BLEU	CodeBLEU
	Retriever module	0	57.56	56.58
Sequence-based	LPN	6.10	67.10	–
	RNN	3.03	64.53	58.56
	Transformer	3.03	62.46	51.63
	ReEdit *	9.10	70.00	–
Tree-based	Seq2Tree	1.50	53.40	–
	TRANX	16.20	75.80	–
	ASN	18.20	77.60	–
	ReCode *	19.60	78.40	–
	TreeGen	25.80	79.30	–
Pre-trained	CodeBERT	3.03	66.50	59.39
	GraphCodeBERT	3.03	66.32	58.87
	CodeGPT	15.15	80.90	66.69
	GPT-CC	15.15	74.58	63.95
	CodeParrot	19.70	76.99	65.40
	PyCodeGPT	24.24	81.03	68.70
	CERT-PyCodeGPT	16.67	78.91	67.73
	CodeGen	24.24	78.80	67.43
	REDCODER *	21.21	80.08	67.31
	CodeT5-small	21.20	77.91	64.60
	CodeT5-base	25.84	81.28	68.42
	SKCODER	**30.30 (↑ 17.26%)**	**83.12 (↑ 2.26%)**	**70.97 (↑ 3.73%)**

TABLE III
RESULTS ON THE MAGIC DATASET (JAVA). WE OMIT SOME BASELINES AS THEY CANNOT BE APPLIED TO THE JAVA LANGUAGE.

Type	Approach	EM	BLEU	CodeBLEU
	Retriever module	0	53.64	64.23
Sequence-based	LPN	4.80	61.40	–
	RNN	16.26	71.96	61.83
	Transformer	12.20	73.10	66.61
Pre-trained	CodeBERT	19.42	78.69	71.73
	GraphCodeBERT	27.41	82.33	74.76
	CodeGPT	27.40	78.68	70.04
	REDCODER *	9.79	58.81	50.38
	CodeT5-small	26.95	78.38	71.11
	CodeT5-base	28.91	80.46	73.11
	SKCODER	**35.39 (↑ 22.41%)**	**85.39 (↑ 6.13%)**	**82.42 (↑ 10.27%)**

work [3] and is initialized with pre-trained weights of CodeT5-base [3].

- **Training & Testing.** We train the SKCODER with two NVIDIA A100 GPUs. The batch size is set to 32. During training, we use Top-5 similar code snippets to build the training data of our sketcher and editor. In the inference, we only use the Top-1 similar code, employ the beam search, and set the beam size to 10.

Note that initializing using pre-trained weights is common in previous studies [34], [7], [26], [27], [28] and can effectively improve the performance of models. To make a fair comparison, we also reuse the pre-trained weights in our experiments.

V. RESULTS AND ANALYSES

In our first research question, we evaluate the performance of our SKCODER with respect to previous code generation approaches.

RQ1: How does SKCODER perform compared to SOTA baselines?

Setup. We evaluate baselines (Section IV-D) and our SKCODER on three code generation datasets (Section IV-B). The evaluation metrics are described in Section IV-C, *i.e.,* the EM, BLEU, CodeBLEU, Pass@1, and AvgPassRatio. For all metrics, higher scores represent better performance.

Results. Table II, Table III and Table IV show the experimental results on three datasets, respectively. "–" denotes

TABLE IV
RESULTS ON THE AIXBENCH-L DATASET (JAVA). WE OMIT SOME BASELINES AS THEY CANNOT BE APPLIED TO THE JAVA LANGUAGE.

Type	Approach	Pass@1	AvgPassRatio
	Retriever module	2.86	7.93
Sequence-based	RNN	4.00	13.33
	Transformer	6.29	12.43
Pre-trained	CodeBERT	9.14	23.35
	GraphCodeBERT	10.86	24.99
	CodeGPT	17.71	35.67
	REDCODER *	16.00	33.14
	CodeT5-small	12.57	25.11
	CodeT5-base	15.43	24.53
	SKCODER	**20.00 (↑ 12.9%)**	**38.70 (↑ 8.49%)**

that the models have not been evaluated using this metric, to the best of our knowledge. "*" represents the copy-based baselines, which also use the retrieved similar code. The percentages in parentheses are the relative improvements compared to the strongest baselines. On Magic and AixBench-L datasets, we omit some baselines because they are designed for specific languages and cannot work in the Java dataset.

Analyses. (1) Our SKCODER achieves the best results among all baselines. Our SKCODER can generate more correct programs. Compared to the SOTA model - CodeT5-base, SKCODER outperforms it by up to 22.41% in EM and 29.62% in Pass@1. Note that the EM and Pass@1 are very strict metrics and are hard to be improved. The significant improvements prove the superiority of our SKCODER in automatic code generation. (2) The retrieved code is beneficial to code generation. Our retriever module performs well in the BLEU and CodeBLEU, but it is poor in the EM and Pass@1. It validates our motivation that the similar code contains lots of reusable contents and irrelevant parts. By introducing the retrieved code, code generation models can be further improved. For example, on the HearthStone dataset, ReEdit improves its base model (*i.e.,* RNN) by up to 200%, and ReCode improves its base model (*i.e.,* TRANX) by up to 20.99%. (3) Our SKCODER outperforms the SOTA copy-based baselines. The SOTA copy-based baseline is the REDCODER, which uses multiple similar code snippets to augment code generation models. While our SKCODER only uses the Top-1 similar code. Compared to the REDCODER, SKCODER improves it by 42.86% in EM, 25% in Pass@1, and 16.78% in AvPassRatio. This is because REDCODER is likely to repeat the similar code without necessary modifications. While our SKCODER utilizes a sketcher to extract the relevant content as a sketch, ignoring irrelevant parts. The sketch is further edited based on the input description. Thus, our SKCODER is closer to developers' code reuse behavior and can generate more correct programs.

On the HearthStone and Magic datasets, we notice that the improvements on the EM are higher than those on other metrics. We carefully compare the output of different models and find that baselines and our SKCODER all can correctly generate the body of programs. But baselines often err on some details, such as parameters. Thus, our SKCODER can

TABLE V
THE RESULTS OF ABLATION STUDY.

Editor	Retriever	Sketcher	HearthStone EM	HearthStone BLEU	HearthStone CodeBLEU	Magic EM	Magic BLEU	Magic CodeBLEU	AixBench-L Pass@1	AixBench-L AvgPassRate
RNN	✗	✗	3.03	64.53	57.56	16.26	71.96	61.83	4.00	13.33
	✓	✗	3.03 (↑ 0%)	68.39	59.12	16.51 (↑ 1.54%)	72.79	63.82	5.14 (↑ 28.5%)	10.61
	✓	✓	**4.54 (↑ 49.83%)**	**71.50**	**61.76**	**17.91 (↑ 10.15%)**	**73.72**	**65.04**	**8.57 (↑ 114.3%)**	**13.42 (↑ 2.7%)**
CodeT5-small	✗	✗	21.20	77.91	64.60	26.95	78.38	71.11	12.57	25.11
	✓	✗	27.86 (↑ 31.42%)	79.84	68.76	31.73 (↑ 17.74%)	80.85	77.10	14.29 (↑ 13.68%)	26.06 (↑ 3.78%)
	✓	✓	**30.30 (↑ 42.90%)**	**83.08**	**69.35**	**33.89 (↑ 25.75%)**	**85.15**	**80.08**	**18.29 (↑ 45.51%)**	**34.05 (↑ 35.6%)**
CodeT5-base	✗	✗	25.24	81.28	68.42	28.91	80.46	73.11	15.43	24.52
	✓	✗	27.81 (↑ 10.18%)	82.06	69.35	32.43 (↑ 12.18%)	83.11	78.97	17.71 (↑ 14.78%)	34.75 (↑ 41.72%)
	✓	✓	**30.30 (↑ 20.05%)**	**83.12**	**70.97**	**35.39 (↑ 22.41%)**	**85.39**	**80.62**	**20.00 (↑ 29.62%)**	**38.70 (↑ 57.83%)**

generate more exactly correct programs, and achieve lower improvements on n-gram similarity metrics (*i.e.,* BLEU and CodeBLEU). The results also verify that compared to generating the code from scratch, editing a well-formed sketch is easier to generate the correct code.

Answer to RQ1: SKCODER achieves the best results among all baselines. In particular, SKCODER generates 30.30%, 35.39%, and 20% correct programs on three datasets, outperforming the SOTA code generation models by 17.26%, 22.41%, and 12.9%. The significant improvements prove our sketch-based approach is more promising in automatic code generation.

In RQ2, we aim to figure out the contributions of different modules in our SKCODER. Besides, we plan to investigate the effectiveness of our approach on different code generation models.

RQ2: What are the contributions of different modules in our approach?

Setup. In this RQ, we select three code generation models as the base editor, including RNN, CodeT5-small, and CodeT5-base. They cover mainstream network architectures, *i.e.,* RNN, Transformer, and pre-trained models. For each editor, we conduct an ablation study by gradually adding the retriever and sketcher.

Results. The experimental results is shown in Table V. ✓ and ✗ represent adding and removing corresponding modules, respectively. An individual editor is just a vanilla code generation model that maps an NL description to the source code. After adding a retriever, the model takes the retrieved code as an additional input. After further introducing a sketcher, the model is our sketch-based approach.

Analyses. (1) All three modules are necessary to perform the best. After adding a retriever, the performance of all models is improved. For example, on HearthStone, the retriever brings a 10.18% improvement in the EM for the CodeT5-base. It validates that the retrieved code contains lots of valuable information that benefits code generation models. After introducing a sketcher, all models obtain better results. For example, on the HearthStone, the CodeT5-base is improved by 20.05% in the EM. It proves that compared to copying from the retrieved code, our sketch-based code generation approach can better mine the knowledge in the retrieved code.

TABLE VI
THE PERFORMANCE OF DIFFERENT SKETCHERS.

Approach	HearthStone EM	HearthStone BLEU	HearthStone CodeBLEU	Magic EM	Magic BLEU	Magic CodeBLEU
Without sketcher	27.81	82.06	69.35	32.43	82.01	78.87
Sketcher-1	27.93 (↑ 0.43%)	82.39	70.81	33.06 (↑ 1.94%)	83.04	80.15
Sketcher-2	29.03 (↑ 4.39%)	82.77	70.27	34.46 (↑ 5.95%)	83.91	80.19
Our Sketcher	**30.30 (↑ 9.13%)**	**83.12**	**70.97**	**35.39 (↑ 9.13%)**	**85.39**	**80.62**

Input NL description:
```
check if all elements in list var_0 are identical
```
Ground truth:
```
all(x == var_0[0] for x in var_0)
```
Retrieved similar code:
```
all(isinstance(x, int) for x in var_0)
```
Sketcher-1 (anonymizing user-defined terms):
```
all(isinstance(v_1, int) for v_1 in v_2)
```
Sketcher-2 (keeping overlapping tokens):
```
all((x) for x in var_0)
```
Our Sketcher (longest common subsequence):
```
all( <pad> for x in var_0)
```

Fig. 5. Examples of three sketches.

(2) Our approach is effective to multiple code generation models. As shown in Table V, our approach supports different code generation models and brings obvious improvements. Specifically, in terms of the Pass@1, our approach improves the RNN by up to 114.3%, the CodeT5-small by 45.31%, and the CodeT5-base by 29.62%. In the future, our approach can be used to enhance more powerful code generation models.

Answer to RQ2: All three modules are essential for the performance of our approach. Besides, our approach is effective to different code generation models and improves them by 114.3%, 45.31%, and 29.62% in Pass@1.

Code sketches are not explicitly defined in existing datasets and how to build a sketch is an open question, Thus, we design several plausible design choices for the sketcher and investigate which one is better.

RQ3: What is the better design choice for the code sketch?

Setup. In this RQ, we provide three sketchers (*i.e.,* sketcher-1, sketcher-2, and our sketcher). The sketcher-1 utilizes a parse to anonymize the user-defined terms in the similar code (*i.e.,* string, constant, variable) and obtains a code sketch. The sketcher-2 trains a neural network to predict overlapping tokens between the similar code and the ground truth. The overlapping tokens are collected to build a sketch. Our sketcher

trains a neural network to predict the longest common subsequence (LCS) between the similar code and the ground-truth. The predicted LCS is viewed as a sketch. We present some examples of different sketchers in Figure 5.

Results. The experimental results are shown in Table VI. We present the results of our SKCODER with different sketchers and the result without a sketcher.

Analyses. (1) Introducing a sketcher can better utilize the retrieved code. Compared to the model without a sketcher, the models with sketchers perform better. It shows that the sketcher can better mine the knowledge from the retrieved code and our sketch-based approach is more promising than copy-based approaches. (2) Our sketcher performs best among all baselines. On both datasets, our sketcher brings 2x improvements (*e.g.,* 9.13% vs. 0.43%) over other sketchers. This is because our sketcher can accurately extract the relevant content and leave irrelevant details, while other sketchers cannot. As shown in Figure 5, sketch-1 outputs a sketch by anonymizing the user-defined terms. But the anonymized code still contains irrelevant parts (*e.g.,* `isinstance`) and even loses some reusable tokens (*e.g.,* `var_0`). Sketch-2 only keeps tokens that may occur in the ground truth. It ignores the sequentiality of tokens, and the generated sketch probably is disorder and confusing (*e.g.,* `(x)`). By contrast, the sketch produced by our sketcher is well-formed and provides a clear code structure.

Answer to RQ3: Code sketches are beneficial to reuse the knowledge in the retrieved code. Among multiple plausible sketchers, our sketcher performs best and brings a maximum of 9.13% improvement in the EM.

VI. HUMAN EVALUATION

The ultimate goal of code generation models is to assist developers in writing the source code. Thus, in this section, we conduct a human evaluation to assess our SKCODER.

Setup. Following previous work [24], we manually evaluate the generated code by different models in three aspects, including *correctness* (whether the code satisfies the given requirement), *code quality* (whether the code does not contain bad code smell) and *maintainability* (whether the implementation is standardized and has good readability). For each aspect, the score is integers, ranging from 0 to 2 (from bad to good). We randomly select 50 test samples and collect programs generated by 10 models on these samples. Finally, we obtain 500 programs (50*10) for evaluation. The evaluators are computer science Ph.D. students and are not co-authors. They all have programming experience ranging from 3+ years. The 500 code snippets are divided into 5 groups, with each questionnaire containing one group. We randomly list the code and the corresponding input description on the questionnaire. Each group is evaluated anonymously by two evaluators, and the final score is the average of two evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts.

Results and Analyses. The results of the human evaluation are shown in Table VII. Our SKCODER is better than all baselines in three aspects. Specifically, SKCODER outperforms the SOTA model - CodeT5-base by 16.95% in correctness, 12.45% in code quality, and 12.09% in maintainability. All the p-values are substantially smaller than 0.005, which shows the improvements are statistically significant. The improvements prove the superiority of our SKCODER in assisting developers in coding. Besides, we notice that the copy-based model - REDCODER performs well in maintainability but is poor in correctness and code quality. This is because REDCODER can generate natural programs by copying from the retrieved code. But some copied content is irrelevant and leads to incorrect code.

TABLE VII
THE RESULTS OF HUMAN EVALUATION. ALL THE P-VALUES ARE SUBSTANTIALLY SMALLER THAN 0.005.

Approach	Correctness	Code quality	Maintainability
GraphCodeBERT	0.9277	0.9872	1.3049
CodeGPT	0.9798	1.0229	1.3306
REDCODER *	1.0177	1.2038	1.5796
CodeGen	1.1250	1.3610	1.5573
PyCodeGPT	1.1098	1.3661	1.5442
CodeParrot	0.9704	1.0814	1.3668
GPT-Code-Clippy	0.9646	1.0585	1.3672
CERT-PyCodeGPT	0.9629	1.0439	1.3882
CodeT5-base	1.1719	1.3908	1.5848
SKCODER	**1.3705** (↑ **16.95%**)	**1.5639** (↑ **12.45%**)	**1.7764** (↑ **12.09%**)

VII. DISCUSSION

A. Case Study

Figure 6 presents some code snippets generated by different models on the HearthStone dataset. From the examples, we obtain the following findings. (1) The retrieved similar code provides a well-formed code structure and contains some irrelevant details (*e.g.,* `ImpGangBoss`). (2) As a copy-based approach, REDCODER wrongly repeat the inappropriate statement (*i.e.,* `effects=[Effect(...), ActionTag(...)])` without modifications. It causes the generated code is inconsistent with the input description. (3) Our sketcher accurately keeps the relevant content and replaces irrelevant details with placeholders. The extracted sketch provides a clear start-point for editing. (4) Based on the input description, our SKCODER further edits the sketch into the desired code. For example, the input description specifies the card's effect (*i.e.,* `whenever you hero takes damage on your turn, gain +2/+2`). SKCODER modifies the `Damaged()` and `ActionTag()` calls in the sketch and adds more details (*e.g.,* `And(IsHero(), OwersTurn())`. Besides, the editor adds some components that are not in the sketch, such as `Character`.

B. Threats to Validity

There are three main threats to the validity of our work.

The generalizability of our experimental results. To mitigate this threat, we carefully design the experimental datasets, metrics, and baselines. For the datasets, we follow previous studies [1], [5], [2] and pick three representative code generation datasets. The three datasets are collected from real software projects and communities and cover two

Input description: NAME: Floating Watcher ATK: 4 DEF: 4 COST: 5 DUR: -1 TYPE: Minion PLAYER: Warlock RACE: Demon RARITY: Common DESCRIPTION: Whenever your hero takes damage on your turn, gain +2/+2.

Retriever (similar code):

```
class ImpGangBoss(MinionCard):
    def __init__(self):
        super().__init__("Imp Gang Boss", 3,
            CHARACTER_CLASS.WARLOCK, CARD_RARITY.COMMON, minion_type=MINION_TYPE.DEMON)
    def create_minion(self, player):
        return Minion(2, 4,
            effects=[Effect(Damaged(), ActionTag(Summon(Imp()), PlayerSelector()))])
```

Sketcher (code sketch from similar code):

```
class <pad> (MinionCard):
    def __init__(self):
        super().__init__( <pad> , <pad> ,
            CHARACTER_CLASS.WARLOCK, CARD_RARITY.COMMON, minion_type=MINION_TYPE.DEMON)
    def create_minion(self, player):
        return Minion(2, 4,
            effects=[Effect(Damaged(), ActionTag( <pad> , <pad> ))])
```

REDCODER:

```
class FloatingWatcher(MinionCard):
    def __init__(self):
        super().__init__("Floating Watcher", 5,
            CHARACTER_CLASS.WARLOCK, CARD_RARITY.COMMON, minion_type=MINION_TYPE.DEMON)
    def create_minion(self, player):
        return Minion(4, 4,
            effects=[Effect(Damaged(), ActionTag(Summon(Imp()), PlayerSelector()))])
```

SkCoder (editing the sketch) & Ground-truth:

```
class FloatingWatcher(MinionCard):
    def __init__(self):
        super().__init__("Floating Watcher", 5,
            CHARACTER_CLASS.WARLOCK, CARD_RARITY.COMMON, minion_type=MINION_TYPE.DEMON)
    def create_minion(self, player):
        return Minion(4, 4,
            effects=[Effect(CharacterDamaged(And(IsHero(), OwnersTurn())),
                ActionTag(Give([Buff(ChangeAttack(2)),
                                Buff(ChangeHealth(2))]), SelfSelector()))])
```

Fig. 6. Examples of code snippets generated by different models. We highlight the parts that SKCODER modifies on the sketch.

popular programming languages (*i.e.,* Java and Python). For the metrics, we select five widely used metrics, including the EM, BLEU, CodeBLEU, Pass@1, and AvgPassRatio. Existing work [40] has proven the reliability of these metrics. To verify the superiority of our approach, we select 20 code generation models as our baselines for the comparison. They cover the most of representative work in the past six years. Besides, we run each approach three times and report the average results.

The implementation of models. It is widely known that deep learning models are sensitive to the implementation details, including hyper-parameters and network architectures. In this work, we need to implement baselines and our approach. For the baselines, we use the source code provided by their original papers and ensure that the model's performance is comparable with their reported results. For our approach, we implement a version that employs mainstream neural networks (details in Section IV-E). Due to the high training cost, we do a small-range grid search on several hyper-parameters (*i.e.,* learning rate and batch size), leaving other hyper-parameters the same as those in previous studies [18], [34], [3]. Thus, there may be room to tune more hyper-parameters and network architectures of our approach for more improvements.

The impact of retrieved code. The retrieved code is an important element in our approach. Intuitively, when the retrieved code is less similar to the target code, the performance of our model may suffer. To address this threat, we have two thoughts. (1) A large-scale study on 13.2 million real code files found the proportion of reused code is up to 80% [8]. Therefore, we believe that it is quite possible to retrieve the similar code in real development scenarios. (2) Even if the retrieved code is dissimilar to the target code, our SKCODER can selectively focus on the retrieved code based on current requirements. To prove this point, we randomly select code snippets from the retrieval corpus as the retrieved code and train a variant named SKCODER-random. The results are shown in Table VIII. SKCODER-random has a drop compared to SKCODER but still substantially outperforms CodeT5-base. It proves that our SKCODER can adaptively extract valuable content from the retrieved code and has strong robustness.

TABLE VIII
THE PERFORMANCE OF SKCODER-RANDOM.

Approach	EM	BLEU	CodeBLEU
CodeT5-base	28.91	80.46	73.11
SKCODER-random	33.48 (↑ 15.81%)	82.07	79.08
SKCODER	35.39 (↑ 22.41%)	85.39	80.62

VIII. RELATED WORK

Code generation aims to generate the source code that satisfies a given natural language description or requirement. Existing work can be divided into three categories: sequence-based models, tree-based models, and pre-trained models.

Sequence-based Models. Sequence-based models treat the source code as a sequence of tokens and use neural networks to generate the source code token-by-token based on the input description. Ling et al. [12] generate the source code with a structured attention mechanism to process the semi-structural input. Hashimoto et al. [6] train a task-dependent retriever to retrieve the similar code, and then use the similar code as an additional input to the generator. Wei et al. [41] propose a code generation model based on dual learning, which performs better with the help of code summarization.

Tree-based Models Program is strictly structured, and can be parsed into a tree, *e.g.,* Abstract Syntax Tree (AST). Tree-based models generate a parse tree of the program based on the NL description and then convert the parse tree into the corresponding code. Dong et al. [31] generate the AST by expanding every non-terminal with an LSTM model. Rabinovich et al. [32] generate the AST with a decoder that has a dynamically-determined modular structure paralleling the structure of the output AST. Yin et al. [1] generate the tree-construction action sequence with an LSTM model, and construct the AST from the action sequence. Sun et al. [2] encode the natural language and grammar rules that have been generated with specially designed Transformer blocks, and predict the next grammar rule accordingly.

Pre-trained Models Recent years have witnessed the emergence of pre-trained models [42], [43], [44]. These models are pre-trained on massive data of source code and then fine-tuned on code generation task. Pre-trained models can be divided into three categories.

(1) *Encoder-only pre-trained models* only contains an encoder and is mainly used in code representation. They are usually pre-trained with language comprehension tasks, *e.g.,* masked language modeling or replaced token detection. The recently proposed encoder-only pre-trained models include the CodeBERT [33], GraphCodeBERT [34], etc. (2) *Decoder-only pre-trained models* are pre-trained to predict the next token based on the input context. GPT series [45] are excellent decoder-only models for natural language processing, and there are many efforts to adapt similar ideas to code. Lu et al. [35] adapt GPT-2 [36] model on the source code, resulting in CodeGPT. Chen et al. [46] fine-tune GPT-3 [47] models on the code to produce CodeX and GitHub Copilot [48]. Neither CodeX nor GitHub Copilot is open-sourced, which leads to several attempts to replicate CodeX in industry and academia, resulting in CodeParrot [27], GPT-CC [28], PyCodeGPT [26], and CodeGen [38]. CodeParrot and CodeGen are trained from scratch. PyCodeGPT and GPT-CC are fine-tuned from GPT-Neo[37]. Zan et al. [26] propose a variant of PyCodeGPT. They first generate a sketch that anonymizes user-defined constants, and then generate the complete program from the NL and the sketch. (3) *Encoder-decoder pre-trained models* are composed of an encoder and a decoder. They can support both code representation and code generation tasks. Various successful encoder-decoder architecture in natural language processing has been transferred into the source code, resulting in powerful models, *e.g.,* CodeT5 [3] and PLBART [39].

Inspired the code reuse, some studies introduce the similar code to augment code generation models. Hayati et al. [5] retrieve the similar code with the input, and copy n-gram actions from the similar code during decoding. Hashimoto et al. [6] and Parvez et al. [7] retrieve similar code snippets and feed them along with the input description to a generator. They train the generator to learn to copy some reusable content from the similar code. We refer to these studies as copy-oriented approaches. Different from copy-oriented approaches, our sketch-oriented SKCODER mimics the developers' code reuse behavior, extracts content that is relevant to input requirement and ignores irrelevant parts in the similar code. The extracted content is viewed as a code sketch and further edited to the target code with guidance of input requirement.

IX. CONCLUSION AND FUTURE WORK

During software development, human developers often reuse similar code snippets. In this paper, we propose a novel sketch-based code generation approach named SKCODER to mimic developers' code reuse behavior. Different from previous copy-based approaches, SKCODER can extract the relevant content from the retrieved similar code and builds a code sketch. The sketch is further edited by adding more requirement-specific details. We conduct experiments on two public code generation datasets and a new Java dataset collected by this work. The new dataset contains 200k NL-code pairs and each test sample is equipped with a set of unit tests. Experimental results show that SKCODER substantially outperforms state-of-the-art baselines. The ablation study proves the effectiveness of code sketches and our approach is effective to different neural networks. In the future, we will explore more effective sketchers and apply our sketch-based idea to large-scale pre-trained models.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program under Grant No. 2021ZD0110303, the National Natural Science Foundation of China under Grant Nos. 62192731, 61751210, 62072007, 62192733, 61832009, and 62192730.

REFERENCES

[1] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (Demo Track)*, 2018.

[2] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[3] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[4] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[5] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[6] T. B. Hashimoto, K. Guu, Y. Oren, and P. S. Liang, "A retrieve-and-edit framework for predicting structured outputs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[7] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 2719–2734.

[8] A. Mockus, "Large-scale code reuse in open source software," in *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 7–7.

[9] [Online]. Available: https://stackoverflow.com/

[10] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[11] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[12] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 599–609.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[14] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[15] [Online]. Available: https://github.com/LJ2lijia/SkCoder

[16] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[17] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.

[18] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 155–166.

[19] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 6227–6240.

[20] [Online]. Available: https://lucene.apache.or

[21] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[22] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[23] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[24] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei, "Aixbench: A code generation benchmark dataset," *arXiv preprint arXiv:2206.13179*, 2022.

[25] [Online]. Available: https://github.com/

[26] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: Continual pre-training on sketches for library-oriented code generation," in *Proceedings of the 31-th International Joint Conference on Artificial Intelli-gence (IJCAI 2022)*.

[27] [Online]. Available: https://huggingface.co/codeparrot/codeparrot

[28] [Online]. Available: https://github.com/CodedotAl/gpt-code-clippy

[29] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[31] L. Dong and M. Lapata, "Language to logical form with neural attention," in *54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL), 2016, pp. 33–43.

[32] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1139–1149.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[34] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2020.

[35] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[37] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow, march 2021," *URL https://doi. org/10.5281/zenodo*, vol. 5297715.

[38] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[39] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[40] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *arXiv preprint arXiv:2208.03133*, 2022.

[41] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[42] J. Li, G. Li, Z. Li, Z. Jin, X. Hu, K. Zhang, and Z. Fu, "Codeeditor: Learning to edit source code with pre-trained models," *arXiv preprint arXiv:2210.17040*, 2022.

[43] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Towards enhancing in-context learning for code generation," *arXiv preprint arXiv:2303.17780*, 2023.

[44] J. Li, G. Li, Y. Li, and Z. Jin, "Enabling programming thinking in large language models toward code generation," *arXiv preprint arXiv:2305.06599*, 2023.

[45] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[46] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[47] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[48] [Online]. Available: https://github.com/features/copilot