GiFT: Gibbs Fine-Tuning for Code Generation

Haochen Li **Wanjin Feng** **Xin Zhou*** **Zhiqi Shen**
Nanyang Technological University, Singapore
{haochen003, xin.zhou, zqshen}@ntu.edu.sg wanjin_feng@outlook.com

Abstract

Training Large Language Models (LLMs) with synthetic data is a prevalent practice in code generation. A key approach is self-training, where LLMs are iteratively trained on self-generated correct code snippets. In this case, the self-generated codes are drawn from a conditional distribution, conditioned on a specific seed description. However, the seed description is not the only valid representation that aligns with its intended meaning. With all valid descriptions and codes forming a joint space, codes drawn from the conditional distribution would lead to an underrepresentation of the full description-code space. As such, we propose **Gi**bbs **F**ine-**T**uning (GiFT), a novel self-training method inspired by Gibbs sampling. GiFT allows self-generated data to be drawn from the marginal distribution of the joint space, thereby mitigating the biases inherent in conditional sampling. We provide a theoretical analysis demonstrating the potential benefits of fine-tuning LLMs with code derived from the marginal distribution. Furthermore, we propose a perplexity-based code selection method to mitigate the imbalanced long-tail distribution of the self-generated codes. Empirical evaluation of two LLMs across four datasets demonstrates that GiFT achieves superior performance, particularly on more challenging benchmarks. Source code is available at https://github.com/Alex-HaochenLi/GiFT.

1 Introduction

Code generation, the automated synthesis of code snippets from natural language specifications, significantly enhances software development productivity (Han et al., 2024; Jiang et al., 2024b). Recent advances in large language models (LLMs), trained on massive web-derived code and text corpora, exhibit notable capabilities for code understanding and generation (Jiang et al., 2024a; Wang et al., 2024; Liu et al., 2024b). While scaling training data is beneficial, high-quality data has been found to play a more important role in boosting LLM performance (Wei et al., 2023, 2024).

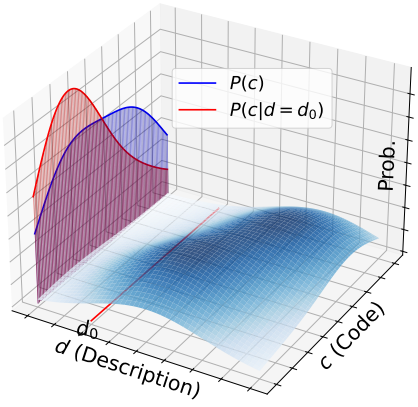

Figure 1: For the intention of d_0, the set of all valid descriptions and codes forms a space. The distribution gap between conditional distribution (Red) and marginal distribution (Blue) indicates the bias introduced when fine-tuned LLMs with codes conditional on d_0, as some codes are rarely sampled.

However, curating large-scale, high-quality datasets by manual annotation is challenging due to substantial costs. Consequently, researchers switch gears to synthetic data. Two principal approaches to generate data are: a) knowledge distillation from stronger LLMs (Wei et al., 2023; Yu et al., 2024; Luo et al., 2023), and b) self-training, wherein models generate their own training data (Yuan et al., 2023; Zelikman et al., 2022; Wei et al., 2024). The prerequisite of stronger LLMs for distillation restricts its general applicability, thus many works focus more on self-training, as we studied in this paper.

The self-training process utilizes a seed dataset, denoted as $\{d_i\}_{i=1}^{N}$. For each description d_i, an LLM generates multiple candidate code snippets, which are evaluated against test cases. Snippets passing all tests are used to fine-tune the same LLM.

* Corresponding author

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12271–12284

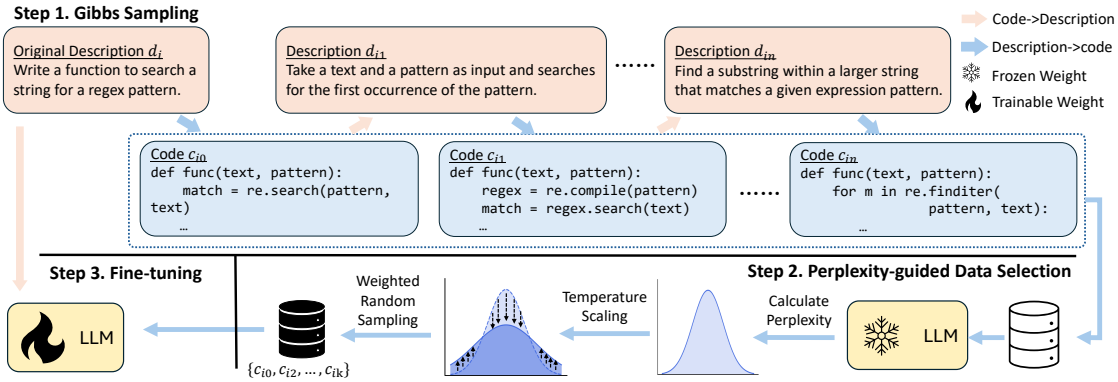

Figure 2: Overview of GiFT. For each description d_i in the seed dataset, we first translate it between descriptions and codes iteratively to draw codes from the marginal distribution based on the intention of d_i. Then, we calculate the perplexity of each generated code and employ weighted random sampling to select codes with codes from the tail being more likely to be selected for fine-tuning. Finally, all selected codes are paired with d_i for fine-tuning. The example shown in this figure is taken from MBPP-sanitized/6.

This process iterates until performance plateaus or degrades. However, the natural language description d_i represents only one possible articulation of the underlying intention. Consider the goal of matching a pattern within a string. This underlying goal—what we refer to as the "intention"—can be expressed through various descriptions such as "*Write a function to search a string for a regex pattern*" or "*Find a substring within a larger string that matches a given regular expression pattern*". Considering the set of all valid descriptions and corresponding code implementations that satisfy the underlying intention of d_i as a description-code space, generating code exclusively from d_i can be viewed as sampling from the conditional distribution $P(c|d_i)$. We argue that this approach is suboptimal. Instead, we propose that sampling from the marginal distribution of codes within the joint description-code space, denoted as $P(c)$, would yield superior results. Figure 1 illustrates the potential benefit of sampling from the marginal distribution compared to the conditional distribution. The figure highlights the possibility of oversampling certain code implementations and undersampling others when relying solely on the conditional probability $P(c|d_i)$.

In this paper, we first theoretically justify the benefit of fine-tuning LLMs with samples drawn from the marginal distribution, by showing that an additional expectation of loss is implicitly taken to reduce the bias introduced by samples from the conditional distribution. Direct sampling from the marginal distribution is intractable in practice. We gain inspiration from Gibbs sampling (Geman and Geman, 1984), an MCMC algorithm, that iteratively samples each variable from its conditional distribution while keeping the others fixed, gradually approximating the joint distribution. Simulating the Gibbs sampling in the context of code generation, we propose **Gi**bbs **F**ine-**T**uning (GiFT). From a seed description, code is generated and then summarized into a new description, used for subsequent code generation. This process is repeated to get a set of self-generated description-code pairs, which could be considered drawn from the joint distribution. The code components in pairs can be considered drawn from the marginal distribution [1].

Self-generated code often suffers from data imbalances detrimental to LLM fine-tuning. One source of imbalance is the varying number of generated codes for descriptions of differing difficulty. We address this by selecting a fixed number of codes per description. The other more fundamental imbalance arises from the long-tailed nature of the marginal distribution from which code is sampled (Ding et al., 2024; Dohmatob et al., 2024). High-probability (head) codes are over-sampled, while low-probability (tail) codes are under-sampled. This disparity can lead to model collapse during iterative self-training (Shumailov et al., 2023). To address this, we use perplexity as a proxy for the likelihood of being sampled: higher perplexity indicates rarer, tail-distributed code. During training, we select more high-perplexity codes from the tail. Figure 2 illus-

[1] In this paper, we primarily study the impact of using code samples from either conditional or marginal distributions on fine-tuning. The incorporation of self-generated descriptions is discussed in Section 5.

trates the GiFT overview.

We evaluate GiFT on DeepSeek-Coder-6.7B (Guo et al., 2024) and CodeLlama-7B (Roziere et al., 2023) over APPS+ (Introductory-level and Interview-level) (Dou et al., 2024), MBPP+ (Liu et al., 2024a), and CodeInsight (Beau and Crabbé, 2024) datasets. Experimental results demonstrate the superiority of drawing codes from the marginal distribution instead of the conditional distribution (+1.2% on MBPP+, +2.3% on CodeInsight, +9.8% on APPS+ dataset), and perplexity-guided data selection benefits self-training over iterations.

2 Related Work

We classify related works into distillation and self-training based on whether the synthetic data is generated by stronger LLMs or the LLM undergoing training itself.

Distillation Code Alpaca (Chaudhary, 2023), similar to Self-Instruct (Wang et al., 2023), leverages the in-context learning ability of ChatGPT to generate new description-code pairs. WizardCoder (Luo et al., 2023) prompts ChatGPT with five tailored heuristics to improve the difficulty of existing descriptions in Code Alpaca. Magicoder (Wei et al., 2023) and WaveCoder (Yu et al., 2024) highlight the importance of data diversity and quality by prompting ChatGPT to create new pairs based on open-sourced codes on the web instead of LLM-generated Code Alpaca. MathGenie (Lu et al., 2024) improves from the solution side where it augments the solutions by prompting an external LLM with heuristics and then back-translates augmented solutions into math problems in order to create new problems. However, stronger LLMs are not always available, which limits the generalizability of distillation methods.

Self-training Self-training refers to making LLMs learn from their own outputs based on a set of seed descriptions. Self-training approaches can be categorized into two directions based on whether additional data is synthesized on the description side or the code side. On the description side, Instruction Backtranslation (Li et al., 2023b) and InverseCoder (Wu et al., 2024) ask an LLM to generate synthesized descriptions for unlabeled codes for instruction tuning. On the code side, Self-Taught Reasoner (STaR) (Zelikman et al., 2022) is a pioneering work that generates a single rationale for each reasoning problem. LMSI (Huang et al., 2022) and Rejection Fine-tuning (RFT) (Yuan et al., 2023) enhance STaR by generating multiple rationales per problem. While STaR, LMSI, and RFT rely on the ground truth answer to filter out incorrect rationales, SelfCodeAlign (Wei et al., 2024) additionally asks LLMs to generate test cases for synthesized codes to conduct self-validation. Reinforced Self-Training (ReST) (Gulcehre et al., 2023) and ReST^{EM} (Singh et al., 2023) expand the RFT process into an iterative one, where the generate-then-fine-tune process is repeated multiple times until no further improvement is observed.

Though there are intermediate descriptions generated in GiFT, we do not use those intermediate descriptions for fine-tuning, as GiFT is mainly proposed to improve the data quality on the code side. GiFT is orthogonal to the self-training methods on the code side as each synthetic description can benefit from higher-quality codes generated in GiFT. Besides, GiFT is beneficial under the distillation setting. We empirically demonstrate the effectiveness of data from GiFT in Section 5.

3 Gibbs Fine-Tuning

Preliminaries We first introduce how iterative self-training works. Given a seed dataset $\mathcal{D} = \{d_i\}_{i=1}^{N}$, an LLM $\mathcal{M}$ is used to generate n code snippets for each d_i:

$$\{c_{ij}\}_{j=1}^{n} \sim P_{\mathcal{M}}(c|d_i) \tag{1}$$

Then correct codes that pass all the test cases are selected as $\mathcal{C}$ for supervised fine-tuning (SFT). The SFT loss $\mathcal{L}$ for d_i could be written as:

$$\begin{aligned}\mathcal{L}(d_i^*) &= -\mathbb{E}_{\mathcal{C}\sim P_{\mathcal{M}}(c|d_i^*)} \log P_{\mathcal{M}}(c|d_i) \\ &= -\sum_{c\in\mathcal{C}} P_{\mathcal{M}}(c|d_i^*) \log P_{\mathcal{M}}(c|d_i)\end{aligned} \tag{2}$$

Here d_i^* refers to the description source that c_{ij} is generated from. In this case, $d_i^* = d_i$. The SFT loss is calculated over the seed dataset $\mathcal{D}$ to update $\mathcal{M}$, and the updated $\mathcal{M}$ will be used to generate codes in the next iteration. This process is repeated until no further improvement in performance is observed.

Theoretical Insight The problem in the code generation process is that all the self-generated codes are drawn from a conditional distribution $c_{ij} \sim P(c|d_i)$ instead of the joint space of descriptions and codes based on the intention behind d_i. We argue that it is better to draw codes from the

marginal distribution of that space. If we fine-tune LLMs with codes from marginal distribution, we have:

$$
\begin{aligned}
&\mathcal{L}_{marg} \\
&= -\mathbb{E}_{c\sim P_c} \log P_{\mathcal{M}}(c|d_i) \\
&= -\sum_{c} P_c(c) \log P_{\mathcal{M}}(c|d_i) \\
&= -\sum_{c}\sum_{d_{ij}} P_{\mathcal{M}}(c|d_{ij})P(d_{ij}) \log P_{\mathcal{M}}(c|d_i) \\
&= -\sum_{d_{ij}} \Big[\sum_{c} P_{\mathcal{M}}(c|d_{ij}) \log P_{\mathcal{M}}(c|d_i)\Big] P(d_{ij}) \\
&= -\sum_{d_{ij}} \mathcal{L}(d_{ij})P(d_{ij}) \qquad (3) \\
&= -\mathbb{E}_{d_{ij}\sim P_d}\mathcal{L}(d_{ij}) \qquad (4)
\end{aligned}
$$

According to the law of total expectation, we could find that $\mathcal{L}_{marg}$ is estimated over all possible descriptions d_{ij} in the joint space, instead of only d_i in Eq. 2. The additional expectation in $\mathcal{L}_{marg}$ reduces the bias of $\mathcal{L}(d_i)$ in learning to generate codes for the intention behind d_i.

Besides, we analyze the variance of self-generated codes c from either the marginal distribution or the conditional distribution. According to the law of total variance, we have:

$$Var(c) = \mathbb{E}_{d_{ij}}[Var(c|d_{ij})] + Var(\mathbb{E}_{d_{ij}}[c|d_{ij}])$$

Since $Var(\mathbb{E}_{d_{ij}}[c|d_{ij}]) \geq 0$, the variance of codes drawn from the marginal distribution is greater than or equal to the expected variance of codes drawn from the conditional distribution conditioned on a certain d_{ij} (e.g. the seed description d_i). What's more, $Var(\mathbb{E}_{d_{ij}}[c|d_{ij}])$ could become even larger if an LLM is sensitive to input descriptions, which further widens the gap between $Var(c)$ and $\mathbb{E}_{d_{ij}}[Var(c|d_{ij})]$. Here the variance of c reflects the diversity of self-generated codes. More diverse codes are found to benefit LLM fine-tuning (Yuan et al., 2023).

Gibbs Sampling Though we have demonstrated that marginal distribution is better than conditional distribution, direct sampling from marginal distribution is not straightforward, as we only have one certain d_i in the seed dataset. We gain inspiration from Gibbs sampling (Geman and Geman, 1984), a Markov chain Monte Carlo algorithm, that is commonly used to approximate joint distributions based on conditional distributions. Take a bivariate distribution as an example. It approximates joint distributions by drawing an instance of one variable conditional on the current value of the other variable, then drawing an instance of the other variable conditional on the new value of the first variable, and repeating this process for several rounds.

In GiFT, we consider the code-to-text translation and text-to-code translation as the conditional sampling process. We keep translating between descriptions and codes to simulate Gibbs sampling. During this process, all the intermediate description-code pairs could be considered as being drawn from the joint distribution. If we take all codes from the pairs, those codes can be considered drawn from the marginal distribution of the joint space. Specifically, for each description d_i in the seed dataset, we start from the description side to generate corresponding codes c_{i1}, and then summarize c_{i1} into description d_{i1}. We repeat this process for n times. The whole process could be formulated as:

$$
\begin{aligned}
c_{i1} &= \mathcal{M}(d_i) \qquad & d_{i1} &= \mathcal{M}(c_{i1}) \\
&\vdots & & \qquad (5) \\
d_{in-1} &= \mathcal{M}(c_{in-1}) \qquad & c_{in} &= \mathcal{M}(d_{in-1})
\end{aligned}
$$

The prompting templates for code generation and summarization are shown in Appendix A. To improve the efficiency of the Gibbs sampling, we generate 3 codes in each code generation step but only select one correct code for the following rounds. If none of the 3 codes passes all the test cases, we use the code from the last round for the next code summarization step.

Perplexity-guided Data Selection After the Gibbs sampling process, for each d_i, we have a set of codes that could be paired with it for fine-tuning. As shown in Eq. 3, $P(d_{ij})$ plays a pivotal role in the estimation of $\mathcal{L}_{marg}$. In practice, $P(d_{ij})$ is reflected in the selection of codes.

Simply selecting all correct codes is detrimental. On the one hand, we are more likely to sample more codes for easy descriptions in the seed dataset and less for harder ones. Fine-tuning with all codes will bias LLMs towards easy descriptions (Singh et al., 2023), so we only select K codes for each description. For descriptions with fewer than K codes, we resample existing codes to ensure balance. On the other hand, there is data imbalance in the code set of each d_i. According to Ding et al. (2024), the marginal distribution within each code set is found to follow a long-tail distribution. Employing

random sampling to select K codes makes codes from the tail occupy only a small proportion of the training data since they are seldom generated, which bias $\mathcal{L}_{marg}$ towards the head. In iterative self-training, this bias will be exacerbated where the knowledge distribution of the LLM shifts to be more peaked.

We propose to use perplexity (Brown et al., 1990) as a measurement to guide data selection. As we know, LLMs prefer tokens with higher probabilities in each generation step, despite using temperature to flatten the probability distribution. The probability of generating a code c with l tokens given d could be formulated as:

$$P_{\mathcal{M}}(c|d) = \prod_{t=1}^{l} P_{\mathcal{M}}(c_t|c_{<t}, d)$$

And the perplexity (ppl) is calculated by:

$$\text{ppl}(c|d, \mathcal{M}) = \exp\Big(-\frac{1}{l}\sum_{t=1}^{l}\log P_{\mathcal{M}}(c_t|c_{<t}, d)\Big)$$

We could find that the perplexity of c and the probability of generating c have a strong negative correlation. In other words, codes with lower perplexity are more likely to come from the head. Thus, to mitigate imbalance during data selection, we employ weighted random sampling and assign more weights for high perplexity (i.e. tail) codes:

$$w_{ij} = \frac{\exp(\text{ppl}(c_{ij})/T)}{\sum_{j=1}^{n_i}\exp(\text{ppl}(c_{ij})/T)}$$

where n_i is the number of correct codes for d_i and T is the scaling temperature. Finally, the selected codes $\{c_{ij}\}_{j=1}^{K}$ are paired with d_i for fine-tuning LLMs with the SFT loss. The workflow of GiFT in each iteration is shown in Appendix B Algorithm 1.

4 Experimental Setups

Datasets We evaluate GiFT on three datasets, APPS+ (Dou et al., 2024), MBPP (Austin et al., 2021), and CodeInsight (Beau and Crabbé, 2024). APPS+ is a sanitized version of APPS (Hendrycks et al., 2021) where wrong descriptions or test cases are removed from the original dataset. For MBPP, we use MBPP-sanitized for training and MBPP+ (Liu et al., 2024a) for testing. The APPS dataset consists of problems collected from different open-access coding websites, MBPP is full of general programming problems, and CodeInsight is collected from StackOverflow focusing on standard library usage. In this paper, we consider the problems in APPS+ with the difficulty of "introductory" and "interview" as two independent datasets. All four datasets are written in Python and their statistics are shown in Appendix C.1. We take the widely adopted Pass@1 as the evaluation metric.

Baselines We compare GiFT with two baseline methods. (1) Rejection Fine-Tuning (**RFT**) (Yuan et al., 2023) uses rejection sampling that generates multiple codes depending on each seed description. As ReST (Gulcehre et al., 2023) could be considered as iterative RFT, we denote this baseline as RFT in our experiments. (2) In RFT+Rewriting Description (**RFT+RD**), we first ask LLMs to rewrite the seed description and then apply RFT to both the original description and rewritten descriptions. Though no related works employ this method, we consider it as an alternative way to approximate the marginal distribution. We apply these three methods to DeepSeek-Coder-6.7B (Guo et al., 2024) and CodeLlama-7B (Roziere et al., 2023).

Implementation Details For GiFT, we repeat the description-to-code and code-to-description process for 20 times, and we generate 3 codes from each description and only select at most one correct code for the next round. To ensure fair generation times, for RFT, we generate 20×3 codes for each seed description. And for RFT+RD, we rewrite the seed description into 5 new descriptions and generate 10 codes for each new description and the original description. For all the generation processes of LLMs, we set a temperature of 1.0. We set the temperature $T = 2$ in weights calculation for random sampling. For three methods, we select $K = 8$ codes per description for fine-tuning and employ resampling for descriptions with fewer than 8 codes. More details can be found in Appendix C.2.

5 Experiments

Overall Results We apply RFT, RFT+RD, and GiFT to DeepSeek-Coder-6.7B and CodeLlama-7B across four datasets with a 3-iteration self-training, the results are shown in Figure 3. Note that for RFT+RD and GiFT, we also consider codes generated from RFT as candidates for being selected, since we find that incorporating codes from RFT could further boost the performance of GiFT, even though merely using GiFT data has already outper-

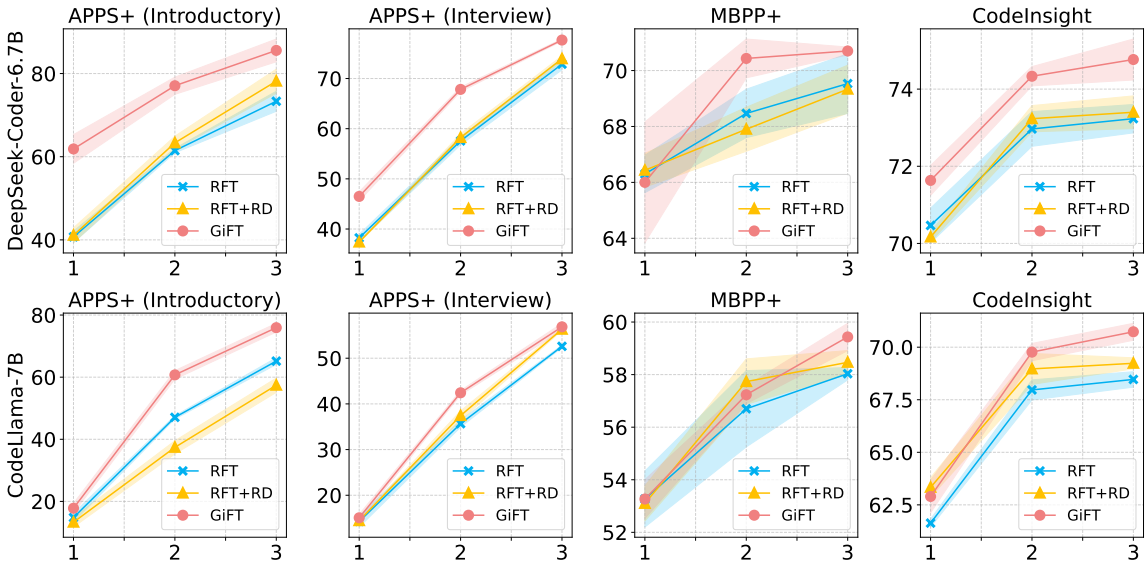

Figure 3: Pass@1 (%) of applying RFT, RFT+RD, and GiFT to Deepseek-Coder-6.7B and CodeLlama-7B on 4 code generation datasets. The x-axis represents the iteration number and the shaded area represents the standard deviation.

formed baseline methods. We discuss the impact of these RFT-generated codes in Appendix D.1.

We could see that GiFT outperforms RFT and RFT+RD with a significant margin on all evaluated datasets, which indicates the effectiveness of GiFT. Generally, the improvement brought by GiFT is more significant on more challenging datasets like APPS+. We think this is because LLM's output distribution for complicated descriptions is more peaked, which exacerbates the bias in loss calculation. This speculation is supported by the results shown at the bottom of Figure 5. We can find that the perplexity distribution of self-generated codes of APPS+ Introductory is much more peaked compared to that of MBPP+.

Given the fact that GiFT is superior compared to RFT, and RFT+RD outperforms RFT on most of the datasets, we demonstrate that drawing self-generated codes based on multiple possible descriptions that represent the intention of the seed description is better than drawing solely based on the seed description. In other words, mitigating the bias introduced in the loss calculation of examples from conditional distribution is beneficial for LLM fine-tuning.

Analysis for RFT+RD Though RFT+RD outperforms RFT on most of the datasets, drawing codes based on multiple rewritten descriptions is still not comparable with GiFT.

We find the reason is that the rewriting ability of LLMs is not satisfiable. There are often errors and information loss in rewritten descriptions, which makes the codes translated from rewritten descriptions often wrong. We calculate the pass rate of self-generated codes in the first iteration to indicate the correctness of self-generated descriptions. For RFT+RD, we separately calculate the pass rate of codes from the seed description and five rewritten descriptions. The results are shown in Table 1. We can see that the pass rate of codes generated from rewritten descriptions is significantly lower, which indicates that the rewritten descriptions are often incorrect. Since we do not train LLMs to rewrite descriptions, this phenomenon is expected to exist in the following iterations of self-training. Given this reason, we believe that there will be no significant improvement despite scaling the RFT+RD method with more rewritten descriptions.

Datasets	RD		GiFT
	Seed	Rewritten	
APPS+ (Intro.)	17.79	4.8	>10.31
APPS+ (Inter.)	3.22	0.38	>2.28
MBPP+	53.71	24.6	>24.07
CodeInsight	43.87	9.84	>24.18

Table 1: Pass rate (%) of self-generated codes from the seed description, rewritten description, and GiFT. In GiFT, we generate 3 codes per description and save at most 1 for the next round. Thus, the true pass rate of GiFT is higher than the value in this table.

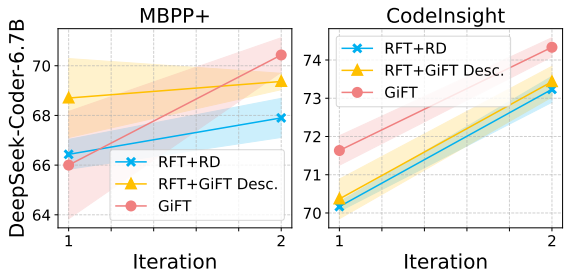

Figure 4: Pass@1 (%) of applying RFT+RD with descriptions from a single step of Gibbs sampling to Deepseek-Coder-6.7B on MBPP+ and CodeInsight.

RFT+RD with Descriptions from a Single-step GiFT Given that the rewritten descriptions are often incorrect, we evaluate the performance of another stronger setting for RFT+RD. That is, we use the self-generated descriptions from a single step of Gibbs sampling as the rewritten descriptions in RFT+RD. The results of DeepSeek-Coder-6.7B on MBPP+ and CodeInsight are shown in Figure 4. The improvement of stronger RFT+RD over the vanilla RFT+RD verifies our claim that it is better to leverage the outstanding ability of LLMs to translate the code back to a description instead of rewriting the description to a new one. However, we could see that GiFT still outperforms the stronger RFT+RD. We think this is because all of the descriptions are from the same code. Instead, in GiFT, descriptions are from various codes in each Gibbs sampling step.

Impact of T in Data Selection Recall that in perplexity-guided data selection, we set $T = 2$ to encourage the selection of more codes from the tails of the distribution to mitigate the tail narrowing problem (Ding et al., 2024). On the contrary, we could set T as a negative value to select more codes from the head. By setting $T = \pm 2$, we explore the impact of data source (head or tail) for LLM fine-tuning. Note that we conduct extended experiments by setting $T = \pm 5$ in Appendix D.2.

We show the performance of DeepSeek-Coder-6.7B on APPS+ Introductory and MBPP+ in Figure 5. Furthermore, we visualize the perplexity distribution of self-generated codes at the third iteration for APPS+ Introductory and MBPP+. We could find that selecting more codes from the head outperforms the tail at the first several iterations, but is surpassed from the third iteration. We speculate that selecting more codes from the head reinforces LLM's knowledge at the head hence accelerating training at the beginning, but with the expense

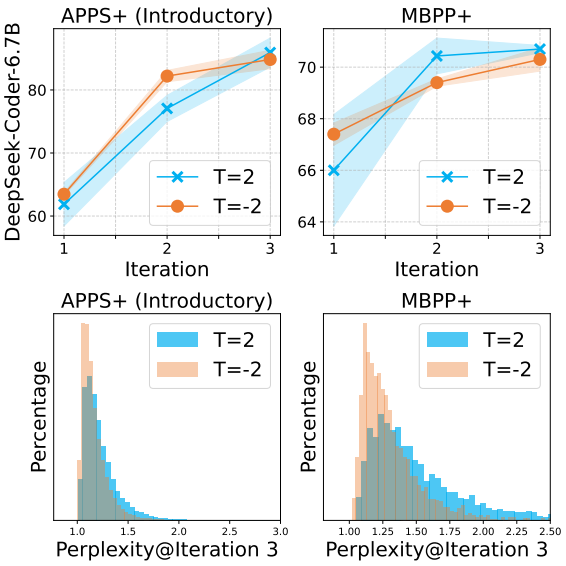

Figure 5: **Top:** Pass@1 (%) of applying GiFT to Deepseek-Coder-6.7B on APPS+ (Introductory) and MBPP+ with $T = \pm 2$. **Bottom:** Perplexity distribution of self-generated codes at the 3rd iteration for APPS+ (Introductory) and MBPP+.

of discarding or forgetting knowledge at the tail. Over iterations, the tail-narrowing phenomenon is exacerbated and hinders further improvement. When selecting more codes from the tail, LLMs could achieve an overall better performance though they improve slower. As we show at the bottom of Figure 5, after three iterations, LLM could generate more low-perplexity codes on two datasets if we set $T = 2$.

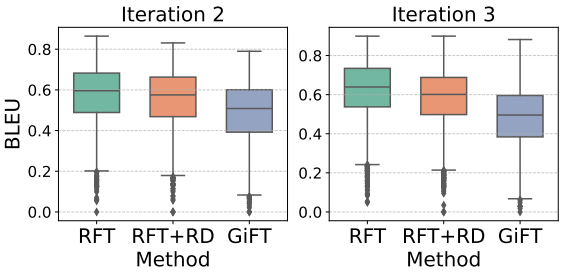

Figure 6: Boxplot of BLEU for self-generated codes from DeepSeek-Coder-6.7B on MBPP+ at the 2nd and 3rd iteration.

Similarity Analysis for Self-generated Code We use BLEU as a measurement of code diversity to show that one of the benefits of using GiFT is the increase in data diversity. For each seed description, we calculate the BLEU score between any two self-generated codes and average them to indicate one code's similarity to others. We show

the BLEU results of MBPP+ at the 2nd and 3rd iteration of DeepSeek-Coder-6.7B in Figure 6. We can observe that generating codes from various descriptions leads to more diverse codes for fine-tuning. Besides, as the number of iterations increases, RFT tends to generate more similar code, while the diversity holds for GiFT.

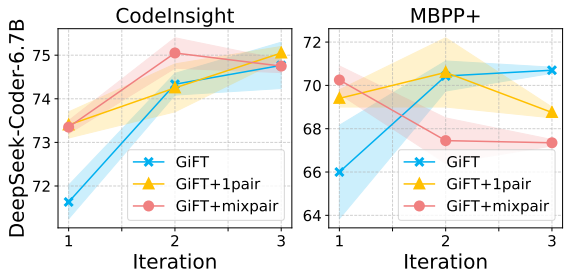

Figure 7: Comparison of incorporating self-generated descriptions and vanilla GiFT on DeepSeek-Coder-6.7B over CodeInsight and MBPP+.

Incorporating Self-generated Descriptions into Fine-tuning In GiFT, we only take the self-generated codes for fine-tuning after Gibbs sampling. Here we investigate the impact of incorporating self-generated descriptions into fine-tuning. Theoretically, if self-generated descriptions can match self-generated codes, LLMs are expected to achieve an even better performance, since LLMs benefit from not only diverse codes but also diverse descriptions as inputs.

We discover two alternatives, for each seed description, we add 8 self-generated descriptions, and 1) each one is paired with the code generated from it (denoted as GiFT-1pair). 2) each one is paired with 8 codes randomly sampled from the self-generated code set of the seed description (denoted as GiFT-mixpair). Note that not all self-generated descriptions are correct. We only select self-generated descriptions that can result in correct codes. We compare these two settings with the vanilla GiFT on DeepSeek-Coder-6.7B over CodeInsight and MBPP+ and the results are shown in Figure 7. It was observed that incorporating self-generated descriptions into fine-tuning leads to better performance at the first iteration, yet is outperformed by GiFT in subsequent iterations.

We suspect that this is because LLMs are relatively tolerant of noisy data at the beginning, but as they have more expertise, their requirements for data quality become increasingly higher. We find there are mainly two sources of noisy pairs. First, some self-generated descriptions are just incorrect. Since we additionally provide some test cases in the docstring, LLMs may generate correct codes by inferring through given test cases and ignoring the incorrect description. Second, a self-generated description may not match all codes in the self-generated code set, possibly due to specifications on implementation requirements. To filter out noisy pairs, we may calculate the similarity between description and code using code search models (Li et al., 2023a, 2022) or LLM-as-a-Judge (Zheng et al., 2023; Gu et al., 2024). Since we mainly aim to show the benefit of fine-tuning LLMs using the seed description paired with codes from the marginal distribution, we leave this as our future work.

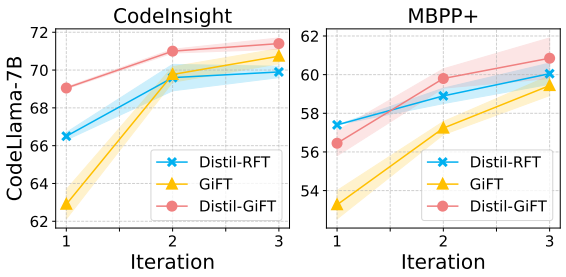

Figure 8: Comparison of using GiFT and RFT data from DeepSeek-Coder-6.7B to distill CodeLlama-7B over CodeInsight and MBPP+. The yellow line shows the performance of self-training with GiFT.

GiFT for Distillation As we discussed in the introduction, since the prerequisite of stronger LLMs limits the generalizability of distillation methods, we focus on self-training methods, as many recent works do. Yet, we are also interested in exploring whether drawing codes from the marginal distribution is beneficial for distillation. To simulate the distillation process, we use self-generated codes from DeepSeek-Coder-6.7B to fine-tune CodeLlama-7B, as we find that DeepSeek-Coder is stronger than CodeLlama on evaluated datasets. We keep other settings the same as they are in the main experiments. The comparison between distillation with RFT and GiFT on MBPP+ and CodeInsight is shown in Figure 8. It is observed that GiFT also outperforms RFT under the distillation setting. This superiority meets our expectations because the benefit of fine-tuning LLMs with codes drawing from the marginal distribution is not limited to self-training methods, as we analyzed in the theoretical insights.

Ablation study for Perplexity-guided Data Selection The proposed perplexity-guided data se-

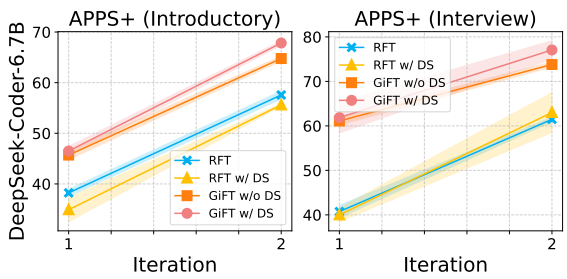

Figure 9: Pass@1 (%) of applying the proposed data selection method to RFT and GiFT on APPS+ (Introductory) and APPS+ (Interview). DS is short for data selection.

lection method can be applied not only to GiFT but also to baseline methods. The results of applying the proposed data selection method to RFT and GiFT with $T = 2$ on DeepSeek-Coder-6.7B are shown in Figure 9. We can see that the data selection method does not bring significant improvement to RFT compared with GiFT.

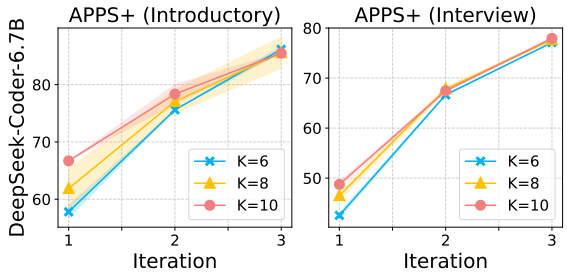

Figure 10: Pass@1 (%) of applying GiFT to DeepSeek-Coder-6.7B on APPS+ (Introductory) and APPS+ (Interview) with $K = 6, 8, 10$, respectively.

Impact of K in Data Selection In this paper, we set $K = 8$ for all the reported experimental results. In this section, we explore the impact of K in GiFT. We set $K = 6, 8, 10$ for DeepSeek-Coder-6.7B on APPS+ Introductory and APPS+ Interview and the results are shown in Figure 10. We can find that pairing each seed description with more codes significantly improves LLM performance at the beginning of iterative self-training. Yet this benefit diminishes as the iteration progresses, and finally, LLM performance converges to similar performance. We think that the curves will converge to the upper bound of LLM's potential. Thus, we argue that increasing K in GiFT accelerates iterative self-training.

6 Discussion

Here we discuss the generalization of GiFT to other tasks. The applicability of GiFT and its superiority over RFT depends on two factors. First, there is a joint input-output sampling space, in which same intention has multiple possible forms of presentation, and such presentation largely decides the self-generated outputs from LLM. GiFT is suitable for code generation because descriptions and codes naturally form such a joint space (Li et al., 2024). On the other hand, take question answering as an example, while there are numerous ways to phrase the same question, the answers tend to be highly similar due to the uniqueness of factual truths. This characteristic naturally mitigates biases introduced by conditional sampling.

Second, LLMs should be able to translate accurately between inputs and outputs. Since LLMs are found to be good at translating between descriptions and codes (Sun et al., 2025; Zan et al., 2022; Jiang et al., 2024a), GiFT performs well on code generation. Nevertheless, for mathematical reasoning, whether LLMs can reliably generate a math problem based on the given solution should be carefully evaluated before we apply GiFT. If the translation from solutions to problems lacks precision, the Gibbs sampling process in GiFT may be highly inefficient.

7 Conclusion

In this paper, we first theoretically demonstrate the benefit of fine-tuning LLMs with codes from the marginal distribution of the joint description-code space instead of the conditional distribution conditioned on the seed description. Then, we propose GiFT, which iteratively translates natural language descriptions and codes between each other to approximate the marginal distribution. Furthermore, we leverage perplexity to guide data selection to mitigate the data imbalance problem in Gibbs sampling. Experimental results on two LLMs across four datasets demonstrate the effectiveness of GiFT.

Acknowledgement

We thank the anonymous reviewers for their helpful comments and suggestions. This research is supported by the RIE2025 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) (Award I2301E0026), administered by A*STAR, as well as supported by Alibaba Group and NTU Singapore through Alibaba-NTU Global e-Sustainability CorpLab (ANGEL).

Limitations

There are mainly three limitations in this work. First, GiFT is only evaluated on LLMs with a size of around 7B across Python datasets. However, as we demonstrated in the theoretical analysis, the loss bias from the conditional distribution is independent of model sizes and programming languages. Thus, we expect that GiFT is also effective in bigger or smaller LLMs and other programming languages. Second, GiFT relies heavily on test cases to filter out wrong self-generated codes. In this paper, we mainly evaluate GiFT on datasets that already provide test cases in the training set. We do not evaluate GiFT on the most recent high-quality datasets like OSS-Instruct (Wei et al., 2023). A possible solution is to ask LLMs to generate test cases and codes at the same time, which is studied by recent works (Chen et al., 2022, 2024; Liu et al., 2024c). Third, GiFT outperforms RFT by translating between natural language descriptions and codes, which introduces additional code summarization steps in the self-generation process. In our future work, we will focus on improving the sampling efficiency of GiFT by introducing a dynamic sampling strategy. For example, we do not generate a fixed number of codes and select at most one correct code for the next round. Instead, we dynamically adjust the number of generated codes based on the estimated accuracy to ensure that at least one correct code is expected. Thus, LLMs tend to generate fewer code candidates for easier descriptions in each round.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Nathanaël Beau and Benoît Crabbé. 2024. Codeinsight: A curated dataset of practical coding solutions from stack overflow. *arXiv preprint arXiv:2409.16819*.

Peter F Brown, John Cocke, Stephen A Della Pietra, Vincent J Della Pietra, Frederick Jelinek, John Lafferty, Robert L Mercer, and Paul S Roossin. 1990. A statistical approach to machine translation. *Computational linguistics*, 16(2):79–85.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. *GitHub repository*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. 2024. B4: Towards optimal assessment of plausible code solutions with plausible tests. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1693–1705, New York, NY, USA. Association for Computing Machinery.

Yiwen Ding, Zhiheng Xi, Wei He, Zhuoyuan Li, Yitao Zhai, Xiaowei Shi, Xunliang Cai, Tao Gui, Qi Zhang, and Xuanjing Huang. 2024. Mitigating tail narrowing in llm self-improvement via socratic-guided sampling. *arXiv preprint arXiv:2411.00750*.

Elvis Dohmatob, Yunzhen Feng, Pu Yang, Francois Charton, and Julia Kempe. 2024. A tale of tails: Model collapse as a change of scaling laws. *arXiv preprint arXiv:2402.07043*.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

Stuart Geman and Donald Geman. 1984. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. 2023. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. Archcode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13520–13552.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns,

Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022. Exploring representation-level augmentation for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Haochen Li, Xin Zhou, Anh Luu, and Chunyan Miao. 2023a. Rethinking negative pairs in code search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12760–12774, Singapore. Association for Computational Linguistics.

Haochen Li, Xin Zhou, and Zhiqi Shen. 2024. Rewriting the code: A simple method for large language model augmented code search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1371–1389, Bangkok, Thailand. Association for Computational Linguistics.

Xian Li, Ping Yu, Chunting Zhou, Timo Schick, Omer Levy, Luke Zettlemoyer, Jason Weston, and Mike Lewis. 2023b. Self-alignment with instruction backtranslation. *arXiv preprint arXiv:2308.06259*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024b. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

Zhihan Liu, Shenao Zhang, Yongfei Liu, Boyi Liu, Yingxiang Yang, and Zhaoran Wang. 2024c. Dstc: Direct preference learning with only self-generated tests and code to improve code lms. *arXiv preprint arXiv:2411.13611*.

Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms. *arXiv preprint arXiv:2402.16352*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2023. The curse of recursion: Training on generated data makes models forget. *arXiv preprint arXiv:2305.17493*.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. 2023. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 419–431, Los Alamitos, CA, USA. IEEE Computer Society.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

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)*, pages 13484–13508.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao

Yuan, Rui Zhang, Xishan Zhang, et al. 2024. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct. *arXiv preprint arXiv:2407.05700*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Prompting Templates for Code-to-Text and Text-to-Code Translation

An example of a prompt for code generation is shown in Table 2. We construct a template for code completion where we provide a function head and a function docstring. The function docstring could be replaced with descriptions from previous Gibbs sampling results. We additionally provide one input-output pair for MBPP-sanitized and CodeInsight.

An example of a prompt for code summarization is shown in Table 3. We provide in-context examples to help LLM learn summarization. The in-context example is selected from a pool. We construct the pool by asking LLMs to summarize the same dataset without in-context examples and then filter out meaningless or too-long responses. The code following "###Code:" could be replaced with codes from previous Gibbs sampling results.

```
def first_repeated_char(str1):
    """ Write a python function to find the
first repeated character in a given string.
    >>> first_repeated_char("abcabc")
    "a"
    """
```

Table 2: A prompt example of MBPP-sanitized/1 for code generation.

```
###Code:
{example_code}
###Description of the given code:
{example_description}


###Code:
def first_repeated_char(str1):
    ### BEGIN SOLUTION
    letters_found = []

    for char in str1:
        if char in letters_found:
            return char
        else:
            letters_found.append(char)
    ### END SOLUTION

###Description of the given code:
```

Table 3: A prompt example used of MBPP-sanitized/1 for code summarization. {example_code} and {example_description} are randomly selected from a pool.

B Algorithm Workflow

The algorithm workflow of GiFT in each iteration is shown in Algorithm 1. This workflow is repeated where the updated LLM $\mathcal{M}^*$ in each iteration is used as the initialization LLM for the next one.

C Experimental Setup

C.1 Dataset Statistics

The dataset statistics are shown in Table 5. Note that we use MBPP sanitized version for training while MBPP+ (a version with more test cases for each problem) for testing.

C.2 Implementation Details

For the prompting template of generating codes in RFT and RFT+RD, we follow Table 2. An example

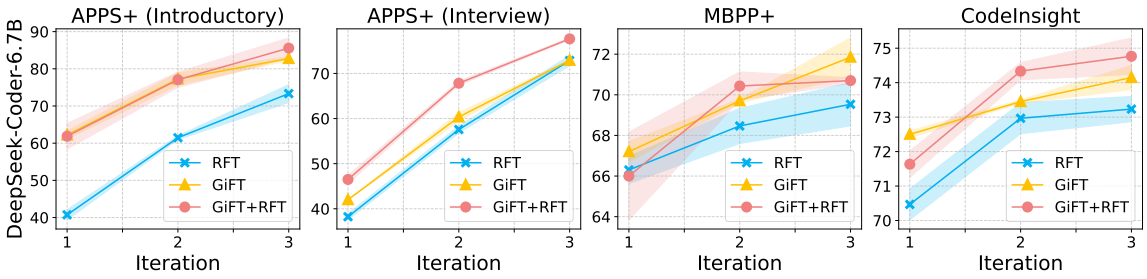

Figure 11: Pass@1 (%) of applying RFT, pure GiFT, and GiFT+RFT to Deepseek-Coder-6.7B across four code generation datasets. The x-axis represents the iteration number and the shaded area represents the standard deviation.

Rewrite the given Description ###Description: Write a python function to find the first repeated character in a given string. ###New Description:

Table 4: A prompt example of MBPP-sanitized/1 for rewriting description.

Dataset	APPS+ Introductory	APPS+ Interview	MBPP	CodeInsight
Train	1,998	3,736	170	1,547
Test	90	367	378	1,856

Table 5: Statistics of the dataset used in our experiment.

of a prompt for rewriting descriptions is shown in Table 4. DeepSeek-Coder-6.7B is initialized with the checkpoint at https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base. CodeLlama-7B is initialized with the checkpoint at https://huggingface.co/meta-llama/CodeLlama-7b-hf. We fine-tune the LLMs with DeepSpeed ZeRO-2 optimization with a batch size of 1 for each GPU. The maximum length is set to be 1,024 for MBPP and Code Insight and 1,536 for APPS. We use AdamW as the optimizer with a learning rate of 2e-5 and set the gradient accumulation steps as 16. We fine-tune LLMs for 2 epochs for APPS+ and 1 epoch for MBPP and CodeInsight. All experiments are running with 3 random seeds 1234, 12345, and 123456. Experiments are conducted on 8 Nvidia Tesla A100 GPUs.

D More Experimental Results

D.1 Impact of RFT data in GiFT

As we mentioned, we find that incorporating RFT data in GiFT could further boost the performance of GiFT, even though merely fine-tuning LLMs with codes from GiFT has already outperformed RFT. We compare the performance of RFT, pure GiFT, and GiFT+RFT on DeepSeek-Coder-6.7B across four datasets in Figure 11. We suspect that this is due to the lack of correct codes for some seed descriptions. As a result, we have to resample existing codes to ensure that there are K codes for each seed description during fine-tuning. The incorporation of RFT data mitigates this drawback and improves data diversity. In practice, we can ask LLMs to generate several times at the first round of GiFT, since in the first round of GiFT, LLMs generate codes from the seed description, which is the same in input of RFT.

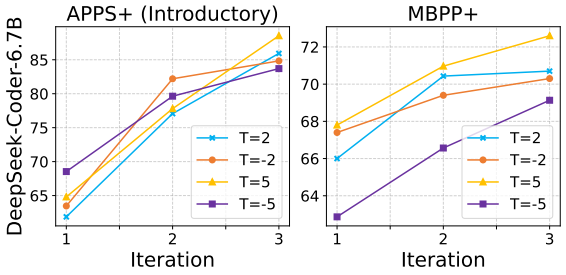

Figure 12: Pass@1 (%) of applying GiFT to Deepseek-Coder-6.7B on APPS+ (Introductory) and MBPP+ with $T = \pm 2, \pm 5$.

D.2 Extended Experiments of T

We additionally conduct experiments with bigger and smaller T values to further study the impact of T in data selection. Specifically, we set $T = \pm 5$ and the results are shown in Figure 12. We can observe that a larger T which makes codes from the tail more likely to be selected leads to even better performance. On the contrary, $T = -5$ leads to worse performance. We argue that T should be set within a moderate range and specifically tuned for each dataset.

Algorithm 1 Workflow of GiFT in each iteration

Input: A seed dataset $\mathcal{D} = \{d_i\}_{i=1}^{N}$, an LLM $\mathcal{M}$.
Parameter: Gibbs sampling iterations n, selection threshold K, temperature T.
Output: An updated LLM $\mathcal{M}^*$ for next GiFT iteration.

$\mathcal{C} \leftarrow \emptyset$.
for each description $d_i \in \mathcal{D}$ **do** ▷ Gibbs Sampling
  $c_{i1} \leftarrow \mathcal{M}(d_i)$
  for $k \leftarrow 1$ to n **do**
    $d_{ik} \leftarrow \mathcal{M}(c_{ik})$ ▷ Summarize code into description
    $c_{ik+1} \leftarrow \mathcal{M}(d_{ik})$ ▷ Generate code from description
  end for
  $\mathcal{C}_i \leftarrow \{c_{i1}, c_{i2}, \ldots, c_{in}\}$
end for
$\mathcal{C}^* \leftarrow \emptyset$
for each description $d_i \in \mathcal{D}$ **do** ▷ Perplexity-Guided Selection
  for each code $c_{ij} \in \mathcal{C}_i$ **do**
    $\texttt{ppl}(c_{ij}) \leftarrow \exp\Big(-\frac{1}{|c_{ij}|}\sum_{t=1}^{|c_{ij}|}\log P_{\mathcal{M}}(c_t|c_{<t}, d_i)\Big)$ ▷ Compute perplexity
  end for
  for each code $c_{ij} \in \mathcal{C}_i$ **do**
    $w_{ij} \leftarrow \frac{e^{\text{ppl}(c_{ij})/T}}{\sum_{j=1}^{n_i} e^{\text{ppl}(c_{ij})/T}}$ ▷ Compute weight
  end for
  $\{c_{ij}\}^K \leftarrow \texttt{weighted random sampling}(\{c_{ij}, w_{ij}\})$
  $\mathcal{C}_i^* \leftarrow \mathcal{C}_i^* \cup \{c_{ij}\}^K$
end for
$\mathcal{D}^* \leftarrow \{(d_i, c) | d_i \in \mathcal{D}, c \in \mathcal{C}_i^*\}$ ▷ Construct dataset for SFT
$\mathcal{M}^* \leftarrow SFT(\mathcal{M}, \mathcal{D}^*)$ ▷ Supervised Fine-Tuning
return $\mathcal{M}^*$ ▷ Return fine-tuned model