SCoder: Empowering Code LLMs through Bootstrapping Small-Scale Data Synthesizers

Anonymous ACL submission

Abstract

Existing code large language models (LLMs) often rely on large-scale instruction data distilled from proprietary LLMs for fine-tuning, which typically incurs high costs. In this paper, we explore the potential of small-scale open-source LLMs (e.g., 7B) as synthesizers for high-quality code instruction data construction. We first observe that the data synthesis capability of small-scale LLMs can be enhanced by training on a few superior data synthesis samples from proprietary LLMs. Building on this, we propose a novel multi-round self-distillation approach to bootstrap small-scale LLMs, transforming them into powerful synthesizers that reduce reliance on proprietary LLMs and minimize costs. Concretely, we design multi-checkpoint sampling and multi-aspect scoring strategies to self-distill data synthesis samples and filter them in each round. Based on these filtered samples, we further select high-value ones by introducing an optimizer-aware influence estimation method, which estimates the influence of each self-distilled sample by calculating its gradient similarity to the superior samples from proprietary LLMs. Based on the code instruction data from our small-scale synthesizers, we introduce SCoder, a family of code generation models fine-tuned from DeepSeek-Coder. SCoder achieves state-of-the-art code generation capabilities, demonstrating the effectiveness of our method.

1 Introduction

Code generation has long been a central challenge in computer science, and has attracted wide attention from the research community. Recent advancements in code large language models (LLMs) (Chen et al., 2021; Li et al., 2022, 2023; Chowdhery et al., 2023; Rozière et al., 2023; Lozhkov et al., 2024) have led to significant breakthroughs. These models can generate code that closely aligns with user intent and are increasingly being widely adopted.

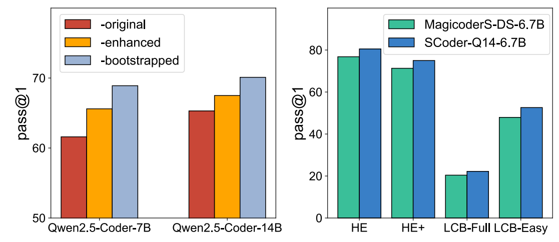

Figure 1: **Left**: The performance of code generation models on HumanEval using data provided by different synthesizers (Qwen2.5-Coder-7B or -14B). **Right**: The performance of our SCoder and the baseline. SCoder uses 60K instruction data generated by a small-scale synthesizer and the baseline uses 75K instruction data generated by proprietary LLMs. All code generation models are fine-tuned from DeepSeek-Coder-6.7B-Base.

Typically, instruction tuning is a crucial step to develop high-performance code LLMs. Therefore, extensive research on code LLMs focuses on constructing high-quality instruction data. A common approach involves distilling knowledge from proprietary LLMs. For instance, Code Alpaca (Chaudhary, 2023) and WizardCoder (Luo et al., 2024) are fine-tuned with instruction data distilled from GPT-3.5, using Self-Instruct (Wang et al., 2023) and Evol-Instruct (Xu et al., 2024), respectively. Additionally, MagicoderS (Wei et al., 2024) is fine-tuned on data distilled from both GPT-3.5 and GPT-4, using OSS-Instruct to generate coding problems and solutions based on the given code snippets. While these methods have proven effective, they all suffer from the cost-expensive issue caused by the distillation of large-scale instruction data from the proprietary LLMs like GPT-3.5 and GPT-4.

In this paper, we explore the potential of small-scale (7B, 8B, and 14B) open-source LLMs as synthesizers for code instruction data construction, where small-scale is a relative concept compared to extremely large models like GPT-3.5 and GPT-4. Previous works have shown that small LLMs can assist in pre-training data synthesis for non-code domains (Yang et al., 2024). However, instruction data typically takes a different form from pre-training data and requires higher quality standards (Wang et al., 2025). To validate the feasibility of small LLMs in synthesizing code instruction data, we conduct a preliminary experiment. First, we use small-scale LLMs as original synthesizers and further train them on a limited set of proprietary LLM-distilled samples as enhanced synthesizers. Then, we fine-tune code generation models using data provided by them. The results on the left of Figure 1 show that the instruction data provided by the enhanced synthesizer outperforms that of the original, highlighting that a few superior samples can unleash the data synthesis potential of small models. However, distilling more proprietary samples to further improve the synthesis capability of small synthesizers would again trigger the cost-expensive issue. Therefore, a crucial question arises: ***Can we continuously improve the data synthesis capability of small-scale synthesizers without relying on proprietary LLMs' samples?***

To address this, we propose an effective multi-round self-distillation bootstrap method that iteratively improve the code instruction data synthesis capability of small-scale LLMs. Specifically, starting with an enhanced synthesizer, we employ a two-step approach in each round to obtain high-quality self-distilled data synthesis samples for further training. First, we develop multi-checkpoint sampling and multi-aspect scoring strategies to obtain and filter self-distilled samples, maintaining their diversity and reliability. Then, we introduce an optimizer-aware influence estimation method to further select high-value ones by computing the gradient similarity between each self-distilled sample to the superior samples from proprietary LLMs. We validate our method on small-scale LLMs like Qwen2.5-Coder-7B/14B-Ins (Hui et al., 2024), improving their data synthesis capabilities as shown in the left of Figure 1, and transforming them into powerful data synthesizers.

Based on the code instruction data provided by our small-scale synthesizers, we introduce SCoder, a family of code generation models fine-tuned from DeepSeek-Coder-6.7B-Base (Guo et al., 2024). Experimental results on HumanEval (+) (Chen et al., 2021; Liu et al., 2023), MBPP (+) (Austin et al., 2021), LiveCodeBench (Jain et al., 2024), and BigCodeBench (Zhuo et al., 2024) show that SCoder outperforms or matches state-of-the-art code LLMs, which use instruction data from proprietary LLMs. Overall, our contributions can be summarized as follows:

1) We propose a multi-round self-distillation bootstrap method to fully unleash the data synthesis potential of small-scale LLMs and develop small-scale synthesizers for code instruction data construction.

2) To obtain diverse, reliable and high-value self-distilled data for synthesizer training in each round, we introduce a two-step process that combines multi-checkpoint sampling, multi-aspect scoring, and optimizer-aware influence estimation.

3) We fine-tune the code generation models, SCoder, based on the data provided by our small-scale synthesizers. Experimental results on multiple benchmarks show the effectiveness of our method.

2 Related Work

2.1 Code Large Language Models

Code generation based on LLMs has seen significant advancements in recent years. Prominent closed-source models, such as Codex (Chen et al., 2021), GPT-4 (OpenAI, 2023), PaLM (Chowdhery et al., 2023), and Gemini (Anil et al., 2023), have demonstrated impressive results across various code generation benchmarks. Meanwhile, open-source models, including CodeGen (Nijkamp et al., 2023), CodeGeeX (Zheng et al., 2023), StarCoder (Li et al., 2023), CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024), and CodeQwen (Hui et al., 2024), have also made substantial contributions to the field. The success of these models has not only advanced code generation capabilities but also facilitated more efficient and automated software development.

Typically, these models are developed through continual pre-training (Rozière et al., 2023), followed by supervised fine-tuning (SFT) (Yu et al., 2023). While pre-training leverages vast amounts of unannotated real-world code corpora, fine-tuning requires high-quality labeled instruction data, the construction of which remains a critical challenge (Ding et al., 2024).

2.2 Code Instruction Data Synthesis

Creating diverse and complex instruction data, especially in the coding domain, is a challenging task that requires specialized knowledge. While human-written instruction datasets, such as those used in OctoPack (Muennighoff et al., 2024) and PIE (Shypula et al., 2024), have proven effective, they are labor-intensive and difficult to scale. As a result, current works often rely on powerful proprietary LLMs to automatically generate code instruction data. For instance, Code Alpaca (Chaudhary, 2023) employs the Self-Instruct (Wang et al., 2023) method to generate instruction data from a pool of seed tasks, while WizardCoder (Luo et al., 2024) uses the Evol-Instruct (Xu et al., 2024) technique, which synthesizes diverse and complex instruction data through evolutionary heuristics. Magicoder (Wei et al., 2024) employs OSS-Instruct, which utilizes open-source code snippets as seeds to generate high-quality programming problems and solutions, thereby enhancing the diversity and realism of the generated data. WaveCoder (Yu et al., 2023) proposes a generator-discriminator framework to generate instruction data, while OpenCodeInterpreter (Zheng et al., 2024) leverages interactions between users, LLMs, and compilers to create diverse, multi-turn instruction data. Although these methods are effective, they often rely on costly proprietary LLMs for data distillation, leading to significant expenses (Wu et al., 2024). In this work, we explore the potential of small-scale open-source LLMs to generate high-quality code instruction data in a more cost-effective manner, reducing the reliance on expensive proprietary models while maintaining comparable performance.

3 Methodology

3.1 Overview

In this work, we aim to train a set of small-scale code instruction data synthesis models, named synthesizers, capable of generating the high-quality code instruction data, i.e., the code problem-solution pair (q, s) given an open-source code snippet c and an instruction synthesis prompt p. To achieve this, we first construct a clean and noise-free code snippet pool $\mathcal{C} = \{c_i\}$, following the data pre-processing pipeline of StarCoder2 (Lozhkov et al., 2024). Next, we distill a limited number of instruction data synthesis samples, denoted as $\mathcal{D}_p = \{(p, c_i^p, q_i^p, s_i^p)\}$, from proprietary LLMs to obtain enhanced synthesizers. Finally, we propose a bootstrap method based on multi-round self-distillation to continually train the synthesizers using self-distilled data, denoted as $\mathcal{D}_s = \{(p, c_i^s, q_i^s, s_i^s)\}$. The prompt p and more details of code snippet pool $\mathcal{C}$ are provided in Appendix D and A, respectively.

Synthesizer	HumanEval	MBPP
Llama3.1-8B-Ins	60.4	64.7
+*Enhanced*	64.2	69.3
Qwen2.5-Coder-7B-Ins	61.6	70.8
+*Enhanced*	65.6	72.1
Qwen2.5-Coder-14B-Ins	65.3	73.7
+*Enhanced*	67.5	75.8

Table 1: The performance of code generation model fined-tuned on 40K code instruction data provided by different synthesizers.

3.2 Preliminary Study

We conduct a preliminary study to validate whether small LLMs can acquire a certain level of data synthesis capability by distilling a limited number of proprietary LLM samples. To obtain proprietary samples with sufficient knowledge coverage, we adopt a classification-based diversified code snippet sampling technique. Specifically, we employ 10 pre-defined task categories and calculate the similarity between each code snippet and the task category descriptions with the help of a state-of-the-art embedding model INSTRUCTOR (Su et al., 2023). Based on the embedding similarity, each code snippet is assigned to its most relevant task category. We then randomly sample 1K code snippets from each category to ensure sufficient knowledge diversity. Finally, these selected code snippets are used to prompt proprietary LLMs generating code instruction data synthesis samples $\mathcal{D}_p = \{(p, c_i^p, q_i^p, s_i^p)\}$, where (p, c_i^p) denotes input and (q_i^p, s_i^p) denotes output.

We use Llama3.1-8B-Ins and Qwen2.5-Coder-7B/14B-Ins as the original synthesizers and train them on $\mathcal{D}_p$ to obtain enhanced synthesizers. Based on code instruction data provided by these synthesizers, we fine-tune DeepSeek-Coder-6.7B-Base as the code generation model. The results are shown in Table 1, the enhanced synthesizers exhibit a significant improvement in data synthesis capability, even with only 10K proprietary LLM samples. This demonstrates the strong potential of small models for code instruction data synthesis.

3.3 Bootstrapping with Multi-Round Self-Distillation

To further boost small LLMs for synthesizing higher-quality code instruction data without distilling additional proprietary LLM samples, in this section we propose an effective bootstrap method based on multi-round self-distillation. Specifically,

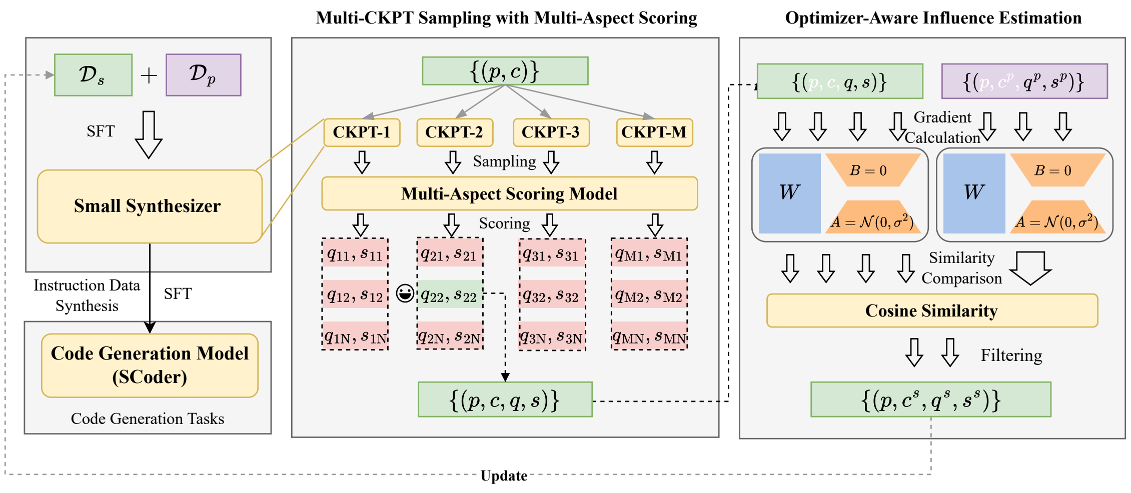

Figure 2: **Overview of our multi-round self-distillation bootstrap method.** In each round, we sample outputs from multiple checkpoints and evaluate them with a multi-aspect scorer for diversity and reliability. We then use an optimizer-aware influence estimation method to select the most valuable samples, which is done by evaluating the gradient similarity between the self-distilled and proprietary LLM-distilled code instruction data.

we start with the mentioned enhanced synthesizers, considering this as the 0-th round of the bootstrap. Then, in subsequent each iteration, we first collect diverse and reliable self-distilled data synthesis samples by multi-checkpoint sampling and multi-aspect scoring strategies. These samples are generated by the synthesizers from the previous round. Next, to further identify the most valuable samples, we introduce an optimizer-aware influence estimation method, which quantifies each sample's influence by computing its gradient similarity with proprietary LLM samples. Finally, these high-quality samples are used to train the synthesizer itself, enhancing its ability to generate code instruction data. The overview of our method is shown in Figure 2.

Multi-Checkpoint Sampling with Multi-Aspect Scoring. As our approach iteratively trains on self-distilled data synthesis samples, ensuring their quality and diversity is essential. Therefore, we first develop a multi-checkpoint sampling strategy. Specifically, given the synthesis prompt p and a code snippet c, we obtain $M \times N$ diverse problem-solution pairs $\{(q_{ij}, s_{ij})\}$ by sampling N times from M checkpoints of synthesizers, where $i \in [1, M]$ and $j \in [1, N]$. Compared to the strategy Best-of-N (Stiennon et al., 2022), which selects candidates from a single checkpoint, our approach expands the search space and improves both the reliability and diversity of the selected data.

Next, to rank and select the best candidate pair corresponding to code snippet, we introduce a multi-aspect scoring model, namely scorer. Given a candidate pair (q_{ij}, s_{ij}), the scorer evaluates it across Z aspects, producing a feature vector $\mathbf{x}_{ij} = \{x_{ij}^z\}$, where $x_{ij}^z \in [0, 9]$ represents the integer score in the z-th aspect, such as problem-solution consistency [1]. Furthermore, considering that different aspects are independent and integer-based scores provide only a hard signal that lacks granularity for distinguishing data quality, we propose a weighted scoring aggregation method, which assigns each aspect a weight w^z and computes the final aggregated real-valued score $Score_{ij}$ as:

$$Score_{ij} = \sum_{z=1}^{Z} w^z x_{ij}^z. \tag{1}$$

To determine the optimal weight vector $\mathbf{w} = \{w^z\}$, we conduct K experiments based on the instruction data generated by synthesizers. For each experiment, we compute the average multi-aspect scores $\bar{\mathbf{x}}_k$ of the instruction data and use the data to fine-tune DeepSeek-Coder-6.7B-Base. The fine-tuned model is then evaluated on an out-of-distribution (OOD) test set to obtain the corresponding performance score y_k. Given the data $\{(\bar{\mathbf{x}}_k, y_k)\}$, we estimate $\mathbf{w}$ by solving the following ridge regression problem:

$$\mathbf{w} = \arg\min_{\mathbf{w}} \sum_{k=1}^{K} (y_k - \mathbf{w} \cdot \bar{\mathbf{x}}_k)^2 + \lambda \|\mathbf{w}\|^2, \tag{2}$$

[1]The prompt for the multi-aspect scorer are provided in Appendix D.

where λ is a regularization term to prevent overfitting, and the learned weights indicate the relative importance of each scoring aspect in determining the effectiveness of instruction data.

Optimizer-Aware Influence Estimation While multi-checkpoint sampling with multi-aspect scoring ensures reliability and diversity, the influence of each selected self-distilled sample on base model fine-tuning can vary. Inspired by previous works (Pruthi et al., 2020; Xia et al., 2024), we introduce an optimizer-aware influence estimation method to further identify the most valuable samples by estimating the fine-tuning influence of the code instruction data they contain.

Concretely, based on the influence formulation (Pruthi et al., 2020), the influence of a self-distilled code instruction data $d = (q, s)$ on the prediction of a test instance t in a base model parameterized by θ can be estimated by computing the similarity between their gradients:

$$\mathrm{Inf}(d, t) \propto \mathrm{Sim}(\nabla l(d, \theta), \nabla l(t, \theta)). \qquad (3)$$

However, code generation tasks are inherently broad and diverse, and some of them may lack well-established benchmarks. To address this, we instead estimate the influence of d by computing its gradient similarity to the code instruction data $\{d^p = (q^p, s^p)\}$ from proprietary LLM samples $\mathcal{D}_p$. The idea is that proprietary LLMs (e.g., GPT-4o) have undergone extensive optimization through various strategies, making their distilled instruction data highly effective in improving model performance across diverse tasks.

Specifically, inspired by previous work (Xia et al., 2024), we first train an LLM-based reference model on the proprietary instruction data $\{d^p = (q^p, s^p)\}$ using LoRA (Hu et al., 2022), which allows for low-rank adaptation, significantly reducing trainable parameters and ensuring the efficiency for the following gradient computations. We then compute the gradient of each self-distilled instruction data d with respect to the LoRA parameters θ_{lora}, denoted as $\nabla l_{ref}(d, \theta_{lora})$. To further improve efficiency, following prior work (Park et al., 2023), we apply a projection matrix initialized with a Rademacher distribution to reduce gradient dimensionality, resulting in $\hat{\nabla} l_{ref}(d, \theta_{lora})$. According to the Johnson-Lindenstrauss Lemmas (Johnson et al., 1984), this transformation can preserve gradient distances while ensuring the usefulness of lower-dimensional features. Similarly, we compute the projected gradients for each proprietary instruction data d^p, denoted as $\hat{\nabla} l_{ref}(d^p, \theta_{lora})$. Finally, we approximate the influence of d by calculating its cosine similarity to the average gradient of $\{d^p\}$:

$$V(d) = \mathsf{Cosine}\Bigg(\hat{\nabla} l_{ref}(d, \theta_{lora}), \frac{1}{N_p}\sum_{i=1}^{N_p} \hat{\nabla} l_{ref}(d_i^p, \theta_{lora})\Bigg), \qquad (4)$$

where N_p is the size of $\{d^p\}$.

4 Experiments

4.1 Benchmarks

We employ multiple widely adopted benchmarks for a comprehensive evaluation, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) (along with their EvalPlus (Liu et al., 2023) versions), LiveCodeBench (V4) (Jain et al., 2024), and BigCodeBench (Zhuo et al., 2024). We assess model performance using the pass@1 metric.

4.2 Baselines

We compare SCoder with several powerful baselines, including two proprietary models: GPT-4-Turbo-20240409 (OpenAI, 2024a) and GPT-o1-Preview-20240912 (OpenAI, 2024b), as well as seven open-source models built on DeepSeek-Coder-6.7B-Base (Guo et al., 2024): DeepSeek-Coder-6.7B-Instruct, WaveCoder-Ultra-6.7B (Yu et al., 2023), MagicoderS-DS-6.7B (Wei et al., 2024), OpenCodeInterpreter-DS-6.7B (Zheng et al., 2024), AlchemistCoder-DS-6.7B (Song et al., 2024), InverseCoder-DS-6.7B (Wu et al., 2024), and WizardCoder-GPT-4-6.7B (Luo et al., 2024).

4.3 Implementation Details

We provide a simplified version of the implementation details here; a more detailed version can be found in Appendix C.

Small-Scale Data Synthesizer. We train Llama3.1-8B-Ins, Qwen2.5-Coder-7B-Ins, and Qwen2.5-Coder-14B-Ins as data synthesizers. Each model is initially trained on 10K GPT-4o data D_p, followed by two rounds of bootstrapping using

[2] https://evalplus.github.io/leaderboard.html

[3] https://livecodebench.github.io/leaderboard.html

[4] https://huggingface.co/spaces/bigcode/bigcodebench-leaderboard

Synthesizer	Data Size	HumanEval	MBPP	LiveCodeBench	BigCodeBench
DeepSeek-Coder-6.7B-Base					
None	0	47.6†	72.0†	16.2†	41.8†
Fine-Tuning DeepSeek-Coder-6.7B-Base on 40K Synthesized Data					
Llama3.1-8B-Instruct	0	60.4	64.7	16.5	42.1
+*Enhanced*	10K	64.2	69.3	17.3	42.8
+*1 Iter*	20K	65.5	71.1	17.4	43.1
+*2 iter*	40K	**67.4**	**73.4**	**17.8**	**43.5**
Qwen2.5-Coder-7B-Instruct	0	61.6	70.8	17.0	42.7
+*Enhanced*	10K	65.6	72.1	18.2	43.8
+*1 Iter*	20K	66.3	72.9	18.4	44.1
+*2 iter*	40K	**68.9**	**74.7**	**18.9**	**44.7**
Qwen2.5-Coder-14B-Instruct	0	65.3	73.7	18.7	43.2
+*Enhanced*	10K	67.5	75.8	19.4	44.5
+*1 Iter*	20K	68.4	76.3	19.3	45.1
+*2 iter*	40K	**70.1**	**76.5**	**19.7**	**45.9**

Table 2: Performance of code generation models built on instruction data generated by small synthesizers on HumanEval, MBPP, LiveCodeBench (Full), and BigCodeBench (Complete-Full). Data size refers to the amount of data used to train the synthesizer. † denotes results from the benchmark leaderboards[2][3][4].

Models	HumanEval		MBPP		LiveCodeBench		BCB (Comp)		BCB (Inst)	
	Base	Plus	Base	Plus	Full	Easy	Full	Hard	Full	Hard
Proprietary Models										
GPT-4-Turbo-20240409	90.2†	86.6†	85.7‡	73.3‡	42.0†	82.4†	58.2†	35.1†	48.2†	32.1†
GPT-o1-Preview-20240912	96.3†	89.0†	95.5†	80.2†	58.5†	94.1†	/	34.5†	/	23.0†
DeepSeek-Coder-6.7B-Base										
DeepSeek-Coder-6.7B-Base	47.6†	39.6†	72.0†	58.7†	16.2†	38.7†	41.8†	13.5†	/	/
Fine-Tuned Models based on DeepSeek-Coder-6.7B-Base										
DeepSeek-Coder-6.7B-Instruct	74.4†	71.3†	74.9†	65.6†	19.8†	45.8†	43.8†	15.5†	35.5†	10.1†
WaveCoder-Ultra-6.7B	75.0†	69.5†	74.9†	63.5†	19.7	46.8	43.7†	**16.9**†	33.9†	12.8†
MagicoderS-DS-6.7B	76.8†	71.3†	$\underline{79.4}^{\dagger}$	$\underline{69.0}^{\dagger}$	20.4	47.9	$\underline{47.6}^{\dagger}$	12.8†	36.2†	13.5†
OpenCodeInterpreter-DS-6.7B	77.4†	71.3†	76.5†	66.4†	18.9	46.6	44.6†	**16.9**†	37.1†	13.5†
AlchemistCoder-DS-6.7B	$\underline{79.9}^{\ddagger}$	$\underline{75.6}^{\ddagger}$	77.0‡	60.2‡	17.4	44.7	42.5	14.2	33.5	13.2
InverseCoder-DS-6.7B	$\underline{79.9}^{\ddagger}$	**76.8**‡	78.6‡	$\underline{69.0}^{\ddagger}$	20.3	46.6	45.7	14.9	35.4	9.5
WizardCoder-GPT-4-6.7B	77.4	73.8	75.4	64.8	21.0	49.6	45.1	15.5	37.3	10.8
SCoder-L-DS-6.7B	78.2	73.8	77.6	65.4	21.1	51.7	46.2	15.1	37.9	13.4
SCoder-Q7-DS-6.7B	78.7	74.3	79.1	66.5	$\underline{21.4}$	$\underline{52.2}$	47.4	15.5	$\underline{38.6}$	$\underline{14.5}$
SCoder-Q14-DS-6.7B	**80.5**	75.0	**81.0**	**69.3**	**22.2**	**52.6**	**49.2**	$\underline{16.2}$	**40.6**	**16.9**

Table 3: Performance comparison of different models on multiple code generation benchmarks. Three SCoder models are fine-tuned using data generated by our small synthesizers, where L, Q7, and Q14 denote three different synthesizers after two rounds of bootstrap. BCB, Comp, and Inst denote BigCodeBench, Complete, and Instruct. ‡ denotes results from the InverseCoder work (Wu et al., 2024). The best results are in **bold** and the second best results are underlined.

20K and 40K self-distilled data D_s, respectively. Training is performed with a learning rate of 1×10^{-5} and a global batch size of 128, while the temperature is set to 0.2 during inference.

SCoder. For a fair comparison, we first train DeepSeek-Coder-6.7B-Base on the 110K evol-codealpaca-v1 data for 2 epochs, and then fine-tune it for 3 epochs on the 60K data synthesized by our small synthesizers to obtain SCoder. The 110K evol-codealpaca-v1 data is also widely used in baselines, as shown in Table 5.

4.4 Main Results

As shown in Table 2, our proposed method significantly enhances the instruction data synthesis capabilities of small models with only two rounds

Models	HumanEval	MBPP	LiveCodeBench	BigCodeBench
SCoder-Q7-DS-6.7B	**78.7**	**79.1**	**21.4**	**47.4**
w/o multi-checkpoint sampling	74.9	73.8	18.7	44.3
w/o multi-aspect scoring	72.3	<u>76.7</u>	<u>19.9</u>	<u>45.5</u>
w/o optimizer-aware influence estimation	<u>75.1</u>	74.4	18.2	43.2
SCoder-Q14-DS-6.7B	**80.5**	**81.0**	**22.2**	**49.2**
w/o multi-checkpoint sampling	75.6	74.4	20.4	<u>46.3</u>
w/o multi-aspect scoring	74.9	<u>75.8</u>	<u>20.8</u>	45.1
w/o optimizer-aware influence estimation	<u>76.1</u>	74.9	20.0	44.8

Table 4: Ablation study on HumanEval, MBPP, LiveCodeBench (Full), and BigCodeBench (Complete-Full). The best results are in **bold** and the second best results are underlined.

Model	Common Data	Specific Data
WizardCoder-GPT-4	110K (GPT-4)	0K
WaveCoder-Ultra		20K (GPT-4)
MagicoderS		75K (GPT-3.5)
AlchemistCoder		>80K (GPT-3.5)
InverseCoder		90K (self-generated)
SCoder (ours)		60K (small model-generated)

Table 5: Comparison of data used by different models. The source of the data is indicated in parentheses.

of bootstrap, regardless of their model family or scale. For example, the fine-tuning performance of the 40K data synthesized by Llama3.1-8B-Ins on the base model achieves a 5.0% improvement on HumanEval and a 5.9% improvement on MBPP after two rounds of bootstrap. This demonstrates that our approach, leveraging well-designed sampling and filtering strategies, enables small models to acquire self-distilled data synthesis samples with strong reliability, broad diversity, and high value. As a result, they progressively evolve into effective data synthesizers while minimizing dependence on proprietary LLM distillation.

Furthermore, Table 3 shows that SCoder, trained on data generated by bootstrapped small-scale data synthesizers, outperforms or matches other state-of-the-art open-scource baselines across multiple benchmarks. For example, SCoder-Q14-DS-6.7B surpasses the best open-source baselines by 5.9% and 9.7% on average in the challenging LiveCodeBench and BigCodeBench, respectively. Notably, the open-source baselines typically utilize a larger amount of proprietary LLM-distilled instruction data as listed in Table 5, further validating the effectiveness of our method in constructing strong small-scale data synthesizers.

4.5 Ablation Study

We conduct ablation studies based on SCoder-Q7-DS-6.7B and SCoder-Q14-DS-6.7B. The results presented in Table 4 demonstrate the importance of our extensive sampling and refined filtering strategies.

First, without multi-checkpoint sampling (i.e., sampling an equal number of outputs solely from the last checkpoint of the previous round), the performance of both code generation models on HumanEval and LiveCodeBench drops by at least 4.8% and 8.1%, respectively. This indicates that a limited sampling space reduces the likelihood of obtaining high-quality self-distilled data, thereby hindering the effectiveness of the bootstrap process. Furthermore, when either multi-aspect scoring or optimizer-aware influence estimation is removed from the data selection process, the performance on MBPP and BigCodeBench drops by up to 7.5% and 8.9%, respectively. This highlights that both strategies are essential for ensuring the reliability, diversity, and value of self-distilled data, and removing either significantly impacts the overall effectiveness.

4.6 Data Scaling

To further evaluate the data synthesis quality of small data synthesizers, we investigate the data scaling law using the bootstrapped Qwen2.5-Coder-14B-Ins. As shown in Figure 3, increasing the data size leads to significant improvements of the code generation model fine-tuned on DeepSeek-Coder-6.7B-Base, surpassing DeepSeek-Coder-6.7B-Instruct on most benchmarks. This further validates the effectiveness of our approach in constructing high-quality small-scale data synthesizers.

4.7 Further Discussion

In this section, we provide a more fine-grained analysis of the effectiveness of our method.

First, we compare the impact of different selec-

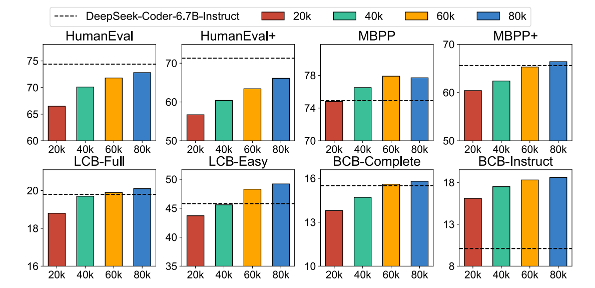

Figure 3: Impact of data scaling. The dashed lines represent the performance of DeepSeek-Coder-6.7B-Instruct across various benchmarks.

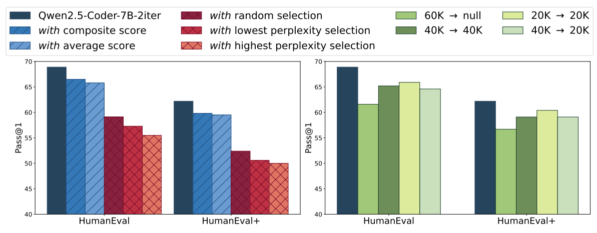

Figure 4: Comparison of different selection methods and the number of self-distilled data used in different bootstrap rounds. The y-axis denotes the performance of the code generation models fine-tuned on 40K synthesized data.

tion strategies during the bootstrap process. As shown in the left of Figure 4, for multi-aspect scoring, replacing the aggregated score with either the raw composite score from the scorer or the simple average of scores leads to a decline in the synthesizer's data synthesis performance. Moreover, substituting the optimizer-aware influence estimation with alternative selection methods, such as random selection or lowest/highest perplexity selection, results in an even more substantial performance drop. These findings highlight the effectiveness of our selection strategy in identifying reliable and high-value self-distilled samples, thereby ensuring the success of the bootstrap process.

Second, as the synthesizer's capability improves with more bootstrap iterations, we progressively increase the number of self-distilled samples used in training across two iterations (20K → 40K). Here, we compare different settings, including removing multi-round iteration (60K → Null), progressively decreasing the sample size (40K → 20K), increasing the sample size in the first iteration (40K → 40K), and decreasing the sample size in the second iteration (20K → 20K). As shown in the right of Figure 4, in all cases, performance declines, indicating that a well-balanced and progressively increasing data schedule plays a crucial role in maximizing the effectiveness of the bootstrap process.

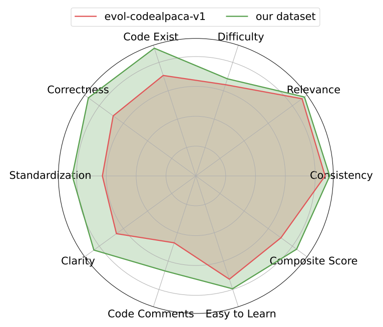

Figure 5: Quality comparison between the evol-codealpaca-v1 dataset and our synthesized dataset.

4.8 Data Quality Analysis

To further validate the quality of data generated by the synthesizers, we sampled 100 code instruction data from evol-codealpaca-v1 and the bootstrapped Qwen2.5-Coder-14B-Ins, respectively and use GPT-4o-20240513 and GPT-4-turbo-20240409 to score the data across 10 aspects based on the prompt provided in Appendix D. The average results, shown in Figure 5, demonstrate that our synthesized data achieves higher scores across all aspects, further confirming the effectiveness of our method in building high-quality small-scale code instruction synthesizers.

5 Conclusion

In this paper, we propose a multi-round self-distillation bootstrap method to fully unlock the data synthesis potential of small-scale LLMs, transforming them into powerful instruction data synthesizers while reducing reliance on proprietary LLMs and minimizing costs. We design a multi-checkpoint sampling and multi-aspect scoring method to ensure the diversity and reliability of self-distilled samples, followed by an optimizer-aware influence estimation method to select high-value ones for training. We validate our method on Llama3.1-8B-Ins and Qwen2.5-Coder-7B/14B-Ins, demonstrating their effectiveness as data synthesizers. Based on the data generated by these small-scale synthesizers, we introduce SCoder, a family of code generation models that achieves strong performance on HumanEval (+), MBPP (+), LiveCodeBench, and BigCodeBench, showcasing the potential of small models in code instruction data synthesis.

6 Limitations

Although our proposed multi-round self-distillation bootstrap method shows effectiveness in fully exploiting small models' potential for code instruction data synthesis, limitations remain. For example, while the method demonstrates strong performance across benchmarks, the synthesis process could still benefit from further exploration into other data synthesis methods, such as Self-Instruct (Wang et al., 2023) and Evol-Instruct (Xu et al., 2024), which we plan to explore in future work.

References

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, and et al. 2023. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, and et al. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, and et al. 2023. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113.

Bosheng Ding, Chengwei Qin, Ruochen Zhao, Tianze Luo, Xinze Li, Guizhen Chen, Wenhan Xia, Junjie Hu, Anh Tuan Luu, and Shafiq Joty. 2024. Data augmentation using llms: Data perspectives, learning paradigms and challenges. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 1679–1705. Association for Computational Linguistics.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and et al. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

William B Johnson, Joram Lindenstrauss, et al. 1984. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, and et al. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and et al. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and et al. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

OpenAI. 2024a. Gpt-4. `https://openai.com/index/gpt-4-research/`.

OpenAI. 2024b. Learning to reason with llms. `https://openai.com/index/learning-to-reason-with-llms/`.

Sung Min Park, Kristian Georgiev, Andrew Ilyas, Guillaume Leclerc, and Aleksander Madry. 2023. TRAK: attributing model behavior at scale. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 27074–27113. PMLR.

Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. 2020. Estimating training data influence by tracing gradient descent. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, and et al. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Zifan Song, Yudong Wang, Wenwei Zhang, Kuikun Liu, Chengqi Lyu, Demin Song, Qipeng Guo, Hang Yan, Dahua Lin, Kai Chen, and Cairong Zhao. 2024. Alchemistcoder: Harmonizing and eliciting code capability by hindsight tuning on multi-source data. *Preprint*, arXiv:2405.19265.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. 2022. Learning to summarize from human feedback. *Preprint*, arXiv:2009.01325.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2023. One embedder, any task: Instruction-finetuned text embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 1102–1121. Association for Computational Linguistics.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, et al. 2025. Epicoder: Encompassing diversity and complexity in code generation. *arXiv preprint arXiv:2501.04694*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *CoRR*, abs/2407.05700.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. 2024. LESS: selecting influential data for targeted instruction tuning. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. 2024. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12834–12859. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, and et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

A Code Snippet Gathering

To ensure the validity of our experimental results, we first construct a clean and noise-free code snippet pool that serves as the foundation for code instruction data synthesis. Specifically, inspired by the data preprocessing pipeline of StarCoder2 (Lozhkov et al., 2024), we follow the steps below to construct the code snippet pool $\mathcal{C}$ from the Stack V1, a collection of source code in over 300 programming languages.

- **Code Snippet Extraction:** We first extract all Python functions that include docstrings from the Stack V1 dataset. To ensure a high level of diversity while minimizing redundancy, we perform near-deduplication using MinHash, Locality-Sensitive Hashing (LSH), and Jaccard similarity with a threshold of 0.5.

- **Invalid Function Filtering:** We remove any functions that do not contain a return statement or contain syntax errors. Additionally, we supplement the remaining functions with necessary dependency packages and remove functions that import problematic packages (e.g., os or sys), which could lead to issues in execution.

- **Quality Evaluation:** We further evaluate the remaining functions using the StarCoder2-15B as a classifier to filter out examples with bad documentation or low-quality code.

- **Data Decontamination:** Finally, we employ an n-gram filtering technique to remove any functions that contain solutions or prompts from the benchmarks used in this work.

B Task Category

Following the Magicoder (Wei et al., 2024), we use the following ten task categories for classifying code snippets: "Algorithmic and Data Structure Problems", "Mathematical and Computational Problems", "Database and SQL Problems", "System Design and Architecture Problems", "Security and Cryptography Problems", "Performance Optimization Problems", "Web Problems", "Domain Specific Problems", "User Interface and Application Design Problems", and "Data Science and Machine Learning Problems".

C Implementation Details

Multi-Aspect Scorer. We sample 2.5K code instruction data from Llama3.1-8B-Ins, Qwen2.5-Coder-7B-Ins, Qwen2.5-Coder-14B-Ins, and the evol-codealpaca-v1 dataset (Luo et al., 2024), respectively. Using the prompt in Appendix D, we distill scoring results from GPT-4o-20240806 from $Z = 10$ aspects and train Llama3.1-8B-Base for 3 epochs with a learning rate of 1×10^{-5} and a global batch size of 64, obtaining the multi-aspect scorer. During inference, we set the temperature to 0. To derive the weight vector $\mathbf{w}$, we conduct $K = 20$ experiments and evaluate the results on LiveCodeBench (202410-202501).

Reference Model. We train Llama3.1-8B-Base as the reference model on 10K GPT-4o-20240806 data ($\mathcal{D}_p$) for 3 epochs with a learning rate of 2×10^{-5} and a global batch size of 32. For LoRA configurations, we set lora_r $= 128$, lora_alpha $= 512$, and apply LoRA to the target modules: q_proj, k_proj, v_proj, and o_proj.

Small-Scale Data synthesizer. We train Llama3.1-8B-Ins, Qwen2.5-Coder-7B-Ins, and Qwen2.5-Coder-14B-Ins as data synthesizers. Each model is first trained on 10K GPT-4o-20240806 data ($\mathcal{D}_p$) before undergoing two rounds of bootstrapping. In each round, we sample $N = 3$ data synthesis samples from $M = 5$ different checkpoints, respectively. The first round trains on 20K self-distilled samples, while the second round uses 40K. Each training runs for 3 epochs with a learning rate of 1×10^{-5} and a batch size of 128. During inference, we set the temperature to 0.2.

SCoder. To maintain consistency with the baselines, we use DeepSeek-Coder-6.7B-Base as the base model and distill 60K code instruction samples from each of the three bootstrapped small-scale synthesizers. For a fair comparison, we also incorporate the evol-codealpaca-v1 dataset, an open-source Evol-Instruct implementation with approximately 110K data, widely used in baselines such as WizardCoder-GPT-4, WaveCoder-Ultra, MagicoderS, AlchemistCoder, and InverseCoder. The training data size comparison across different models is presented in Table 5.

To obtain SCoder, we first fine-tune DeepSeek-Coder-6.7B-Base on the 110K evol-codealpaca-v1 data for 2 epochs with an initial learning rate of 5×10^{-5} and a global batch size of 512. We then further fine-tune it on the 60K small model-generated data for 3 epochs with an initial learning rate of 1×10^{-5} and a batch size of 64. Both phases of training utilize a linear learning rate scheduler with a 0.05 warmup ratio and the AdamW optimizer. Training is conducted on 16 A100-80G GPUs.

D Prompts

The data synthesis prompt is inspired by Wei et al. (2024) and is shown in Figure 6. The multi-aspect scoring prompt is inspired by Hui et al. (2024) and is shown in Figure 7.

Data Synthesis Prompt

Please gain inspiration from the following random code snippet to create a high-quality programming problem. Present your output in two distinct sections: [Problem Description] and [Solution].
Code snippet for inspiration:
```
<<code>>
```
Guidelines for each section:
1. [Problem Description]: This should be **completely self-contained**, providing all the contextual information one needs to understand and solve the problem. Assume common programming knowledge, but ensure that any specific context, variables, or code snippets pertinent to this problem are explicitly included.
2. [Solution]: Provide a comprehensive and correct solution that accurately addresses the [Problem Description] you have provided. First, analyze the problem, then provide the specific code, and finally, explain the code.

Figure 6: Data Synthesis Prompt.

Multi-Aspect Scoring Prompt

You are responsible for training a large language model with coding abilities. Given a code instruction and a corresponding response, you need to evaluate the quality of this code data pair. Score the data based on its value for training a large code language model, using a scale from 0 to 9, where 0 represents the worst and 9 represents the best.

Constraints:
1. The evaluation score must be judged among these ten scores [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], and decimal scores cannot appear.
2. The output should have [Reason] part. You need to Generate the evaluation reason first and then generate the score.
3. You should concentrate on the quality only. The following irrelevant matters **should not** influence the quality evaluation.
   3.1 whether the data is domain-specific or not should not be considered, given that the code language model need to deal with inputs with diverse domains.
   3.2 whether the data contains non-English content or not should not be considered, given the code language model may need to deal with multilingual inputs.
   3.3 whether the data has timeliness statements or not should not be considered, given the code language model may need to deal with issues with timeliness.
4. For each data pair, evaluate the following scoring criteria individually and provide an overall composite score:
   4.1 Consistency: Whether Q&A are consistent and correct for fine-tuning.
   4.2 Relevance: Whether Q&A are related to the computer field.
   4.3 Difficulty: Whether Q&A are sufficiently challenging.
   4.4 Code Exist: Whether the code is provided in question or answer.
   4.5 Code Correctness: Evaluate whether the provided code is free from syntax errors and logical flaws.
   4.6 Code Standardization: Consider factors like proper variable naming, code indentation, and adherence to best practices.
   4.7 Code Clarity: Assess how clear and understandable the code is. Evaluate if it uses meaningful variable names, proper comments, and follows a consistent coding style.
   4.8 Code Comments: Evaluate the presence of comments and their usefulness in explaining the code's functionality.
   4.9 Easy to Learn: Determine its educational value for a student whose goal is to learn basic coding concepts.
   4.10 Composite Score: Considering the above factors, an overall quality score is assigned to the data pair, weighted by the importance of each criterion.

The instruction and response you need to evaluate is as following:

```
[Instruction]
<<instruction>>
[Instruction End]
[Response]
<<response>>
[Response End]
```

Your response should be in the following format:

```
[Reason]
{reason}
[Score]
{
   "Consistency": {score},
   "Relevance" : {score},
   "Difficulty": {score},
   "Code Exist" : {score},
   "Code Correctness": {score},
   "Code Standardization" : {score},
   "Code Clarity": {score},
   "Code Comments" : {score},
   "Easy to Learn": {score},
   "Composite Score" : {score}
}
[End]
```

Figure 7: Multi-Aspect Scoring Prompt.