SCoder: Iterative Self-Distillation for Bootstrapping Small-Scale Data Synthesizers to Empower Code LLMs

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 iterative self-distillation approach to bootstrap small-scale LLMs, transforming them into powerful synthesizers that reduce reliance on proprietary LLMs and minimize costs. Concretely, in each iteration, to obtain diverse and high-quality self-distilled data, we design multi-checkpoint sampling and multi-aspect scoring strategies for initial data selection. Furthermore, to identify the most influential samples, we introduce a gradient-based influence estimation method for final data filtering. Based on the code instruction datasets from the small-scale synthesizers, we develop SCoder, a family of code generation models fine-tuned from DeepSeek-Coder. SCoder models achieve 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.

Typically, instruction tuning on base models (e.g., DeepSeek-Coder-Base) is a crucial step in developing 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-intensive issue caused by the distillation of large-scale instruction data from the proprietary LLMs like GPT-3.5 and GPT-4.

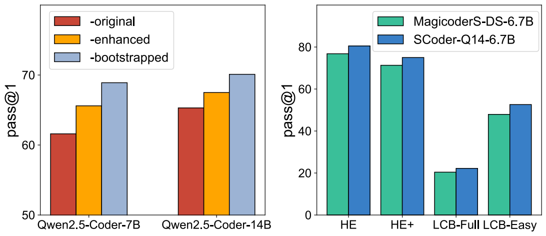

Figure 1: **Left**: The performance of code generation models on HumanEval using data provided by different synthesizers (Qwen2.5-Coder-7B or -14B). **Right**: The performance of our SCoder and the baseline. SCoder uses 60K instruction data generated by a small-scale synthesizer, and the baseline uses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

In this paper, we explore the potential of relatively small-scale (7B, 8B, and 14B) open-source LLMs as synthesizers for code instruction data construction. 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-intensive 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 a progressive self-distillation method that iteratively bootstraps the code instruction data synthesis capability of small-scale LLMs. Specifically, starting with an enhanced synthesizer, we employ a two-step approach in each iteration to obtain high-quality self-distilled data synthesis samples for further training. First, we design *multi-checkpoint sampling* and *multi-aspect scoring* strategies to obtain diverse and reliable self-distilled samples. Then, we introduce a *gradient-based influence estimation* method to further select the influential ones by comparing the gradients induced by self-distilled samples with those induced by 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 datasets provided by our small-scale synthesizers, we develop 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 that use the instruction data from proprietary LLMs. Overall, our contributions can be summarized as follows:

- We propose a novel iterative self-distillation approach that transforms small-scale LLMs into effective synthesizers of code instruction data. Using the instruction data generated by these synthesizers, we train a family of code generation models (SCoder), which achieve performance comparable to that of models relying on proprietary LLM-distilled data.

- To obtain diverse and high-quality self-distilled data, we design multi-checkpoint sampling and multi-aspect scoring strategies for initial data selection. To further identify the most influential samples, we introduce a gradient-based influence estimation method for final data filtering.

- We fine-tune the code generation models (SCoder) based on the datasets generated 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 made significant strides 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 shown impressive performance across various code generation benchmarks. Meanwhile, open-source models like 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. These models not only enhance code generation capabilities but also promote more efficient and automated software development.

Typically, such models are developed through continual pre-training (Rozière et al., 2023), followed by supervised fine-tuning (SFT) (Yu et al., 2023). While pre-training utilizes large-scale, unannotated code corpora, SFT relies on high-quality labeled instruction data, whose construction remains a key challenge (Ding et al., 2024).

2.2 Code Instruction Data Synthesis

Creating diverse and complex code instruction data is challenging and requires domain expertise. While human-written datasets used in OctoPack (Muennighoff et al., 2024) and PIE (Shypula et al., 2024) are effective, they are labor-intensive and hard to scale. To address this, many recent works leverage powerful proprietary LLMs for automatic instruction generation. For example, Code Alpaca (Chaudhary, 2023) adopts Self-Instruct (Wang et al., 2023), WizardCoder (Luo et al., 2024) uses Evol-Instruct (Xu et al., 2024), and Magicoder (Wei et al., 2024) utilizes OSS-Instruct to create realistic, diverse programming tasks from open-source code. Similarly, WaveCoder (Yu et al., 2023) introduces a generator-discriminator framework, while OpenCodeInterpreter (Zheng et al., 2024) leverages user-LLM-compiler interactions to synthesize multi-turn instruction data. Despite their effectiveness, these approaches often depend on costly proprietary models (Wu et al., 2024). In this work, we explore using small-scale open-source LLMs to generate high-quality code instruction data more cost-effectively, reducing reliance on expensive proprietary models while maintaining strong 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 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 preprocessing 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 an iterative bootstrap approach to continuously 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 the code snippet pool $\mathcal{C}$ are provided in Appendix D and A, respectively.

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.

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 the code generation model fine-tuned on 40K code instruction data provided by different synthesizers.

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 Iterative 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 iterative self-distillation. Specifically, we start with the mentioned enhanced synthesizers, considering this as the 0-th iteration of the bootstrap. Then, in 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 it-

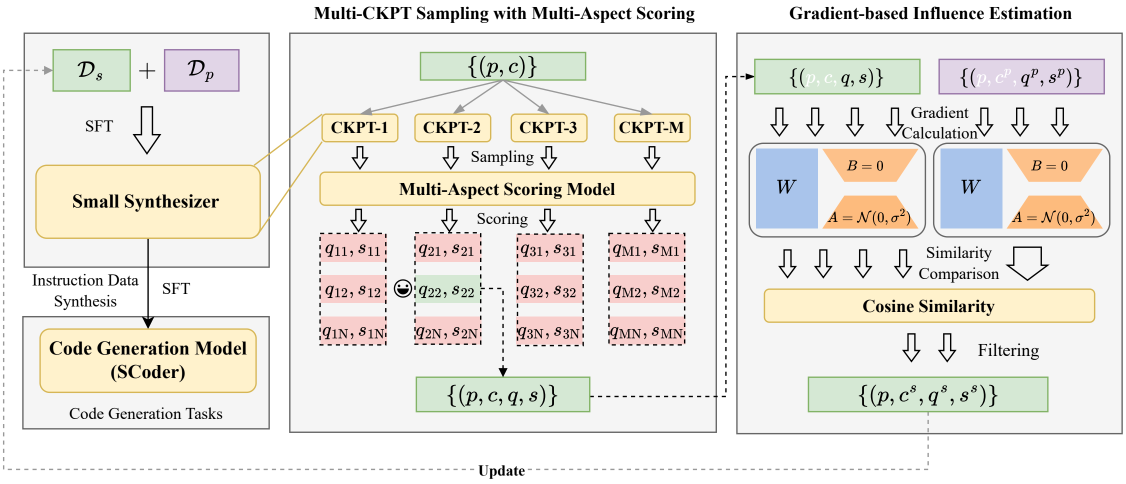

Figure 2: **Overview of our iterative self-distillation bootstrap method.** In each iteration, we sample outputs from multiple checkpoints and evaluate them with a multi-aspect scorer for diversity and reliability. We then use a gradient-based influence estimation method to select the most influential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

eration. Next, to further identify the most influential samples, we introduce a gradient-based 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 illustrated in Figure 2, and a detailed theoretical analysis of the iterative self-distillation is provided in Appendix E.

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 the 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:

$$Score_{ij} = \sum_{z=1}^{Z} w^z x_{ij}^z. \qquad (1)$$

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:

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

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.

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

Gradient-based Influence Estimation While multi-checkpoint sampling with multi-aspect scoring ensures diversity and reliability, the influence of each selected self-distilled sample on the fine-tuning of the base model can vary. Inspired by previous works (Pruthi et al., 2020; Xia et al., 2024), we introduce a gradient-based 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:

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

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 $\{d^p\}$:

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

where N_p is the size of $\{d^p\}$. Eventually, the data samples with the highest influence will be selected and used for training.

4 Experiments

4.1 Benchmarks

We evaluate model performance using the pass@1 metric on several standard benchmarks: 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). Evaluation strictly follows each benchmark's official settings and prompts.

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 first trained on 10K GPT-4o data D_p, then bootstrapped with 20K and 40K self-distilled

[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 (target 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†	79.4†	69.0†	20.4	47.9	47.6†	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	79.9‡	75.6‡	77.0‡	60.2‡	17.4	44.7	42.5	14.2	33.5	13.2
InverseCoder-DS-6.7B	79.9‡	**76.8**‡	78.6‡	69.0‡	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	21.4	52.2	47.4	15.5	38.6	14.5
SCoder-Q14-DS-6.7B	**80.5**	75.0	**81.0**	**69.3**	**22.2**	**52.6**	**49.2**	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 iterations 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.

data D_s. Training uses a learning rate of 1×10^{-5}, global batch size 128, and inference temperature 0.2.

SCoder. For fair comparison, we train DeepSeek-Coder-6.7B-Base on 110K evol-codealpaca-v1 for 2 epochs, then fine-tune it on 60K synthesized data (from small synthesizers) for 3 epochs to obtain SCoder. The 110K data is commonly used in baselines (Table 5). More target model results are in Appendix H.

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 iterations of bootstrap, regardless of their model family

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	76.7	19.9	45.5
w/o gradient-based influence estimation	75.1	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	46.3
w/o multi-aspect scoring	74.9	75.8	20.8	45.1
w/o gradient-based influence estimation	76.1	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.

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 iterations 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 broad diversity, strong reliability, and high influence. 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-source 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. A more detailed cost efficiency analysis of our method is provided in Appendix F.

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 iteration), 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 gradient-based 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 and influence 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-

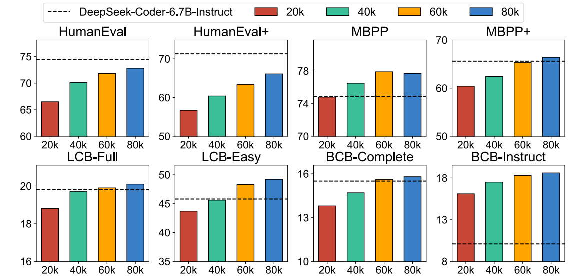

Figure 3: Impact of data scaling. The dashed lines represent the performance of DeepSeek-Coder-6.7B-Instruct across various benchmarks.

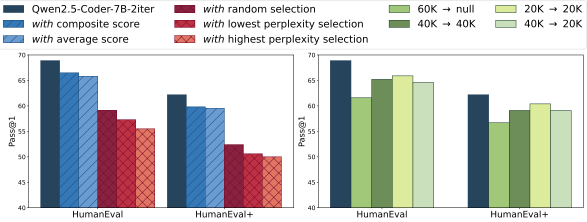

Figure 4: Comparison of different selection methods and the number of self-distilled data used in different bootstrap iterations. The y-axis denotes the performance of the code generation models fine-tuned on 40K synthesized data.

tion strategies during the bootstrap process. As shown on 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 gradient-based 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 influential 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 on 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.

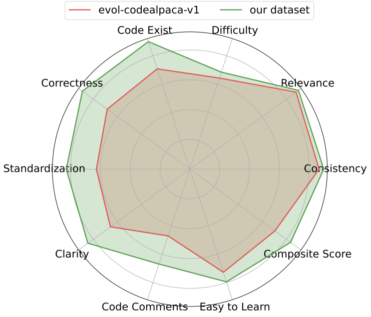

Figure 5: Quality comparison between the evol-codealpaca-v1 dataset and our synthesized dataset.

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 used 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 an iterative self-distillation bootstrap method to fully unlock the data synthesis potential of small-scale LLMs, transforming them into powerful code instruction data synthesizers while reducing reliance on proprietary LLMs and minimizing costs. We design multi-checkpoint sampling and multi-aspect scoring strategies to ensure the diversity and reliability of self-distilled samples, followed by a gradient-based influence estimation method to select influential 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

Despite the demonstrated effectiveness of our iterative self-distillation bootstrap method in fully leveraging the code instruction data synthesis capability of small-scale LLMs, certain limitations persist. For example, the current synthesis framework does not incorporate alternative data generation paradigms, such as Self-Instruct (Wang et al., 2023) and Evol-Instruct (Xu et al., 2024), which have shown promise in previous work. Investigating the integration of such approaches constitutes an important direction for future work.

Furthermore, this study limits its empirical validation to the domain of code generation. While the underlying methodology may apply to other domains, several challenges arise. For example, although synthesizers can efficiently generate large-scale code instruction data by leveraging vast amounts of open-source code snippets, achieving efficient data synthesis for other tasks may require additional consideration and tailored design. Therefore, further exploration is needed to fully assess feasibility in other domains, and we plan to present related findings 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. We further investigate the impact of different reference models on data selection in Appendix G.

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 iterations of bootstrapping. In each iteration, we sample $N = 3$ data synthesis samples from $M = 5$ different checkpoints, respectively. The first iteration trains on 20K self-distilled samples, while the second iteration 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.

E Theoretical Analysis of Iterative Self-Distillation

In this section, we provide a rigorous theoretical analysis of the iterative self-distillation framework from two perspectives: convergence behavior and its interpretation in terms of Nash equilibrium and the exploration-exploitation trade-off.

E.1 Problem Setup

Let $(\mathcal{M}, \|\cdot\|)$ be a complete metric space representing the space of model parameters. Let $M_0 \in \mathcal{M}$ be a fixed initial model. Define the data generation process as a mapping $\mathcal{G} : \mathcal{M} \to \mathcal{P}$, where $\mathcal{P}$ denotes the space of data distributions. The training operator is defined as $\mathcal{T} : \mathcal{M} \times \mathcal{P} \to \mathcal{M}$, mapping a model and a dataset to an updated model.

At each self-distillation iteration i, the process proceeds as follows:

$$D_i = \mathcal{G}(M_i), \tag{5}$$

$$M_{i+1} = \mathcal{T}(M_0, D_i). \tag{6}$$

where D_i is the data generated by model M_i, and each new model M_{i+1} is trained from scratch using the fixed initialization M_0 and dataset D_i.

E.2 Convergence Analysis

We analyze the convergence behavior of the model sequence $\{M_i\}$ by examining the composed operator $\Phi(M) = \mathcal{T}(M_0, \mathcal{G}(M))$, which encapsulates the entire update process at each iteration of self-distillation. This operator provides a clear description of how the model M evolves after one iteration of self-distillation, starting from the fixed model M_0.

Assumptions: We impose the following assumptions:

- **(A1)** *Training Lipschitz Continuity:* There exists $L_T > 0$ such that for all $D, D' \in \mathcal{P}$, the training process satisfies:

$$\|\mathcal{T}(M_0, D) - \mathcal{T}(M_0, D')\| \leq L_T \|D - D'\|. \tag{7}$$

- **(A2)** *Data Generation Lipschitz Continuity:* There exists $L_G > 0$ such that for all $M, M' \in \mathcal{M}$, the data generation process satisfies:

$$\|\mathcal{G}(M) - \mathcal{G}(M')\| \leq L_G \|M - M'\|. \tag{8}$$

- **(A3)** *Contraction Condition:* The product of the Lipschitz constants satisfies:

$$L_T L_G < 1. \tag{9}$$

Under assumptions (A1) and (A2), we can establish the following lemma: The composed operator $\Phi(M) = \mathcal{T}(M_0, \mathcal{G}(M))$ is Lipschitz continuous with a constant of $L_T L_G$. Specifically, for any two models M and M', we have the following inequality:

$$\begin{aligned} &\|\Phi(M) - \Phi(M')\| \\ =&\|\mathcal{T}(M_0, \mathcal{G}(M)) - \mathcal{T}(M_0, \mathcal{G}(M'))\| \\ \leq& L_T \|\mathcal{G}(M) - \mathcal{G}(M')\| \\ \leq& L_T L_G \|M - M'\|. \end{aligned} \tag{10}$$

Now, we impose the contraction condition (A3), which ensures that Φ is a contraction mapping. Since $L_T L_G < 1$, we can apply Banach's Fixed-Point Theorem to guarantee the existence of a unique fixed point $M^* \in \mathcal{M}$ such that $M^* = \Phi(M^*)$. Given this, we can analyze the convergence of the model sequence $\{M_i\}$, where $M_{i+1} = \Phi(M_i)$. For any $i \geq 0$, the distance between M_{i+1} and M^* is given by:

$$\begin{aligned} \|M_{i+1} - M^*\| &= \|\Phi(M_i) - \Phi(M^*)\| \\ &\leq L_T L_G \|M_i - M^*\|. \end{aligned} \tag{11}$$

By recursively applying this inequality, we obtain:

$$\|M_{i+1} - M^*\| \leq (L_T L_G)^i \|M_0 - M^*\|. \tag{12}$$

Since $L_T L_G < 1$, the factor $(L_T L_G)^i$ decays exponentially, and thus the sequence $\{M_i\}$ converges to M^* at a linear rate.

Therefore, under the assumptions of Lipschitz continuity of both the training and data generation processes, and the contraction condition, the model sequence converges to a unique fixed point M^*, with linear convergence determined by the product of the Lipschitz constants $L_T L_G$.

E.3 Nash Equilibrium Interpretation

Beyond convergence, the fixed point M^* of the self-distillation process can also be interpreted through a game-theoretic lens as a *Nash equilibrium*.

Consider each iteration of self-distillation as a two-player interaction:

- **Teacher:** A model $M \in \mathcal{M}$ that generates synthetic data via $\mathcal{G}(M)$.
- **Student:** A fixed model M_0 that is retrained on the teacher's generated data via $\mathcal{T}(M_0, \mathcal{G}(M))$.

The process evolves according to the update rule in Equation 5 and 6 where the teacher at iteration i is M_i, and the student is always initialized as M_0. The student updates its parameters based on the synthetic data provided by the teacher, effectively defining a best-response map from the teacher's strategy to a new model.

At convergence, the fixed point M^* satisfies:

$$M^* = \mathcal{T}(M_0, \mathcal{G}(M^*)), \tag{13}$$

which indicates that when the teacher generates data using M^*, retraining the student M_0 on that data simply reproduces the same model M^*. Thus, neither the teacher nor the student can unilaterally change their behavior to improve the outcome, satisfying the condition for a Nash equilibrium.

This perspective emphasizes that iterative self-distillation converges to a stable teacher–student pair, where the synthetic data and the resulting trained model are mutually consistent.

E.4 Exploration–Exploitation Trade-off

The iterative nature of self-distillation inherently embeds an exploration–exploitation mechanism.

- **Exploration:** In each iteration i, the teacher model M_i generates a new dataset $D_i = \mathcal{G}(M_i)$, which may differ significantly from previous iterations. This promotes exploration of new data distributions, especially in the early stages when M_i is far from convergence.

Data Synthesizer	HE	LCB-V4-Full
Llama3.1-8B	60.4	16.5
+2 iter	67.4	17.8
+3 iter	67.2	17.9
Qwen2.5-Coder-7B	61.6	17.0
+2 iter	68.9	18.9
+3 iter	69.1	18.8

Table 6: Finetuning performance of DeepSeek-Coder-6.7B-Base on 40K data synthesized by different synthesizers.

- **Exploitation:** At every iteration, the student model is always retrained from the fixed initialization M_0. This exploits prior knowledge encoded in M_0, focusing learning on the current data D_i.

As training progresses, the diversity of generated data typically decreases, and the model converges to a stable state M^*. In this sense, the process naturally transitions from high-entropy exploration to low-entropy exploitation. This dynamic provides a theoretical rationale for the empirical success of iterative self-distillation.

E.5 Discussion

Although the convergence and equilibrium are guaranteed under idealized assumptions (e.g., Lipschitz continuity, contraction property), in practical scenarios with non-convex models and imperfect optimization, strict convergence is not guaranteed. However, our empirical results suggest that the self-distillation process stabilizes in practice and leads to consistently improved model performance, as shown in Table 2.

Furthermore, we extended the self-distillation process to three iterations. In the third iteration, we used 40K self-distilled samples for training. As shown in Table 6, the performance of the synthesizers becomes stable when the number of self-distillation iterations reaches two or more, indicating that additional iterations yield diminishing returns while maintaining strong generation quality.

F Cost Efficiency of Our Method

In this section, we detail the cost advantages of our proposed approach, which relies on training a lightweight data synthesizer rather than directly distilling a proprietary large language model (LLM). Our method significantly reduces reliance on expensive LLM queries, improving both efficiency and accessibility.

Specifically, we use only 10K proprietary LLM samples during the initial bootstrapping phase. This is a substantial reduction compared to prior works, which typically require 150K–200K proprietary samples, as shown in Table 5. By contrast, once the bootstrapped synthesizer is trained, we can generate high-quality instruction data at scale without further calls to proprietary models.

The main computational cost of our method lies in fully fine-tuning the data synthesizer. In comparison, model inference (for sampling and multi-aspect scoring) and gradient similarity calculations are relatively lightweight. For instance, constructing the gradient library for each iteration takes approximately 3 hours on a single NVIDIA A100 80GB GPU.

Taking Qwen2.5-Coder-7B-Instruct as an example, we fine-tuned on 110K self-distilled samples throughout the entire bootstrap process, which took around 6.5 hours on 8× A100 80GB GPUs. Based on Google Cloud's official pricing[5], the total cost is estimated to be only $263.58. In contrast, using proprietary model APIs such as the GPT-4o-20240806 API for instruction synthesis incurs significantly higher costs; given average input/output lengths of 253 and 752 tokens respectively (as statistically measured from 10K distilled samples from the proprietary model), the same budget would only allow for generating approximately 30K samples. This highlights the efficiency of our approach: once trained, the synthesizer enables large-scale data generation at a fraction of the cost.

G Influence of Different Reference Models

In our main experiments, we primarily used Llama3.1-8B-Base as the reference model to compute gradient-based influence scores for guiding data selection. To assess whether the choice of reference model significantly impacted the outcome, we conducted additional experiments using different reference models while keeping the data synthesizer (Qwen2.5-Coder-7B-Instruct) and the target model (DeepSeek-Coder-6.7B-Base) unchanged.

We trained the data synthesizers for 2 iterations and used them to generate 40K code instruction data for training the target model. As shown in

[5] https://cloud.google.com/products/calculator

Reference Model	HE	LCB-V4-Full
Llama3.1-8B	68.9	18.9
Llama2-7B	68.4	18.5
Llama2-13B	69.2	18.8

Table 7: Performance of the target model (DeepSeek-Coder-6.7B-Base) using different reference models for influence estimation. The data synthesizer is fixed to Qwen2.5-Coder-7B-Instruct.

Models	HE		LCB-V4	
	Base	Plus	Full	Easy
Llama3.1-8B-Ins	65.9	57.9	18.0	46.7
SCoder-Q14-Llama-8B	70.1	64.6	19.1	48.3
Qwen2.5-Coder-7B-Ins	88.4†	84.1†	24.7	36.6
SCoder-Q14-Qwen-7B	85.8	80.0	29.4	65.2

Table 8: Training results of different target models. † denotes results from the official technical report.

Table 7, the performance variations across different reference models are relatively small. This indicates that our method is stable and largely insensitive to the specific scale or version of the reference model, further validating its robustness and practicality.

H Data validity on more target models

To further demonstrate the generalization capability of the data synthesized by the small synthesizers, we additionally selected Llama3.1-8B-Base and Qwen2.5-Coder-7B-Base as target models. Following the settings described in Appendix C, we set the bootstrapped Qwen2.5-Coder-14B-Ins as the synthesizer and trained SCoder-Q14-Llama-8B and SCoder-Q14-Qwen-7B respectively. As shown in Table 8, SCoder achieves significant improvements over the corresponding instruction models across the majority of evaluation metrics. Considering that Qwen2.5-Coder-7B-Ins was trained on millions of instruction data while we only used 60K data generated by the small synthesizer, this still demonstrates the effectiveness of our approach.

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.