AlchemistCoder: Harmonizing and Eliciting Code Capability by Hindsight Relabeling on Multi-source Data

Anonymous ACL submission

Abstract

Open-source Large Language Models (LLMs) and their specialized variants, particularly Code LLMs, have recently delivered impressive performance. However, previous Code LLMs are typically fine-tuned on a single dataset, which may insufficiently elicit the potential of pre-trained Code LLMs. This paper presents ***AlchemistCoder***, a series of Code LLMs with better code generation and generalization abilities fine-tuned on multi-source data. To harmonize the inherent conflicts among the various styles and qualities in multi-source data, we introduce data-specific prompts, termed ***AlchemistPrompts***, inspired by hindsight relabeling, to improve the consistency between instructions and responses. We further propose to incorporate the data evolution process itself into the fine-tuning data to enhance the code comprehension capabilities of LLMs, including instruction evolution, data filtering, and code review. Extensive experiments demonstrate that ***AlchemistCoder*** holds a clear lead among all models of the same size (6.7B/7B) and rivals or even surpasses larger models (15B/33B/70B), showcasing the efficacy of our method in refining instruction-following capabilities and advancing the boundaries of code intelligence.

1 Introduction

Closed-source Large Language Models (LLMs) like ChatGPT and GPT-4 (OpenAI, 2022, 2023) exhibit impressive code intelligence by learning on large-scale and diverse code corpus, which also benefits many other applications, such as math reasoning (Chen et al., 2022), embodied control (Liang et al., 2023), and agent (Yang et al., 2024). Since open-source LLMs (Touvron et al., 2023) still lag behind closed-source LLMs (OpenAI, 2023) in this field, there has been growing interest in investigating the acquisition of code capabilities by developing specialized Code LLMs (Roziere et al., 2023; Guo et al., 2024).

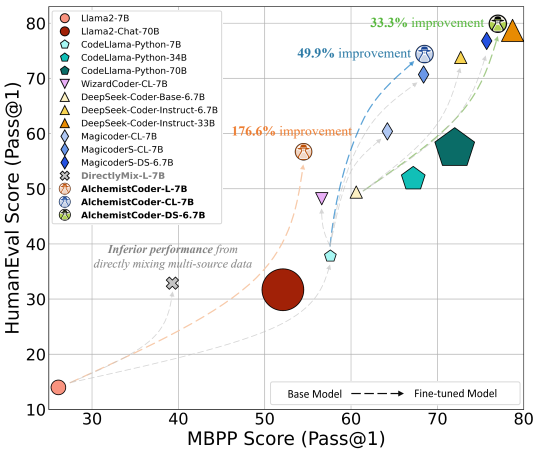


Figure 1: Performance scatter plot (*top right* is better) of open-source models on mainstream code benchmarks, HumanEval and MBPP. We use different shapes/sizes to represent different series/parameters of models, and draw dashed lines to indicate the improvements of the fine-tuned Code LLMs compared to the base models. -L/-CL/-DS respectively indicate the models fine-tuned on Llama 2/CodeLlama-Python/DeepSeekCoder-Base.

The training of Code LLMs mainly goes through pre-training and fine-tuning stages (Roziere et al., 2023). Pioneer works (Chen et al., 2021; Nijkamp et al., 2022; Li et al., 2023) have amassed extensive code data for pre-training, while recent open-source models (Luo et al., 2023b; Wei et al., 2023) highlight the effectiveness of high-quality or targeted code fine-tuning datasets. Despite these advancements, current fine-tuning methods mainly rely on a particular kind of code-related question-answering dataset, unlike the pre-training stage that integrates code-related corpus from various sources (Roziere et al., 2023). Such a discrepancy indicates that the fine-tuning data may not be diverse enough to fully stimulate the capabilities of base models, resulting in limited performance, generalization, and robustness.

To tackle these challenges, we first explore integrating data from multiple sources and find that directly mixing (*e.g.*, the DirectlyMix-L-7B model in

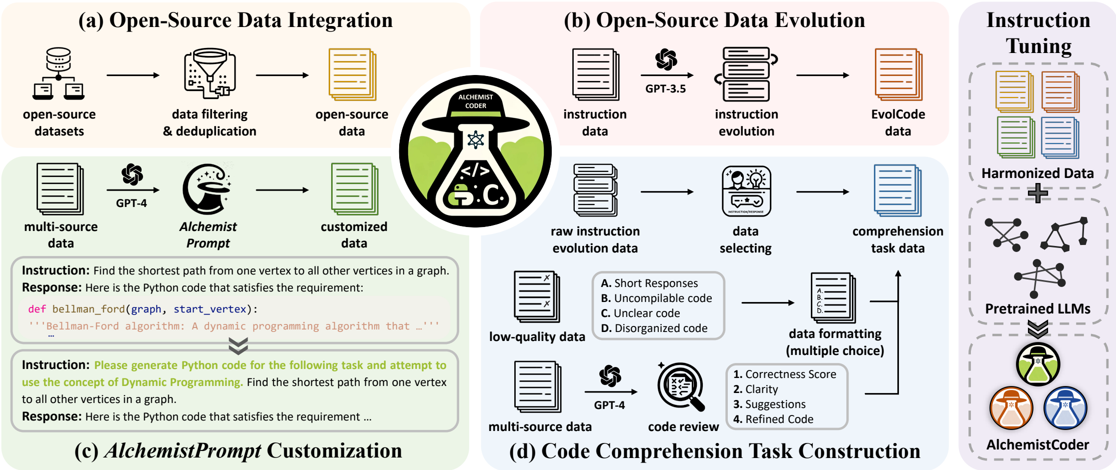


Figure 2: **Overview for developing *AlchemistCoder* series.** We first integrate high-quality open-source data (a) and conduct data evolution based on them (b). Then, we adopt *AlchemistPrompt* to harmonize their inherent conflicts (c) and construct code comprehension data (d). We use a mix of these data to fine-tune various pre-trained LLMs to obtain our *AlchemistCoder* models.

Fig. 1) does not produce the desired effect due to inherent conflict of multi-source data. Therefore, we propose to adopt hindsight relabeling (Andrychowicz et al., 2017; Zhang et al., 2023) for multi-source data mixing, which designs data-specific prompts to harmonize the inherent conflicts of different data sources so that they can be used together to elicit the performance of base models more sufficiently. We term this form of prompts as *AlchemistPrompt*s, inspired by the power and definition of *Alchemists*:

> *"Alchemist: Someone Who Transforms Things for the Better."* —— Merriam Webster Dictionary

Specifically, we first integrate some high-quality open-source code instruction data and conduct data evolution (Luo et al., 2023b) based on some of them (Fig. 2(a, b)) to obtain multi-source data for fine-tuning. Due to the diverse styles, coding languages, and varying data quality of different sources, directly mixing multi-source data will cause the trained model to be unaligned to specific coding languages and styles, resulting in inferior performance. Therefore, as shown in Fig. 2(c), for instruction-response pairs of different sources, we adopt one LLM to generate *AlchemistPrompt*s that more accurately and explicitly describe the characteristics as requirements of the response to enrich the instructions. This makes the response more consistent with and goal-conditioned on the new instructions in each data source and eventually transforms the fine-tuning process on multi-source data from *cloning different responses to similar questions* to *learning to follow diverse instructions*. Consequently, *AlchemistPrompt*s not only enhance the instruction-following capability of LLMs but also allow the integrated multi-source data to effectively boost different aspects of the base models.

Apart from the conventional problem-solution data, we argue that the evolution of code data (Luo et al., 2023b) reflects higher-level capabilities and is also valuable for the learning of Code LLMs. Thus, we decompose the process of data evolution into three tasks incorporated for training, including instruction evolution, data filtering, and code review (Fig 2 (d)), enabling further improvements of code comprehension capabilities.

We conduct extensive experiments with various base models (Touvron et al., 2023; Roziere et al., 2023; Guo et al., 2024) and develop the instruction-tuned *AlchemistCoder* series. As shown in Fig. 1, on two mainstream code benchmarks, HumanEval and MBPP, *AlchemistCoder* holds a clear lead among all models of the same size (6.7/7B), and rivals or even surpasses larger models (15/33/70B), demonstrating harmonized and formidable code capabilities. We further analyze the efficacy of *AlchemistPrompt*s and observe that *AlchemistPrompt*s mitigates the instruction/response misalignment of the data. More surprisingly, *AlchemistPrompt*s allow the code corpus also significantly improve the general capability of Code LLMs, as demonstrated by the improvements on MMLU, BBH, and GSM8K.

2 Method

To more comprehensively elicit the capability of the base LLMs, we first construct multi-source data for fine-tuning (§ 2.1), which is harmonized by *AlchemistPrompt*s to take effect(§ 2.2). Code com-

Instruction: Merge the elements of two given arrays into one, but ensure that the merged array contains only the common elements between the two arrays.

Response (a): Here is an example implementation in **JavaScript**:

```
function mergeArrays(array1, array2)
{
    return array1.filter(item => array2.includes(item));
}
```

Response (b): Sure, here's a **Python** function to do that:

```
def merge_common_elements(arr1, arr2):
    merged_arr = []
    for element in arr1:
        if element in arr2 and element not in merged_arr:
            merged_arr.append(element)
    return merged_arr
```

Figure 3: Example of conflicts from multiple sources.

prehension tasks are also constructed to further improve the performance(§ 2.3). We also discuss the details and statistics of the filtered and harmonized multi-source data in § 2.4.

2.1 Multi-source Data Construction

To fully elicit the capability of code LLMs, we first collect the fine-tuning data from multiple sources (Fig. 2(a)) and adopt the instruction evolution (Luo et al., 2023b) to improve the complexity of the instructions (Fig. 2(b)). However, integrating multi-source data for instruction tuning is challenging. Naturally, one code-related question can be solved by different coding languages with various algorithms or response styles (*e.g.*, with or without reasoning). When naively combing data curated by different developers with different LLMs, the model will learn to answer similar questions with different coding languages and response styles, as shown in Fig. 3. On the one hand, learning diverse responses may elicit different capability aspects of the base models. On the other hand, since the learned responses to similar instructions are quite different due to different implicit human intentions, the LLMs tend to be unaligned (to our expectation) after the fine-tuning on the directly mixed data (*e.g.*, we cannot expect which coding language the LLMs will use in real-world applications), resulting in inferior performance. Therefore, directly mixing multi-source data is not a promising solution and can be detrimental.

2.2 AlchemistPrompt

To better facilitate the models' learning from multi-source data, we propose customizing data-specific meta prompts called *AlchemistPrompt*s, to harmonize the inherent conflicts of data (Fig. 2(c)), inspired by hindsight relabeling (Andrychowicz et al., 2017; Zhang et al., 2023). Specifically, we harmonize multi-source data via employing GPT-4 (OpenAI, 2023) to play the role of an *Alchemist* for generating *AlchemistPrompt*s, using the prompt as illustrated in Fig. 4 to obtain data-specific *AlchemistPrompt*s. For instance, for an instruction of 'Write code to find the shortest path from one vertex to all other vertices in a graph', if the response involves Python code employing a Bellman-Ford algorithm with dynamic programming, we would expect to customize the instruction with an *AlchemistPrompt* of 'Please generate Python code for the following task and attempt to use the concept of Dynamic Programming'.

AlchemistPrompt Generation

Here are {{ sample number }} sets of instructions-response paired coding samples. Each set begins with [SAMPLE BEGIN] and ends with [SAMPLE END]. Please read them carefully and complete the **TASK**.

[SAMPLE 1 BEGIN]
<|Instruction|>:
{ *AlchemistPrompt* }
{{ sample 1 instruction }}
<|Response|>:
{{ sample 1 response }}
[SAMPLE 1 END]
...

TASK:
Please fill in { *AlchemistPrompt* } with appropriate content (within 50 words) to make the instructions clearer and better match the responses.
NOTE (very important):
1. If you find code in the response, try to analyze (including but not limited to programming languages, algorithm concepts, and code characteristics) and briefly reflect the results in { *AlchemistPrompt* }.
2. You are not allowed to modify anything other than the contents within { *AlchemistPrompt* }.
3. You are asked to generate only one { *AlchemistPrompt* }, which should be universally applicable to all the {{ sample number }} samples.

Figure 4: Detailed prompt designed for generating data-specific *AlchemistPrompt*s.

The adjustment to data by *AlchemistPrompt* is relatively minor and well-calibrated, and our ablation study indicates that optimal performance can be achieved by incorporating *AlchemistPrompt*s into only 5% of all the samples. It attains a balance between the diversity and domain gap resulting from the fusion of multi-source data. Crucially, by retrospectively analyzing previous responses and reinterpreting them as alternate goals, the *AlchemistPrompt*s serve to elevate the condition/goal of the data. This hindsight integration (Andrychowicz et al., 2017; Zhang et al., 2023; Liu et al., 2023a) allows for a more nuanced and adaptive learning process, not only enhances the models' comprehension of data but also refines their instruction-following capabilities.

2.3 Code Comprehension Task

The existing training datasets for Code LLMs (Li et al., 2022; Chaudhary, 2023; theblackcat102, 2023; Luo et al., 2023b; Wei et al., 2023) primarily focus on the code generation task consisting of programming problems and their corresponding solutions with code. We argue that apart from this, the process of constructing code data demonstrates higher-level abilities. Thus, it is also valuable for Code LLMs to further improve their performance by enhancing code comprehension from various dimensions. Therefore, we devise three code comprehension tasks relevant to data construction, including instruction evolution, data filtering, and code review (Fig. 2(d)).

Instruction Evolution. Inspired by the concept of instruction evolution (Xu et al., 2023; Luo et al., 2023b), we employ GPT-3.5 (OpenAI, 2022) to construct instruction evolution task data. including increased requirements for instructions and detailed explanations for programming tasks. The introduction of instruction evolution task can help the model intuitively perceive the differences before and after evolutions, deepening the comprehension of programming requirements, code complexity, task decomposition, and other code-related concepts.

Data Filtering. We identify four categories of low-quality data from multiple sources, including (a) responses that are too short and do not contain code, (b) code that fails to compile, (c) code with poor clarity, and (d) code that does not follow the requirement in the instruction regarding its organization in function form. In each instruction of the data filtering task, we present the model with a low-quality sample and ask the model to determine which of the four categories it belongs to. The data filtering task involves reusing the filtered-out data by providing counter examples, enabling the model to generate fewer low-quality responses.

Code Review. In this task, we require the model to review a piece of code and assign scores between 0 and 10 for correctness and clarity separately. Additionally, the model is expected to provide suggestions for code improvement and present the refined code. To obtain higher-quality data, we utilize GPT-4 (OpenAI, 2023) to generate code reviews and select cases that are more representative, specifically those with average scores for correctness and clarity greater than 8 or less than 6. Simultaneously, we also focus on situations where there are severe deficiencies in one aspect, *i.e.*, when correctness or clarity is equal to or less than 4.

2.4 Data Cleaning and Decontamination

In practice, we have devised a set of filtering rules to refine our data cleaning and purification scripts. These involve excluding samples based on various criteria, such as response length (either too short or too long), absence of code or insufficient code content, non-compilable code, code failing test cases (pertinent to certain samples), responses structured in notebook form, and instances with excessive textual descriptions preceding the code. After conducting an extensive series of validation experiments, we conclusively opt to eliminate low-quality data meeting either of the following conditions: (a) responses that are excessively brief and lack accompanying code. Such responses typically offer direct answers to the instructions, omitting both the code solution and explanatory annotations. Additionally, these samples frequently present overly simplistic questions in the instructions; (b) code solutions that are non-compilable or fail test cases (relevant to specific samples).

Simultaneously, following (Gunasekar et al., 2023), we employ N-gram similarity, cosine distance of code segment embeddings, and edit distance of the syntax tree of code segments to calculate the similarity between training data and samples in HumanEval and MBPP. We subsequently eliminate samples with excessively high similarity through this process of data filtering and deduplication, resulting in the removal of approximately 6% of the dataset.

2.5 Harmonized AlchemistCoder Dataset

Our *AlchemistCoder* dataset (~200M tokens) consists of four types of multi-source data, includ-

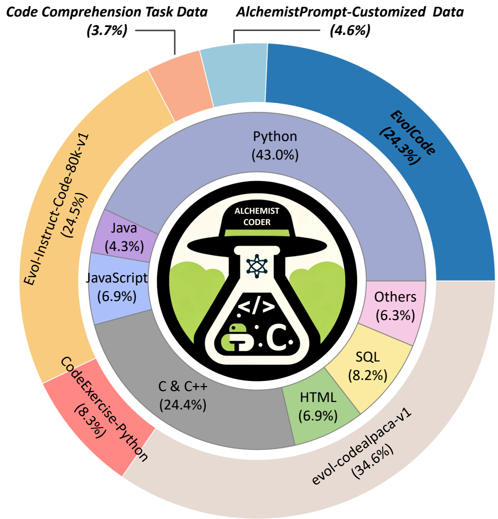


Figure 5: Data distribution analysis of our *AlchemistCoder* dataset. The outer and inner circular diagrams respectively display the distributions of data composition and programming languages, with the data constructed by us marked in bold italic. The data from *AlchemistPrompt*s and code comprehension tasks, accounting for only 8% of the total data volume, is sufficient to endow the harmonized code capability of *AlchemistCoder*.

Table 1: Results of pass@1 on HumanEval (HumanEval+) and MBPP (MBPP+) benchmarks. Models are evaluated in zero-shot on Human Eval and 3-shot on MBPP. We report the results of HumanEval and MBPP consistently from the EvalPlus (Liu et al., 2023b) and the **bold** scores denote the best performance among models of the same size

Model	**Params**	**Base Model**	**HumanEval (+)**	**MBPP (+)**	**Average (+)**
Closed-source Models					
GPT-3.5-Turbo (2022)	-	-	72.6 (65.9)	81.7 (69.4)	77.2 (67.7)
GPT-4-Turbo (2023)	-	-	**85.4 (81.7)**	**83.0 (70.7)**	**84.2 (76.2)**
Open-source Models					
Llama 2-Chat (2023)	70B	Llama 2	31.7 (26.2)	52.1 (38.6)	41.9 (32.4)
CodeLlama-Python (2023)	70B	Llama 2	57.9 (50.0)	72.4 (52.4)	65.2 (51.2)
CodeLlama-Instruct (2023)	70B	CodeLlama	**65.2 (58.5)**	**73.5 (55.1)**	**69.4 (56.8)**
CodeLlama-Python (2023)	34B	Llama 2	51.8 (43.9)	67.2 (50.4)	59.5 (47.2)
WizardCoder-CL (2023b)	34B	CodeLlama-Python	73.2 (56.7)	73.2 (51.9)	73.2 (54.3)
DeepSeek-Coder-Instruct (2024)	33B	DeepSeek-Coder-Base	**78.7 (67.7)**	**78.7 (59.7)**	**78.7 (63.7)**
StarCoder (2023)	15B	-	34.1 (33.5)	55.1 (43.4)	44.6 (38.5)
CodeLlama-Python (2023)	13B	Llama 2	42.7 (36.6)	61.2 (**45.6**)	52.0 (41.1)
WizardCoder-SC (2023b)	15B	StarCoder	**51.9 (45.7)**	**61.9** (44.9)	**56.9 (45.3)**
Llama 2 (2023)	7B	-	14.0 (10.4)	26.1 (17.5)	20.1 (14.0)
StarCoder (2023)	7B	-	24.4 (21.3)	33.1 (29.2)	28.8 (25.3)
CodeLlama-Python (2023)	7B	Llama 2	37.8 (33.5)	57.6 (42.4)	47.7 (38.0)
WizardCoder-CL (2023b)	7B	CodeLlama-Python	48.2 (42.1)	56.6 (42.4)	52.4 (42.3)
DeepSeek-Coder-Base (2024)	6.7B	-	47.6 (41.5)	70.2 (53.6)	58.9 (47.6)
Magicoder-CL (2023)	7B	CodeLlama-Python	60.4 (49.4)	64.2 (46.1)	62.3 (47.8)
MagicoderS-CL (2023)	7B	CodeLlama-Python	70.7 (60.4)	68.4 (49.1)	69.6 (54.8)
Magicoder-DS (2023)	6.7B	DeepSeek-Coder-Base	66.5 (55.5)	75.4 (55.6)	71.0 (55.6)
DeepSeek-Coder-Instruct (2024)	6.7B	DeepSeek-Coder-Base	73.8 (69.5)	72.7 (55.6)	73.3 (62.6)
MagicoderS-DS (2023)	6.7B	DeepSeek-Coder-Base	76.8 (65.2)	75.7 (56.1)	76.3 (60.7)
AlchemistCoder-L (ours)	7B	Llama 2	56.7 (52.4)	54.5 (49.6)	55.6 (51.0)
AlchemistCoder-CL (ours)	7B	CodeLlama-Python	74.4 (68.3)	68.5 (55.1)	71.5 (61.7)
AlchemistCoder-DS (ours)	6.7B	DeepSeek-Coder-Base	**79.9 (75.6)**	**77.0 (60.2)**	**78.5 (67.9)**

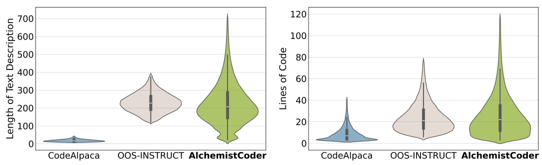


Figure 6: Comparative distribution of text description lengths (left) and code lines (right). Our dataset contains high-quality samples with more diverse distributions.

ing three types of data constructed by us and open-source datasets. Specifically, (a) EvolCode data generated from gpt-3.5-turbo following (Luo et al., 2023b), (b) data customized by *AlchemistPrompts*, (c) data of the code comprehension tasks (*i.e.*, instruction evolution, data filtering, and code review), and (d) open-source datasets including Evol-Instruct-Code-80k-v1 (codefuse ai, 2023b), CodeExercise-Python-27k (codefuse ai, 2023a), and evol-codealpaca-v1 (theblackcat102, 2023). We visualize the distributions of data sources and programming languages involved using two circular graphs in Fig. 5. Concurrently, Fig. 6 reports a distribution of text description lengths and code lines. Compared to CodeAlpaca (Chaudhary, 2023) and OOS-INSTRUCT (Wei et al., 2023), our *AlchemistCoder* dataset presents a notably diverse distribution and maintains moderate overall text description and code lengths, benefiting significantly from the integration of multi-source data along with *AlchemistPrompt*s and code comprehension tasks. This is instrumental in contributing to a comprehensive evolution of code capability.

3 Experiments

In this section, we report results on various benchmarks of code generation and conduct ablation experiments. Furthermore, we present analytical studies to provide a more in-depth demonstration of the efficacy of our *AlchemistCoder*.

3.1 Benchmarks and Implementation Details

Benchmarks. We adopt six code benchmarks: HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023b), HumanEval-X (Zheng et al., 2023), MBPP (Austin et al., 2021), MBPP+ (Liu et al., 2023b), and DS-1000 (Lai et al., 2023). In addition, we access three mainstream benchmarks (MMLU (Hendrycks et al., 2020), BBH (Suzgun et al., 2022), and GSM8K (Cobbe et al., 2021)) to evaluate generalization abilities. All evaluation and benchmark details can be found in AppendixA.

Table 2: Results of pass@1 on HumanEval-X. We present the multilingual code capabilities of our *AlchemistCoder* with the respective base models and competitors. All models possess 6.7B/7B parameters

Model	Python	C++	Go	Java	JS	Avg
Llama 2	14.0	11.0	6.1	11.0	14.0	11.2
CodeLlama	31.7	27.4	12.8	25.6	32.9	26.1
AlchemistCoder-L	**56.7**	**31.1**	**25.6**	**45.1**	**41.5**	**37.1**
CodeLlama-Python	37.8	33.5	30.5	39.6	35.4	35.4
MagicoderS-CL	68.3	47.6	39.6	34.8	**57.9**	49.6
AlchemistCoder-CL	**74.4**	**53.1**	**42.7**	**64.0**	52.4	**57.3**
DeepSeek-Coder-Base	47.6	45.1	38.4	56.1	43.9	46.2
MagicoderS-DS	72.6	**63.4**	51.8	70.7	66.5	65.0
AlchemistCoder-DS	**79.9**	62.2	**59.8**	**72.0**	**68.9**	**68.6**

Table 3: Pass@1 results of models with 6.7/7B parameters on DS-1000. pd, np, tf, sp, skl, torch, and plt represent Pandas, Numpy, Tensorflow, Scipy, Sklearn, Pytorch, and Matplotlib, respectively

Model	pd	np	tf	sp	skl	torch	plt	All
Llama 2	2.4	7.3	6.7	6.6	2.6	1.5	7.7	5.0
CodeLlama	12.0	**27.7**	17.8	**13.2**	12.2	**20.6**	15.5	17.0
AlchemistCoder-L	**13.4**	22.7	**31.1**	11.3	**25.2**	8.8	**29.0**	**20.2**
CodeLlama-Python	16.2	16.4	15.6	17.9	20.0	22.1	38.7	21.0
MagicoderS-CL	25.1	40.9	35.6	29.3	36.5	38.2	51.0	36.7
AlchemistCoder-CL	**30.9**	**43.6**	**46.7**	**30.2**	**37.4**	**41.2**	**52.3**	**40.3**
DeepSeek-Coder-Base	21.3	35.0	26.7	23.6	34.8	25.0	34.8	28.7
MagicoderS-DS	30.6	46.8	44.2	30.2	33.0	29.7	45.2	37.1
AlchemistCoder-DS	**32.0**	**51.7**	**44.5**	**33.1**	**38.4**	**33.8**	**49.8**	**40.5**

Baselines. We compare with the following competitive baselines. Closed-Source Models: GPT-3.5-Turbo (OpenAI, 2022) and GPT-4-Turbo (OpenAI, 2023). Open-Source Models: Llama 2 (Touvron et al., 2023), CodeLlama (Roziere et al., 2023), StarCoder (Li et al., 2023), WizardCoder (Luo et al., 2023b), DeepSeek-Coder (Guo et al., 2024), and Magicoder (Wei et al., 2023).

Supervised Fine-Tuning. We adopt Llama-2-7B, CodeLlama-Python-7B, and DeepSeek-Coder-Base-6.7B as the base models and fine-tune all the base models for 2 epochs using 32 NVIDIA A100-80GB GPUs. We set the initial learning rate at 1e-4. We use Adam optimizer (Loshchilov and Hutter, 2017) and choose a batch size of 2 with a sequence length of 8192.

3.2 Evaluation on Code Generation Task

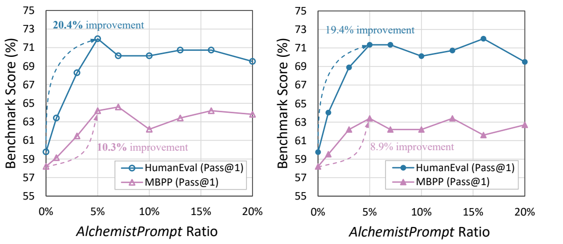
Results on Python Code Generation. We first access HumanEval and MBPP to evaluate the capability of the *AlchemistCoder* series for python code generation. These benchmarks necessitate models to generate code based on the function definitions and subsequently pass the test cases. Models are evaluated in zero-shot on HumanEval and 3-shot on MBPP. The comprehensive comparisons in Tab. 1 and Fig. 1 demonstrate the impressive capabilities of *AlchemistCoder* models. From the results, *AlchemistCoder-L* attains a remarkable performance boost of 42.7% and 28.4% pass@1 scores on HumanEval and MBPP respectively, compared to Llama 2-7B. Notably, *AlchemistCoder-DS* elevates the pass@1 scores to 79.9% and 77.0% on these benchmarks, holding an overall improvement of 33.3%. Moreover, our *AlchemistCoder* series with 7B parameters outperforms larger models (*e.g.*, WizardCoder-CL-34B and CodeLlama-Instruct-70B) and rivals with GPT-3.5-Turbo, significantly bridging the performance gap between closed-source and open-source models.

Results on Multilingual Code Generation. We compare the pass@1 accuracy of the base models and the corresponding fine-tuned *AlchemistCoder* models on Humaneval-X (Zheng et al., 2023). The results presented in Tab. 2 demonstrate that the *AlchemistCoder* series exhibits great improvements (exceeding 50%) for multilingual code generation, delivering comprehensive code capabilities.

Results on Code Generation for Data Science. We further conduct evaluation of data science code completion on DS-1000 (Lai et al., 2023). According to Tab. 3, *AlchemistCoder* models achieve up to 19.2% overall performance gain over base models. Particularly, *AlchemistCoder-CL* achieves an astonishing overall accuracy of 40.3% with relatively better performance in all libraries, demonstrating powerful capabilities in data science workflows.

3.3 Ablation Study

The Recipe of *AlchemistPrompt*s. As illustrated in Sec. 2.2, *AlchemistPrompt*s can further align the instructions and responses of data samples and harmonize the domain gap between multiple sources. To find the appropriate recipe of *AlchemistPrompt*s that maintains a balance between data diversity and domain gap, we conduct ablation experiments on the proportion (0% to 20%) of data customized by *AlchemistPrompt*s. We adopt two settings: (a) augment the original data with its customized variant and report the results of fine-tuning for 2 epochs on CodeLlama-Python-7B; (b) replace the original data and report the results of fine-tuning for the same steps (*i.e.*, keeping the number of tokens used consistent). As shown in Fig. 7, *AlchemistCoder* is particularly enhanced when the proportion of customized data increases from 1% to 5%, and nearly peaks in performance at 5%. Thus, we introduce *AlchemistPrompt*s into 5% of the training set to balance the performance gain and the generation cost. Additionally, both two strategies effectively



(a) Augmentation (b) Replacement

Figure 7: Ablation experiments on the proportion (0% to 20%) of data customized by *AlchemistPrompt*s conducted on CodeLlama-Python-7B. (a) Augment the original data. (b) Replace the original data.

Table 4: Ablation study on the utility of code understanding tasks for the *AlchemistCoder-CL-7B* model, across the HumanEval and MBPP benchmarks

Alchemist Prompt	Instruction Evolution	Data Filtering	Code Review	HumanEval (Pass@1)	MBPP (Pass@1)
-	-	-	-	59.8	58.2
✓	-	-	-	72.0	63.4
✓	✓	-	-	71.3	65.8
✓	✓	✓	-	73.8	67.7
-	✓	✓	✓	65.2	64.6
✓	✓	✓	✓	**74.4**	**68.5**

enhance the performance and validate the efficacy of our approach. To push the limit of *AlchemistCoder*, we employ the augmentation strategy in our performance experiments.

Efficacy of the Code Comprehension Tasks. We conduct an ablation study on the key components of the code comprehension tasks to ascertain their individual contributions to the overall performance. As reported in Tab. 4, compared to the baselines (the first and second rows), the model demonstrates enhanced performance on both benchmarks following the incremental incorporation of code comprehension task data. Notably, the improvement (5.1% regard to the pass@1 metric) is particularly remarkable on MBPP. This indicates the significant contribution of all code comprehension tasks to further programming capabilities.

3.4 Analytical Study

***AlchemistPrompt*s Harmonize the Discrepancy Between Instructions and Responses.** To in-depth verify the efficacy of *AlchemistPrompt*s, we calculate the perplexities of the model in generating responses under given conditions (instructions), *i.e.*, the difference between $\text{Perplexity}(\text{conditional\_instruction} + \text{response})$ and $\text{Perplexity}(\text{response})$, called Conditional Perplexity Discrepancy (CPD). Specifically, we adopt the instructions before and after customization by *AlchemistPrompt*s for comparison, and provide the Kernel Density Estimation of CPD in Fig. 8.

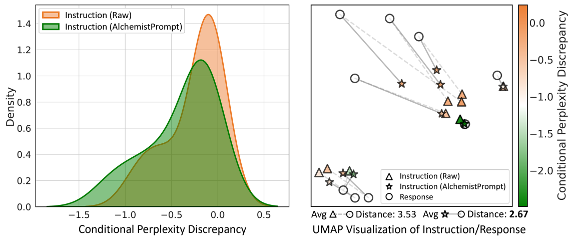


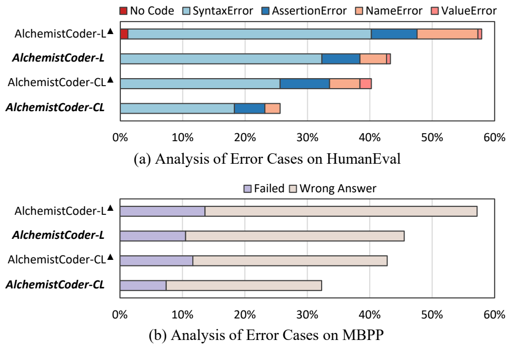
Figure 8: In-depth analysis of the efficacy from *AlchemistPrompt*s. Left: Kernel Density Estimation of conditional perplexity discrepancy. Right: UMAP visualization of 10 instruction/response groups.

Table 5: Results on various benchmarks, encompassing interdisciplinary knowledge (MMLU), comprehensive reasoning (BBH), and mathematical abilities (GSM8K)

Model	Params	MMLU	BBH	GSM8K	Avg
Llama 2	7B	41.1	34.6	16.8	30.8
CodeLlama	7B	31.5	**42.7**	14.4	29.5
AlchemistCoder-L	7B	**43.9**	**42.7**	**25.0**	**37.2**
CodeLlama-Python	7B	26.1	26.7	6.6	19.8
MagicoderS-CL	7B	33.0	**41.5**	18.8	31.1
AlchemistCoder-CL	7B	**42.1**	39.3	**20.2**	**33.9**
DeepSeek-Coder-Base	6.7B	34.0	12.8	22.0	22.9
MagicoderS-DS	6.7B	34.4	43.8	14.3	30.8
AlchemistCoder-DS	6.7B	**38.5**	**45.6**	**28.0**	**37.4**

Clearly, the latter (green) gains smaller overall CPD values, indicating that *AlchemistPrompt*s are beneficial for prediction and can provide effective contextual information. Furthermore, we randomly select 10 groups of these samples and use UMAP (McInnes et al., 2018) to map their feature representations into a 2-D space in the right of Fig. 8. From the fact that the solid lines are generally shorter than the dashed lines, our *AlchemistPrompt*s can harmonize the discrepancy between instructions and responses, leading to higher-quality data for attaining improved instruction-following ability.

***AlchemistCoder* Models are Better Generalists.** To further analyze the comprehensive capabilities of our *AlchemistCoder*, we conduct evaluations on more diversified benchmarks, including MMLU (Hendrycks et al., 2020) for multitask language understanding, BBH (Suzgun et al., 2022) for comprehensive reasoning, and GSM8K (Cobbe et al., 2021) for mathematical ability. The results are presented in Tab. 5 and illustrate that the *AlchemistCoder* models exhibit an overall performance increase of 6.4%, 13.6%, and 14.5% over the base models Llama 2, CodeLlama-Python, and DeepSeek-Coder-Base, respectively. Notably, CodeLlama-Python presents inferior performance on these benchmarks relative to Llama 2, indicating the discrepancy between natural language processing and code capabilities of open-source mod-



(a) Analysis of Error Cases on HumanEval

(b) Analysis of Error Cases on MBPP

Figure 9: Analysis of error cases on HumanEval and MBPP. The horizontal axis denotes the proportion of error cases among all test cases. The symbol ▲ represents the version that has not incorporated *AlchemistPrompt*s and the code comprehension task data.

els. Such divergence can be attributed to "catastrophic forgetting" (Dong et al., 2023; Luo et al., 2023a; Kotha et al., 2023), often occurring when fine-tuning is exclusively concentrated on domain-specific data. By leveraging harmonized multi-source data, our *AlchemistCoder* series achieves more multifaceted and comprehensive capabilities.

Error Case Analysis. To meticulously dissect the improvements brought by our method, we provide an analysis of error cases on HumanEval and MBPP. We compare models before and after the introduction of *AlchemistPrompt*s and code understanding task data. The bar chart shown in Fig. 9 (a) indicates that these two types of key data help to better handle compilation errors (*i.e.*, SyntaxError, NameError, and ValueError), and eliminate the occurrence of no code written in the responses. On the other hand, the results of Fig. 9 (b) on MBPP suggest that the *AlchemistCoder* series incorporated with these two types of data attains stronger programming logic, as evidenced by the clear reduction in the 'Wrong Answer' error cases.

4 Related Work

Code Large Language Models. Early researches (Chen et al., 2021; Nijkamp et al., 2022; Li et al., 2023) focus on collecting massive amounts of code data to develop pretrained Code LLMs. Recent efforts (Luo et al., 2023b; Yu et al., 2023; Wei et al., 2023) are dedicated to fine-tune these pretrained models with specific instructional data to further the coding abilities. For instance, WizardCoder (Luo et al., 2023b) and Magicoder (Wei et al., 2023) construct their instruction tuning datasets based on CodeAlpaca (Chaudhary, 2023) and the stack (Kocetkov et al., 2022) dataset, respectively. In this work, we develop the *AlchemistCoder* series by instruction tuning on optimized multi-source data instead of single-category data as in previous methods, endowing astonishing and harmonized code capability.

Instruction Tuning. Instruction tuning aims to enhance LLMs via fine-tuning pre-trained LLMs using samples of instruction/response pairs. Obtaining high-quality data for instruction tuning is typically challenging and extensive works have been dedicated to this endeavor. For instance, Alpaca (Taori et al., 2023) employs self-instruct (Wang et al., 2022) to generate instruction-following demonstrations. WizardLM (Xu et al., 2023) introduces Evol-Instruct and transforms the instruction data into more complex variants. In addition to Evol-Instruct, we also incorporate the data construction process itself as a form of data into the training. Moreover, although previous works (Ivison et al., 2023; Wang et al., 2023b,a) utilize multiple fine-tuning datasets, we harmonize multi-source data at a fine-grained level.

Learning from Hindsight. The concept of learning from hindsight (Liu et al., 2023a) has been explored in goal-conditioned learning (Kaelbling, 1993; Ganguli et al., 2022). Andrychowicz *et al.* (Andrychowicz et al., 2017) introduce Hindsight Experience Replay (HER) to re-label rewards and facilitate learning from sparse feedback retrospectively. Korbak *et al.* (Korbak et al., 2023) study the influence of human preferences during pre-training, showing improved performance when models are aligned with human preferences. Previous work primarily serves as an alternative to RLFT, utilizing HER to leverage (suboptimal) historical data for model learning. We focus on constructing multi-source data and harmonizing the inherent conflicts within multi-source data through hindsight, to fully tap into the potential of base models.

5 Conclusion

In this paper, we develop *AlchemistCoder*, a series of enhanced Code LLMs fine-tuned on multi-source data. To achieve this, we introduce data-specific *AlchemistPrompt*s leveraging the idea of hindsight to harmonize multi-source data, and design code comprehension tasks to provide data with more dimensions. Performance experiments verify the harmonized and superior code capabilities of the *AlchemistCoder* models. Additionally, extensive analytical studies have been well-designed and delved deeply into the efficacy of our method.

6 Limitations

Currently, GPT-4 holds an advantage in generating high-quality responses, and thus has been chosen as our *Alchemist* model. Although our experiments have verified that customizing only 5-7% of the data can achieve a leap in performance, the generation of *AlchemistPrompt*s is still a significant cost. We will explore fine-tuning open-source models to achieve the free generation of *AlchemistPrompt*s in the future.

7 Ethical Considerations

We use publicly available datasets, benchmarks, and models for training and evaluation, free from any possible harm toward individuals or groups. The generated data from LLMs are relevant to code-related tasks and no personal identification information is involved. Furthermore, we adopt ChatGPT to polish the writing and assist with language.

References

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. 2017. Hindsight experience replay. *Advances in neural information processing systems*, 30.

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*.

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

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

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *CoRR*, abs/2211.12588.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

codefuse ai. 2023a. The codeexercise-python-27k dataset. `https://huggingface.co/datasets/codefuse-ai/CodeExercise-Python-27k`.

codefuse ai. 2023b. The evolinstrutcode dataset. `https://huggingface.co/datasets/codefuse-ai/Evol-instruction-66k`.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv preprint arXiv:2310.05492*.

Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, et al. 2022. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *arXiv preprint arXiv:2209.07858*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y 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*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Hamish Ivison, Yizhong Wang, Valentina Pyatkin, Nathan Lambert, Matthew Peters, Pradeep Dasigi, Joel Jang, David Wadden, Noah A Smith, Iz Beltagy, et al. 2023. Camels in a changing climate: Enhancing lm adaptation with tulu 2. *arXiv preprint arXiv:2311.10702*.

Leslie Pack Kaelbling. 1993. Learning to achieve goals. In *IJCAI*, volume 2, pages 1094–8. Citeseer.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Tomasz Korbak, Kejian Shi, Angelica Chen, Rasika Vinayak Bhalerao, Christopher Buckley, Jason Phang, Samuel R Bowman, and Ethan Perez. 2023. Pretraining language models with human preferences. In *International Conference on Machine Learning*, pages 17506–17533. PMLR.

Suhas Kotha, Jacob Mitchell Springer, and Aditi Raghunathan. 2023. Understanding catastrophic forgetting in language models via implicit inference. *arXiv preprint arXiv:2309.10105*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *ICRA*, pages 9493–9500. IEEE.

Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. 2023a. Chain of hindsight aligns language models with feedback. *arXiv preprint arXiv:2302.02676*, 3.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. 2023a. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *arXiv preprint arXiv:2308.08747*.

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

Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt/.

OpenAI. 2023. Gpt-4 technical report. Technical report.

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

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Stanford alpaca: An instruction-following llama model.

theblackcat102. 2023. The evolved code alpaca dataset. https://huggingface.co/datasets/theblackcat102/evol-codealpaca-v1.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023a. Openchat: Advancing open-source language models with mixed-quality data. *arXiv preprint arXiv:2309.11235*.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. 2023b. How far can camels go? exploring the state of instruction tuning on open resources. *arXiv preprint arXiv:2306.04751*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

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

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R. Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, Heng Ji, and Chengxiang Zhai. 2024. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *CoRR*, abs/2401.00812.

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. *arXiv preprint arXiv:2312.14187*.

Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E Gonzalez. 2023. The wisdom of hindsight makes language models better instruction followers. *arXiv preprint arXiv:2302.05206*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

A Benchmark and Evaluation Details

A.1 HumanEval/HumanEval+

HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al., 2023b) are benchmarks for assessing LLMs' code generation, focusing on functional correctness. HumanEval+ expands on HumanEval by significantly increasing test cases through EvalPlus, using LLM and mutation strategies for more rigorous evaluation. This approach reveals performance drops in models like GPT-4 and ChatGPT against challenging tests, emphasizing the need for diverse test scenarios to accurately evaluate LLMs' coding abilities. For evaluation on HumanEval and HumanEval+, we adopt the prompt designed for HumanEval/HumanEval+ tasks shown in Fig. A1. Following prior works (Zheng et al., 2023; Chen et al., 2023; Wei et al., 2023), we use the greedy decoding strategy and focus on comparing the pass@1 metric.

Prompt

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Complete the following python function.

{ instruction }

Here is the completed function:

Figure A1: Prompt used to evaluate on HumanEval and HumanEval+.

A.2 MBPP/MBPP+

The MBPP (Mostly Basic Python Programming) benchmark (Austin et al., 2021) consists of around 1,000 Python challenges, crowd-sourced to test basic programming skills, including fundamentals and standard library use. Aimed at beginners, each challenge offers a description, solution, and three tests for verifying solution accuracy. MBPP+ (Liu et al., 2023b) is an extension of the MBPP benchmark, utilizing a subset of hand-verified problems from MBPP-sanitized to ensure tasks are well-defined and unambiguous. For evaluation on MBPP and MBPP+, we adopt the three-shot prompt shown in Fig. A2.

A.3 HumanEval-X

HumanEval-X (Zheng et al., 2023) is a comprehensive benchmark that assesses the capabilities of code generation models across multiple programming languages, including Python, C++, Java, JavaScript, and Go. It consists of 820 meticulously created data samples, each accompanied by test cases, making it an invaluable resource for evaluating and improving multilingual code generation models. The benchmark aims to provide insights into the models' proficiency in solving diverse coding challenges and their accuracy in generating functionally correct code in different languages. For evaluation on HumanEval-X, we do not use specific prompts and follow the original test prompts.

A.4 DS-1000

The DS-1000 benchmark (Lai et al., 2023) adapts 1000 different data science coding problems each with unit tests from StackOverflow and checks both execution semantics and surface-form constraints. These realistic problems are drawn from seven popular data science libraries in Python, including Matplotlib (plt), NumPy (np), Pandas (pd), SciPy (scp), Scikit-Learn (sk), PyTorch (py), and TensorFlow (tf). DS-1000 has two modes: completion and insertion, and here we only evaluate completion, as the basic CodeLlama-Python does not support insertion. For evaluation on DS-1000, we do not use specific prompts and follow the original test prompts.

A.5 MMLU

The Massive Multitask Language Understanding (MMLU) benchmark (Hendrycks et al., 2020) is an evaluation framework designed to measure the depth and breadth of knowledge that LLMs possess. It accomplishes this by testing these models across 57 varied tasks in both zero-shot and few-shot scenarios. The tasks encompass a wide array of topics, including basic math, American history, computer science, law, and more, challenging the models to leverage their acquired knowledge to solve complex problems. MMLU seeks to emulate the multifaceted way in which human knowledge and problem-solving skills are assessed, offering a comprehensive gauge of a model's ability to understand and apply information across multiple domains. For evaluation on MMLU, we do not use specific prompts and follow the original test prompts.

A.6 BBH

The BIG-Bench Hard (BBH) Benchmark (Suzgun et al., 2022) is a specialized evaluation framework tailored to rigorously test the capabilities of LLMs. This benchmark targets a selection of tasks that have historically proven challenging for LLMs, focusing on areas where models typically do not exceed average human performance. The BBH Benchmark aims to push the boundaries of what LLMs can achieve by emphasizing complex reasoning, deep understanding, and nuanced interpretation, setting a high bar for model development and performance evaluation. For evaluation on BBH, we do not use specific prompts and follow the original test prompts.

A.7 GSM8K

The GSM8K (Grade School Math 8,000) benchmark (Cobbe et al., 2021) serves as a rigorous evaluation framework for testing the mathematical problem-solving prowess of LLMs. This benchmark comprises a dataset of 8,500 diverse and high-quality math word problems at the grade school level, designed to challenge LLMs with tasks necessitating advanced, multi-step reasoning abilities. GSM8K's primary aim is to gauge how well these models can parse, understand, and solve math problems, thereby offering a comprehensive measure of their capacity for logical reasoning and mathematical computation. By incorporating such a specialized benchmark, researchers can better understand the extent to which LLMs can mimic human-like reasoning in solving complex mathematical scenarios. For evaluation on GSM8K, we do not use specific prompts and follow the original test prompts.

Prompt

```
Question:
You are an expert Python programmer, and here is your task:
{ few_shot_instruction }
 Your code should pass these tests:
{ few_shot_test_case }
Your code should start with a [BEGIN] tag and end with a [DONE] tag.
Answer:
{ few_shot_response }
Question:
You are an expert Python programmer, and here is your task:
{ few_shot_instruction }
 Your code should pass these tests:
{ few_shot_test_case }
Your code should start with a [BEGIN] tag and end with a [DONE] tag.
Answer:
{ few_shot_response }
Question:
You are an expert Python programmer, and here is your task:
{ few_shot_instruction }
 Your code should pass these tests:
{ few_shot_test_case }
Your code should start with a [BEGIN] tag and end with a [DONE] tag.
Answer:
{ few_shot_response }
Question:
You are an expert Python programmer, and here is your task:
{ few_shot_instruction }
 Your code should pass these tests:
{ few_shot_test_case }
Your code should start with a [BEGIN] tag and end with a [DONE] tag.
Answer:
```

Figure A2: Three-shot prompt used to evaluate on MBPP and MBPP+.

B AlchemistCoder Dataset Details

B.1 *AlchemistPrompt*

We provide two samples of *AlchemistPrompt*s in Fig. A4 and Fig. A5.

B.2 Code Comprehension Task Data

B.2.1 Instruction Evolution Data

We provide two samples of instruction evolution task data in Fig. A6 and Fig. A7.

B.2.2 Data Filtering Data

We provide two samples of data filtering task data in Fig. A8 and Fig. A9.

B.2.3 Code Review Data

We design prompt as illustrated in Fig. A3 to obtain high-quality code review task data and we provide

Code Review Data Generation

You are asked to act as a professional code reviewer and your task is to professionally and accurately review the given content and assign a score. The involved programming languages include but are not limited to Python, C, C++, Java, JavaScript, HTML, Haskell, SQL, C#, and PHP.

Please adhere to the following review requirements:
- **Correct:** No syntax and logic errors. The implementations should follow the given function names.
- **Clarity:** Variables should have meaningful names. The arguments and return values of functions should have type annotations.

NOTE (important):
1. Please evaluate and score the code from the aspects of 'correct' and 'clarity', and adhere to the following output format:
[Correct]
<your score (ranges from 0 to 10)>. <your review>
[Clarity]
<your score (ranges from 0 to 10)>. <your review>
[Suggestions]
<your additional suggestions for further improvement>
[Refined Code]
<The refined code according to your suggestions (DO NOT modify the function declarations appearing in the code)>
2. If there is no code in the given content, please answer with [N/A].
3. DO NOT respond with content that is outside of the specified output format.

Given Content:
{{ code from multi-source data}}

Figure A3: Detailed prompt designed for generating code review data.

two samples in Fig. A10 and Fig. A11.

Instruction

Design a function that finds the greatest common divisor (GCD) of two given numbers: 10, 20

Response

Sure, here is a Python function that uses the Euclidean algorithm to find the GCD of two numbers:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return abs(a)

print(gcd(10, 20)) # Output: 10
```

This function keeps replacing `a` with `b` and `b` with `a % b` (the remainder of `a / b`) until `b` is `0`, at which point `a` is the greatest
common divisor. So, the GCD of 10 and 20 would be 10.

You can call the above function with any two numbers you want to find the GCD of.

AlchemistPrompt

Please provide a Python code example for the following task including the expected output after the code execution.

Figure A4: Example #1 of *AlchemistPrompt*s.

Instruction

Compare two strings and output true if they are an anagram. The input strings can contain both lowercase and uppercase letters.

Response

```
def is_anagram(string1, string2):
    string1 = string1.lower()
    string2 = string2.lower()
    string1 = string1.replace(" ", "")
    string2 = string2.replace(" ", "")

    list1 = list(string1)
    list2 = list(string2)
    list1.sort()
    list2.sort()

    sorted_string1 = ''.join(list1)
    sorted_string2 = ''.join(list2)

    # Compare the sorted strings
    if sorted_string1 == sorted_string2:
        return True
    else:
        return False
```

AlchemistPrompt

Write a Python function to check if two input strings are anagrams of each other, considering both lower and uppercase letters. An anagram is a word or phrase formed by rearranging the letters of a different word or phrase.

Figure A5: Example #1 of *AlchemistPrompt*s.

Instruction

Please increase the difficulty of the given programming test question a bit.
You can increase the difficulty using, but not limited to, the following methods:
If the original problem can be solved with only a few logical steps, please add more reasoning steps.

Given Prompt:
Construct an SQL query to add a new column called "Gender" to the table "Employees" with values "Male" and "Female", but also make sure that the values for the "Gender" column are randomly assigned for each employee.

Response

Construct an SQL query to add a new column called "Gender" to the table "Employees" with values "Male" and "Female", but also make sure that the values for the "Gender" column are randomly assigned for each employee. Additionally, ensure that the ratio of male to female employees is approximately 1:1.

Figure A6: Example #1 of instruction evolution task data.

Instruction

Please increase the difficulty of the given programming test question a bit.
You can increase the difficulty using, but not limited to, the following methods:
If the original problem can be solved with only a few logical steps, please add more reasoning steps.

Given Prompt:
Iterate over a list and count the number of occurrences of each item. The list may contain duplicates and non-integer elements. Return a dictionary with the count of each item. The input list can have a maximum length of 1000.

Response

Iterate over a list and count the number of occurrences of each item. The list may contain duplicates and non-integer elements. However, now you need to also handle nested lists and dictionaries within the main list. You should recursively iterate through the nested data structures and count the occurrences of each item. Return a dictionary with the count of each item. Additionally, consider optimizing the code for performance, as the input list can have a maximum length of 10,000.

Figure A7: Example #2 of instruction evolution task data.

Instruction

Below is a code task and its corresponding solution. Please utilize your understanding of code to carefully review this solution.

Task:

Before we proceed, let's briefly talk about Python list indexing. Python uses zero-based indexing, which means the first element of a list is at index 0. The list 'a' here is a mutable sequence of integers, and we are going to modify it using a for loop.
Consider a list `a = [1,3,2,2,3,4]`. What would be the output of the following code?

```
a = [1,3,2,2,3,4]

for i in range(len(a)):
    a[i] = a[i -1]
```

Solution:

The correct output of the given code would be: [3, 1, 3, 2, 2, 3]

The above solution is considered to be of ***low quality***. Choose from the following four options what category this solution belongs to:
a. The solution is too short (less than 20 words) and does not contain code.
b. There are code compilation errors in the solution.
c. The clarity of the code in the solution is too poor.
d. The solution contains multiple sections of code and is not organized into modularized functions.

Your choice (answer only the choice without the reason):

Response

a. The solution is too short (less than 20 words) and does not contain code.

Figure A8: Example #1 of data filtering task data.

Instruction

Below is a code task and its corresponding solution. Please utilize your understanding of code to carefully review this solution.

Task:
Write a function that takes as input a tuple of numbers (t_1,...,t_{N+1}) and returns a tuple of length N where the i-th element of the tuple is equal to t_i * t_{i+1}. The function should also handle negative numbers and zeros, and return an error message if the input is not a tuple.

Solution:
Here is a function implementation in Python:

```
def tuple_product(input_tuple):
    if not isinstance(input_tuple, tuple):
        return "Error: Input is not a tuple"
    return tuple(input_tuple[i] * input_tuple[i + 1] for i in
range(len(input_tuple) - 1))

print(tuple_product((1, 2, 3, 4, 5))
# Outputs: (2, 6, 12, 20)

print(tuple_product((-1, 2, -3, 4, -5)))
# Outputs: (-2, -6, -12, -20)
```

The above solution is considered to be of ***low quality***. Choose from the following four options what category this solution belongs to:
a. The solution is too short (less than 20 words) and does not contain code.
b. There are code compilation errors in the solution.
c. The clarity of the code in the solution is too poor.
d. The solution contains multiple sections of code and is not organized into modularized functions.

Your choice (answer only the choice without the reason):

Response

b. There are code compilation errors in the solution.

Figure A9: Example #2 of data filtering task data.

Instruction

The prompt for generating data is omitted.

Given Content:

You can use the `generate_json` function to get JSON string:

```
import json
def process_value(value):
    if isinstance(value, str):
        return value.upper()
    elif isinstance(value, int):
        return value * 2
    elif isinstance(value, list):
        return [process_value(element) for element in value]
    elif isinstance(value, dict):
        return {key.upper(): process_value(val) for key, val in 
value.items()}
    else:
        return value
def generate_json(dictionary):
    modified_dictionary = process_value(dictionary)
    json_string = json.dumps(modified_dictionary)
    return json_string
```

Response

[Correct]

9. The code is correct. It correctly processes the values according to the specified rules and returns the expected JSON string.

[Clarity]

8. The code is clear and easy to understand. The function names are descriptive and the code follows a logical flow.

[Suggestions]

- It would be helpful to include type annotations for the function arguments and return values. This would make the code more clear and easier to understand.

[Refined Code]

```
import json
from typing import Any, Dict, List, Union

def process_value(value: Union[str, int, List[Any], Dict[str, Any]]) -> 
Union[str, int, List[Any], Dict[str, Any]]:
    if isinstance(value, str):
        return value.upper()
    elif isinstance(value, int):
        return value * 2
    elif isinstance(value, list):
        return [process_value(element) for element in value]
    elif isinstance(value, dict):
        return {key.upper(): process_value(val) for key, val in 
value.items()}
    else:
        return value

def generate_json(dictionary: Dict[str, Any]) -> str:
    modified_dictionary = process_value(dictionary)
    json_string = json.dumps(modified_dictionary)
    return json_string
```

Figure A10: Example #1 of code review task data. For clarity, the prompt for generating data is omitted.

Instruction

The prompt for generating data is omitted.

Given Content:

The error in the code is an incorrect indentation when printing the indentation spaces. The number of spaces printed should be based on the line number, but the code mistakenly indents by a fixed number of spaces. This leads to misalignment and missing characters in some lines. Additionally, there are missing asterisks and periods in the inner loop, which causes the alternating pattern to be incorrect. By fixing the indentation and adding the missing characters, the tower of stars will be displayed correctly.

```
def print_star_tower():
    for i in range(1, 11):
        # Error: Incorrect indentation
         print(' ' * i, end='') # Indentation should be based on line number
        for j in range(1, i*3):
            if j % 2 == 0:
                print('.', end='') # Error: Missing asterisk
            else:
                print('*', end='') # Error: Missing period
        print('')

print_star_tower()
```

Response

[Correct]

3. The code has syntax and there are logic errors that cause the tower of stars to be displayed incorrectly.

[Clarity]

8. The code uses meaningful variable names and is easy to understand.

[Suggestions]

- The code could benefit from adding type annotations to the function and its arguments.
- The logic errors could be fixed by adjusting the indentation and adding the missing characters in the inner loop.

[Refined Code]

```
def print_star_tower() -> None:
    for i in range(1, 11):
        print(' ' * i, end='')
        for j in range(1, i*3):
            if j % 2 == 0:
                print('.', end='')
            else:
                print('*', end='')
        print()
```

In the refined code, the indentation has been fixed and the missing characters have been added. Additionally, type annotations have been added to the function and its arguments to improve clarity.

Figure A11: Example #2 of code review task data. For clarity, the prompt for generating data is omitted.