CODEM: Less Data Yields More Versatility via Ability Matrix

Anonymous ACL submission

Abstract

In the era of code large language models (code LLMs), data engineering plays a pivotal role during the instruction fine-tuning phase. To train a versatile model, previous efforts devote tremendous efforts to crafting instruction data that covers all the downstream scenarios. Nonetheless, this will incur significant expenses in data construction and model training. Therefore, this paper introduces CODEM, a novel data construction strategy, which can efficiently train a versatile model using less data via our newly proposed *ability matrix*. CODEM uses ability matrix to decouple code LLMs' abilities into two dimensions, constructing a lightweight training corpus that only covers a subset of target scenarios. Extensive experiments on HumanEvalPack and MultiPL-E reveal that code LLMs can combine the single-dimensional abilities to master composed abilities, validating the effectiveness of CODEM.

1 Introduction

Code large language models (code LLMs) have been booming recently (Zan et al., 2023; Zhang et al., 2023b). An abundance of code LLMs are released in succession, e.g., Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), StarCoder (Li et al., 2023), and CodeLlama (Rozière et al., 2023). Recent trends have witnessed the versatility of code LLMs, aiming to train a multilingual multitasking model. To meet this need, some efforts (Di et al., 2023; Zheng et al., 2023a) typically created the corresponding instruction training data for every downstream language and task, to fine-tune the model. However, this will entail significant costs in constructing data and training models, considering the countless downstream scenarios (Zheng et al., 2023b; Cassano et al., 2022). For instance, if we enable the model to support 3 tasks and 6 languages, we need to laboriously craft training data for a total of 18 diverse scenarios. Moreover, our expectations for the versatile abilities of code LLMs are continually growing (Muennighoff et al., 2023; Cassano et al., 2022; Zheng et al., 2023a).

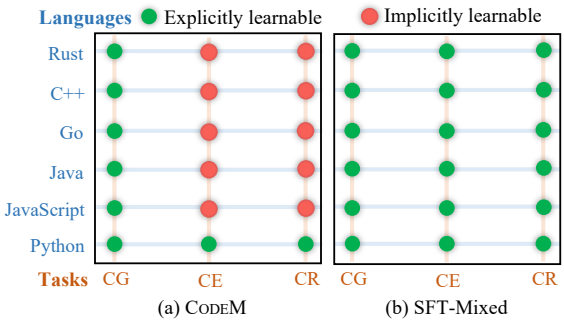

Figure 1: Ability matrix of CODEM and its baseline SFT-Mixed. Code generation, code explanation, and code repair are abbreviated as CG, CE, and CR.

To more efficiently fine-tune code LLMs, we propose a novel data construction strategy, namely CODEM, which can empower **CODE** LLMs with powerful **M**ultilingual **M**ultitasking abilities using less training data via our newly proposed *ability **M**atrix*. This matrix assists CODEM in decoupling the versatile abilities of code LLMs into two ability dimensions (understanding languages and completing tasks). Then, CODEM only requires constructing the training datasets that cover each single-dimensional ability (e.g., Python, CG, or CE), rather than that of all composed abilities (e.g., Python+CG, Java+CE, or Go+CR). We suppose that those composed abilities not covered by the constructed dataset can be implicitly learnable based on two conjectures: (1) code LLMs can generalize based on explicitly learnable abilities; (2) code LLMs can imitate based on the relationships between multiple explicitly learnable abilities.

As shown in Figure 1, compared to its baseline SFT-Mixed, CODEM only needs to cover a subset of 18 composed abilities by picking out one row and one column. CODEM aims to master all composed abilities (e.g., Java+CE) by combining the explicitly learned single-dimensional abilities (e.g.,

Java and CE), and showcases uncompromising performance in these implicitly learned abilities. To evaluate CODEM, we first craft the corresponding instruction training data for each of these 18 scenarios. Then, we conduct experiments to validate CODEM's effectiveness on HumanEvalPack. Extensive results demonstrate that CODEM can rival SFT-Mixed, while utilizing less than half the data.

To more comprehensively verify CODEM, we explore the proposed two conjectures for CODEM. We first verify the generalization of code LLMs across 15 languages, 7 tasks, and even 3 domains on MultiPL-E. Our findings reveal that the model trained on one scenario training data can produce a universal generalization. Secondly, to verify whether the code model can imitate, we deliberately remove key data from CODEM's training corpus to disrupt the preconditions for imitation, and observe performance changes. The results prove the existence of imitation behavior in code LLMs.

In a nutshell, our contributions can be listed as follows: (1) We introduce CODEM, a simple yet effective data construction strategy, which can yield more versatility of code LLMs using less training data via our newly proposed ability matrix. This *matrix* is capable of decoupling multiple scenarios into two ability dimensions, thereby aiding CODEM in achieving efficient training. (2) We carry out extensive experiments on HumanEvalPack and MultiPL-E to validate CODEM's effectiveness, as well as offer some valuable analyses. (3) Our work has been open-sourced at https://anonymous.4open.science/r/CodeM-5190.

2 CODEM

2.1 Task Definition

The goal of CODEM is to obtain a versatile code LLM via efficient instruction fine-tuning. By combining multiple programming languages and coding tasks, we can derive a *scenario set*, formatted as $S = L \times T$, where L denotes a set of target programming languages and T denotes a set of coding tasks. Each element $s = (l, t) \in S$ corresponds to a concrete scenario, requiring code LLMs to complete a task t with the language l. Thus, we in total obtain $|L| \cdot |T|$ different scenarios. Given a scenario set S, a versatile model is expected to support all scenarios and achieve a balanced performance across different scenarios.

Although the scenario set can have countless variations in practice, we mainly focus on one specific version to validate CODEM. Following HumanEvalPack, we regard 3 coding tasks across 6 languages as our target scenario set S in this paper. The 6 languages include Python, JavaScript, Java, Go, C++, and Rust and the 3 tasks include code generation (CG), code explanation (CE) and code repair (CR), leading to a total of 18 scenarios.

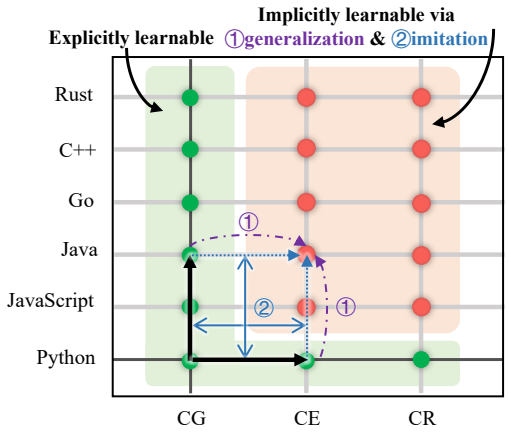

Figure 2: Illustration of ability matrix.

2.2 Approach

CODEM is essentially a data construction strategy for efficient fine-tuning, aiming to empower code LLMs with powerful multilingual and multitasking abilities using less training data. To train a versatile model, a naive idea is to collect training data for each scenario. However, this approach will suffer from overabundant downstream scenarios, resulting in considerable computational resource consumption. CODEM introduces a new concept of ability matrix, aiming to select a subset of scenarios from the fullset S to train the model, while still maintaining an uncompromising performance.

Ability Matrix As depicted in Figure 2, the ability matrix refers to an $|L| \times |T|$ matrix which covers all the abilities requested by the target scenario set. In the ability matrix, the two axes denote two ability dimensions. In detail, the vertical axe corresponds to the ability dimension of understanding the target programming languages, while the horizontal one corresponds to that of completing the target tasks. Based on the two dimensions, we define the $|L| \cdot |T|$ intersections between them as composed abilities.

Upon the above defined ability matrix, CODEM picks out one row and one column to select a scenario subset ($|L| + |T| - 1$ scenarios) from our target scenario set ($|L| \cdot |T|$ scenarios). For each se-

lected scenario, we collect the corresponding training dataset and merge them to obtain a training corpus. This training corpus covers all the single-dimensional abilities without requiring coverage of all composed abilities. Given such a training corpus, the model is expected to combine the explicitly learned single-dimensional abilities, thereby generalizing to all composed abilities. We argue that the ability combination can be realized based on two conjectures (also illustrated in Figure 2): (1) **code LLMs can generalize**: the model can generalize well in every ability dimension, e.g., Python to Java and CG to CE; (2) **code LLMs can imitate**: the model can imitate based on the relationships between multiple abilities that have been explicitly learned. e.g., by imitating the relationship of CG and CE in the Python version, the model can generalize from CG in the Java version (explicitly learnable) to the CE in the Java version (implicitly learnable). If the two conjectures hold, all the composed abilities in the ability matrix can be reachable even though only a subset of them are explicitly learnable. The validity of the claimed two conjectures will be further discussed in Section 4.3.

3 Training Corpus Construction

Regarding the 3 tasks (CG, CE, and CR) across 6 languages (Python, JavaScript, Java, Go, C++, and Rust) mentioned in Section 2.1, totaling 18 scenarios, we craft corresponding instruction dataset for each of these by harnessing the capability of OpenAI's GPT-3.5-turbo. In detail, we start from CodeAlpaca[1], an instruction dataset of code generation, from which we extract those Python-related instances as our seed data. Based on these seed data, we meticulously design prompts for each of the 18 scenarios to request OpenAI's GPT-3.5-turbo, deriving the corresponding instruction datasets. More details about prompt engineering can be seen in Appendix A. In our experiments, each training dataset contains 9.6K data pairs, where a concrete data pair is displayed in Appendix Figure 6.

4 Experiments

4.1 Experimental Setup

4.1.1 Benchmarks

We use HumanEvalPack (Zheng et al., 2023a) as our primary benchmark to evaluate the versatility of code LLMs brought about by CODEM. It provides a series of evaluations for 3 tasks (code generation, code explanation, and code repair) across 6 programming languages (Python, JavaScript, Java, Go, C++, and Rust). Besides that, we also conduct experiments based on MultiPL-E (Cassano et al., 2022), which focuses on the code generation evaluation across 11 programming languages, including Python, C, C++, JavaScript, TypeScript, PHP, Go, Rust, Bash, Java, and Racket. The two benchmarks are created by adapting HumanEval (Chen et al., 2021), which is a hand-written benchmark for Python code generation comprising 164 programming problems with comprehensive test cases.

4.1.2 Evaluation Metrics

Following HumanEvalPack (Muennighoff et al., 2023) and MultiPL-E (Cassano et al., 2022), we adopt pass@1 (Chen et al., 2021) as our metric to evaluate all models. Each model generates one answer using the greedy decoding strategy for each programming problem, and the answer would be executed on the given test cases. Only when all the test cases are passed, the programming problem can be considered solved with the generated code. In this setting, pass@1 can be formulated as $\frac{|P_c|}{|P|}$, where $|P|$ denotes the total number of programming problems and $|P_c|$ represents the number of solved problems. In essence, the pass@1 metric we use can be considered as the accuracy.

4.1.3 Baselines

To prove the effectiveness of CODEM, we design some baselines. The downstream benchmark for CODEM is HumanEvalPack, which covers 18 scenario tasks; CODEM chose 8 of these scenarios to train code LLMs. Hence, the primary baseline of CODEM is using data from 18 scenarios, called SFT-Mixed, as shown in Figure 1 (b). Of note, for a fair comparison, SFT-Mixed contains a total of 9.6K data pairs by default, aligning with the size of each training dataset mentioned in Section 3. To thoroughly showcase CODEM's superiority in terms of data volume, we also craft multiple versions of SFT-Mixed, each with different amounts of data pairs, spanning 4.3K, 19.2K, 28.8K, 48K, 76.8K, and 100K. Each of 18 scenarios in SFT-Mixed has the same data volume (e.g., 533 in 9.6K version of SFT-Mixed). In addition to the mixed-scenario training data, we also compared the models trained on single-scenario data from each of the 18 scenarios. For example, SFT-CG (Py version) in

[1] https://huggingface.co/datasets/sahil2801/CodeAlpaca-20k

Table 1 is trained on the 9.6K instruction data pairs of Python code generation mentioned in Section 3. Furthermore, we also compare a variety of off-the-shelf models, including CodeGeeX2 6B (Zheng et al., 2023a), CodeLlama 7B (Rozière et al., 2023), DeepSeekCoder 7B (Guo et al., 2024), WizardCoder 15B (Guo et al., 2024), CodeFuse 15B (Di et al., 2023), OctoCoder 15B (Muennighoff et al., 2023), StarCoder 1B, 3B, and 7B (Li et al., 2023). CODEM is based on StarCoder 7B for instruction tuning by default. For more implementation details, please refer to Appendix C.

4.2 Main Results

CODEM uses the data corresponding to any row and column in Figure 1, covering a total of 8 scenarios, to train code LLMs. Figure 1 comprises a total of 6 rows and 3 columns. We thus can derive 18 different versions of CODEM, where each of the rows and columns can be combined with each other. For instance, the model, trained on the Python row and the CG column in Figure 1, is named CODEM-CG#Py. Table 1 and Appendix Table 5 display the results of CODEM with varying amounts of training data and all baselines on HumanEvalPack. Among them, CODEM (4.3K) and SFT-Mixed (9.6K) each contain 533 data pairs per scenario, where the former spans 8 scenarios and the latter 18. By examining these results, we find that all CODEM (4.3K) variants perform on par with or even sometimes outperform SFT-Mixed (9.6K) across every downstream task. For instance, CODEM-CE#Java (4.3K) achieves an absolute pass@1 improvement of 2.4% over SFT-Mixed (9.6K) in the code explanation task of the Python version. Meanwhile, CODEM of 9.6K version exceeds SFT-Mixed with equal training scale and even rivals the 29K SFT-Mixed in some downstream tasks. As an illustration, CODEM-CR#C++ (9.6K) eclipses SFT-Mixed (29K) in the C++ code generation, and the Python&C++ code repair task. All these findings highlight CODEM's effectiveness and superiority in constructing training data, even with a lower volume of data. Moreover, we observe that models trained on mixed scenario data, such as CODEM and SFT-Mixed, consistently surpass those trained on single scenario data like SFT-CG/CE/CR in all downstream evaluation tasks. The results underscore the importance of data diversity, suggesting that mixed scenario data can complement each other and further elevate the model's capabilities. In addition, we also observe that CODEM (9.6K) with a 7B parameter size, can hold its ground against WizardCoder 15B and CodeFuse 15B in certain downstream tasks, even with fewer parameters. This further concluded that CODEM possesses remarkable advantages compared to its baselines. Overall, CODEM leverages ability matrix to reduce redundancy among multiple training scenarios, and to develop a versatile model using limited training data. This offers insightful guidance for constructing concise instruction data.

4.3 Conjecture Verification

We implement CODEM based on two conjectures as mentioned in Section 2.2: (1) code LLMs can generalize; (2) code LLMs can imitate. We will explore the validity of them via experiments.

4.3.1 Can Code LLMs Generalize?

We would like to verify whether code LLMs possess generalization capabilities under different training scenarios. For the sake of comprehensiveness of our experiments, in addition to the scenarios provided by HumanEvalPack, we also select a broader range of scenarios. Those new scenarios cover more programming languages of the code generation task (C, TypeScript, PHP, Bash, Racket, Haskell, SQL, HTML, and XML), more coding tasks (code translation, test case generation, code commenting, code-related questing answering), and even more domains (math and natural language). For each scenario, we create the corresponding dataset (see Appendix A and B for more details). We fine-tune StarCoder 7B on each dataset separately, and then evaluate each fine-tuned model on MultiPL-E and HumanEvalPack.

Table 1 and Table 2 respectively display the evaluation results on MultiPL-E and HumanEvalPack of the models trained on various scenarios. It can be observed that code LLMs can generalize across different languages, tasks, and even domains. As an example, SFT-JS, trained on JavaScript, achieves an absolute improvement of 13.1% on pass@1 for the Python code generation task, compared to its base model StarCoder 7B, and even exceeds SFT-Python. We also observe that training on structured query languages (SFT-SQL) and markup languages (SFT-HTML and SFT-XML) can also yield performance improvements in general-purpose programming languages like Python and Java. For instance, SFT-HTML leads to an absolute improvement of 6.8% on pass@1 for C++ code generation. Further-

Model	HumanEvalPack																		Avg.
	Py			JS			Java			Go			C++			Rust			
	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	
Existing Competitive Models																			
CodeGeeX2 6B	34.1	20.7	23.7	11.1	31.0	13.4	22.7	35.3	17.6	18.8	9.7	14.0	27.9	33.5	7.9	19.8	9.7	3.6	19.7
CodeLlama 7B	29.8	31.7	28.6	35.4	31.7	26.8	31.0	40.8	37.1	19.4	18.2	29.2	25.4	34.1	25.6	23.7	19.5	4.8	27.4
DeepSeekCoder 7B	31.0	33.5	27.4	38.5	33.5	26.2	29.7	45.7	40.2	20.1	21.3	30.4	25.4	34.1	26.8	22.4	20.1	8.5	28.6
WizardCoder 15B	59.6	60.9	51.2	39.7	55.4	42.0	36.0	51.2	40.8	27.2	42.6	51.2	40.3	42.0	43.2	35.2	14.6	9.7	41.3
CodeFuse 15B	52.7	59.7	51.8	44.7	56.7	43.2	35.4	48.7	35.9	29.8	41.4	48.7	36.0	45.7	43.2	28.2	14.0	9.7	40.3
OctoCoder 15B	15.5	35.1	30.4	10.5	24.5	28.4	15.1	27.3	30.6	9.7	21.1	30.2	11.8	24.1	26.1	10.2	14.8	16.5	21.2
StarCoder 1B	22.9	12.8	3.6	21.7	11.6	0.6	17.7	15.2	3.6	13.6	8.5	1.2	18.0	14.0	0.6	17.9	3.6	3.0	10.6
StarCoder 3B	29.1	17.0	13.4	24.8	8.5	17.6	25.9	18.9	12.8	20.1	10.9	10.3	19.2	23.1	7.3	21.1	12.8	0.6	16.3
StarCoder 7B	29.1	21.3	26.8	24.8	25.6	29.2	25.9	27.4	19.5	20.1	16.1	22.5	19.2	26.8	11.5	21.1	10.9	5.4	21.3
Py version (Supervised Fine-Tuning is abbreviated as SFT, ditto below)																			
SFT-CG	40.3	36.5	46.3	34.1	26.2	45.1	27.2	28.0	42.6	23.3	18.2	45.1	28.5	26.2	41.4	30.1	18.2	22.5	32.2
SFT-CE	32.2	38.4	39.7	27.3	29.2	40.8	25.9	31.7	42.0	20.1	20.1	42.0	23.6	29.8	33.5	23.7	17.0	19.5	29.8
SFT-CR	39.7	34.1	51.8	27.9	26.8	45.7	27.2	29.2	48.7	22.7	18.9	48.7	26.0	26.8	43.2	26.9	15.8	26.8	32.6
JS version																			
SFT-CG	38.5	34.7	43.2	36.6	27.4	46.3	27.8	28.6	43.9	24.0	17.6	45.1	29.8	27.4	40.8	28.2	18.9	23.7	32.4
SFT-CE	31.0	36.5	37.8	28.5	31.0	43.9	23.4	30.4	42.6	19.4	20.7	42.0	24.2	28.6	34.1	22.4	18.9	18.9	29.7
SFT-CR	36.0	34.1	48.1	29.1	28.0	49.3	26.5	27.4	48.7	22.0	19.5	48.7	25.4	26.8	42.6	27.5	17.6	25.0	32.4
Java version																			
SFT-CG	37.2	35.3	41.4	33.5	25.6	45.7	29.7	30.4	47.5	22.7	18.9	43.9	27.3	26.8	41.4	30.7	17.6	23.7	32.2
SFT-CE	30.4	36.5	35.9	26.7	28.6	40.8	26.5	33.5	46.3	20.7	19.5	42.6	22.9	29.8	32.9	23.0	20.1	19.5	29.8
SFT-CR	36.6	33.5	47.5	28.5	27.4	45.1	28.4	29.2	52.4	21.4	19.5	48.1	26.0	26.2	43.9	26.2	18.2	25.6	32.4
Go version																			
SFT-CG	37.8	34.7	40.8	34.7	25.6	45.1	26.5	28.0	43.2	27.9	20.7	49.3	27.9	27.4	40.8	28.8	18.9	21.9	32.2
SFT-CE	29.8	37.8	36.5	26.0	29.2	41.4	24.0	29.8	42.0	24.6	24.3	46.9	24.8	29.2	33.5	21.7	20.7	17.6	30.0
SFT-CR	34.1	32.3	48.7	26.7	26.8	43.2	25.3	28.0	47.5	28.5	20.1	54.2	26.7	25.6	42.0	25.6	18.9	26.2	32.2
C++ version																			
SFT-CG	36.6	32.9	42.0	35.4	25.0	43.9	27.8	27.4	41.4	24.6	17.6	44.5	32.9	29.8	43.2	30.7	19.5	21.9	32.1
SFT-CE	30.4	39.6	35.3	25.4	27.4	42.6	25.3	31.0	43.2	19.4	21.3	42.6	28.5	34.7	39.0	23.7	19.5	18.2	30.4
SFT-CR	33.5	35.3	46.9	27.9	24.3	42.0	26.5	29.2	47.5	22.0	18.2	49.3	26.7	30.4	45.1	26.2	17.6	25.6	31.9
Rust version																			
SFT-CG	36.6	29.8	43.2	33.5	26.2	44.5	27.2	28.0	42.6	22.0	17.0	42.6	29.1	25.6	39.6	33.3	19.5	25.0	31.4
SFT-CE	32.2	37.8	36.5	26.0	26.8	42.0	24.6	30.4	43.9	19.4	21.3	41.4	21.7	29.8	33.5	26.9	22.5	21.3	29.9
SFT-CR	29.1	31.7	48.1	27.3	25.6	43.2	25.9	28.6	46.9	21.4	18.9	47.5	26.0	26.2	41.4	31.4	20.1	30.4	31.6
Other Domains																			
SFT-Math	30.4	24.3	39.0	31.6	26.8	37.8	25.9	28.0	43.2	22.0	17.0	43.9	27.3	22.5	32.9	23.7	14.6	22.5	28.5
SFT-NLQA	34.7	25.0	37.8	32.9	25.6	38.4	26.5	29.2	40.8	25.3	17.6	39.6	26.7	23.1	32.3	25.0	14.0	23.7	28.8
Mixed Data (baseline)																			
SFT-Mixed (9.6K)	41.6	37.8	51.2	35.4	32.9	47.5	29.1	33.5	50.0	26.6	25.0	48.7	31.6	32.9	44.5	32.6	21.3	31.7	36.3
SFT-Mixed (29K)	47.8	45.7	53.0	42.8	49.3	51.2	34.8	43.2	54.2	29.8	39.0	52.4	40.3	45.1	47.5	33.9	20.7	32.9	42.4
CODEM (4.3K)																			
CODEM-CG#Py	44.0	38.4	51.2	37.2	31.0	46.3	29.1	31.7	48.7	26.6	23.7	49.3	32.2	30.4	44.5	32.0	19.5	29.8	35.9
w/o CG×Py	40.9	36.5	50.6	36.0	29.8	45.1	28.4	28.6	46.9	25.3	23.7	48.1	30.4	28.0	42.0	30.1	17.6	27.4	34.2
w/o CG×Java	43.4	39.0	51.2	37.8	31.7	45.7	27.8	28.0	46.9	27.2	24.3	48.7	32.2	31.0	45.7	30.1	18.9	30.4	35.6
w/o CE×Py	43.4	35.9	51.8	37.2	30.4	46.3	29.1	28.0	49.3	26.6	22.5	49.3	31.6	27.4	43.9	32.6	16.4	29.2	35.0
CODEM-CG#JS	43.4	36.5	50.0	37.8	34.1	48.1	30.3	32.9	47.5	27.9	25.0	48.1	31.6	31.7	43.9	32.6	20.7	28.0	36.1
CODEM-CE#Java	41.6	40.2	48.1	36.0	32.9	45.7	29.7	35.3	50.0	25.3	25.6	48.7	30.4	32.3	43.2	30.7	21.3	28.6	35.9
CODEM-CE#Go	40.9	37.8	50.6	34.7	33.5	46.3	28.4	34.1	48.1	27.2	26.2	50.6	30.4	33.5	44.5	31.4	21.3	29.8	36.1
CODEM-CR#C++	43.4	35.9	51.8	34.1	31.0	48.7	27.8	32.9	50.6	26.6	25.0	49.3	31.0	32.9	45.7	32.0	20.1	31.0	36.1
CODEM-CR#Rust	41.6	37.1	52.4	36.0	32.9	47.5	29.1	32.3	50.0	25.9	24.3	50.0	29.8	31.7	45.1	33.3	22.5	32.3	36.3
CODEM (9.6K)																			
CODEM-CG#Py	46.5	43.2	53.6	42.2	42.0	48.1	35.4	40.8	51.8	29.8	32.9	49.3	40.3	40.2	45.7	36.5	23.1	33.5	40.8
CODEM-CG#JS	45.9	43.9	53.0	44.0	44.5	49.3	34.1	41.4	50.6	29.2	32.9	50.0	41.6	40.2	45.7	37.1	23.7	32.3	41.1
CODEM-CE#Java	42.8	45.1	51.8	40.9	46.9	48.7	34.8	43.2	52.4	27.9	34.1	50.0	39.7	42.6	46.3	35.2	24.3	33.5	41.1
CODEM-CE#Go	42.8	42.0	50.6	41.6	45.7	49.3	33.5	42.0	51.2	30.5	34.7	51.2	38.5	41.4	44.5	35.8	24.3	32.9	40.7
CODEM-CR#C++	45.3	39.0	54.2	39.7	43.9	50.6	32.9	39.6	52.4	28.5	31.7	52.4	40.9	40.8	48.7	33.9	19.5	31.7	40.7
CODEM-CR#Rust	41.6	40.8	53.0	41.6	43.9	50.0	34.1	40.8	53.0	27.9	33.5	51.8	40.9	40.2	46.9	37.1	25.0	35.3	41.0

Table 1: Evaluation results on HumanEvalPack. More results can be seen in Appendix Table 5. Code generation, code explanation, code repair, Average, Python, and JavaScript are abbreviated as CG, CE, CR, Avg., Py, and JS.

more, we discover that different tasks can generalize to each other. For example, SFT-CR, trained on the Python code repair task, achieves an absolute improvement of 6.8% on pass@1 for C++ code generation. By analyzing the results in Table 1, our findings reveal that CG and CR can always improve each other substantially, compared to CE. One plausible reason is that the essence of CG and CR lies in generating code, whereas CE involves understanding code and producing natural language output. More surprisingly, instruction datasets of non-coding domains can also bolster code LLMs in coding. SFT-NLQA and SFT-Math improve the JavaScript code generation performance by 11.8% and 6.8% on pass@1, respectively. That might be due to the enhancement in the model's fundamental abilities, e.g. natural language understanding (Rozière et al., 2023) and reasoning (Dong et al., 2023). Overall, our findings consistently affirm the generalization of code LLMs, thereby solidifying the foundation of CODEM.

4.3.2 Can Code LLMs Imitate?

We design several ablation experiments to demonstrate whether code LLMs possess the imitation capability. The default setting of CODEM-CG#Py covers the Python row and the CG column, as stated in Figure 2. We intentionally exclude an intersection point of the Python row and the CG column from the training corpus of CODEM-CG#Py (abbreviated as w/o CG×Py in Table 1). Upon our conjectures, this will lead to a drop in CODEM-CG#Py's performance on all evaluation tasks: without the training instances of CG×Py scenario, the model can not reach those composed abilities that are not explicitly learnable through imitation. By observing the results in Table 1, compared to CODEM-CG#Py, excluding the CG×Py corpus indeed leads to a decline in performance, further confirming the correctness of our conjectures. Furthermore, we remove the CG×Java corpus from the training corpus of CODEM-CG#Py (w/o CG×Java), which results in the absence of any Java-related data. The results in Table 1 indicate that such the setting only has a negative impact on Java-related tasks, without affecting others. This observation also justifies our conjectures. Additionally, we exclude the Python CE corpus (w/o CE×Py) and observe the same pattern. In summary, the experimental findings confirm the imitation capability of code LLMs and further underscore CODEM's effectiveness.

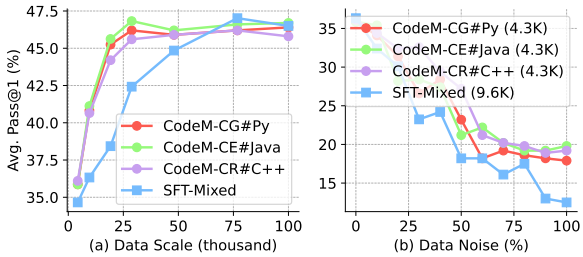

Figure 3: Performance analysis of CODEM and its baselines in terms of data scale and data noise.

4.4 Closer Analysis

Data Scale CODEM takes the advantage of using less data to yield more versatility. To demonstrate CODEM's advantage, we plot the performance of CODEM and its baseline SFT-Mixed across various data scales on HumanEvalPack, as shown in Figure 3 (a). We observe that, at any data scale, various versions of CODEM consistently outperform SFT-Mixed. For example, with a dataset scale of 19.2K, CODEM-CE#Java outperforms SFT-Mixed by 7.2% in pass@1, achieving performance on par with the 48K version of SFT-Mixed. This observation demonstrates CODEM's advantage in terms of data scale. Additionally, with sufficient data, CODEM can achieve peak performance comparable to that of SFT-Mixed. This indicates that with CODEM, the model can seamlessly combine single-dimensional abilities within ability matrix, achieving uncompromising performance on composed abilities without explicit learning. Compared to SFT-Mixed, CODEM can converge to its peak performance more rapidly, where the former reaches its peak at 100K and the latter at 28.8K, further underscoring CODEM's advantage.

Data Quality We intend to explore the robustness of CODEM and its baseline with respect to data quality. In Figure 3 (b), we deliberately introduce noise for CODEM (4.3K) and SFT-Mixed (9.6K), by creating data pairs with inconsistent instructions and responses, at different noisy levels from 0% to 100% at 10% intervals. We observe that at most noisy levels, CODEM outperforms its baseline. Also, as the noisy level increases, CODEM tends to stabilize around a 20% pass@1 rate, whereas SFT-Mixed continues to drop to 12.5% and is still on a downward trend. This exhibits CODEM's greater robustness to noise data.

Data Redundancy In section 4.2, compared to SFT-Mixed, we claim that CODEM reduces data redundancy by selecting one row and one column

Model	MultiPL-E Benchmark											Avg.
	Py	C	C++	JS	TS	PHP	Go	Rust	Bash	Java	Racket	
Existing Competitive Models												
CodeGeeX2 6B	34.1	24.6	27.9	11.1	22.6	23.6	18.8	19.8	6.9	22.7	11.1	20.3
CodeLlama 7B	29.8	24.6	25.4	35.4	33.3	23.6	19.4	23.7	10.7	31.0	11.1	24.4
DeepSeekCoder 7B	31.0	25.9	25.4	38.5	32.0	27.3	20.1	22.4	12.0	29.7	12.4	25.2
WizardCoder 15B	48.4	34.5	40.3	39.7	45.9	40.9	27.2	35.2	16.4	36.0	14.2	34.4
CodeFuse 15B	52.7	33.3	36.0	44.7	42.1	39.1	29.8	28.2	13.9	35.4	14.9	33.6
StarCoder 1B	15.5	9.2	11.8	10.5	15.7	9.3	9.7	10.2	2.5	15.1	5.5	10.5
StarCoder 3B	22.9	16.6	18.0	21.7	25.1	21.1	13.6	17.9	4.4	17.7	7.4	16.9
StarCoder 7B	29.1	22.8	19.2	24.8	28.3	24.2	20.1	21.1	6.3	25.9	11.8	21.2
Programming Languages (code generation version)												
SFT-Py	$40.3^{11.2}$	$27.1^{4.3}$	$28.5^{9.3}$	$34.1^{9.3}$	$37.7^{9.4}$	$34.1^{9.9}$	$23.3^{3.2}$	$30.1^{9.0}$	$8.2^{1.9}$	$27.2^{1.3}$	$16.7^{4.9}$	27.9
SFT-C	$37.8^{8.7}$	$29.6^{6.8}$	$34.1^{14.9}$	$35.4^{10.6}$	$36.4^{8.1}$	$34.1^{9.9}$	$29.2^{9.1}$	$26.9^{5.8}$	$10.1^{3.8}$	$32.2^{6.3}$	$15.5^{3.7}$	29.2
SFT-C++	$36.6^{7.5}$	$31.4^{8.6}$	$34.7^{15.5}$	$36.0^{11.2}$	$37.7^{9.4}$	$31.0^{6.8}$	$24.6^{4.5}$	$27.5^{6.4}$	$9.4^{3.1}$	$32.9^{7.0}$	$14.9^{3.1}$	28.8
SFT-JS	$42.2^{13.1}$	$25.9^{3.1}$	$32.2^{13.0}$	$36.0^{11.2}$	$41.5^{13.2}$	$35.4^{11.2}$	$24.0^{3.9}$	$28.8^{7.7}$	$8.2^{1.9}$	$29.7^{3.8}$	$15.5^{3.7}$	29.0
SFT-TS	$36.0^{6.9}$	$27.1^{4.3}$	$26.0^{6.8}$	$34.7^{9.9}$	$37.1^{8.8}$	$31.0^{6.8}$	$22.7^{2.6}$	$26.2^{5.1}$	$10.1^{3.8}$	$29.1^{3.2}$	$12.4^{0.6}$	26.6
SFT-PHP	$36.0^{6.9}$	$26.5^{3.7}$	$28.5^{9.3}$	$35.4^{10.6}$	$36.4^{8.1}$	$37.8^{13.6}$	$22.0^{1.9}$	$28.2^{7.1}$	$7.5^{1.2}$	$28.4^{2.5}$	$13.0^{1.2}$	27.2
SFT-Go	$34.7^{5.6}$	$28.3^{5.5}$	$30.4^{11.2}$	$34.1^{9.3}$	$35.2^{6.9}$	$31.0^{6.8}$	$29.2^{9.1}$	$26.2^{5.1}$	$10.1^{3.8}$	$29.1^{3.2}$	$16.7^{4.9}$	27.7
SFT-Rust	$37.2^{8.1}$	$29.6^{6.8}$	$29.1^{9.9}$	$34.1^{9.3}$	$35.8^{7.5}$	$34.7^{10.5}$	$24.6^{4.5}$	$31.4^{10.3}$	$7.5^{1.2}$	$30.3^{4.4}$	$13.6^{1.8}$	28.0
SFT-Bash	$40.9^{11.8}$	$27.7^{4.9}$	$32.9^{13.7}$	$41.6^{16.8}$	$42.1^{13.8}$	$31.6^{7.4}$	$28.5^{8.4}$	$30.1^{9.0}$	$26.5^{20.2}$	$29.1^{3.2}$	$12.4^{0.6}$	31.2
SFT-Java	$40.3^{11.2}$	$27.7^{4.9}$	$30.4^{11.2}$	$32.9^{8.1}$	$34.5^{6.2}$	$31.6^{7.4}$	$21.4^{1.3}$	$29.4^{8.3}$	$8.8^{2.5}$	$32.2^{6.3}$	$14.2^{2.4}$	27.6
SFT-Racket	$40.9^{11.8}$	$29.0^{6.2}$	$33.5^{14.3}$	$36.0^{11.2}$	$35.2^{6.9}$	$29.8^{5.6}$	$27.9^{7.8}$	$27.5^{6.4}$	$8.2^{1.9}$	$27.8^{1.9}$	$21.7^{9.9}$	28.9
SFT-Haskell	$37.8^{8.7}$	$25.9^{3.1}$	$29.1^{9.9}$	$33.5^{8.7}$	$33.9^{5.6}$	$31.6^{7.4}$	$23.3^{3.2}$	$27.5^{6.4}$	$10.1^{3.8}$	$26.5^{0.6}$	$12.4^{0.6}$	26.5
SFT-SQL	$31.0^{1.9}$	$20.9^{1.9}$	$23.6^{4.4}$	$27.9^{3.1}$	$32.0^{3.7}$	$25.4^{1.2}$	$22.0^{1.9}$	$26.9^{5.8}$	$8.8^{2.5}$	$24.0^{1.9}$	$11.1^{0.7}$	23.1
SFT-HTML	$32.2^{3.1}$	$20.3^{2.5}$	$26.0^{6.8}$	$32.9^{8.1}$	$32.0^{3.7}$	$29.8^{5.6}$	$22.0^{1.9}$	$25.6^{4.5}$	$7.5^{1.2}$	$28.4^{2.5}$	$12.4^{0.6}$	24.5
SFT-XML	$36.0^{6.9}$	$19.7^{3.1}$	$24.8^{5.6}$	$31.6^{6.8}$	$31.4^{3.1}$	$24.8^{0.6}$	$22.7^{2.6}$	$25.0^{3.9}$	$8.2^{1.9}$	$26.5^{0.6}$	$11.8^{0.0}$	23.9
Other Tasks (Python version)												
SFT-CE	$32.2^{3.1}$	$21.6^{1.2}$	$23.6^{4.4}$	$27.3^{2.5}$	$30.1^{1.8}$	$25.4^{1.2}$	$20.1^{0.0}$	$23.7^{2.6}$	$8.2^{1.9}$	$25.9^{0.0}$	$10.5^{1.3}$	22.6
SFT-CR	$39.7^{10.6}$	$24.6^{1.8}$	$26.0^{6.8}$	$27.9^{3.1}$	$32.0^{3.7}$	$28.5^{4.3}$	$22.7^{2.6}$	$26.9^{5.8}$	$9.4^{3.1}$	$27.2^{1.3}$	$14.9^{3.1}$	25.4
SFT-CT	$29.8^{0.7}$	$22.2^{0.6}$	$22.9^{3.7}$	$29.8^{5.0}$	$30.1^{1.8}$	$26.0^{1.8}$	$20.7^{0.6}$	$24.3^{3.2}$	$6.9^{0.6}$	$26.5^{0.6}$	$11.8^{0.0}$	22.8
SFT-TestCase	$34.1^{5.0}$	$22.8^{0.0}$	$26.7^{7.5}$	$30.4^{5.6}$	$30.8^{2.5}$	$24.8^{0.6}$	$21.4^{1.3}$	$22.4^{1.3}$	$8.2^{1.9}$	$29.7^{3.8}$	$13.0^{1.2}$	24.0
SFT-Comment	$29.8^{0.7}$	$22.8^{0.0}$	$25.4^{6.2}$	$29.1^{4.3}$	$29.5^{1.2}$	$26.0^{1.8}$	$20.1^{0.0}$	$24.3^{3.2}$	$8.2^{1.9}$	$24.0^{1.9}$	$13.0^{1.2}$	22.9
SFT-CodeQA	$34.7^{5.6}$	$27.7^{4.9}$	$27.9^{8.7}$	$34.1^{9.3}$	$31.4^{3.1}$	$24.8^{0.6}$	$26.6^{6.5}$	$25.6^{4.5}$	$12.6^{6.3}$	$28.4^{2.5}$	$12.4^{0.6}$	26.0
Other Domains												
SFT-Math	$30.4^{1.3}$	$22.2^{0.6}$	$27.3^{8.1}$	$31.6^{6.8}$	$32.0^{3.7}$	$24.2^{0.0}$	$22.0^{1.9}$	$23.7^{2.6}$	$8.2^{1.9}$	$28.4^{2.5}$	$13.0^{1.2}$	23.9
SFT-NLQA	$34.7^{5.6}$	$25.9^{3.1}$	$26.7^{7.5}$	$36.6^{11.8}$	$33.4^{5.1}$	$27.3^{3.1}$	$25.3^{5.2}$	$25.0^{3.9}$	$8.8^{2.5}$	$27.8^{1.9}$	$13.6^{1.8}$	25.9

Table 2: Evaluation results on MultiPL-E. The numbers in red and green represent the absolute increase and decrease compared to the base model StarCoder 7B. We abbreviate Python, JavaScript, TypeScript, Average, code explanation, code repair, and code-related/natural-language question answering as Py, JS, TS, Avg., CE, CR, CodeQA/NLQA.

from 6×3 matrix. To prove this, we select not only one row (language) and column (task) from Figure 1 but also $\mathbf{m}$ rows and $\mathbf{n}$ columns[2], where $\mathbf{m} \in \{1,2,3,4,5,6\}$ and $\mathbf{n} \in \{1,2,3\}$. We summarize the experimental results in Figure 4. We observe that increasing the selected scenarios ($\mathbf{m}$>1 and $\mathbf{n}$>1) does not yield performance gain, compared to CODEM ($\mathbf{m}$=1 and $\mathbf{n}$=1). For example, the pass@1 is 35.8% for $\mathbf{m}$=6, $\mathbf{n}$=2, compared to 35.9% for $\mathbf{m}$=1, $\mathbf{n}$=1. This suggests there exists redundancy in the training data among the 18 scenarios, further underscoring CODEM's advantages.

[2]We randomly select rows and columns in the experiments.

Base Models CODEM can be built upon different base models, such as StarCoder (1B, 3B, 7B, 15B), CodeLlama (7B, 13B, 34B), and DeepSeekCoder (1B, 7B, 33B). We thus plot the results of CODEM-CG#Py (4.3K) and its baseline SFT-Mixed (9.6K) in Figure 5. We observe that CODEM consistently improves the average pass@1 of various base models on HumanEvalPack, each achieving an improvement of over 10%. For instance, CODEM-CG#Py brings an absolute improvement of 19.5% for StarCoder 7B in the average pass@1. Notably, in all settings, CODEM with merely 4.3K training data can compete with SFT-Mixed with 9.6K, demonstrating the efficiency of CODEM. In this paper,

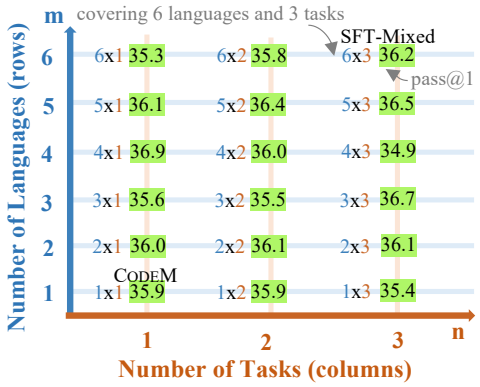

Figure 4: Pass@1 (%) performance of models trained using scenario data from $\mathbf{m}$ rows and $\mathbf{n}$ columns in Figure 1, where $\mathbf{m} \in \{1, 2, 3, 4, 5, 6\}$ and $\mathbf{n} \in \{1, 2, 3\}$.

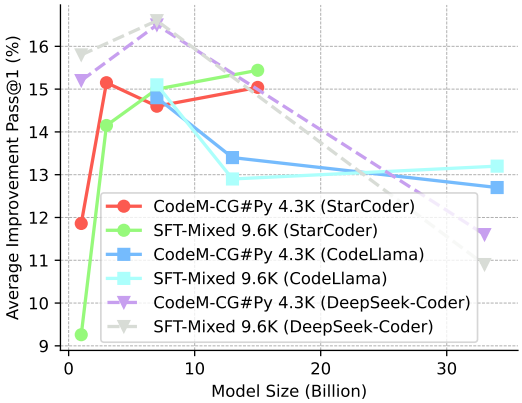

Figure 5: Average improvement pass@1 on HumanEvalPack of CodeM trained on various base models.

CodeM's base model defaults to StarCoder, due to its fully open-source pre-training data, unlike CodeLlama and DeepSeekCoder.

5 Related Work

5.1 Code Large Language Models

Codex (Chen et al., 2021) with 12-billion parameters is able to solve Python programming problems automatically. This remarkable success triggered a significant buzz in both the academic and industrial realms. Followed by Codex, a plenty of code LLMs are released, including AlphaCode (Li et al., 2022), PaLM-Coder (Chowdhery et al., 2022), CodeGen (Nijkamp et al., 2023), InCoder (Fried et al., 2023), CodeT5 (Wang et al., 2021), PanGu-Coder (Christopoulou et al., 2022), PyCodeGPT (Zan et al., 2022), SantaCoder (Allal et al., 2023), CodeGeeX (Zheng et al., 2023a), StarCoder (Li et al., 2023), CodeLlama (Rozière et al., 2023), phi-1/1.5/2 (Gunasekar et al., 2023), CodeFuse (Di et al., 2023), and DeepSeek-Coder (Guo et al., 2024). These above models are trained on a large-scale code corpus and achieve impressive code generation performance. Recent works (Ouyang et al., 2022; Zhang et al., 2023a) have witnessed the instruction tuning technique that can teach LLMs how to follow instructions. In the realm of code generation, WizardCoder (Luo et al., 2023), PanGu-Coder2 (Shen et al., 2023), CodeLlama-instruct (Rozière et al., 2023), Phind (Name, 2023), and DeepSeekCoder-instruct (Guo et al., 2024) also harness this technique to unlock their code related potential. In this paper, we fine-tune these off-the-shelf models on our crafted multi-lingual multitasking instruction data, to derive a versatile and powerful model.

5.2 Instruction Data

Instruction fine-tuning can unlock the potential of LLMs. In this process, constructing what kind of instruction data is a highly fascinating research topic (Zhang et al., 2023a; Zhao et al., 2023). Some studies (Zhou et al., 2023; Cao et al., 2023; Chen et al., 2023) claim that high-quality instruction data will yield significant performance improvements. Beyond data quality, some efforts also highlight the importance of data diversity (Bukharin and Zhao, 2024; Kapania et al., 2023). Recent works also propose some tricks for training LLMs from the perspective of data composition (Dong et al., 2024) and training sequence (Wang et al., 2023; Guo et al., 2024). Unlike prior studies, this paper aims to investigate how to enable code LLMs to achieve more versatility performance with less data, on multiple tasks across multiple languages.

6 Conclusion and Future Work

In this paper, we propose CodeM, which aims to empower code LLMs with powerful multilingual multitasking capabilities using less training data by leveraging ability matrix. The matrix assisting CodeM divides the model's intrinsic abilities into two dimensions: languages and tasks, then guides the construction of instruction training data. Extensive experiments on HumanEvalPack and MultiPL-E demonstrate the effectiveness and superiority of CodeM. Furthermore, we validate two conjectures of CodeM: generalization and imitation, and obtain many insightful findings. In our future work, we would like to delve into code LLMs to uncover more effective and intriguing data construction strategies.

Limitations

While this paper showcases numerous intriguing findings on the generalization of code LLMs, it also poses a few limitations as follows:

- CODEM constructs instruction data from two dimensions: programming languages and tasks, with the expectation that the two dimensions can be better blended together to empower a more versatile code LLM. Technically speaking, CODEM is also capable of handling additional dimensions, such as data domains and data structures, which will be explored in our forthcoming research.

- To ensure the consistency of data origin and quality, as well as the fairness of the experiments, the majority of our experiments are conducted using GPT-3.5-generated data. But employing CODEM in practice might also involve data from other sources, potentially posing threats to our approach. Also, we do not verify the correctness of content generated by GPT-3.5, which is in line with previous works (Luo et al., 2023; Shen et al., 2023).

- Our work, including training, inference, and API requests to OpenAI's GPT-3.5-turbo, requires a high cost in computational resources. Therefore, we will open-source our efforts to foster the rapid advancement of this field.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alexander Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, J. Poirier, Hailey Schoelkopf, Sergey Mikhailovich Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Franz Lappert, Francesco De Toni, Bernardo Garc'ia del R'io, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luisa Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Christopher Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *ArXiv*, abs/2301.03988.

Alexander Bukharin and Tuo Zhao. 2024. Data diversity matters for robust instruction tuning.

Yihan Cao, Yanbin Kang, and Lichao Sun. 2023. Instruction mining: High-quality instruction data selection for large language models. *arXiv preprint arXiv:2307.06290*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sy Duy Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *ArXiv*, abs/2208.08227.

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. 2023. AlpaGasus: Training a better alpaca with fewer data.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhong-Yi Li,

Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li yu Yan, Pingyi Zhou, Xin Wang, Yu Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jia Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program synthesis with function-level language modeling. *ArXiv*, abs/2207.11280.

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*.

Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2023. CodeFuse-13B: A pretrained multilingual code large language model.

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.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2024. How abilities in large language models are affected by supervised fine-tuning data composition.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need.

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

Shivani Kapania, Alex S Taylor, and Ding Wang. 2023. A hunt for the snark: Annotator diversity in data practices. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–15.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you!

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097.

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*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Author's Name. 2023. Code llama beats gpt-4: A deep dive. Accessed: 2023-10-31.

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

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and

Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open foundation models for code.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. PanGu-Coder2: Boosting large language models for code with ranking feedback.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

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.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual pre-training on sketches for library-oriented code generation. In *International Joint Conference on Artificial Intelligence*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023a. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. Unifying the perspectives of nlp and software engineering: A survey on language models for code.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023a. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: Less is more for alignment.

A Prompts for Requesting OpenAI's GPT-3.5-turbo

To collect the training datasets for our selected scenarios where each instance is an instruction-response pair (an example is presented in Figure 6), we craft two types of prompts to request GPT-3.5-turbo: the *instruction request prompt* for instruction generation, and the *response request prompt* for requesting a corresponding response. The prompt formats differ between different languages, tasks, and domains.

For *instruction request prompts*, which are presented in Appendix Table 3, we typically ask OpenAI's GPT-3.5-turbo to craft a new task based on given seed tasks and optional additional information. The placeholder {seed task} in the prompt

Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

Instruction:
Design a web page that displays a message when loaded. The message should be "Hello, World". Use HTML code to achieve this.

Response:
```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h1>Hello, World</h1>
</body>
</html>
```

Figure 6: A HTML training example of our crafted instruction-response pairs.

templates corresponds to a seed task[3]; {TabFact topic} corresponds to a brief topic randomly selected from the dataset `TabFact`[4]; {language} corresponds to a specified programming language.

For *response request prompts*, which are presented in Appendix Table 4, we typically utilize the obtained instructions to request a corresponding answer from GPT-3.5-turbo. The placeholder {programming task} in the prompt templates corresponds to an instruction obtained via instruction requests or retrieving from the existing dataset (e.g., `SFT-Python`); {code snippet} corresponds to a suitable code snippet, which can be retrieved from the already crafted dataset (e.g., `SFT-Python`).

B Scenario Selection for Generalization Verification

As mentioned in Section 3 and 4.3.1, in this paper, we have crafted instruction data for diversified training scenarios, covering a total of 15 programming languages, 7 code-related tasks, and 3 domains. This section will explain the reasons behind the selection of these scenarios.

15 Programming Languages We meticulously select 15 languages, including Python, C, C++, JavaScript, TypeScript, PHP, Go, Rust, Bash, Java, Racket, Haskell, SQL HTML, and XML. To derive a diversified set of training scenarios, these selected languages cover a broad range of language features, considering aspects of programming paradigm, type system, memory management, etc., as shown in Appendix Table 6. In addition, we also take into account the share of these languages in the pre-trained corpus[5] of our base model, ranging from 0% to 15% all over. For each of the 15 languages, we have created an instruction dataset of code generation correspondingly (see Appendix A for more details).

7 Tasks We select 7 code-related tasks: code generation, code explanation, code repair, code translation, test case generation, code commenting, and code-related questing answering. We select these tasks because they represent a comprehensive range of skills required in software development and maintenance (Zheng et al., 2023b; Zhang et al., 2023b). For each of the 7 tasks, we have created multiple language versions of instruction data, including Python, JavaScript, Java, Go, C++, and Rust (see Appendix A for more details).

3 Domains We select 3 domains including code, math, and natural language. We choose the latter two domains based on the assumption that arithmetic reasoning and natural language understanding may also bolster the code-related capabilities of code LLMs (Rozière et al., 2023; Name, 2023). As for the instruction dataset of two non-code domains, we directly extract 9.6K data pairs from the GSM8K (Cobbe et al., 2021) and WizardLM (Xu et al., 2023) datasets.

C Implementation Details

C.1 Training and Inference

We fine-tune StarCoder using PyTorch (Paszke et al., 2019), transformers (Wolf et al., 2019), and DeepSpeed (Rajbhandari et al., 2020) with FP16 enabled. During instruction tuning, we set the batch size to 8, the epoch to 4, the max length to 1024, the warmup ratio to 0.03, the gradient accumulation steps to 16, the save steps to 10, and the learning rate to 2e-5 with cosine scheduler. When fine-tuning on one training dataset, we report the

[3]The seed task is CodeAlpaca mentioned in Section 3.

[4]https://huggingface.co/datasets/tab_fact

[5]https://huggingface.co/datasets/bigcode/the-stack

Model-Task	Prompt to obtain instruction
SFT-Python SFT-C SFT-C++ SFT-JS SFT-TS SFT-PHP SFT-Go SFT-Rust SFT-Bash SFT-Java SFT-Racket SFT-Haskell	I want you act as a Programming Contest Designer. Your objective is to rewrite a given programming task into a more complex version to make it more educational.\nYou can increase the difficulty using, but not limited to, the following method:\n- Add new constraints and requirements to the original problem, adding approximately 10 additional words.\n- If the original problem can be solved with only a few logical steps, please add more reasoning steps.\n\nYour response is the rewritten programming task (#Rewritten Task#).\nThe #Rewritten Task# must be reasonable and must be understood and responded by humans, and also solvable with {language} code. It should not be dependent on the #Given Task#. Your rewriting cannot omit the non-text parts such as the table and code in #Given Task#. Also, please do not omit the input in #Given Task#.\n**The rewritten task and the given task should have the similar length.**\n**The rewritten task should ask for a function-level code solution.**\n'#Given Task#', '#Rewritten Task#', 'given task', and 'rewritten task' are NOT allowed to appear in #Rewritten Task#.\n\n#Given Task#\n{seed task}\n\n#Rewritten Task#\n
SFT-SQL SFT-HTML SFT-XML	I want you act as a prompt engineer. Your objective is to create an {language} code generation task by drawing inspiration from the #Given Topic#. The task should be educational to junior programmers, just like the #Reference Task#, but can also involve some advanced skills of this specific programming language.\n\nYour response is the #Code Generation Task#, asking an AI code assistant to generate an {language} code snippet.\nThe #Code Generation Task# must be reasonable and must be understood and responded by humans, and also solvable with {language} code. The #Code Generation Task# is not necessarily related to the #Given Topic#, but should be in the same domain of it.\n'#Code Generation Task#, '#Given Topic#', 'code generation task', and 'given topic' are NOT allowed to appear in #Code Generation Task#.\n\n#Given Topic#\n{TabFact topic}\n\n#Reference Task#\n{seed task}\n\n#Code Generation Task#\n
SFT-CR	I want you act as a Programming Task Designer. Your objective is to create a code fix task based on a given programming task.\nYou SHOULD increase the difficulty of the given programming task and rewrite it into a code fix task, including a programming task and a piece of buggy code. You can create the code fix task using the following method:\n- Add new constraints and requirements to the original problem, adding approximately 10 additional words.\n- If the original problem can be solved with only a few logical steps, please add more reasoning steps.\n\nYour response is the rewritten code fix programming task.\nThe #Code Fix Task# contains a description which must be reasonable and must be understood and responded by humans, attached with a piece of buggy code in {language}. #Code Fix Task# should not be dependent on the #Given Task#. Your rewriting cannot omit the non-text parts such as the table and code in #Given Task#. Also, please do not omit the input in #Given Task#.\n'#Given Task#', '#Code Fix Task#', 'given task', and 'code fix task' are NOT allowed to appear in #Code Fix Task#.\n\n#Given Task#\n{seed task}\n\n#Code Fix Task#\n

Table 3: Prompts of crafting instructions of training datasets by requesting OpenAI's GPT-3.5. The "Model-Task" column corresponds to the "Model" column in Tables 1 and Table 2.

results of the last checkpoint. In our experiments, all results are truncated to one decimal place. The number of samples in MultiPL-E for Python, C, C++, JavaScript, TypeScript, PHP, Go, Rust, Bash, Java, and Racket are 161, 162, 161, 161, 159, 161, 154, 156, 158, 158, and 161, while HumanEvalPack uniformly contains 164 programming problems for various evaluation scenarios.

We use the same prompt for training and inference for each task. To align the forms across di-

Model-Task	Prompt to obtain response for instruction
SFT-PLs SFT-CG SFT-CR	Below is an instruction that describes a programming task. Write a response that appropriately completes the request. You should use {language} to do this. You are NOT allowed to use any other programming languages.\n\n### Instruction:\nCreate a code solution for this problem:\n{programming task}\n\n### Response:\n
SFT-CE	Below is a code snippet. Write a response that detailedly explains the code snippet.\n\n### Code Snippet:\n{code snippet}\n\n### Response:\n
SFT-CT	You are a senior programmer. Your objective is to translate the given #Python Code# to a {language} version accurately.\n\nYour response is the #{language} Code#, which should contain a {language} code snippet.\n'#Python Code#' and 'Python Code' are NOT allowed to appear in #{language} Code#.\n\n#Python Code#\npython\n{code snippet}\n\n\n#{language} Code#\n
SFT-TestCase	You are a Software Test Engineer. Your goal is to craft comprehensive test cases for the given #Programming Task#, validating a Python function which aims to solve this given task.\n\nYour response should include two parts: #Function Signature# and #Test Cases#. #Function Signature# contains an appropriate signature for this solution function, while #Test Cases# is a code snippet containing multiple different test cases. Each test case should format as a Python assert statement.\n\n#Programming Task#\n{programming task}\n
SFT-Comment	You are a senior Python programmer. Your objective is to add appropriate comments for the given Python code snippet. The comments should be concise and educational to junior programmers.\n\nThe added comments can be a description of the code snippet, a brief explanation of the code snippet, or details about specific statements. You can also add comments to the code snippet to make it more readable and declare some detials. You are NOT allowed to change the code snippet.\nThe given #Code Snippet# is to solve the given #Programming Task#. Your response is the #Commented Code#.\n\n#Programming Task#\n{programming task}\n\n#Code Snippet#\npython\n{code snippet}\n\n\n#Commented Code#\n
SFT-CodeQA	I want you act as a Programming Tutorial author. Your objective is to extract programming-related topics (e.g. specific syntax, developing skills, and etc.) from a #Given Programming Task#, and then give an educational question (which is independent of the given task) for your tutorial exercises.\n\nYour #Question# should be concise and have a definite answer. It should be about a specific topic related to programming and can be answered with pure natural language (not more than 300 words). The topic should be rare and educational.\nFollowed by #Question#, you SHOULD also give a concise #Response# to answer this question. #Response# should NOT contain any code.\n\n#Given Programming Task#\n{programming task}\n\n#Question#\n

Table 4: Prompts of crafting responses of training datasets by requesting OpenAI's GPT-3.5. The "Model-Task" column corresponds to the "Model" column in Table 1 and Table 2. "PLs" in "SFT-PLs" represents one of the 15 programming languages mentioned in Section 3 and 4.3.1: Python, C, C++, JavaScript, TypeScript, PHP, Go, Rust, Bash, Java, Racket, Haskell, SQL, HTML, and XML.

verse tasks, we design similar prompts for each task, while we also employ distinct descriptions in the prompt as a prefix of the task instruction to differentiate them. As shown in Appendix Table 7, we utilize instructions generated by GPT-3.5-turbo as the content of `### Instruction` section for most tasks. In the case of `SFT-TestCase`, we also provide the function signature in the prompt for the models' reference. In the case of `SFT-Comment`, we provide a code snippet without any code annotation, as well as a constant prompt: "Please add appropriate comments for the given code snippet."

Model	HumanEvalPack																		Avg.
	Py			JS			Java			Go			C++			Rust			
	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	CG	CE	CR	
Mixed Data																			
SFT-Mixed (9.6K)	41.6	37.8	51.2	35.4	32.9	47.5	29.1	33.5	50.0	26.6	25.0	48.7	31.6	32.9	44.5	32.6	21.3	31.7	36.3
SFT-Mixed (29K)	47.8	45.7	53.0	42.8	49.3	51.2	34.8	43.2	54.2	29.8	39.0	52.4	40.3	45.1	47.5	33.9	20.7	32.9	42.4
CODEM (4.3K)																			
CODEM-CG#Java	42.2	37.1	51.2	36.0	31.7	44.5	31.0	32.9	47.5	26.6	25.0	48.1	29.1	31.7	42.6	29.4	18.2	31.7	35.4
CODEM-CG#Go	44.0	36.5	49.3	35.4	32.9	43.9	27.2	32.3	50.0	27.2	23.1	48.7	31.6	32.9	43.9	31.4	21.3	30.4	35.7
CODEM-CG#C++	43.4	37.8	52.4	37.2	33.5	48.1	30.3	32.3	49.3	27.2	23.7	48.7	32.2	32.3	44.5	30.1	21.9	29.8	36.4
CODEM-CG#Rust	43.4	37.1	48.1	33.5	32.9	46.9	29.1	35.9	51.2	25.9	22.5	47.5	32.2	29.8	43.2	33.3	22.5	27.4	35.7
CODEM-CE#Py	43.4	39.6	49.3	36.0	33.5	46.3	29.1	36.5	48.1	25.9	25.6	50.0	30.4	31.0	43.2	32.0	18.2	31.0	36.1
CODEM-CE#JS	42.2	39.6	52.4	36.0	34.1	47.5	25.9	32.9	51.2	27.9	24.3	48.1	31.0	31.0	45.1	30.7	18.9	28.0	35.9
CODEM-CE#C++	42.2	39.0	49.3	36.0	31.7	48.1	28.4	35.9	49.3	25.3	25.0	49.3	29.1	34.1	46.3	32.6	21.9	29.8	36.3
CODEM-CE#Rust	40.3	37.8	48.7	33.5	30.4	43.9	27.2	35.9	50.0	24.0	25.6	50.0	31.0	30.4	45.7	29.4	23.7	29.2	35.4
CODEM-CR#Py	42.2	35.9	52.4	35.4	32.3	46.9	30.3	36.5	48.1	25.3	23.1	51.2	32.9	33.5	43.9	30.7	22.5	30.4	36.3
CODEM-CR#JS	40.9	37.1	50.0	34.7	31.7	49.3	28.4	33.5	49.3	24.6	24.3	47.5	29.8	31.7	45.1	30.1	19.5	28.6	35.3
CODEM-CR#Java	40.3	37.1	52.4	37.2	32.3	46.9	29.7	32.3	51.8	24.0	25.0	48.1	29.8	30.4	44.5	32.0	21.9	32.3	36.0
CODEM-CR#Go	41.6	37.8	48.7	34.7	31.7	47.5	31.0	31.7	47.5	27.2	26.2	50.6	29.8	33.5	44.5	31.4	23.1	32.9	36.2
CODEM (9.6K)																			
CODEM-CG#Java	43.4	39.6	51.2	39.7	43.9	48.7	36.0	43.9	52.4	29.2	31.7	49.3	39.7	40.2	43.9	32.6	21.9	33.5	40.0
CODEM-CG#Go	44.7	41.4	53.6	41.6	45.7	49.3	31.6	39.0	51.8	31.8	35.9	49.3	38.5	42.0	45.1	33.3	21.3	35.3	40.6
CODEM-CG#C++	45.3	42.0	50.0	42.2	43.9	50.6	32.9	41.4	51.2	30.5	31.0	50.0	42.8	40.8	48.7	33.9	22.5	32.3	40.7
CODEM-CG#Rust	42.8	41.4	50.0	42.8	43.2	51.2	33.5	40.2	53.0	27.9	32.9	50.0	37.2	39.0	45.7	35.2	22.5	35.9	40.2
CODEM-CE#Py	45.9	45.1	51.2	39.7	43.9	48.7	33.5	39.0	50.0	30.5	31.7	48.7	40.9	42.6	44.5	32.6	23.1	34.1	40.3
CODEM-CE#JS	45.3	39.6	54.2	38.5	47.5	49.3	34.8	42.6	53.0	29.8	33.5	49.3	39.1	41.4	45.1	34.6	23.1	33.5	40.8
CODEM-CE#C++	45.3	43.9	53.6	41.6	46.3	51.2	32.9	41.4	50.0	31.1	32.3	49.3	40.3	43.2	44.5	33.9	25.6	35.3	41.2
CODEM-CE#Rust	45.9	42.6	54.2	42.8	43.2	49.3	35.4	40.2	52.4	28.5	32.9	48.7	37.2	43.9	46.9	35.2	25.0	34.7	41.1
CODEM-CR#Py	44.7	41.4	54.2	41.6	46.3	48.7	33.5	43.2	50.6	29.8	32.3	51.2	41.6	40.8	48.1	33.3	22.5	32.3	40.9
CODEM-CR#JS	44.0	42.0	53.0	39.1	45.1	51.8	34.1	39.0	51.2	31.1	32.3	50.0	39.7	40.8	46.3	34.6	23.7	34.7	40.7
CODEM-CR#Java	45.3	41.4	52.4	40.9	47.5	47.5	33.5	41.4	54.8	29.8	35.3	49.3	40.3	42.6	46.9	33.3	24.3	34.1	41.1
CODEM-CR#Go	44.7	43.9	51.8	42.2	44.5	51.2	34.8	42.6	53.6	30.5	34.1	51.8	40.9	39.6	46.3	32.0	21.3	35.9	41.2

Table 5: (Continuation of Table 1) More evaluation results on HumanEvalPack of CODEM and its baselines.

Language	Programming Paradigm					Purpose			Type System		Mem. Mgmt.		Compilation		GP	Prop.
	IMP	DECL	PROC	OOP	FUNC	GEN	MRK	DATA	STAT	DYN	GC	OWN	COM	INTR		
Python	✓		✓	✓		✓				✓	✓			✓		7.9%
C	✓		✓			✓			✓			✓	✓			7.0%
C++	✓			✓		✓			✓			✓	✓		✓	6.4%
JS	✓			✓		✓				✓	✓			✓		8.4%
TS	✓			✓		✓			✓		✓			✓	✓	3.5%
PHP	✓		✓	✓		✓				✓	✓			✓		7.9%
Go	✓		✓	✓	✓	✓			✓		✓		✓		✓	3.1%
Rust	✓			✓	✓	✓			✓			✓	✓		✓	1.2%
Bash	✓		✓			✓				✓				✓		0.4%
Java	✓			✓		✓			✓		✓		✓		✓	11.3%
Racket	✓			✓	✓	✓				✓	✓			✓		0.0%
Haskell		✓			✓	✓			✓		✓		✓		✓	0.3%
SQL	✓		✓					✓	✓					✓		1.4%
HTML		✓					✓							✓		3.8%
XML		✓					✓							✓		0.0%

Table 6: Taxonomy of 15 languages mentioned in Section 3 and 4.3.1 and data proportion of programming languages in Stack (Li et al., 2023). Abbreviations: Mem. Mgmt. (Memory Management), GP (Generic Programming), IMP (Imperative), DECL (Declarative), PROC (Procedural), OOP (Object-Oriented), FUNC (Functional), GEN (General-purpose), MRK (Markup), DATA (Data Query), STAT (Static), DYN (Dynamic), GC (Garbage Collected), OWN (Ownership), COMP (Compiled), INTR (Interpreted); Prop. (Data Proportion in Stack).

Model-Task	Prompt
SFT-PLs SFT-Math SFT-PyNoise	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\n{instruction}\n\n### Response:\n
SFT-CodeQA SFT-NLQA SFT-NLNoise	Below is an instruction that proposes a question. Write a response that appropriately answers the question.\n\n### Instruction:\n{instruction}\n\n### Response:\n
SFT-CE	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nPlease write a response that detailedly explains the Python code snippet.\n{instruction}\n\n### Response:\n
SFT-CR	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\n{instruction}\n\n### Response:\n
SFT-CT	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nPlease translate the Python code snippet to a PHP version accurately.\n{instruction}\n\n### Response:\n
SFT-TestCase	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nPlease write unit tests to test a function which is intended for solving the below programming problem.\n{instruction}\nThe function signature is:\n{signature}\n\n### Response:\n
SFT-Comment	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nPlease add appropriate comments for the following code snippets.\n{input}\n\n### Response:\n

Table 7: Prompts of training and inference. The "Model-Task" column matches the "Model" column in Table 1 and Table 2. "PLs" in "SFT-PLs" refers to one of the 15 programming languages outlined in Section 3 and 4.3.1.

C.2 Other Details

(1) To ensure that the data generated by GPT-3.5-turbo meets our requirements, we manually review them. It is noteworthy that we remove those HTML code responses where there are embedded JavaScript scripts. (2) HumanEval-Pack (Muennighoff et al., 2023) offers two versions of evaluation sets for the code repair task, including HumanEvalFixDocs and HumanEvalFix-Tests. We opt for the former to conduct our experiments. (3) Among the competitive models evaluated in our study (Table 1 and Table 2), CodeLlama (Rozière et al., 2023) and CodeFuse (Di et al., 2023) have multiple versions. The specific versions we used are `CodeLlama-7b-hf`[6] and `CodeFuse-StarCoder-15B`[7].

[6] https://huggingface.co/codellama/CodeLlama-7b-hf

[7] https://huggingface.co/codefuse-ai/CodeFuse-StarCoder-15B