WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning

Zhaojian Yu[1*] **Xin Zhang**[2†] **Ning Shang**[2] **Yangyu Huang**[2] **Can Xu**[2]
Yishujie Zhao[1*] **Wenxiang Hu**[2] **Qiufeng Yin**[2]
[1]Tsinghua University
[2]Microsoft
yzj23@mails.tsinghua.edu.cn, xinzhang3@microsoft.com
https://github.com/microsoft/WaveCoder

Abstract

Recent work demonstrates that, after instruction tuning, Code Large Language Models (Code LLMs) can obtain impressive capabilities to address a wide range of code-related tasks. However, current instruction tuning methods for Code LLMs mainly focus on the traditional code generation task, resulting in poor performance in complex multi-task scenarios. In this paper, we concentrate on multiple code-related tasks and present **WaveCoder**, a series of Code LLMs trained with Widespread And Versatile Enhanced instruction data. To enable the models to tackle complex code-related tasks, we propose a method to stably generate diverse, high-quality instruction data from open source code dataset in multi-task scenarios and obtain **CodeSeaXDataset**, a dataset comprising 19,915 instruction instances across 4 code-related tasks, which is aimed at improving the generalization ability of Code LLM. Our experiments demonstrate that WaveCoder models significantly outperform other open-source models in terms of the generalization ability across different code-related tasks. Moreover, WaveCoder-Ultra-6.7B presents the state-of-the-art generalization abilities on a wide range of code-related tasks.

1 Introduction

Recently, Large Language Models (LLMs) such as ChatGPT, GPT-4 (OpenAI, 2023), and Gemini [1] have attained unprecedented performance levels in a broad array of NLP tasks. These models utilize a self-supervised pre-training process, and subsequent supervised fine-tuning to demonstrate exceptional zero/few-shot capabilities, effectively following human instructions across various tasks.

For code-related tasks, several previous works, including Codex (Chen et al., 2021), StarCoder (Li et al., 2023a), CodeLLaMa (Roziere et al., 2023) and DeepseekCoder (Guo et al., 2024), have successfully demonstrated that pre-training on code corpus can significantly improve the model's capability to tackle code-related problems. After the process of pre-training, instruction tuning (Wei et al., 2022; Aribandi et al., 2022; Chung et al., 2022) has shown its effectiveness in the aspect of improving the quality of LLM responses. To specifically enhance the performance of Code LLMs on code-related tasks through instruction tuning, many existing methods for instruction data generation have been designed. For example, Code Alpaca (Chaudhary, 2023) utilizes the method of self-instruct (Wang et al., 2023a) within the coding domain, leveraging the few-shot capabilities of teacher LLM to generate instruction data. Similarly, WizardCoder (Luo et al., 2024) applies the evol-instruct (Xu et al., 2024) approach based on Code Alpaca, demonstrating a novel and effective method for the generation of instruction data. These applications underscore the potential of utilizing teacher LLMs to produce instructional content effectively, thereby offering an avenue for the creation of instruction data in the code domain. However, the quality of the data they generate heavily relies on the performance of the teacher LLM and the limited initial seeds, which often produces a large amount of duplicate instruction instances and reduce the effectiveness of instruction tuning (Xu et al., 2022; Yan et al., 2023; Lee et al., 2022). To break away from dependence on teacher LLMs, Octopack (Muennighoff et al., 2024) constructs a code instruction dataset leveraging the natural structure of Git commits. Nonetheless, ensuring the quality of data in git messages presents a considerable challenge, and the comprehensive screening of data through artificial filtering rules is often a complex task. Additionally, these endeavors are predominantly centered on traditional code generation tasks and lack the capability to produce detailed, task-

[*] Work was done during an internship at Microsoft.
[†] Corresponding author.
[1] https://deepmind.google/technologies/gemini

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 5140–5153
August 11-16, 2024 ©2024 Association for Computational Linguistics

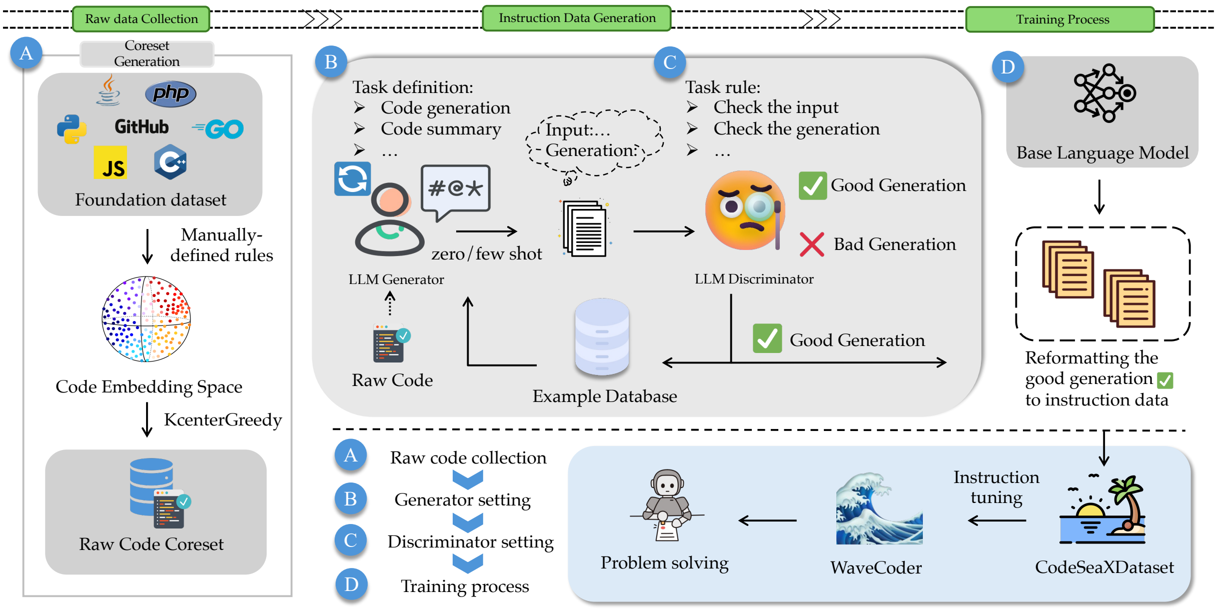


Figure 1: The overview of the widespread and versatile enhancement for Code LLM. Part B and C indicates the LLM-based Generator and LLM-based Disciminator where the generator can leverage different examples in example database by in-context learning.

specific instructions in multi-task scenarios.

In this paper, we primarily focus on multiple code-related tasks, aiming to generate high-quality and diverse instructional data tailored to specific task requirements. Addressing the aforementioned challenges, we refine the instruction data by classifying the instruction instances to four universal code-related tasks in CodeXGLUE (Lu et al., 2021): 1) Code Summarization, 2) Code Generation, 3) Code Translation, 4) Code Repair and propose a widespread and versatile enhanced instruction generation method that could make full use of open source code data and stably generate high quality and diverse instruction data in multi-task scenarios. By this generation strategy, we obtain a dataset of 19,915 instruction instances across four code-related tasks, termed **CodeSeaXDataset**.

To validate our approach, we train StarCoder (Li et al., 2023a), CodeLLaMa (Roziere et al., 2023), and DeepseekCoder (Guo et al., 2024) with our initial CodeSeaXDataset dataset and get **WaveCoder**. Following a thorough assessment on HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), HumanEvalPack (Muennighoff et al., 2024) benchmarks, experimental results show that our **WaveCoder** exhibits outstanding generalization ability based on widespread and versatile enhanced instruction tuning. Moreover, to further explore the improvements brought by data quality, we use GPT-4 (OpenAI, 2023) to regenerate response for the instruction in CodeSeaXDataset. Fine-tuned with the enhanced 20K CodeSeaXDataset dataset, we obtain **WaveCoder-Pro-6.7B** which achieve 72.0% pass@1 on HumanEval (Chen et al., 2021) and surpass open source Code LLMs but still behind SoTA Code LLM. Combining enhanced CodeSeaXDataset with WaveCoder-evol-codealpaca, the decontaminated Magicoder-evol-codealpaca [2] dataset, we present **WaveCoder-Ultra-6.7B**, with SoTA generalization capabilities on multiple code-related tasks.

2 CodeSeaXDataset: Four-task Code-related Instruction Data

2.1 Tasks Details

Given the code-related tasks from CodeXGLUE (Lu et al., 2021), we select four of the most universally representative and common tasks from the three generative tasks (code-to-text, text-to-code, and code-to-code) for further exploration including Code Summarization, Code Generation, Code Translation, and Code Repair. Detailed descriptions of these tasks can be found below.

Code Summarization (code-to-text). This task aims to create a brief summary of a given code. The raw code is used as input and the teacher model's response is reformulated into an instruction format.

[2] https://huggingface.co/datasets/ise-uiuc/Magicoder-evol-codealpaca-110K

Table 1: The proportion of generated data in generation phase.

Task	Num	Per(%)	Prompt
Code Generation	11370	57.1	Implementing functions that perform specific operations given input.
Code Summarization	3165	15.8	Write clear and concise documentation for the given code.
Code Repair	3144	15.8	Identify and fix errors in the given code.
Code Translation	2236	11.2	Rewrite the given code from one programming language to another.

Table 2: The proportion of programming language in raw code.

Task	Percentage(%)
Python	29.44
PHP	21.34
Go	19.68
Java	18.53
JavaScript	5.56
Others (Ruby,C++,C#)	5.45

Code Generation (text-to-code, code-to-code). In this task, the model is expected to generate code based on a user's demand description. Therefore, the teacher model is expected to generate instructions and solution code given the raw code as a instruction-solution pair. The generated solution code is then considered as the output.

Code Translation (code-to-code). This task involves converting one programming language into another. The task-specific prompt and raw code are given to the teacher model, then the model generates instructions and the translated code.

Code Repair (code-to-code). The aim of this task is to provide correct code based on potential issues in the given code. The teacher model is expected to generate solutions for the incorrect code, typically with the correct code and some descriptions, which are then taken as the output.

2.2 Widespread and Versatile Enhanced Instruction Generation

In past research work (Zhou et al., 2023; Gupta et al., 2023), many researchers have discovered that data quality and diversity often play a more important role in instruction tuning process than data amount. The improvement of data quality and diversity are directly related to the performance of the fine-tuned LLM. Therefore, to ensure the data quality and diversity of instruction instance, we propose a widespread and versatile enhanced instruction generation method including two the following parts: 1) a method that can retain the diversity of instruction data by retainig the diversity of raw code to the utmost extent. 2) a LLM-based Generator-Discriminator framework to stably generate high-quality instruction data.

2.2.1 Raw Code Collection

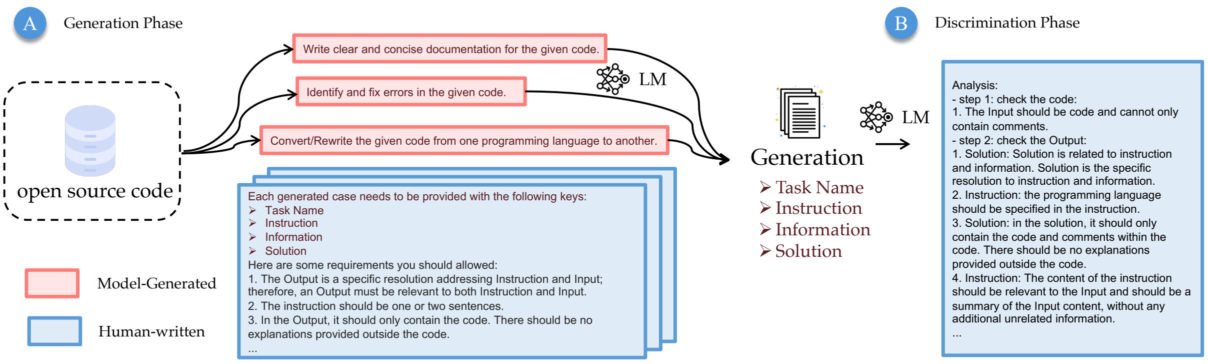
To ensure the quality and diversity of raw code, we manually define some filtering rules and utilize a cluster method KCenterGreedy (Sener and Savarese, 2018; Chen et al., 2023) to get the raw code collection from the open source code dataset. In this work, we select CodeSearchNet [3], which contains 2 million of <comment, code> pairs from open-source libraries hosted on GitHub, as our foundation dataset and process it with the following steps:

Manually defined filtering rules. In order to select high-quality code for instruction-tuning, we make the following rules to filter the foundation dataset: i) In this work, we filtered the code to make sure that the length of required code is neither too long nor too short. ii) Followed Code Alpaca (Chaudhary, 2023), we have eliminated the raw code containing words from the blacklist, which could potentially reduce the performance of the resulting model.

Coreset selection method. To ensure the data diversity when select raw code samples, we employed KCenterGreedy (Sener and Savarese, 2018) algorithm, which has been proven efficient in obtaining a set of core samples of one distribution, to select representative samples from the open source code dataset based on the code embeddings encoded by the same embedding model (`roberta-large-v1` (Liu et al., 2019)).

By incorporating such a method into the open source code dataset, the diversity of the generated data no longer relies solely on capability of the teacher LLM itself or initial seed. Moreover, due to the application of the KCenterGreedy algorithm, the diversity of languages is also significantly retained, as shown in Table 2.

[3] https://huggingface.co/datasets/code_search_net



Write clear and concise documentation for the given code.

Identify and fix errors in the given code.

Convert/Rewrite the given code from one programming language to another.

Each generated case needs to be provided with the following keys:
➢ Task Name
➢ Instruction
➢ Information
➢ Solution
Here are some requirements you should allowed:
1. The Output is a specific resolution addressing Instruction and Input; therefore, an Output must be relevant to both Instruction and Input.
2. The instruction should be one or two sentences.
3. In the Output, it should only contain the code. There should be no explanations provided outside the code.
...

Analysis:
- step 1: check the code:
1. The Input should be code and cannot only contain comments.
- step 2: check the Output:
1. Solution: Solution is related to instruction and information. Solution is the specific resolution to instruction and information.
2. Instruction: the programming language should be specified in the instruction.
3. Solution: in the solution, it should only contain the code and comments within the code. There should be no explanations provided outside the code.
4. Instruction: The content of the instruction should be relevant to the Input and should be a summary of the Input content, without any additional unrelated information.
...

Figure 2: The overview of the our LLM-based Generator-Discriminator framework. In part A, the output of Generator includes 4 keys: Task name, Instruction, Information, Solution. All keys will be analyzed in the Discrimination Phase and the analysis can be reused as examples in next turn.

2.2.2 LLM-based Generator-Discriminator Framework

After the process of raw code collection, the data diversity from raw code has been retained, where the next step is to generate instruction data for supervised fine-tuning from the raw code. To further ensure the quality of generated instruction data, shown in Figure 2, we propose a LLM-based Generator-Discriminator framework where the generator can leverage an extensive amount of unsupervised open source code to generate supervised instruction data and the discriminator can generate analysis for each component in instruction data.

Generation Phase. In the generation phase, we utilize GPT-4 to generate definitions for each code-related task. As shown in Figure 2, following the model-generated task definition, we manually develop the generation requirements for the each code-related task. Integrating both the task definition and all associated requirements into the generation prompt, we take the raw code as an input and select different examples from the example database to generate instruction data by GPT-3.5.

Discrimination Phase. During the exploration of the instruction generation process, we noticed that the data quality of the instruction instances cannot be guaranteed through the generation phase alone. In order to enhance the controllability of data generation and further ensure the data quality, we employ GPT-4 as a LLM-based discriminator to continue analyzing and filtering the instruction data. Subsequently, inspired by Zero-shot-CoT (Kojima et al., 2022), we establish a series of rules, exemplified in Figure 4 and disassemble them to some subtopics to ensure the discriminating accuracy where the LLM-based discriminator can analyze the generation step by step. By adopting this method, the discrimination rules can be modified partially to address certain issues. After the discrimination process, as shown in Figure 1, each instruction instance is classified as either a good or bad case and the classification information is subsequently random selected in the following generation as examples. For the reusage of these classified instruction instance, different from self-instruct (Wang et al., 2023a) which solely utilize the initial seed task as good example, we exploit both the good generation and bad generation as few-shot example so that the generator can learn from the mistake in different bad example. Therefore, this framework provides a comprehensive approach to generating and evaluating instruction data, ensuring a high-quality training dataset.

3 Experiments

3.1 Setup

Unlike the previous work (Luo et al., 2024; Shen et al., 2023; Gunasekar et al., 2023) that mainly focus on code generation task, we generate about 20K dataset covers 4 common code-related tasks to enhance the geralization abilties of Code LLM. To obtain WaveCoder models, We choose *StarCoder-15B*, *CodeLLaMa (7B and 13B)*, *DeepseekCoder-6.7B* as the base model and fine-tune all the base model for 3 epochs using NVIDIA A100-80GB GPU. For *StarCoder-15B*, *CodeLLaMa-7B* and *CodeLLaMa-13B*, we set the global batch size to 256 using Tensor Parallel and set the initial learning rate at 2e-5. For *DeepseekCoder-6.7B*, we set the global batch size to 512 using the Fully Sharded Data Parallel (FSDP) module from Pytorch and set the initial learning rate at 5e-5.

Table 3: Results of pass@1 on HumanEval and MBPP benchmark. We use self-reported scores whenever available. The abbreviations "CL", "SC", "DS" refer to the base models CodeLLaMa and StarCoder and DeepseekCoder, respectively. "WaveCoder-Pro-6.7B" and "WaveCoder-Ultra-6.7B" is detailed in the last paragraph of Section 1. Due to the difference in decoding strategies from previous evaluation work, we marked the results of greedy decoding in blue and n = 200 samples in red. -: Not reported in their paper.

Model	**Params**	**Base Model**	**InsT Data**	**HumanEval**	**MBPP (500)**
		Proprietary Models			
GPT-4	-	-	-	85.4 / 67.0	-
ChatGPT	-	-	-	73.2 / 48.1	52.2
		Open-Source Models			
StarCoder	15B	-	✘	33.6	43.3
OctoCoder	15B	StarCoder	13K	46.2	43.5
WizardCoder	15B	StarCoder	78K	57.3	51.8
WaveCoder-SC-15B 🌊	15B	StarCoder	20K	**50.5 (+16.9)**	**51.0 (+7.4)**
CodeLLaMa	7B	-	✘	33.5	41.4
CodeLLaMa-instruct	7B	CodeLLaMa	14K	34.8	44.4
WaveCoder-CL-7B 🌊	7B	CodeLLaMa	20K	**48.1 (+14.6)**	**47.2 (+5.8)**
CodeLLaMa	13B	-	✘	36.0	47.0
CodeLLaMa-instruct	13B	CodeLLaMa	14K	42.5	49.4
WaveCoder-CL-13B 🌊	13B	CodeLLaMa	20K	**55.4 (+19.4)**	**49.6 (+2.6)**
DeepseekCoder	6.7B	-	✘	49.4	60.6
Magicoder-DS	6.7B	DeepseekCoder	75K	66.5	60.4
WaveCoder-DS-6.7B 🌊	6.7B	DeepseekCoder	20K	**64.0 (+14.6)**	**62.8 (+2.2)**
WaveCoder-Pro-6.7B 🌊	6.7B	DeepseekCoder	20K	**72.0 (+22.6)**	**63.6 (+3.0)**
		SoTA Open-Source Models			
DeepseekCoder-instruct*	6.7B	DeepseekCoder	-	73.8	62.8
Magicoder-S-DS	6.7B	DeepseekCoder	185K	76.8	64.6
WaveCoder-Ultra-6.7B 🌊	6.7B	DeepseekCoder	130K	**78.6 (+29.2)**	**64.4 (+3.8)**

Benchmarks and Baselines. To ensure a thorough assessment of the model's generalization ability, we score our model on three code benchmarks across different code related tasks: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and HumanEvalPack (Muennighoff et al., 2024), as illustrated in Appendix D.

Proprietary Models. We present the self-reported results from an array of SoTA LLMs, including ChatGPT (`gpt-3.5-turbo`), GPT-4. If not reported, we use the results from Octopack (Muennighoff et al., 2024) or evaluate by ourselves.

Open Source Models. To ensure an equitable comparison, we opted to select models that have been trained with the similar amount of instruction instances for our comparative analysis.

SoTA Open Source Models. We compared WaveCoder-6.7B with the SoTA open source Code LLM, includes Magicoder-S-DS (Wei et al., 2023) and DeepseekCoder-instruct-6.7B (Wei et al., 2023) on a wide range of code-related tasks. All the result of SoTA open source models is presented from EvalPlus. (Liu et al., 2023) If not reported, we evaluate it by ourselves.

3.2 Result

Evaluation on Code Generation Task. HumanEval and MBPP are two representative benchmarks for code generation task, as illustrated in Appendix D. Table 3 shows the pass@1 score of different LLMs on both benchmarks. From the results, We have the following observations:

1) WaveCoder-Pro-6.7B outperforms other open source models with only 6.7B parameters and 20K instruction data. Trained with GPT-4 enhanced CodeSeaXDataset dataset, WaveCoder-Pro-6.7B achieve 72.0% pass@1 and on HumanEval and 63.6% on MBPP, surpassing all open source models but still behind proprietary models and the SoTA open source models.

2) Refined and diverse instruction data can significantly improve the efficiency of instruction tuning. As delineated in Table 3, WaveCoder demonstrates commendable performance, utilizing a dataset comprising merely about 20K Instruction Tuning Data (InsT Data), which positions it on an equal foot-

Table 4: Results of pass@1 on HumanEvalFix benchmark. We use self-reported scores whenever available. Due to the difference in decoding strategies from previous evaluation work, we marked the results of greedy decoding in blue and n = 20 samples in red.

Model	Python	JavaScript	Java	Go	C++	Rust	Avg.
GPT-4	47.0	48.2	50.0	50.6	47.6	43.3	47.8
StarCoder	8.7	15.7	13.3	20.1	15.6	6.7	13.4
OctoCoder	30.4	28.4	30.6	30.2	26.1	16.5	27.0
WizardCoder	31.8	29.5	30.7	30.4	18.7	13.0	25.7
WaveCoder-SC-15B 🌊	**39.3**	**35.1**	**34.8**	**36.2**	**30.2**	**22.5**	**33.0**
CodeLLaMa-instruct-7B	28.0	23.2	23.2	18.3	0.1	0.1	15.5
CodeLLaMa-CodeAlpaca-7B	37.8	39.0	42.0	37.8	37.2	29.2	37.1
WaveCoder-CL-7B 🌊	**41.4**	**41.4**	**42.0**	**47.1**	**42.7**	**34.7**	**41.5**
CodeLLaMa-instruct-13B	29.2	19.5	32.3	24.4	12.8	0.1	19.7
CodeLLaMa-CodeAlpaca-13B	42.7	43.9	50.0	45.7	39.6	37.2	43.2
WaveCoder-CL-13B 🌊	**48.8**	**48.2**	**50.6**	**51.8**	**45.1**	**40.2**	**47.4**
DeepseekCoder-6.7B	29.9	29.2	39.0	29.2	25.0	21.9	29.0
Magicoder-DS	42.0	43.3	50.6	41.4	38.4	29.2	40.8
DeepseekCoder-CodeAlpaca-6.7B	49.4	51.8	45.1	48.8	44.5	31.7	45.2
WaveCoder-DS-6.7B 🌊	**57.9**	**52.4**	**57.3**	**47.5**	**45.1**	**36.0**	**49.4**
WaveCoder-Pro-6.7B 🌊	**59.1**	**56.7**	**54.2**	**45.1**	**45.7**	**34.1**	**49.2**
Deepseek-instruct-6.7B	56.1	58.5	57.3	49.4	45.1	36.6	50.5
Magicoder-S-DS	56.1	55.4	58.5	51.2	45.7	35.3	50.3
WaveCoder-Ultra-6.7B 🌊	**58.5**	**57.3**	**61.0**	**53.0**	**50.0**	**37.2**	**52.8**

ing with its contemporaries. Despite a discernible shortfall in the code generation benchmarks relative to WizardCoder (50.5 vs 57.3) and Magicoder (64.0 vs 66.5), it is imperative to consider the substantial disparity in the volume of training data. Moreover, it is observed that WaveCoder-pro-6.7B significantly outperforms Magicoder-DS-6.7B (72.0 vs 66.5), demonstrating the effectiveness of data quality and diversity in instruction tuning.

Evaluation on Other Code-related Task. We score WaveCoder with state-of-the-art Code LLMs on HumanEvalPack (Muennighoff et al., 2024) in Table 4 and Table 5, highlighting the the following salient observations:

1) WaveCoder models outperform all open source models on other code-related task. Building upon Starcoder, our proposed WaveCoder-SC has exhibited exceptional performance, transcending the capabilities of both WizardCoder and OctoCoder as evidenced by the HumanEvalFix (33.0 vs 25.7 vs 27.0) and HumanEvalExplain (30.8 vs 27.5 vs 24.5) benchmarks, which is also shown in other base models. Notably, WaveCoder-DS-6.7B achieves 49.4% average pass@1 score on HumanEvalFix and 41.3% on HumanEvalExplain, surpassing all open source models and demonstrating strong generalization capabilities in multi-task scenarios.

2) The enhancement in data refinement and diversification can markedly bolster the efficacy of instruction tuning in multi-task scenarios. Such data refinement, coupled with the categorization of instructions into four code-related tasks, has propelled our models to reach an unforeseen generalization capabilities in various code-related tasks. Remarkably, our WaveCoder-DS-6.7B model outperforms GPT-4 (49.4 vs 47.8) on HumanEvalFix, thereby underscoring the potential of smaller models to achieve near-parity with parameter-heavy models when optimized efficiently.

WaveCoder-Ultra-6.7B. Inspired by Magicoder-S-DS-6.7B (Wei et al., 2023), we combine CodeSeaXDataset with WaveCoder-evol-codealpaca to a 130K dataset. Fine-tuned with this combination of two datasets, we obtain WaveCoder-Ultra-6.7B. As illustrated in Table 3, 4, 5, WaveCoder-Ultra-6.7B has the state-of-the-art generalization abilities on a wide range of code-related tasks, which highlights the significance of our CodeSeaXDataset dataset again and demonstrates the potential of larger datasets.

4 Ablation and Analysis

4.1 Ablation of Code-related Tasks

To explore the relationship between different tasks, we conduct an ablation study about the task type of instruction data. Using DeepseekCoder-Base-

Table 5: Results of pass@1 on HumanEvalExplain benchmark. We use self-reported scores whenever available. Due to the difference in decoding strategies from previous evaluation work, we marked the results of greedy decoding in blue and n = 20 samples in red.

Model	**Python**	**JavaScript**	**Java**	**Go**	**C++**	**Rust**	**Avg.**
GPT-4	64.6	57.3	51.2	58.5	38.4	42.7	52.1
StarCoder	0.0	0.0	0.0	0.0	0.0	0.0	0.0
WizardCoder	32.5	33.0	27.4	26.7	28.2	16.9	27.5
OctoCoder	35.1	24.5	27.3	21.1	24.1	14.8	24.5
WaveCoder-SC-15B 🌊	**37.1**	**33.3**	**40.5**	**23.3**	**31.8**	**19.3**	**30.8**
CodeLLaMa-instruct-7B	33.5	36.0	31.7	21.3	25.0	16.4	27.3
CodeLLaMa-CodeAlpaca-7B	34.7	24.4	37.8	23.2	28.6	19.5	28.0
WaveCoder-CL-7B 🌊	**41.4**	**31.7**	**39.0**	**25.0**	**34.1**	**23.2**	**32.4**
CodeLLaMa-instruct-13B	40.2	26.8	37.2	22.5	28.0	14.6	28.2
CodeLLaMa-CodeAlpaca-13B	32.3	28.0	34.1	18.9	29.9	20.7	27.3
WaveCoder-CL-13B 🌊	**45.7**	**42.0**	**48.2**	**32.3**	**38.4**	**20.7**	**37.9**
DeepseekCoder-6.7B	43.9	40.2	37.8	29.2	34.1	22.5	34.6
Deepseek-CodeAlpaca-6.7B	40.8	37.2	42.1	29.9	31.7	22.5	34.0
Magicoder-DS	55.5	36.6	49.4	36.0	39.6	27.4	40.7
WaveCoder-DS-6.7B 🌊	**48.2**	**47.5**	**49.4**	**32.3**	**48.2**	**22.0**	**41.3**
WaveCoder-Pro-6.7B 🌊	**53.0**	**43.3**	**54.9**	**34.1**	**42.7**	**20.0**	**41.3**
Magicoder-S-DS	60.3	46.3	54.3	38.4	48.1	29.2	46.1
Deepseek-instruct-6.7B	62.2	54.3	61.0	39.6	55.5	33.5	51.0
WaveCoder-Ultra-6.7B 🌊	**56.7**	**50.0**	**54.3**	**34.8**	**51.2**	**36.6**	**47.3**

Table 6: Ablation study on different code-related tasks: CG (**C**ode **G**eneration), CS (**C**ode **S**ummarization), CT (**C**ode **T**ranslation), CR (**C**ode **R**epair). WaveCoder-DS-6.7B utilizes all 4 code-related tasks.

Model	**CG**	**CS**	**CT**	**CR**	**HumanEval**	**HumanEval Fix (Avg.)**	**HumanEval Explain (Avg.)**
DeepseekCoder-Base-6.7B	✘	✘	✘	✘	49.4	29.0	34.6
WaveCoder-DS-6.7B 🌊	✔	✔	✔	✔	**64.0 (+14.6)**	**49.4 (+20.4)**	**41.3 (+7.3)**
-Without Repair	✔	✔	✔	✘	60.9 (-3.1)	15.7 (-33.7)	41.2 (-0.1)
-Without Generation	✘	✔	✔	✔	53.6 (-10.4)	47.4 (-2.0)	40.5 (-0.8)
-Without Translation	✔	✔	✘	✔	60.9 (-3.1)	49.3 (-0.1)	41.6 (+0.3)
-Without Summarization	✔	✘	✔	✔	61.5 (-2.5)	45.6 (-3.8)	28.4 (-12.9)

6.7B as our base model and initial 20K CodeSeaX-Dataset data as our base dataset, we have the following observations from Table 6:

1) Refined instruction data can significantly improve the generalization ability of pre-trained models without a tradeoff. As shown in Table 6, incorporating all 4 code-related tasks into training data, WaveCoder-DS-6.7B achieves the best performance on benchmark of all tasks. For example, the participation of the Code Repair task yields a considerable average improvement of 33.7% absolute for HumanEvalFix without any significant decline in other tasks, and even improved by 3.1% absolute for HumanEval benchmark.

2) Different tasks can promote each other so that the model can show a generalization ability. From Table 6, we can observe that any combination of three tasks resulted in a lower score than all tasks. For example, the addition of the code summarization task offers a modest yet significant average improvement on all benchmarks. Moreover, the absence of any task will cause the score of HumanEval to drop, which also reflects the mutual promotion between different tasks.

4.2 Discussion about Data Leakage

In this section, we explore the potential leakage through three instruction datasets about code (i.e. Code Alpaca, CodeSeaXDataset, Magicoder-evol-codealpaca). To ensure an accurate analysis, we employ SoTA embedding model `GTE-Large` (Li et al., 2023b) to encode the canonical code in test benchmarks and all code in training set. Subsequently, we find the nearest neighbour in train set for each questions in test benchmark. As illustrated in Figure 3, CodeSeaXDataset has the lower

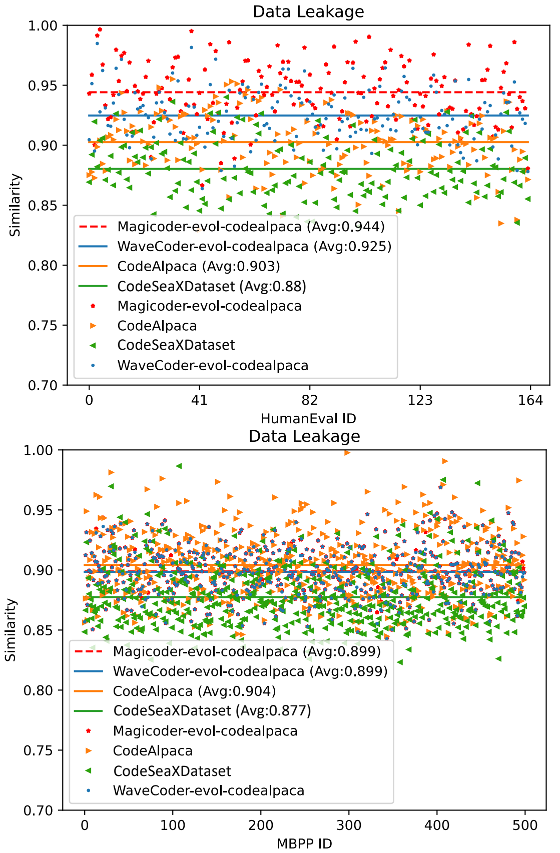


Figure 3: Discussion about data leakage in different training dataset. WaveCoder-evol-codealpaca indicates the decontaminated Magicoder-evol-codealpaca dataset under our strategy.

average cosine similarity than other datasets. Figure 6 in Appendix presents two examples about the data leakage in these training set. Moreover, we analyze all benchmarks and notice a serious data leakage issue between HumanEval and Magicoder-evol-codealpaca dataset. Therefore, we decontaminate Magicoder-evol-codealpaca for each evaluation problem in HumanEval and obtain WaveCoder-evol-codealpaca. As illustrated in Figure 3, WaveCoder-evol-codealpaca has lower similarity than Magicoder-evol-codealpaca.

5 Related Work

Instruction Tuning. Recent studies, such as FLAN (Wei et al., 2022), ExT5 (Aribandi et al., 2022), and FLANT5 (Chung et al., 2022), have underscored the efficacy of integrating diverse tasks within training process to bolster the adaptability of pre-trained models for downstream tasks. Specifically, Flan-PaLM 540B's (Chung et al., 2022) instruction-tuning over 1.8K tasks has demonstrated that a widespread and versatile enhanced instruction dataset markedly enhances language model performance. InstructGPT (Ouyang et al., 2022), with its incorporation of premium instruction data crafted by human annotators, has shown significant promise in aligning model outputs with user intents, prompting further investigation into instruction-tuning mechanisms. Additionally, Stanford Alpaca (Taori et al., 2023) has innovatively employed GPT-generated instruction data via self-instruct (Wang et al., 2023a) for instruction tuning process. WizardLM (Xu et al., 2024) has built upon these advancements by applying the evol-instruct methodology, collectively illuminating the transformative impact of instruction tuning on the overall capabilities of LLM.

Code Large Language Models. Recent advancements in code generation have been propelled by Code LLMs such as CodeGen (Nijkamp et al., 2022), CodeT5 (Wang et al., 2021), StarCoder (Li et al., 2023a), CodeLLaMa (Roziere et al., 2023) and Deepseek-Coder (Guo et al., 2024), which benefit from extensive pre-training on expansive code corpora. Efforts to further enhance efficiency and problem-solving capabilities have led to the development of instruction-tuned models like InstructCodeT5+ (Wang et al., 2023b), WizardCoder (Luo et al., 2024), Pangu-coder2 (Shen et al., 2023), However, all the instruction data they used is from Code Alpaca, which is not refined enough in the context of multi-task environment, which drives us to propose new methods for instruction data generation. Concurrently, with the release of our contemporaneous work Magicoder (Wei et al., 2023), we offer a concise analysis in Section 3.

6 Conclusion

This paper presents WaveCoder, a Code LLM fine-tuned with widespread and versatile enhanced instruction data. By enabling language models to effectively tackle complex code-related tasks, our approach demonstrates the potential of integrating multiple code-related tasks into instruction tuning for Code LLM and generating high-quality and diverse instruction data for specific task requirements in multi-task scenarios. WaveCoder achieves state-of-the-art generalization performance on different code-related tasks surpassing existing open source Code LLMs. Furthermore, our analysis of the relationship of different tasks provides valuable insights for future research, paving the way for more extensive code-related tasks and larger dataset.

Limitations

We present WaveCoder and propose a data generation method which can stably generate high-quality and diversity instruction data from open source dataset in multi-task scenario. One limitation of our work is that the training dataset we used only includes 19,915 instructions, which produces limited enhancements to the model. As illustrated Section 3, we expand the training dataset to a larger amount and the resulted model still have significant improvement. Therefore, future work should focus on more code-related task types and larger dataset.

Ethics Statement

We constructed our CodeSeaXDataset dataset from open source code. For each code snippet we used, we are committed to adhering to the terms of its license, which includes proper attribution ensuring that any modifications or derivative works are also shared under the compatible terms. Moreover, we notice a serious data leakage issue in the Magicoder-evol-codealpaca dataset. To ensure a fair comparison, we remove three nearest neighbours of each question in test benchmark from train set. However, if all similar samples are accidentally removed, the integrity of the data will be damaged, which is harmful to model training. Therefore, this phenomenon should be attributed to the fact that the problems in the current test benchmarks are some of the most basic algorithm logic. To this end, we call for more comprehensive and complex test benchmarks for Code LLMs which will not easily cause data leakage problem.

References

Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q. Tran, Dara Bahri, Jianmo Ni, Jai Gupta, Kai Hui, Sebastian Ruder, and Donald Metzler. 2022. Ext5: Towards extreme multi-task scaling for transfer learning. In *International Conference on Learning Representations (ICLR)*.

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. `https://github.com/sahil280114/codealpaca`.

Hao Chen, Yiming Zhang, Qi Zhang, Hantao Yang, Xiaomeng Hu, Xuetao Ma, Yifan Yanggong, and Junbo Zhao. 2023. Maybe only 0.5% data is needed: A preliminary exploration of low training data instruction tuning. *arXiv preprint arXiv:2305.09246*.

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*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

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*.

Himanshu Gupta, Saurabh Arjun Sawant, Swaroop Mishra, Mutsumi Nakamura, Arindam Mitra, Santosh Mashetty, and Chitta Baral. 2023. Instruction tuned models are quick learners. *arXiv preprint arXiv:2306.05539*.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems (NIPS)*, 35:22199–22213.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445.

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

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023b. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. *International Conference on Learning Representations (ICLR)*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. *International Conference on Learning Representations (ICLR)*.

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. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems (NIPS)*, 35:27730–27744.

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*.

Ozan Sener and Silvio Savarese. 2018. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations (ICLR)*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

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. `https://github.com/tatsu-lab/stanford_alpaca`.

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

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

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 (EMNLP)*, pages 8696–8708.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations (ICLR)*.

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. 2024. Wizardlm: Empowering large language models to follow complex instructions. *International Conference on Learning Representations (ICLR)*.

Jin Xu, Xiaojiang Liu, Jianhao Yan, Deng Cai, Huayang Li, and Jian Li. 2022. Learning to break the loop: Analyzing and mitigating repetitions for neural text generation. *Advances in Neural Information Processing Systems (NIPS)*, 35:3082–3095.

Jianhao Yan, Jin Xu, Chiyu Song, Chenming Wu, Yafu Li, and Yue Zhang. 2023. Understanding in-context learning from repetitions. *arXiv preprint arXiv:2310.00297*.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2023. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*.

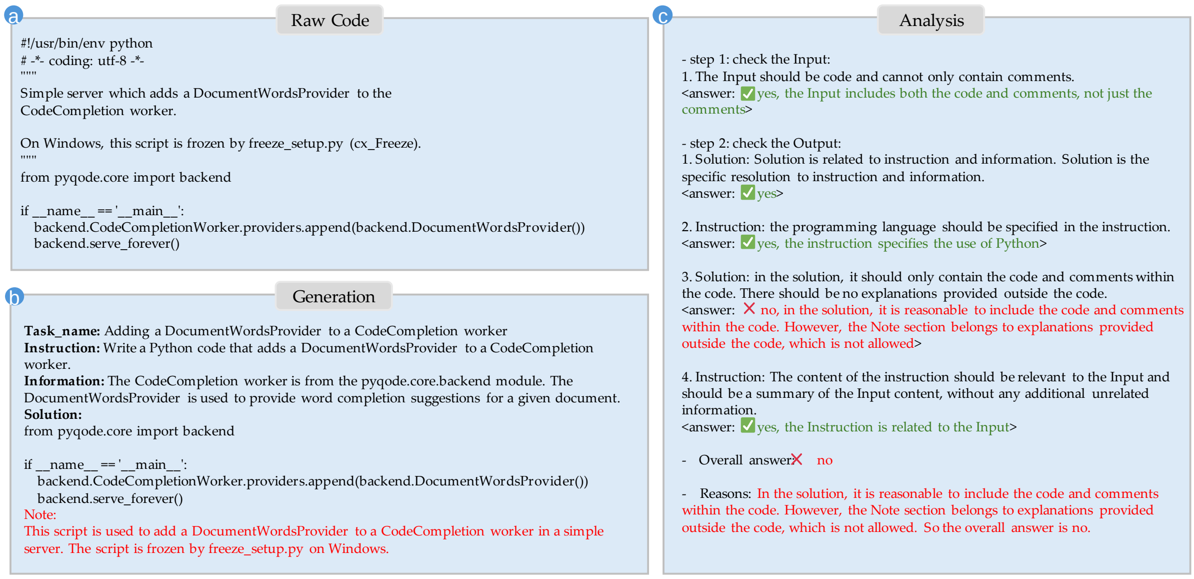
A Prompt

Followed Alpaca (Taori et al., 2023), we set the fine-tuning prompt as follows:

Prompt with Input:
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:{instruction}
### Input:{input}
### Response:

Prompt without Input:
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:{instruction}
### Response:

B Examples of the LLM-based Generator-Discriminator framework


a
Raw Code
b
Generation
c
Analysis

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Simple server which adds a DocumentWordsProvider to the
CodeCompletion worker.

On Windows, this script is frozen by freeze_setup.py (cx_Freeze).
"""
from pyqode.core import backend

if __name__ == '__main__':
   backend.CodeCompletionWorker.providers.append(backend.DocumentWordsProvider())
   backend.serve_forever()
```

Task_name: Adding a DocumentWordsProvider to a CodeCompletion worker
Instruction: Write a Python code that adds a DocumentWordsProvider to a CodeCompletion worker.
Information: The CodeCompletion worker is from the pyqode.core.backend module. The DocumentWordsProvider is used to provide word completion suggestions for a given document.
Solution:

```
from pyqode.core import backend

if __name__ == '__main__':
   backend.CodeCompletionWorker.providers.append(backend.DocumentWordsProvider())
   backend.serve_forever()
```

Note:
This script is used to add a DocumentWordsProvider to a CodeCompletion worker in a simple server. The script is frozen by freeze_setup.py on Windows.

- step 1: check the Input:
1. The Input should be code and cannot only contain comments.
<answer: ✅yes, the Input includes both the code and comments, not just the comments>

- step 2: check the Output:
1. Solution: Solution is related to instruction and information. Solution is the specific resolution to instruction and information.
<answer: ✅yes>

2. Instruction: the programming language should be specified in the instruction.
<answer: ✅yes, the instruction specifies the use of Python>

3. Solution: in the solution, it should only contain the code and comments within the code. There should be no explanations provided outside the code.
<answer: ✗ no, in the solution, it is reasonable to include the code and comments within the code. However, the Note section belongs to explanations provided outside the code, which is not allowed>

4. Instruction: The content of the instruction should be relevant to the Input and should be a summary of the Input content, without any additional unrelated information.
<answer: ✅yes, the Instruction is related to the Input>

- Overall answer:✗ no

- Reasons: In the solution, it is reasonable to include the code and comments within the code. However, the Note section belongs to explanations provided outside the code, which is not allowed. So the overall answer is no.

Figure 4: An example of LLM-based Generator-Discriminator framework on code generation task. The generator produce the instruction data based on the input (a). Subsequently, the discriminator accept the output and generated analysis for it. The output (b) includes four keys, we take the information as input and solution as output in our instruction tuning. The analysis (c) consists of the detailed reason for each rule and an overall answer to check if the sample meet all the requirements.

C Comparison with CodeAlpaca

CodeAlpaca dataset contains 20K multi-task instruction-following data generated by the techniques in the self-instruct (Taori et al., 2023). To ensure a fair and multidimensional comparison, we randomly sampled 1K and 5K from both datasets (CodeAlpaca and CodeSeaXDataset), set the same set of training hyper-parameters set (epoch = 3, learning rate = 1e-4, LoRA rank = 8) and used the same training prompts. To prevent overfitting, we use Low-Rank Adaption (LoRA) (Hu et al., 2022) for fine-tuning if the size of instruction-follow training dataset is less than 5K and perform full fine-tuning on whole 20K dataset.

1) After being fine-tuned with 1K, 5K and 20K of instructional data respectively, the performance of base model improves significantly on HumanEval shown in Figure 5. Taking Starcoder as the base

model, CodeSeaXDataset surpasses the CodeAlpaca (44.9% vs 41.7%, 45.7% vs 48.1% and 47.0% vs 50.5%) shown in Figure 5 (a), which emphasizes the effectiveness of our method on refining instruction data. As shown in Figure 5 (b), The results of different base models on CodeSeaXDataset surpasses the results on CodeAlpaca, which emphasizes the effectiveness of CodeSeaXDataset dataset in enhancing the instruction-following ability of the base model.

2) According to Table 4 and Table 5, All WaveCoder models significantly outperform the model fine-tuned with CodeAlpaca. Remarkably, The pass@1 score of WaveCoder-CL-13B outperforms CodeLLaMa-CodeAlpaca-13B achieving 10.6% absolute improvements on HumanEvalExplain. This emphasizes the effectiveness of defining and classifying code-related tasks on enhancing the generalization ability of Code LLMs.

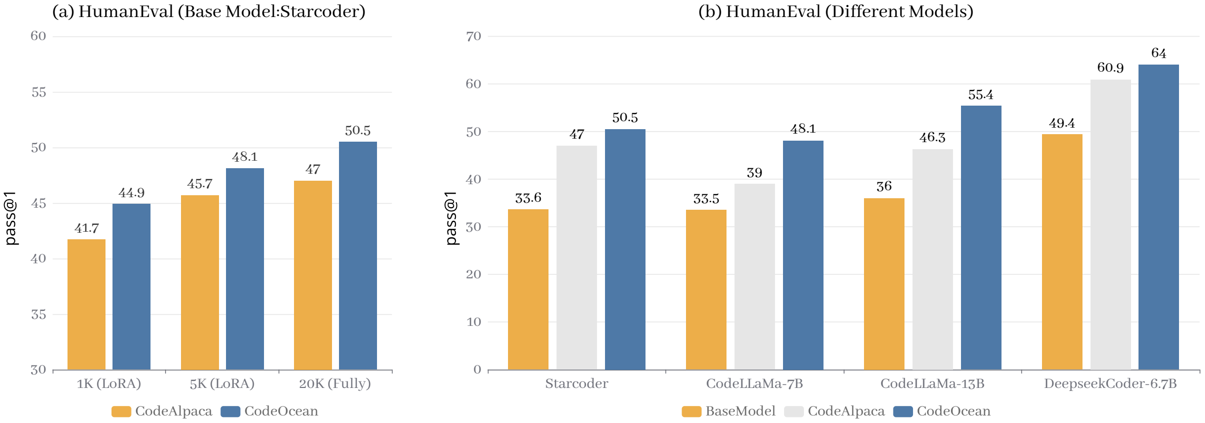


Figure 5: Comparision with CodeAlpaca with different dataset size(a) and different base models(b). CodeSeaX-Dataset outpeforms CodeAlpaca on HumanEval multidimensionally, more detailed analysis is shown in Section 3.3.

D Evaluation Benchmark

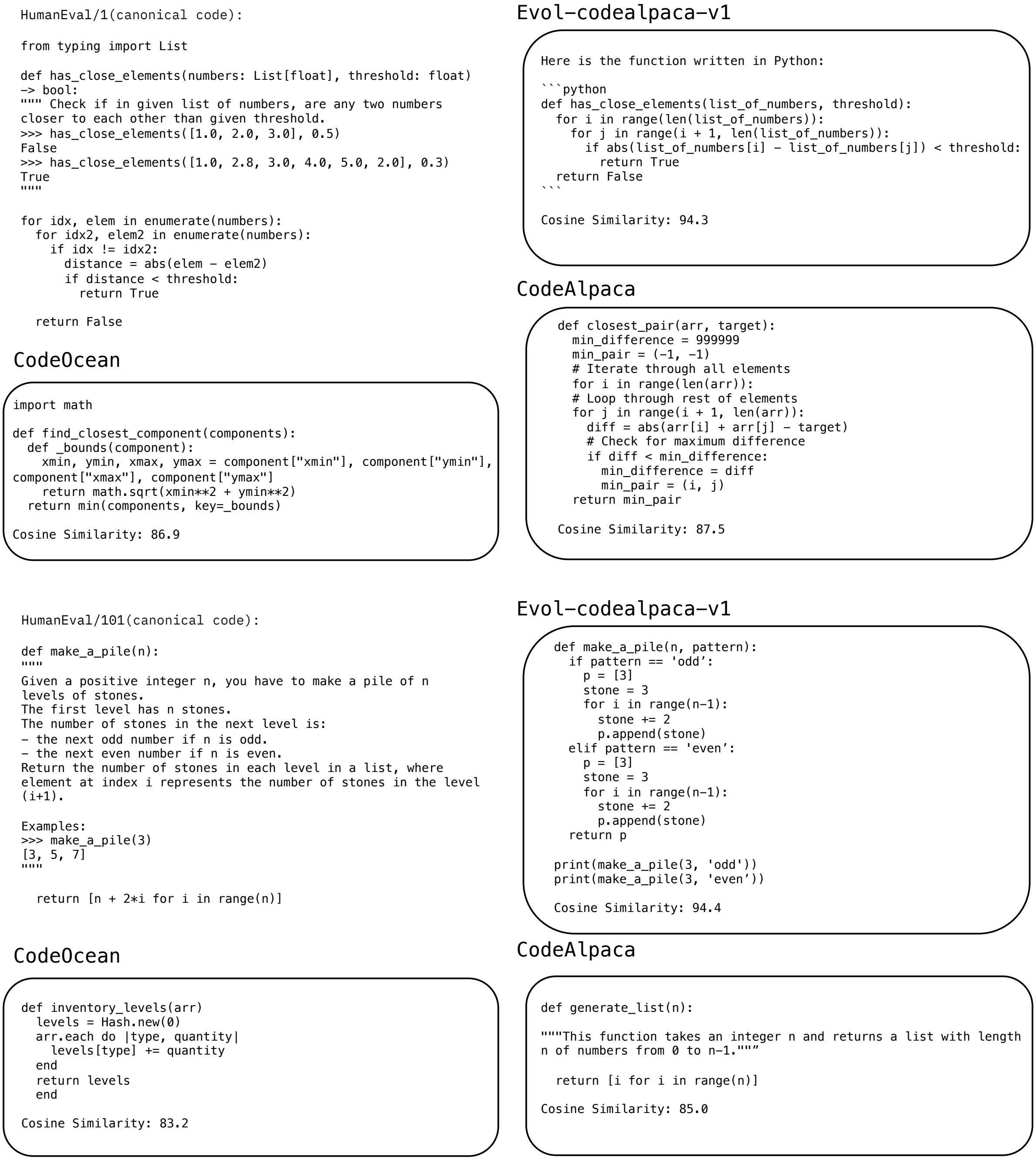
HumanEval [4], which consists of 164 manually-written Python programming problems and an average of 9.6 test cases allocated to each problem is now the most extensively adopted benchmark for Code LLMs.
MBPP [5] consists of around 1,000 crowd-sourced Python programming problems, designed to be solvable by entry level programmers, covering programming fundamentals, standard library functionality, and so on. In this paper, we choose the 500 problems test dataset to evaluate both few-shot inference of fine-tuned models. For whose MBPP (500) result is not reported or not used, we reproduced for them using bigcode-evaluation-harness [6].
HumanEvalPack [7] is an extension of OpenAI's HumanEval to cover 6 total languages across 3 tasks. In this paper, we select the HumanEvalFix to evaluate the code to code ability especially on code repair task and HumanEvalExplain benchmarks to evaluate the code to text ability especially on code summarization task.

[4] https://huggingface.co/datasets/openai_humaneval
[5] https://huggingface.co/datasets/mbpp
[6] https://github.com/bigcode-project/bigcode-evaluation-harness
[7] https://huggingface.co/datasets/bigcode/humanevalpack

LLM-based Generator-Discriminator framework

Input:
The Python code calculates the area of a circle given its radius.
Output:
task_name: Calculate Circle Area
instruction: Write a Python function that calculates the area of a circle given its radius.
information: The formula to calculate the area of a circle is A = pi * r^2, where A is the area and r is the radius.
solution:
import math
def area_of_circle(radius): return math.pi * radius ** 2
Analysis: - Step 1: Check the Instruction: The programming language should be specified in the instruction. <answer: yes, The instruction mentions "Write a Python function," which clearly specifies the programming language as Python.>
- Step 2: Check the Solution:
1. Solution is related to instruction and information. Solution is the specific resolution to instruction and information. <answer: yes, The solution provided calculates the area of a circle given its radius, which is directly related to the instruction and the information provided.>
2. In the solution, it should only contain the code and comments within the code. There should be no explanations provided outside the code. <answer: yes, The solution only contains the code without any explanations outside the code. Comments within the code are not present, but the code is straightforward and easy to understand.>
3. The code that contains algorithmic logic in solution should not often be buried inside complex or poorly documented functions, making them difficult to follow or learn from. <answer: yes, The code that contains algorithmic logic in the solution should not often be buried inside complex or poorly documented functions, making them difficult to follow or learn from.>
4. The code should import the required necessary libraries. <answer: yes, The code imports the necessary math library, which is required for calculating the area of a circle using the mathematical constant pi.>
- Overall answer: yes
- Reasons: All the requirements are met as per the given rules. The programming language is specified, and the solution is related to the instruction and information. The code is simple, easy to follow, and imports the necessary library. The solution provided is a good example of a well-structured and concise code that meets the task's specifications.

Table 7: An example of the LLM-based Generator-Discriminator framework



HumanEval/1(canonical code):

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float)
-> bool:
""" Check if in given list of numbers, are any two numbers
closer to each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True
"""

for idx, elem in enumerate(numbers):
  for idx2, elem2 in enumerate(numbers):
    if idx != idx2:
      distance = abs(elem - elem2)
      if distance < threshold:
        return True

  return False
```

Evol-codealpaca-v1

Here is the function written in Python:

~~~
```python
def has_close_elements(list_of_numbers, threshold):
  for i in range(len(list_of_numbers)):
    for j in range(i + 1, len(list_of_numbers)):
      if abs(list_of_numbers[i] - list_of_numbers[j]) < threshold:
        return True
  return False
```
~~~

Cosine Similarity: 94.3

CodeAlpaca

```
def closest_pair(arr, target):
  min_difference = 999999
  min_pair = (-1, -1)
  # Iterate through all elements
  for i in range(len(arr)):
  # Loop through rest of elements
  for j in range(i + 1, len(arr)):
    diff = abs(arr[i] + arr[j] - target)
    # Check for maximum difference
    if diff < min_difference:
      min_difference = diff
      min_pair = (i, j)
  return min_pair
```

Cosine Similarity: 87.5

CodeOcean

```
import math

def find_closest_component(components):
  def _bounds(component):
    xmin, ymin, xmax, ymax = component["xmin"], component["ymin"],
component["xmax"], component["ymax"]
    return math.sqrt(xmin**2 + ymin**2)
  return min(components, key=_bounds)
```

Cosine Similarity: 86.9

HumanEval/101(canonical code):

```
def make_a_pile(n):
"""
Given a positive integer n, you have to make a pile of n
levels of stones.
The first level has n stones.
The number of stones in the next level is:
- the next odd number if n is odd.
- the next even number if n is even.
Return the number of stones in each level in a list, where
element at index i represents the number of stones in the level
(i+1).

Examples:
>>> make_a_pile(3)
[3, 5, 7]
"""

  return [n + 2*i for i in range(n)]
```

Evol-codealpaca-v1

```
def make_a_pile(n, pattern):
  if pattern == 'odd':
    p = [3]
    stone = 3
    for i in range(n-1):
      stone += 2
      p.append(stone)
  elif pattern == 'even':
    p = [3]
    stone = 3
    for i in range(n-1):
      stone += 2
      p.append(stone)
  return p

print(make_a_pile(3, 'odd'))
print(make_a_pile(3, 'even'))
```

Cosine Similarity: 94.4

CodeAlpaca

```
def generate_list(n):

"""This function takes an integer n and returns a list with length
n of numbers from 0 to n-1."""

  return [i for i in range(n)]
```

Cosine Similarity: 85.0

CodeOcean

```
def inventory_levels(arr)
  levels = Hash.new(0)
  arr.each do |type, quantity|
    levels[type] += quantity
  end
  return levels
  end
```

Cosine Similarity: 83.2

Figure 6: Examples about data leakage.