WizardCoder: EMPOWERING CODE LARGE LANGUAGE MODELS WITH EVOL-INSTRUCT

Ziyang Luo[1*] **Can Xu**[2*] **Pu Zhao**[2] **Qingfeng Sun**[2] **Xiubo Geng**[2]
Wenxiang Hu[2] **Chongyang Tao**[2] **Jing Ma**[1†] **Qingwei Lin**[2] **Daxin Jiang**[2†]
[1]Hong Kong Baptist University [2]Microsoft
{cszyluo, majing}@comp.hkbu.edu.hk, {caxu,puzhao}@microsoft.com
{qins,xigeng,wenxh,chongyang.tao,qlin,djiang}@microsoft.com

ABSTRACT

Code Large Language Models (Code LLMs), such as StarCoder, have demonstrated remarkable performance in various code-related tasks. However, different from their counterparts in the general language modeling field, the technique of instruction fine-tuning remains relatively under-researched in this domain. In this paper, we present *Code Evol-Instruct*, a novel approach that adapts the *Evol-Instruct* method to the realm of code, enhancing Code LLMs to create novel models *WizardCoder*. Through comprehensive experiments on five prominent code generation benchmarks, namely HumanEval, HumanEval+, MBPP, DS-1000, and MultiPL-E, our models showcase outstanding performance. They consistently outperform all other open-source Code LLMs by a significant margin. Remarkably, *WizardCoder 15B* even surpasses the well-known closed-source LLMs, including Anthropic's Claude and Google's Bard, on the HumanEval and HumanEval+ benchmarks. Additionally, *WizardCoder 34B* not only achieves a HumanEval score comparable to GPT3.5 (ChatGPT) but also surpasses it on the HumanEval+ benchmark. Furthermore, our preliminary exploration highlights the pivotal role of instruction complexity in achieving exceptional coding performance.

1 INTRODUCTION

Recently, Large Language Models (LLMs) (Brown et al., 2020; OpenAI, 2023; Chowdhery et al., 2022; Anil et al., 2023; Hoffmann et al., 2022; Rae et al., 2021; Zeng et al., 2022; Zhang et al., 2022; Touvron et al., 2023a) have garnered immense attention and demonstrated impressive success. Notably, OpenAI's GPT3.5 (ChatGPT) stands out as a prominent example. These models, through extensive pre-training on vast internet data and fine-tuning with detailed instruction data (Ouyang et al., 2022), have achieved state-of-the-art (SOTA) zero-shot performance across diverse NLP tasks. This trend also extends to the realm of code understanding and generation, where a multitude of Code LLMs have emerged (Chen et al., 2021a; Li et al., 2022; Fried et al., 2022; Nijkamp et al., 2023b; Zheng et al., 2023; Wang et al., 2021; 2023; Li et al., 2023a; Nijkamp et al., 2023a; Rozière et al., 2023). These models, pre-trained on substantial code data, excel in various code-related tasks, consistently delivering impressive performance.

In contrast to most previous Code LLMs that primarily focus on the pre-training process, there has been limited exploration of fine-grained instruction tuning in the code domain. The introduction of instruction tuning was initially designed to enhance the generalization capabilities of LMs across different tasks via multitask training (Raffel et al., 2020; Wei et al., 2022; Chung et al., 2022; Aribandi et al., 2022; Sanh et al., 2022; Xu et al., 2022; Khashabi et al., 2020). OpenAI's InstructGPT (Ouyang et al., 2022), for instance, involved soliciting human annotators to provide explicit instructions to ensure alignment with users' intentions. Similarly, recent works such as Alpaca (Taori et al., 2023) employed the self-instruct (Wang et al., 2022) method, where GPT3.5 (ChatGPT) generated the instruction data. Vicuna (Chiang et al., 2023) utilized user-shared conversations collected from

[*] Equal contribution. Work done during the internship at Microsoft.
[†] Corresponding author.

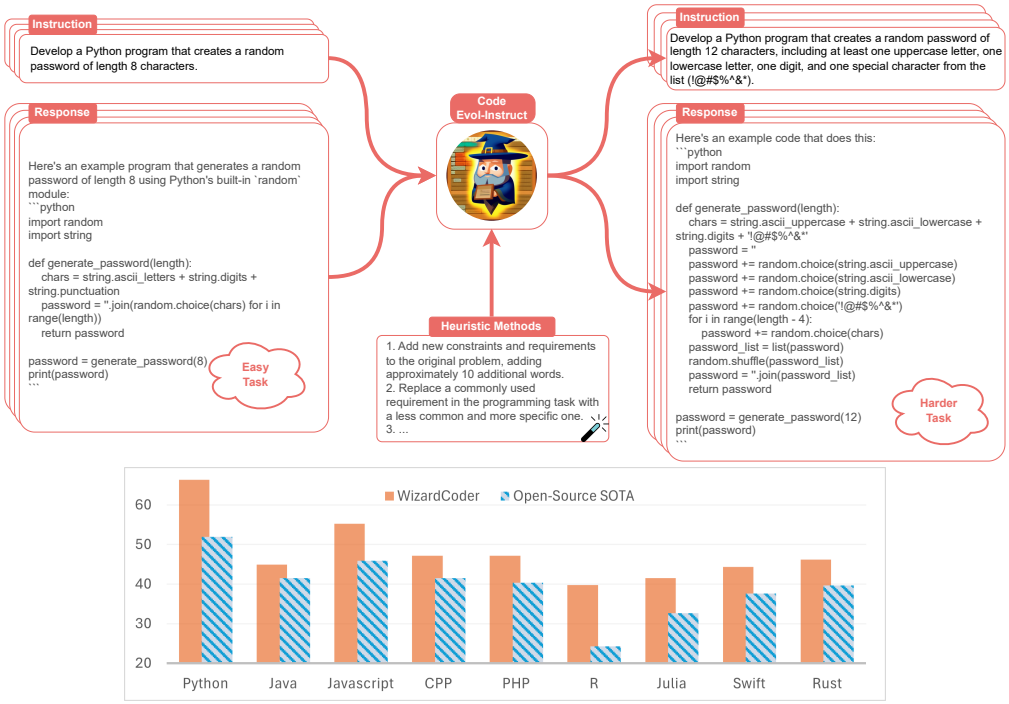


Figure 1: An illustration of our novel *Code Evol-Instruct* and the superior pass@1 performance of our *WizardCoder 34B*, outperforming the open-source SOTA (CodeLlama-34B-Series as of the date before August 24, 2023) by a large margin in 9 different programming languages. The Python score is the mean between HumanEval and MBPP.

ShareGPT.com. WizardLM (Xu et al., 2023) introduces the *Evol-Instruct* method, which involves evolving existing general instruction data to generate more complex and diverse datasets. Drawing inspiration from these previous works in the general domain, our work, *Code Evol-Instruct*, is specifically tailored to the coding domain's distinctive characteristics.

In this study, we aim to enhance the capabilities of the SOTA open-source Code LLMs (i.e., StarCoder and CodeLlama), by introducing our novel *Code Evol-Instruct*. The motivation of this fine-grained instruction-tuning method in the code domain is to automatically increase the complexity of code instruction data, so as to make the best of the internal coding ability of the Code LLMs. Our *Code Evol-Instruct* incorporates several novel methods, including heuristics tailored to coding task features, adversarial sample heuristics, time/space complexity requirements, and evolving stop controls. The whole process includes two steps: initially, we apply our *Code Evol-Instruct* to evolve basic code instruction data, specifically Code Alpaca (Chaudhary, 2023). Subsequently, we fine-tune StarCoder and CodeLlama using our newly generated code instruction-following training set, resulting in our *WizardCoder* models.

Figure 1 and the experimental results obtained from five code generation benchmarks, namely HumanEval (Chen et al., 2021b), HumanEval+ (Liu et al., 2023), MBPP (Austin et al., 2021), DS-100 (Lai et al., 2022), and MultiPL-E (Cassano et al., 2022), demonstrate that our *WizardCoder* models outperform all other open-source Code LLMs (before August 24, 2023), achieving state-of-the-art (SOTA) performance. Remarkably, our *WizardCoder 15B* even surpasses well-known Anthropic's Claude and Google's Bard in terms of pass rates on HumanEval and HumanEval+. Furthermore, *WizardCoder 34B* not only achieves a HumanEval score comparable to GPT3.5 (ChatGPT) but also surpasses it on the HumanEval+ benchmark. Beyond this, our preliminary studies indicate that the complexity of instructions is the key to achieving exceptional coding performance.

The contributions of this work can be summarized as follows:

- We introduce *Code Evol-Instruct*, a novel instruction fine-tuning approach for code, which enhances the performance of the open-source Code LLMs by a large margin.

- We develop *WizardCoder* models, which surpass all other open-source Code LLMs by a substantial margin in coding tasks. Notably, the 15B version even outperforms the well-known closed-source LLMs, such as Claude, and Bard. The 34B version achieves a HumanEval score comparable to GPT3.5 (ChatGPT) and surpasses it on the HumanEval+ benchmark.
- We conduct a preliminary study highlighting the pivotal role of instruction complexity in achieving exceptional coding performance.

2 Related Work

Large Language Models. Recently, LLMs have demonstrated remarkable achievements across a broad spectrum of tasks. Prominent tech companies have made significant strides in developing highly proficient LLMs. These include OpenAI's GPT3&4 (Brown et al., 2020; OpenAI, 2023), Google's PaLM (Chowdhery et al., 2022; Anil et al., 2023), and Bard[1], DeepMind's Chinchilla (Hoffmann et al., 2022), and Gopher (Rae et al., 2021), as well as Anthropic's Claude[2]. However, it is important to note that these models are closed-source and can only be accessed through specific APIs or may not be accessible at all.

The AI community has witnessed the release of several open-source LLMs, where the model weights are made publicly available. EleutherAI has contributed GPT-NeoX-20B (Black et al., 2022) and GPT-J-6B (Wang & Komatsuzaki, 2021). Google has released UL2-20B (Tay et al., 2022). Tsinghua University has introduced GLM-130B (Zeng et al., 2022). Meta has released OPT (Zhang et al., 2022) and LLaMA1&2 (Touvron et al., 2023a;b). It is worth noting that while these open-source models have made valuable contributions, they generally do not exhibit the same level of performance as their closed-source counterparts.

Large Language Models for Code. Recent studies have introduced a significant number of LLMs for code-related tasks to address the challenges of code understanding and generation. OpenAI has unveiled Codex (Chen et al., 2021a) and Code-Davinci (Microsoft, 2023). Google has proposed PaLM-Coder (Chowdhery et al., 2022). They perform outstandingly on the popular code completion benchmarks, like HumanEval (Chen et al., 2021b) and MBPP (Austin et al., 2021). However, these models are closed-source.

On the other hand, there are several open-source Code LLMs available. Salesforce has introduced CodeGen1&2 (Nijkamp et al., 2023b;a), CodeT5 (Wang et al., 2021), and CodeT5+ (Wang et al., 2023). Tsinghua University has contributed CodeGeeX (Zheng et al., 2023), and the BigCode Project has developed StarCoder (Li et al., 2023a). Meta has released the CodeLlama-Series (Rozière et al., 2023), which achieves open-source SOTA performance on several benchmarks. The closely related model, CodeLlama-Instruct, refines its performance through the self-instruct method. These models have demonstrated notable advancements in code-related tasks. However, when compared to the SOTA closed-source models, they still lag behind significantly. In contrast to the aforementioned models, our work demonstrates that further training Code LLMs with our *Code Evol-Instruct* can substantially enhance performance.

Instruction Fine-Tuning. The primary objective of instruction fine-tuning in its early stages was to enhance the cross-task generalization capabilities of LMs. This was achieved by fine-tuning LMs with a substantial corpus of public NLP tasks. T5 (Raffel et al., 2020) was among the first models to explore this approach, training on a multitude of supervised text-to-text tasks. Subsequent works such as FLAN (Wei et al., 2022), ExT5 (Aribandi et al., 2022), T0 (Sanh et al., 2022), and UnifiedQA (Khashabi et al., 2020) further expanded the range of tasks to bolster the overall generalization ability of LMs. Notably, ZeroPrompt (Xu et al., 2022) and FLAN-T5 (Chung et al., 2022) pushed the envelope by incorporating thousands of tasks in their training pipelines. Across these studies, a consistent finding emerges: fine-tuning LMs with diverse NLP task instructions yields significant performance improvements when applied to new tasks.

[1] https://bard.google.com/
[2] https://www.anthropic.com/index/introducing-claude

While fine-tuning LMs with diverse NLP tasks has shown promising results, it often falls short in aligning with the intentions of real-world users. OpenAI has pursued a different approach by soliciting human annotators to provide a large corpus of human instructions, encompassing diverse forms and a wide range of task types. Building upon this dataset, OpenAI trained its GPT3 (Brown et al., 2020) model to create InstructGPT (Ouyang et al., 2022), which better aligns with users' inputs. This line of development has even led to the impressive work known as GPT3.5 (ChatGPT). However, it is important to note that the dataset and model weights associated with these advancements are not publicly available. Alpaca (Taori et al., 2023) takes a different route by adopting the self-instruct method (Wang et al., 2022), leveraging GPT3.5 (ChatGPT) to generate data for training. Vicuna (Chiang et al., 2023) utilizes user-shared conversations collected from ShareGPT.com to train its models. WizardLM (Xu et al., 2023) introduces the *Evol-Instruct* method, which involves evolving existing general instruction data to generate more complex and diverse datasets. Drawing inspiration from this idea, our work, *Code Evol-Instruct*, aligning with the distinctive characteristics of coding domains, is the first instruction fine-tuning method explicitly designed to enhance Code LLMs.

3 WizardCoder: SOTA Open-Source Code LLM

In this section, we elaborate on the methodological details of *WizardCoder*. As illustrated in Figure 1, we first adopt our *Code Evol-Instruct* to iteratively evolve the Code Alpaca dataset. Subsequently, we fine-tune the pre-trained Code LLMs with the evolved data.

3.1 Code Evol-Instruct

Inspired by the *Evol-Instruct* method proposed by WizardLM Xu et al. (2023), this work attempts to automatically enhance the complexity of code instructions, thereby improving the fine-tuning effectiveness of Code LLMs. Diverging from the general domain, our methods are meticulously designed to align with the specific characteristics of coding domains. The evolutionary process introduces the following features:

1. Heuristics aligned with coding task features on platforms like LeetCode, strategically increasing the complexity of coding tasks to enhance the model's capabilities.

2. Introduction of erroneous code as an adversarial sample, inspired by prior research on attacking pre-trained code models Yang et al. (2022); Jha & Reddy (2022), adds a novel and effective method to escalate task complexity.

3. Introduction of a heuristic emphasizing time and space complexity leverages insights from previous studies Madaan et al. (2023), providing a valuable avenue for improving task complexity.

So, the code evolutionary prompt template is as follows:

Prompt for Code Evol-Instruct

Please increase the difficulty of the given programming test question a bit.

You can increase the difficulty using, but not limited to, the following methods:
{method}

{question}

Here, {question} represents the current code instruction awaiting evolution, and {method} is the type of evolution. The five types we used are listed as follows:

Code Evolution Heuristic Methods

Add new constraints and requirements to the original problem, adding approximately 10 additional words.

Replace a commonly used requirement in the programming task with a less common and more specific one.

If the original problem can be solved with only a few logical steps, please add more reasoning steps.

Provide a piece of erroneous code as a reference to increase misdirection.

Propose higher time or space complexity requirements, but please refrain from doing so frequently.

3.2 TRAINING *WizardCoder*

We employ the following procedure to train *WizardCoder*. Initially, we utilize StarCoder 15B (Li et al., 2023a) and CodeLlama-34B-Python (Rozière et al., 2023) as the foundations and proceed to fine-tune them using the code instruction-following training set, which was evolved through *Code Evol-Instruct*. The prompt format for fine-tuning is outlined as follows:

Prompt for Fine-Tuning Format

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:

To construct the training dataset, we initialized it with the instruction-following dataset called Code Alpaca[3]. We iteratively employ the *Code Evol-Instruct* technique on this dataset consisting of around 20k samples to produce evolved data. After each round of data evolution, we merge the evolved data from all previous rounds with the original dataset to finetune Code LLMs. An external dev set serves as the controlled Evol Stop. If the performance drops, we halt the evolution. In Appendix C, we outline the approach employed to prevent data leakage. Additionally, Appendix D showcases some evolved examples for reference.

4 EXPERIMENT

This section begins by providing a comprehensive overview of the baseline models in our experiments. Subsequently, we present the performance of our models on five code generation benchmarks: HumanEval (Chen et al., 2021b), HumanEval+ (Liu et al., 2023), MBPP (Austin et al., 2021), DS-1000 (Lai et al., 2022) and MultiPL-E (Cassano et al., 2022).

4.1 BASELINES

Closed-Source Models. Multiple technology companies have successfully developed highly proficient LLMs while choosing not to publicly release them. These models are referred to as closed-source models. For our research, we incorporate a substantial number of these models as our baselines. Specifically, our baselines encompass the following: (i) OpenAI's GPT3.5(ChatGPT)&GPT4 (OpenAI, 2023), Code-Davinci-002 (Microsoft, 2023), Code-Cushman-001 (Microsoft, 2023), and

[3] https://github.com/sahil280114/codealpaca

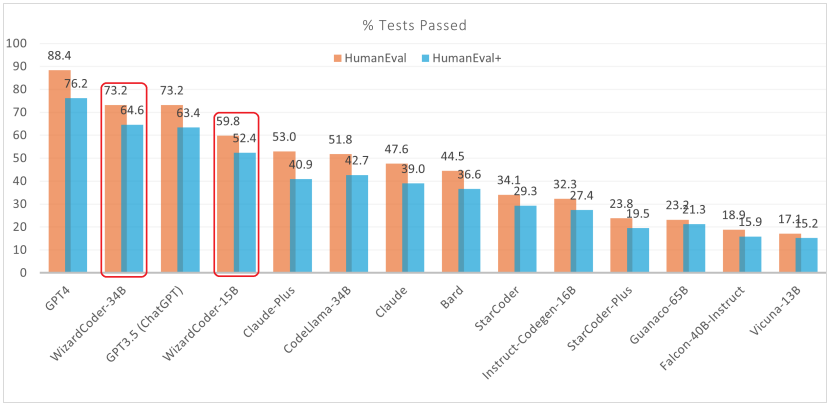


Figure 2: The percentage of pass rates on the HumanEval and HumanEval+ with a single attempt (greedy decoding), following the EvalPlus leaderboard (Liu et al., 2023).

Codex (Chen et al., 2021a); (ii) Google's Bard, PaLM 2 (Anil et al., 2023), PaLM (Chowdhery et al., 2022), and LaMDA (Thoppilan et al., 2022); (iii) Google DeepMind's AlphaCode (Li et al., 2022);(iv) Anthropic's Claude; (v) Huawei's PanguCoder2 (Shen et al., 2023); and (vi) Meta's Unnatural-CodeLlama-34B (Rozière et al., 2023).

Open-Source Models. Several open-source LLMs (OSS) have been made available to the AI community, although their performance generally lags behind the closed-source models a lot. As part of our research, we incorporate a significant number of these open-source models as our baselines. Our baselines encompass the following models: InCoderFried et al. (2022), StarCoder and StarCoder-Plus (Li et al., 2023a), LLaMa1&2 (Touvron et al., 2023a;b), CodeGen (Nijkamp et al., 2023b), CodeGeeX (Zheng et al., 2023), CodeT5+(Wang et al., 2023), and CodeLlama (Rozière et al., 2023). In addition, we also include several models with instructions fine-tuning, including CodeLlama-Instruct (Rozière et al., 2023), OctoCoder (Muennighoff et al., 2023), InstructCodeT5+ (Wang et al., 2023), Instruct-Codegen-16B,[4] Guanaco-65B (Dettmers et al., 2023), Falcon-40B-Instruct (Penedo et al., 2023) and Vicuna-13B (Chiang et al., 2023). More details can be found in the Appendix B.

4.2 Implementation Details

The StarCoder and CodeLlama-34B-Python serve as our basic foundation models. OpenAI's gpt3.5-turbo is used to evolve the dataset and generate responses. The evolved dataset consists of approximately 78k samples. To fine-tune the basic models, we employ specific configurations, including a batch size of 512, a sequence length of 2048, 200 fine-tuning steps, 30 warmup steps, a learning rate of 2e-5, a Cosine learning rate scheduler, and fp16 mixed precision.

4.3 Evaluation on HumanEval, HumanEval+, and MBPP

HumanEval (Chen et al., 2021b), HumanEval+ (Liu et al., 2023), and MBPP (Austin et al., 2021) are key benchmarks in the Code LLM field, featuring diverse Python programming problems validated using test cases. HumanEval comprises 164 problems with an average of 9.6 test cases per problem. HumanEval+ expands the test cases significantly to an average of 774.8 per problem. In contrast, MBPP provides 500 test programming problems with three automated test cases each.[5]

Comparing with the Closed-Source Models. Following the same setting of the EvalPlus leaderboard (Liu et al., 2023). In Figure 2, we compare our *WizardCoder* models with the closed-source models, such as GPT4, Claude, and Bard on this leaderboard. Notably, all models generate code

[4] `https://huggingface.co/sahil2801/instruct-codegen-16B`

[5] For a fair comparison, we present results for GPT3.5(ChatGPT)&GPT4 using Eval-Plus with the latest OpenAI's APIs (Liu et al., 2023) (Figure 2) and OpenAI's report (OpenAI, 2023) (Table 1). Prompt format details are in Appendix A.

Table 1: Results of pass@1(%) on HumanEval and MBPP. We follow the previous works (Chen et al., 2021b) to generate n=200 samples to estimate the pass@1 score of our *WizardCoder* models with the same set of hyper-parameters: temperate=0.2, and top_p=0.95. *: our reproduced results.

Model	**Params**	**HumanEval**	**MBPP**
Closed-source models			
LaMDA (Thoppilan et al., 2022)	137B	14.0	-
AlphaCode (Li et al., 2022)	1.1B	17.1	-
PaLM (Chowdhery et al., 2022)	540B	26.2	36.8
PaLM-Coder (Chowdhery et al., 2022)	540B	36.0	47.0
PaLM 2-S (Anil et al., 2023)	Unknown	37.6	50.0
Codex (Chen et al., 2021a)	2.5B	21.4	-
Codex (Chen et al., 2021a)	12B	28.8	-
Code-Cushman-001 (Microsoft, 2023)	Unknown	33.5	45.9
Code-Davinci-002 (Microsoft, 2023)	Unknown	47.0	58.1
GPT-3.5 (ChatGPT) (OpenAI, 2023)	Unknown	48.1	52.2
PanguCoder2 (Shen et al., 2023)	15B	61.6	-
Unnatural-CodeLlama (Rozière et al., 2023)	34B	62.2	61.2
GPT-4 (OpenAI, 2023)	Unknown	67.0	-
Open-source models			
Llama (Touvron et al., 2023a)	65B	23.7	37.7
Llama2 (Touvron et al., 2023b)	70B	29.9	45.0
CodeGen-Mono (Nijkamp et al., 2023b)	16B	29.3	35.3
CodeGeeX (Zheng et al., 2023)	13B	22.9	24.4
StarCoder (Li et al., 2023a)	15B	33.6	43.6*
CodeT5+ (Wang et al., 2023)	16B	30.9	-
InstructCodeT5+ (Wang et al., 2023)	16B	35.0	-
OctoCoder (Muennighoff et al., 2023)	15B	46.2	-
CodeLlama (Rozière et al., 2023)	34B	48.8	55.0
CodeLlama-Python (Rozière et al., 2023)	34B	53.7	56.2
CodeLlama-Instruct (Rozière et al., 2023)	34B	41.5	57.0
WizardCoder	15B	**57.3**	**51.8**
WizardCoder	34B	**71.5**	**61.2**

solutions for each problem utilizing a single attempt, and the resulting pass rate percentage is reported. To maintain consistency, we employ the same experimental setup by generating answers using greedy decoding and evaluate our *WizardCoder* models using the provided evaluation codes.

As depicted in Figure 2, our *WizardCoder 34B* attains the second position in this benchmark, surpassing GPT3.5 (ChatGPT, 64.6 vs. 63.4) on HumanEval+. Our 15B version outperforms Claude-Plus (59.8 vs. 53.0) and Bard (59.8 vs. 44.5). Furthermore, our *WizardCoder* models demonstrate a remarkable superiority over other open-source LLMs that undergo instruction fine-tuning.

Comparing with the Open-Source Models. In Table 1, we conduct a comprehensive comparison of our *WizardCoder* with other open-source models on the HumanEval and MBPP benchmarks. In contrast to the results presented in Figure 2, we adhere to the approach outlined in previous studies Chen et al. (2021b) by generating n samples for each problem to estimate the pass@1 score. The findings presented in Table 1 clearly demonstrate that our *WizardCoder* exhibits a substantial performance advantage over all the open-source models.

4.4 Evaluation on Multi-Language Coding

We included comprehensive assessment results across 8 distinct programming languages on the MultiPL-E benchmarks. These languages encompass Java, JavaScript, C++, PHP, R, Julia, Swift, and Rust. The empirical results, as presented in Table 2, distinctly demonstrate the superior performance of our *WizardCoder* models across all evaluated programming languages, surpassing the SOTA open-source Code LLMs. This underscores the efficacy of our *Code Evol-Instruct* method.

4.5 Evaluation on DS-1000

The DS-1000 benchmark Lai et al. (2022) comprises 1k distinct data science workflows spanning 7 libraries. It assesses the performance of code generations against test cases and supports two evaluation modes: completion and insertion. In our experiments, we only report insertion scores for

models that support. In Table 3, we present pass@1 (n=40) results for each library, along with an overall score.[6] Based on these results, our conclusion is that *WizardCoder* demonstrates a significant superiority over all other models when tackling data science problems on the DS-1000 benchmark.

Table 2: Results of pass@1(%) on 8 different programming languages on the MultiPL-E (Cassano et al., 2022) benchmarks. All models are evaluated with the same set of hyper-parameters: temperature=0.2, top_p=0.95, max_length=512, and n=50.

Model	**Params**	**Java**	**Js**	**CPP**	**PHP**	**R**	**Julia**	**Swift**	**Rust**
CodeGen-Multi	16B	22.2	19.2	21.0	8.4	6.5	0	1.3	4.2
CodeGeeX	13B	19.1	16.9	16.9	13.5	3.9	0.3	7.3	7.9
Code-Cushman-001	-	31.9	31.3	30.6	29.0	11.0	1.5	22.1	25.2
StarCoderBase	15B	28.5	31.7	30.6	26.8	10.2	21.1	16.7	24.5
StarCoder	15B	30.2	30.8	31.6	26.1	15.5	23.0	22.7	21.8
CodeLlama	34B	40.2	41.7	41.4	40.4	22.7	31.4	35.3	38.7
CodeLlama-Python	34B	39.5	44.7	39.1	39.8	22.4	31.4	34.3	39.7
CodeLlama-Instruct	34B	41.5	45.9	41.5	37.0	24.3	32.7	37.6	39.3
WizardCoder	15B	**35.8**	**41.9**	**39.0**	**39.3**	**33.5**	**34.0**	**33.7**	**27.1**
WizardCoder	34B	**44.9**	**55.3**	**47.2**	**47.2**	**39.8**	**41.5**	**44.3**	**46.2**

Table 3: Performance of *WizardCoder 15B* and baseline models on DS-1000. All models are evaluated with the same set of hyper-parameters: temperature=0.2, top_p=0.5, max_length=1024. Scores are average pass@1 accuracy over 40 samples. Matplotlib (plt) task does not have the right context, so insertion and completion scores are identical.

Format	**Model**	**plt**	**np**	**pd**	**py**	**scp**	**sk**	**tf**	**All**
	# of problems:	155	220	291	68	106	115	45	1,000
Completion	InCoder-6B	28.3	4.4	3.1	4.4	2.8	2.8	3.8	7.4
Completion	CodeGen-mono	31.7	10.9	3.4	7.0	9.0	10.8	15.2	11.7
Completion	Code-Cushman-001	40.7	21.8	7.9	12.4	11.3	18.0	12.2	18.1
Completion	StarCoder	51.7	29.7	11.4	21.4	20.2	**29.5**	24.5	26.0
Completion	*WizardCoder*	**55.2**	**33.6**	**16.7**	**26.2**	**24.2**	24.9	**26.7**	**29.2**
Insertion	InCoder-6B	28.3	4.6	2.9	4.4	2.8	3.1	7.8	7.5
Insertion	StarCoder	51.7	30.8	10.3	21.0	20.2	27.4	20.0	25.4
Insertion	*WizardCoder*	**55.2**	**35.1**	**20.4**	**30.4**	**28.9**	**32.3**	**37.8**	**32.8**

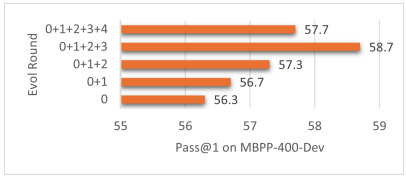
5 ANALYSIS

Evolution Models and Rounds. In Table 4, GPT-4 replaces GPT-3.5 for evolved rounds, boosting HumanEval Pass@1 scores to 73.8 (34B) and 62.2 (15B). Using OSS CodeLlama-Instruct-34B also proves effective, yielding scores of 70.1 (34B) and 55.5 (15B). Despite GPT-4's superior coding performance (88.4 vs. 73.2), the gain in evolved rounds is not proportional (73.8 vs. 73.2). Conversely, CodeLlama's weaker performance narrows when using *Code Evol-Instruct* (73.2 vs. 70.1), highlighting its crucial role. More experiments details are listed in Appendix E. Additionally, Figure 3 presents results for different data evolution rounds. All models are fine-tuned with 200 steps. Due to the limited size of the dev set of MBPP, we merged the training set and dev set, forming the MBPP-400 dev set. The experiments reveal that the highest pass@1 scores on both the MBPP-400 dev set and the HumanEval are achieved subsequent to three rounds of evolution.

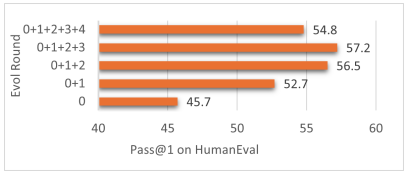
Table 4: Different evolution execution models.

Base Model	Evol Model	Pass@1
StarCoder-15B	GPT-4	62.2
StarCoder-15B	GPT-3.5	59.8
StarCoder-15B	CodeLlama	55.5
CodeLlama-34B	GPT-4	73.8
CodeLlama-34B	GPT-3.5	73.2
CodeLlama-34B	CodeLlama-34B	70.1

[6]Given that this benchmark and its evaluation codes are not designed for the instruction fine-tuned models, we encounter significant challenges in aligning our 34B model with this framework. Moreover, the Codellama-34B base model does not support code insertion. Thus, we only include our 15B model results.



(a) Pass@1 performance on MBPP-400 dev set.



(b) Pass@1 performance on HumanEval.

Figure 3: The impact of the number of data evolution rounds.

Complexity and Quantity. While the enhanced performance attributed to our *Code Evol-Instruct* method has been evident in prior experiments, it remains an open question whether this performance gain is a result of an increase in the number of samples or tokens. During the evolution, each round includes more samples, and the introduction of more complex instructions inevitably leads to an increase in tokens within the training data. To address this question, we fine-tune the models using only the specific round data separately from scratch with a similar number of samples (upper part) or tokens (lower part) in Table 5.

When each round contains the same number of samples or tokens, the models trained with the seed data still lag behind the evolved rounds. Furthermore, combining data from different rounds leads to the best performance. These results suggest that the primary source of the gain is indeed attributable to our *Code Evol-Instruct* method, rather than merely an increase in samples or tokens.

Table 5: Analysis of whether the performance gain comes from more tokens.

Evol	#Samples	Pass@1
Round 0	20.0k	45.7
Round 1	18.8k	56.1
Round 2	19.7k	53.0
Round 3	19.3k	54.3
Round 4	19.0k	51.2
Evol	**#Tokens**	**Pass@1**
Round 0	2.3M	44.5
Round 1	2.3M	51.8
Round 2	2.3M	52.4
Round 3	2.3M	50.0
Round 4	2.3M	49.4

Complexity and Similarity. Apart from the quantity analysis, we also investigate whether evolution leads to the inclusion of data more similar to the test set. To address this, we perform an analysis of the HumanEval test set. We employ test samples as queries to retrieve the top-1 sample from each evolved round's training data, utilizing the SOTA embeddings model, gte-large (Li et al., 2023b). Additionally, we employ GPT4, to provide average similarity scores between the test set and the retrieved top-1 samples. The details are shown in Appendix C.

Figure 4 illustrates that the evolution process does not yield higher similarity scores. Furthermore, similarity scores across all rounds remain relatively low. These findings indicate that the primary source of performance gain is the introduction of more complex data.

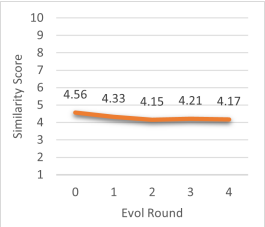


Figure 4: Average similarity scores between HumanEval samples and the top-1 retrieved data, ranging from 1 (completely different) to 10 (identical).

6 Conclusion and Future Work

This paper introduces *WizardCoder* models, the *Code Evol-Instruct* fine-tuned Code LLMs. The experimental results demonstrate that *WizardCoder* models achieve SOTA performance surpassing all existing open-source Code LLMs on five widely recognized code generation benchmarks: HumanEval, HumanEval+, MBPP, DS-1000 and MultiPL-E. Notably, *WizardCoder 15B* model surpasses some of the well-known closed LLMs, such as Claude and Bard. Additionally, *WizardCoder 34B* achieves a HumanEval score comparable to GPT3.5 (ChatGPT) and surpasses it on the HumanEval+ benchmark. Furthermore, our analysis underscores the pivotal role of instruction complexity in enhancing performance. For future work, as depicted in Figure 2, our model still falls significantly behind the SOTA LLM, GPT4. Therefore, future work will further augment the performance of our model.

ACKNOWLEDGMENTS

This work is partially supported by National Natural Science Foundation of China Young Scientists Fund(No. 62206233) and Hong Kong RGC ECS (No. 22200722).

REFERENCES

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernández Ábrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan A. Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vladimir Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, and et al. Palm 2 technical report. *CoRR*, abs/2305.10403, 2023. doi: 10.48550/arXiv.2305.10403. URL https://doi.org/10.48550/arXiv.2305.10403.

Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q. Tran, Dara Bahri, Jianmo Ni, Jai Prakash Gupta, Kai Hui, Sebastian Ruder, and Donald Metzler. Ext5: Towards extreme multi-task scaling for transfer learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=Vzh1BFUCiIX.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. Gpt-neox-20b: An open-source autoregressive language model. *CoRR*, abs/2204.06745, 2022. doi: 10.48550/arXiv.2204.06745. URL https://doi.org/10.48550/arXiv.2204.06745.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *CoRR*, abs/2208.08227, 2022. doi: 10.48550/arXiv.2208.08227. URL https://doi.org/10.48550/arXiv.2208.08227.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021a. URL https://arxiv.org/abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021b. URL https://arxiv.org/abs/2107.03374.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://vicuna.lmsys.org.

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 Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben 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 Garcia, 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 Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *CoRR*, abs/2210.11416, 2022. doi: 10.48550/arXiv.2210.11416. URL https://doi.org/10.48550/arXiv.2210.11416.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *CoRR*, abs/2305.14314, 2023. doi: 10.48550/arXiv.2305.14314. URL https://doi.org/10.48550/arXiv.2305.14314.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022. doi: 10.48550/arXiv.2204.05999. URL https://doi.org/10.48550/arXiv.2204.05999.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022. doi: 10.48550/arXiv.2203.15556. URL https://doi.org/10.48550/arXiv.2203.15556.

Akshita Jha and Chandan K. Reddy. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *AAAI Conference on Artificial Intelligence*, 2022. URL https://api.semanticscholar.org/CorpusID:249240370.

Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. Unifiedqa: Crossing format boundaries with a single QA system. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pp. 1896–1907. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.171. URL https://doi.org/10.18653/v1/2020.findings-emnlp.171.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. *CoRR*, abs/2211.11501, 2022. doi: 10.48550/arXiv.2211.11501. URL https://doi.org/10.48550/arXiv.2211.11501.

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

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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022. doi: 10.48550/arXiv.2203.07814. URL https://doi.org/10.48550/arXiv.2203.07814.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning. *CoRR*, abs/2308.03281, 2023b. doi: 10.48550/arXiv.2308.03281. URL https://doi.org/10.48550/arXiv.2308.03281.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *CoRR*, abs/2305.01210, 2023. doi: 10.48550/arXiv.2305.01210. URL https://doi.org/10.48550/arXiv.2305.01210.

Aman Madaan, Alex Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits. *ArXiv*, abs/2302.07867, 2023. URL https://api.semanticscholar.org/CorpusID:256868633.

Microsoft. Azure openai service models. https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/models, 2023.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124, 2023. doi: 10.48550/arXiv.2308.07124. URL https://doi.org/10.48550/arXiv.2308.07124.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309, 2023a. doi: 10.48550/arXiv.2305.02309. URL https://doi.org/10.48550/arXiv.2305.02309.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=iaYcJKpY2B_.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/arXiv.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

Long Ouyang, Jeffrey 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 Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon LLM: outperforming curated corpora with web data, and web data only. *CoRR*, abs/2306.01116, 2023. doi: 10.48550/arXiv.2306.01116. URL https://doi.org/10.48550/arXiv.2306.01116.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew J. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL https://arxiv.org/abs/2112.11446.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

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. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/arXiv.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9Vrb9D0WI4.

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. Pangu-coder2: Boosting large language models for code with ranking feedback. *CoRR*, abs/2307.14936, 2023. doi: 10.48550/arXiv.2307.14936. URL https://doi.org/10.48550/arXiv.2307.14936.

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

Yi Tay, Mostafa Dehghani, Vinh Q. Tran, Xavier Garcia, Dara Bahri, Tal Schuster, Huaixiu Steven Zheng, Neil Houlsby, and Donald Metzler. Unifying language learning paradigms. *CoRR*, abs/2205.05131, 2022. doi: 10.48550/arXiv.2205.05131. URL https://doi.org/10.48550/arXiv.2205.05131.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed H. Chi, and Quoc Le. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022. URL https://arxiv.org/abs/2201.08239.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023a. doi: 10.48550/arXiv.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023b. doi: 10.48550/arXiv.2307.09288. URL https://doi.org/10.48550/arXiv.2307.09288.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

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

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 8696–8708. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922, 2023. doi: 10.48550/arXiv.2305.07922. URL https://doi.org/10.48550/arXiv.2305.07922.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=gEZrGCozdqR.

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

Hanwei Xu, Yujun Chen, Yulun Du, Nan Shao, Yanggang Wang, Haiyu Li, and Zhilin Yang. Zeroprompt: Scaling prompt-based pretraining to 1, 000 tasks improves zero-shot generalization. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pp. 4235–4252. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.findings-emnlp.312.

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1482–1493, 2022. URL https://api.semanticscholar.org/CorpusID:246210250.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. GLM-130B: an open bilingual pre-trained model. *CoRR*, abs/2210.02414, 2022. doi: 10.48550/arXiv.2210.02414. URL https://doi.org/10.48550/arXiv.2210.02414.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022. doi: 10.48550/arXiv.2205.01068. URL https://doi.org/10.48550/arXiv.2205.01068.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568, 2023. doi: 10.48550/arXiv.2303.17568. URL https://doi.org/10.48550/arXiv.2303.17568.

A PROMPT FORMATS

In this section, we include the prompt for evaluation on different tasks.

Zero-Shot Prompt for Evaluation on HumanEval and HumanEval+

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:
Create a Python script for this problem:
{Question}

Response:

Three-Shot Prompt for Evaluation on MBPP

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:
Create a Python script for this problem:
{Question}
{Test Example 1}
{Test Example 2}
{Test Example 3}

Response:

Zero-Shot Prompt for Evaluation on DS-1000 (Completion)

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:
{Question}
Complete the Python code in "...".

Response:

In the case of DS-1000 (Insertion), adherence to the benchmark's specifications necessitates the utilization of StarCoder's specialized insertion symbol. Consequently, we have found it imperative to align with the same prompt format employed by StarCoder for this particular benchmark.

For the MultiPL-E benchmark, we recognized the need to align with the evaluation codes provided by bigcode-evaluation-harness.[7] Consequently, we opted to adopt the same prompt format utilized by StarCoder.

B BASELINES DETAILS

We include a large amount of models as our baselines. For GPT3.5 (ChatGPT)&GPT4. their results are obtained from GPT4's report and EvalPlus. The results of Code-Davinci-002, Code-Cushman-001, Codex, PaLM, PaLM 2, LaMDA, AlpahaCode, Incoder, StarCoder, LLaMa, CodeGen, CodeGeeX, CodeT5+, and InstructCodeT5+ are from StarCoder or CodeT5+'s paper. The results of Bard are evaluated with Google's API. The results of Claude are evaluated with Anthropic's API. The results

[7] https://github.com/bigcode-project/bigcode-evaluation-harness

of Instruct-Codegen-16B, Guanaco-65B, Falcon-40B-Instruct, and Vicuna-13B are evaluated with the open-sourced checkpoints. The results of CodeLlama-Series are from CodeLlama's paper. The results of OctoCoder are from its paper. The results of PanguCoder2 are also from its paper.

The MBPP score of StarCoder differs from that in its original paper. Through a personal contact, we were informed that StarCoder was evaluated using a cleaned and smaller version of MBPP, comprising only 397 problems, significantly fewer than the original MBPP benchmarks (500). Consequently, we conducted a re-evaluation of StarCoder using the original MBPP.

C Similarity Checking and Data Filtering

The prompt formats to compute the similarity score are as follow:

System Prompt for Similarity Checking

Your task is to evaluate the similarity of the two given coding tasks. Please review the two coding tasks carefully, paying close attention to the overlap in function names, code structures, topics, and contents. Once you have carefully reviewed both coding tasks, provide a similarity score between these two coding tasks. The score should range from 1 to 10 (1: completely different coding tasks; 10: identical coding tasks). You only need to provide your score. The response format is:
Score: '...'

User Input for Similarity Checking

Task1
{task1}

Task2
{task2}

To thoroughly prevent data leakage from the test datasets to the training dataset, we implemented an additional data filtering step. Utilizing the SOTA embeddings model, gte-large, we treated all test samples as queries to extract the top 5 samples from the training data. Following this, GPT-4 was employed to evaluate the similarity between the retrieved samples and the test sample. The task for GPT-4 is simplified to a binary decision—either a "yes" or "no" indicating a match. In case of a positive match, the sample is excluded from the training data.

D Evol Examples

In this section, we present some evolved examples to elucidate the influence exerted by our *Code Evol-Instruct*.

Example 1:

- Round 0: Write a MongoDB query to select all documents in a collection where the field 'category' is 'clothes'.
- Round 1: Write a MongoDB query to select all documents in a collection where the field 'category' is 'clothes' and the 'brand' field is not equal to 'Nike'.
- Round 2: Write a MongoDB query to select all documents in a collection where the field 'category' is 'clothes' and the 'brand' field is not equal to 'Nike', and the 'price' field is greater than or equal to 100 and less than or equal to 500.
- Round 3: Write a MongoDB query to select all documents in a collection where the field 'category' is 'clothes' and the 'brand' field is not equal to 'Nike', and the 'price' field is greater than or equal to 100 and less than or equal to 500, and the 'color' field is either 'red' or 'blue'. Additionally, sort the documents in descending order by the 'date_added' field and limit the result to the first 10 documents.

Example 2:

- Round 0: Using Tensorflow, set up a model to predict the classification of a handwritten digit.
- Round 1: Using Tensorflow, set up a model to predict the classification of a handwritten digit with the constraint that the model must be trained using only 1000 samples from the MNIST dataset.
- Round 2: Using TensorFlow, set up a model to predict the classification of a handwritten digit with the constraint that the model must be trained using only 500 samples from the MNIST dataset. Additionally, implement transfer learning to incorporate the features learned from a pre-trained model.
- Round 3: Using TensorFlow and the MNIST dataset, develop a convolutional neural network (CNN) to accurately classify handwritten digits. However, in this new version of the task, you must limit your training to a maximum of 500 images. Furthermore, incorporate transfer learning by using a pre-trained model to extract features from the data and improve your model's accuracy.

Example 3:

- Round 0: Create a class in Java that can store a list of students and the courses they are enrolled in.
- Round 1: Create a Java program that manages a student database with courses, including options to add, remove, and search for students, as well as courses for each student. The program should also be able to display statistics on the number of students and courses.
- Round 2: Develop a Java program that manages a student database containing courses and grades. Your program should allow users to add, remove, and search for students, as well as add or drop courses for each student. Furthermore, it should display statistics on the number of students and courses, and calculate the average grade for each student. Ensure that your program implements appropriate error-handling mechanisms.
- Round 3: Develop a Java program to manage a library system where users can check-in, check-out and search for books in the library. The system should allow users to add new books and delete existing ones from the library. The program should also provide statistics on the number of books checked-in and checked-out, and calculate the average reading time for each book. Ensure that your program implements appropriate error-handling mechanisms and enforces data integrity constraints such as not allowing users to check-out books that have already been checked-out.

E USING DIFFERENT EVOLUTION EXECUTION MODELS

We explored using the open-source models (OSS) CodeLlama-Instruct-34B for generating evolved instructions. However, it demonstrated relatively low coding performance in response generation. To address this, we fine-tuned it using our code-alpaca dataset and utilized this model for response generation.

F INTERACTION EXAMPLES

Table 6 7 8 showcases examples of interactions with our *WizardCoder*. The examples demonstrate that our model consistently generates accurate responses accompanied by clear explanations.

Table 6: Examples of interaction with our *WizardCoder*.

<table>
<tr><th>Instruction</th><th>Response</th></tr>
<tr><td>Write a Python function to tell me what the date is today.</td><td>Here's a Python function that tells you what the date is today:
<pre>import datetime

def get_date():
    now = datetime.datetime.now()
    return now.strftime("%A, %B %d, %Y")

print(get_date())</pre>This function uses the datetime module to get the current date and time. It then uses the strftime() method to format the date as a string in the format "Day, Month Date, Year". Finally, it returns the formatted string.

You can call this function from your Python code to get the current date and time.</td></tr>
<tr><td>I do not know how to integrate a Python function numerically. Please write a code for me.</td><td>Sure, here's an example code for integrating a Python function numerically using the trapezoidal rule:
<pre>def integrate(f, a, b, n):
    h = (b - a) / n
    s = (f(a) + f(b)) / 2
    for i in range(1, n):
        s += f(a + i * h)
    return h * s</pre>This code takes in a function f, the lower bound a, the upper bound b, and the number of subintervals n. It then calculates the width of each subinterval h, and uses the trapezoidal rule to approximate the integral of f between a and b. The result is returned as a float.
To use this function, you would call it with the function you want to integrate, the lower bound, the upper bound, and the number of subintervals you want to use. For example:
<pre>def f(x):
    return x**2

result = integrate(f, 0, 1, 1000)
print(result)</pre>This would integrate the function f(x) = x**2 between 0 and 1 using 1000 subintervals, and print the result.</td></tr>
</table>

Table 7: Examples of interaction with our *WizardCoder*.

<table>
<tr><th>Instruction</th><th>Response</th></tr>
<tr><td>List the prime numbers between 20 and 30 with Java.</td><td>
Here's the Java code to list the prime numbers between 20 and 30:
<pre>
public class PrimeNumbers {
    public static void main(String[] args) {
        for (int i = 20; i <= 30; i++) {
            boolean isPrime = true;
            for (int j = 2; j < i; j++) {
                if (i % j == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime) {
                System.out.print(i + " ");
            }
        }
    }
}
</pre>
Output:
<pre>
23 29
</pre>
Explanation:
- We use a nested for loop to iterate through all the numbers between 20 and 30.
- For each number, we use another for loop to check if it is prime or not.
- If a number is divisible by any number other than 1 and itself, then it is not prime.
- If a number is not divisible by any number other than 1 and itself, then it is prime.
- We print the prime numbers to the console.
</td></tr>
</table>

Table 8: Examples of interaction with our *WizardCoder*.

<table>
<tr><th>Instruction</th><th>Response</th></tr>
<tr><td>Write a simple neural network with pytorch</td><td>Here is a simple neural network with pytorch:

```python
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 4)
        self.fc2 = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.sigmoid(x)
        x = self.fc2(x)
        return x

net = Net()
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

# Training
for epoch in range(1000):
    inputs = torch.randn(10, 2)
    targets = torch.randn(10, 1)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()

# Testing
inputs = torch.randn(10, 2)
outputs = net(inputs)
print(outputs)
```

In this example, we define a neural network with two fully connected layers and a sigmoid activation function. We use the mean squared error loss function and the stochastic gradient descent optimizer. We train the network for 1000 epochs and then test it with some randomly generated inputs. The output of the network is a tensor of predicted values.</td></tr>
</table>