WizardCoder: Empowering Code Large Language Models with Evol-Instruct

Ziyang Luo[2*]   **Can Xu**[1*]   **Pu Zhao**[1]   **Qingfeng Sun**[1]   **Xiubo Geng**[1]
Wenxiang Hu[1]   **Chongyang Tao**[1]   **Jing Ma**[2]   **Qingwei Lin**[1]   **Daxin Jiang**[1†]

[1]Microsoft
[2]Hong Kong Baptist University
{caxu,puzhao,qins,xigeng,wenxh,chongyang.tao,qlin,djiang}@microsoft.com
{cszyluo, majing}@comp.hkbu.edu.hk

Abstract

Code Large Language Models (Code LLMs), such as StarCoder, have demonstrated exceptional performance in code-related tasks. However, most existing models are solely pre-trained on extensive raw code data without instruction fine-tuning. In this paper, we introduce *WizardCoder*, which empowers Code LLMs with complex instruction fine-tuning, by adapting the *Evol-Instruct* method to the domain of code. Through comprehensive experiments on four prominent code generation benchmarks, namely HumanEval, HumanEval+, MBPP, and DS-1000, we unveil the exceptional capabilities of our model. It surpasses all other open-source Code LLMs by a substantial margin. Moreover, our model even outperforms the largest closed LLMs, Anthropic's Claude and Google's Bard, on HumanEval and HumanEval+. Our code, model weights, and data are public at https://github.com/nlpxucan/WizardLM.

1   Introduction

Recently, Large Language Models (LLMs) [1–9] have garnered significant attention and demonstrated impressive success. Notably, OpenAI's ChatGPT stands out as a prominent example. Leveraging extensive pre-training on vast amounts of internet data and further fine-tuning with detailed instruction data [10], these models have achieved state-of-the-art (SOTA) zero-shot performance across diverse tasks. This trend is also observed in the domain of code understanding and generation. Numerous Code LLMs [11–18] have been proposed to tackle the challenges associated with code-related tasks. These Code LLMs undergo pre-training using substantial amounts of code data, enabling them to excel in various code-related tasks, showcasing impressive performance.

In contrast to most previous Code LLMs that primarily emphasize the pre-training process, there has been limited exploration of fine-grained instruction tuning in the Code domain. The introduction of instruction tuning initially aimed to enhance the generalization capabilities of LMs across different tasks [19–25]. OpenAI's InstructGPT [10], for instance, involved soliciting human annotators to provide explicit instructions to ensure alignment with users' intentions. Similarly, recent works such as Alpaca [26] employed the self-instruct [27] method, where ChatGPT generated the instruction data. Vicuna [28] utilized user-shared conversations collected from ShareGPT.com. WizardLM [29] introduced the *Evol-Instruct* method, which involved evolving existing instruction data to generate more complex and diverse datasets. However, it is worth noting that all these approaches primarily focused on the general domain and lacked specific design considerations for the code domain.

---

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

Preprint. Under review.

Motivated by the *Evol-Instruct* method, this study aims to enhance the capabilities of the SOTA open-source Code LLM, StarCoder [11], by generating intricate code instruction data through code-specific *Evol-Instruct*. To achieve this, we have made several adaptations to the evolutionary prompt process tailored specifically for code-related tasks. These modifications include refining the evolutionary instructions, simplifying the form of evolutionary prompts, and incorporating code debugging and time-space complexity constraints. Initially, our method is applied to evolve the basic code instruction data, Code Alpaca [30]. Subsequently, we conduct fine-tuning of StarCoder using our newly created code instruction-following training set and obtain our *WizardCoder*.

The experimental results obtained from four code generation benchmarks, namely HumanEval [31], HumanEval+ [32], MBPP [33], and DS-100 [34], demonstrate that our *WizardCoder* outperforms all other open-source Code LLMs, achieving state-of-the-art (SOTA) performance. Specifically, we observe a substantial improvement in pass@1 scores, with an increase of +22.3 (57.3 vs. 35.0) in HumanEval and +8.2 (51.8 vs. 43.6) in MBPP. Remarkably, despite its much smaller size, our *WizardCoder* even surpasses Anthropic's Claude and Google's Bard in terms of pass rates on HumanEval and HumanEval+.

The contributions of this work can be summarized as follows:

- We introduce *WizardCoder*, which enhances the performance of the open-source Code LLM, StarCoder, through the application of Code *Evol-Instruct*.
- *WizardCoder* surpasses all other open-source Code LLMs by a substantial margin in terms of code generation, including StarCoder, CodeGen, CodeGee, CodeT5+, InstructCodeT5+, StarCoder-GPTeacher, and Instruct-Codegen-16B.
- *WizardCoder* achieves superior results in code generation compared to the largest closed-source LLMs, such as Claude, Bard, PaLM, PaLM-2, and LaMDA, despite being considerably smaller in size.

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 [1, 2], Google's PaLM [3, 4], and Bard[3], DeepMind's Chinchilla [5], and Gopher [6], as well as Anthropic's Claude[4]. 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 [35] and GPT-J-6B [36]. Google has released UL2-20B [37]. Tsinghua University has introduced GLM-130B [7]. Meta has released OPT [9] and LLaMA [8]. 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 [16] and Code-Davinci [38]. Google has proposed PaLM-Coder [3]. They perform outstandingly on the popular code completion benchmarks, like HumanEval [31] and MBPP [33]. However, these models are closed-source.

On the other hand, there are several open-source Code LLMs available. Salesforce has introduced CodeGen [13], CodeT5 [17], and CodeT5+ [18]. Tsinghua University has contributed CodeGeeX [14], and the BigCode Project has developed StarCoder [11]. 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 without instruction fine-tuning, our work demonstrates that further training Code LLMs with 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 [19] was among the first models to explore this approach, training on a multitude of supervised text-to-text tasks. Subsequent works such as FLAN [20], ExT5 [22], T0 [23], and UnifiedQA [25] further expanded the range of tasks to bolster the overall generalization ability of LMs. Notably, ZeroPrompt [24] and FLAN-T5 [21] 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.

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 [1] model to create InstructGPT [10], which better aligns with users' inputs. This line of development has even led to the impressive work known as ChatGPT. However, it is important to note that the dataset and model weights associated with these advancements are not publicly available. Alpaca [26] takes a different route by adopting the self-instruct method [27], leveraging ChatGPT to generate data for training. Vicuna [28] utilizes user-shared conversations collected from ShareGPT.com to train its models. WizardLM [29] introduces the *Evol-Instruct* method, which involves evolving existing instruction data to generate more complex and diverse datasets. In contrast to these general instruction fine-tuning approaches, our *WizardCoder* successfully applies the *Evol-Instruct* method specifically in the domain of Code LLMs.

3 Approach

In this section, we elaborate on the methodological details of *WizardCoder*. Following WizardLM, we apply the *Evol-Instruct* method to evolve Code Alpaca generated using self-instruct and fine-tune the pre-trained Code LLM StarCoder with the evolved data.

3.1 Evol-Instruct Prompts for Code

Inspired by the Evol-Instruct [29] method proposed by WizardLM, this work also attempts to make code instructions more complex to enhance the fine-tuning effectiveness of code pre-trained large models. To adapt Evol-Instruct to the realm of code, we made the following modifications to the evolutionary prompt:

1. Streamlined the evolutionary instructions by removing deepening, complicating input, and In-Breadth Evolving.

2. Simplified the form of evolutionary prompts by unifying the evolutionary prompt template.

3. Addressing the specific characteristics of the code domain, we added two evolutionary instructions: code debugging and code time-space complexity constraints.

The unified code evolutionary prompt template is as follows:

```
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:

```
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 [11] as the foundation and proceed to fine-tune it using the code instruction-following training set, which was evolved through *Evol-Instruct*. The prompt format for fine-tuning is outlined as follows:

```
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 20K instruction-following dataset called Code Alpaca[5]. We iteratively employ the Evol-Instruct technique on this dataset consisting of 20,000 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 StarCoder and assess the pass@1 metric on HumanEval [31]. Once we observe a decline in the pass@1 metric, we will discontinue the usage of Evol-Instruct and choose the model with the highest pass@1 as the ultimate model.

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 four code generation benchmarks: HumanEval [31], HumanEval+ [32], MBPP [33], and DS-1000 [34].

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&4 [2], Code-Davinci-002 [38], Code-Cushman-001 [38], and Codex [16]; (ii) Google's Bard, PaLM 2 [4], PaLM [3], and LaMDA [40]; (iii) Google DeepMind's AlphaCode [12]; and (iv) Anthropic's Claude.

Open-Source Models. Several open-source LLMs 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: StarCoder [11], LLaMa [8], CodeGen [13],

---
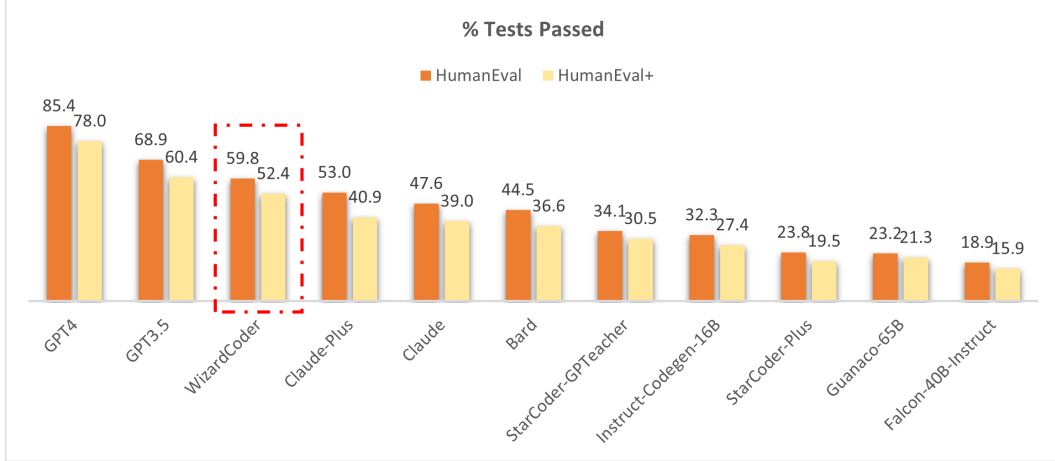[5] https://github.com/sahil280114/codealpaca

Figure 1: The percentage of pass rates on the HumanEval (164 problems) with a single attempt. All baseline scores are retrieved from the LLM-Humaneval-Benchmarks [39]. Our *WizardCoder* generates an answer with greedy decoding.

CodeGeeX [14], CodeT5+[18], and InCoder[15]. In addition, we also include several models with instructions fine-tuning, including StarCoder-GPTeacher,[6] Instruct-Codegen-16B,[7] Guanaco-65B,[8] and Falcon-40B-Instruct.[9]

4.2 Implementation Details

The StarCoder [11] serves as our basic foundation model. 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 [31], HumanEval+ [32] and MBPP [33] are extensively utilized benchmarks within the field of Code LLMs. These benchmarks encompass a vast collection of Python programming problems, employing test cases to validate the code generated by Code LLMs. HumanEval consists of 164 original programming problems, with an average of 9.6 test cases allocated to each problem. To ensure a thorough assessment of the functional correctness of LLM-synthesized code, HumanEval+ extends the number of test cases significantly, averaging at 774.8 test cases per problem. On the other hand, MBPP offers a set of 500 test programming problems, accompanied by three automated test cases per problem. The prompt format for these tasks is as follows:

```
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:
```

Table 1: Results of pass@1(%) on HumanEval and MBPP. Most scores are retrieved from the papers of StarCoder [11] and CodeT5+ [18]. We follow the previous works [31] to generate n samples to estimate the pass@1 score with the same set of hyper-parameters: temperate=0.2, and top_p=0.95. *: we evaluate this model by ourselves.

Model	Params	HumanEval	MBPP
Closed-source models			
LaMDA [40]	137B	14.0	-
AlphaCode [12]	1.1B	17.1	-
PaLM [3]	540B	26.2	36.8
PaLM-Coder [3]	540B	36.0	47.0
PaLM 2-S [4]	-	37.6	50.0
Codex [16]	2.5B	21.4	-
Codex [16]	12B	28.8	-
Code-Cushman-001 [38]	-	33.5	45.9
Code-Davinci-002 [38]	-	47.0	58.1
GPT-3.5 [2]	-	48.1	-
GPT-4 [2]	-	67.0	-
Open-source models			
LLaMa [8]	33B	21.7	30.2
LLaMa [8]	65B	23.7	37.7
CodeGen-Multi [13]	16B	18.3	20.9
CodeGen-Mono [13]	16B	29.3	35.3
CodeGeeX [14]	13B	22.9	24.4
StarCoder [11]	15B	33.6	43.6*
CodeT5+ [18]	16B	30.9	-
InstructCodeT5+ [18]	16B	35.0	-
WizardCoder	15B	**57.3** (+22.3)	**51.8** (+8.2)

Comparing with the Closed-Source Models. The SOTA LLMs for code generation, such as GPT4, Claude, and Bard, are predominantly closed-source. Acquiring access to the APIs of these models proves challenging. In this study, we adopt an alternative approach by retrieving the scores for HumanEval and HumanEval+ from the LLM-Humaneval-Benchmarks [39]. Notably, all the mentioned models generate code 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* using the provided evaluation codes. By adhering to these standardized procedures, we aim to ensure fair and comparable evaluations of our model against existing benchmarks.

As depicted in Figure 1, our *WizardCoder* attains the third position in this benchmark, surpassing Claude-Plus (59.8 vs. 53.0) and Bard (59.8 vs. 44.5). Notably, our model exhibits a substantially smaller size compared to these models. Furthermore, our *WizardCoder* demonstrates a remarkable superiority over other open-source LLMs that undergo instruction fine-tuning, showcasing a significant performance margin.

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 1, we adhere to the approach outlined in previous studies [31] 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.

From the experimental results in Figure 1 and Table 1, we have the following conclusions:

1. *WizardCoder* outperforms the largest closed-source LLMs, including Claude, Bard, PaLM, PaLM-2, and LaMDA, despite being significantly smaller.

6

Table 2: Performance of *WizardCoder* 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**
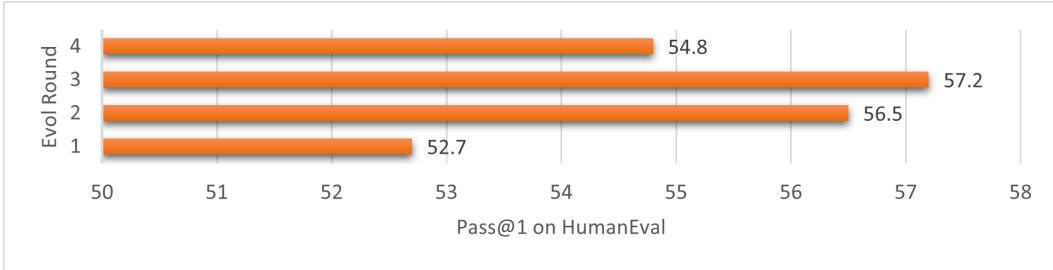
Figure 2: Ablation study on the number of data evolution rounds.

2. *WizardCoder* outperforms all the open-source Code LLMs by a large margin (+22.3 on HumanEval), including StarCoder, CodeGen, CodeGee, and CodeT5+.

3. *WizardCoder* significantly outperforms all the open-source Code LLMs with instructions fine-tuning, including InstructCodeT5+, StarCoder-GPTeacher, and Instruct-Codegen-16B.

4.4 Evaluation on DS-1000

The DS-1000 benchmark [34] comprises 1,000 distinct data science workflows spanning seven 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. The DS-1000 benchmark further classifies problems based on the libraries employed, including Matplotlib (plt), NumPy (np), Pandas (pd), SciPy (scp), Scikit-Learn (sk), PyTorch (py), and TensorFlow (tf). We follow the same prompt format as StarCoder. In Table 2, we present pass@1 (n=40) results for each library, along with an overall score. 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. This observation holds true across nearly all data science libraries.

4.5 Ablation Study

Figure 2 presents an ablation study investigating the impact of the number of data evolution rounds. The first round of evolved data contains 38k samples. The second round contains 58k. The third round contains 78k. The fourth round contains 98k. For consistency, all models undergo fine-tuning with 200 steps. The results reveal that the highest pass@1 score on humaneval is achieved after three rounds of data evolution. Based on this observation, we select the data that evolved during the third round as the ultimate dataset.

4.6 Examples

Table 3 showcases examples of interactions with our *WizardCoder*. The examples demonstrate that our model consistently generates accurate responses accompanied by clear explanations.

5 Conclusion and Future Work

This paper introduces *WizardCoder*, a Code *Evol-Instruct* fine-tuned Code LLM. The experimental results demonstrate that *WizardCoder* achieves SOTA performance surpassing all existing open-source Code LLMs on four widely recognized code generation benchmarks: HumanEval, HumanEval+, MBPP, and DS-1000. Furthermore, *WizardCoder* exhibits superior performance compared to the largest closed LLMs, including Anthropic's Claude and Google's Bard.

Future Work.   Although our *WizardCoder* demonstrates impressive coding performance, as depicted in Figure 1, our model still falls significantly behind the SOTA LLM, GPT4. Therefore, future work will prioritize the enhancement of the Code *Evol-Instruct* method to further augment the performance of our model.

Broader Impact.   Similar to the other LLMs, our *WizardCoder* could also generate unethical, harmful, or misleading information. Therefore, future research to address the ethical and societal implications is needed.

Table 3: Examples of interaction with our *WizardCoder*.

Instruction	Response

Write a Python function to tell me what the date is today.	Here's a Python function that tells you what the date is today:
	
	``python
	import datetime
	
	def get_date():
	    now = datetime.datetime.now()
	    return now.strftime("%A, %B %d, %Y")
	
	print(get_date())
	``
	
	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.

I do not know how to integrate a Python function numerically. Please write a code for me.	Sure, here's an example code for integrating a Python function numerically using the trapezoidal rule:
	
	``python
	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
	``
	
	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:
	
	``python
	def f(x):
	    return x**2
	
	result = integrate(f, 0, 1, 1000)
	print(result)
	``
	
	This would integrate the function f(x) = x**2 between 0 and 1 using 1000 subintervals, and print the result.

List the prime numbers between 20 and 30 with Java.	Here's the Java code to list the prime numbers between 20 and 30:
	
	``java
	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 + " ");
	            }
	        }
	    }
	}
	``
	
	Output:
	
	``
	23 29
	``
	
	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.

References

[1] 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, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[2] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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, 2023.

[9] 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.

[10] 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.

[11] 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*, 2023.

[12] 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.

[13] 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*, 2023.

[14] 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.

[15] 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.

[16] 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, 2021.

[17] 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, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 4235–4252. Association for Computational Linguistics, 2022.

[25] 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, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1896–1907. Association for Computational Linguistics, 2020.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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, 2021.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] Llm humaneval benchmarks. `https://github.com/my-other-github-account/llm-humaneval-benchmarks`, 2023.

[40] 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.