# InstructCoder: Instruction Tuning Large Language Models for Code Editing

**Kaixin Li**[1*] **Qisheng Hu**[1*] **Xu Zhao**[1] **Hui Chen**[2] **Yuxi Xie**[1] **Tiedong Liu**[1]
**Qizhe Xie**[1†] **Junxian He**[3†]
[1]National University of Singapore [2]Singapore University of Technology and Design
[3]Shanghai Jiao Tong University
{likaixin,qishenghu,xu.zhao,xieyuxi,tiedong.liu}@u.nus.edu,
hui_chen@mymail.sutd.edu.sg,
junxianh@sjtu.edu.cn

## Abstract

Code editing encompasses a variety of pragmatic tasks that developers deal with daily. Despite its relevance and practical usefulness, automatic code editing remains an underexplored area in the evolution of deep learning models, partly due to data scarcity. In this work, we explore the use of Large Language Models (LLMs) to edit code based on user instructions. Evaluated on a novel human-written execution-based benchmark dubbed **EditEval**, we found current models often struggle to fulfill the instructions. In light of this, we contribute **InstructCoder**, the first instruction-tuning dataset designed to adapt LLMs for general-purpose code editing, containing high-diversity code-editing tasks such as comment insertion, code optimization, and code refactoring. It consists of over 114,000 instruction-input-output triplets and covers multiple distinct code editing scenarios. The collection process starts with filtered commit data sourced from GitHub Python repositories as seeds. Subsequently, the dataset is systematically expanded through an iterative process, where both seed and generated tasks are used to prompt ChatGPT for more data. Our findings reveal that open-source LLMs fine-tuned on InstructCoder can significantly enhance the accuracy of code edits, exhibiting superior code-editing performance matching advanced proprietary LLMs.

The dataset and the source code are available at `https://github.com/qishenghu/CodeInstruct`.

## 1 Introduction

Developers typically engage in a cyclic routine of writing and revising code. As a crucial element, code editing takes up a great portion of this process, encapsulating diverse sub-tasks such as code optimization, refactoring, and bug fixing, each posing distinct challenges. Automated code editing tools could substantially boost developer productivity by alleviating the burden of monotonous tasks. However, it remains an under-explored area, partly due to the lack of relevant data, hampering substantial progress by deep learning models.

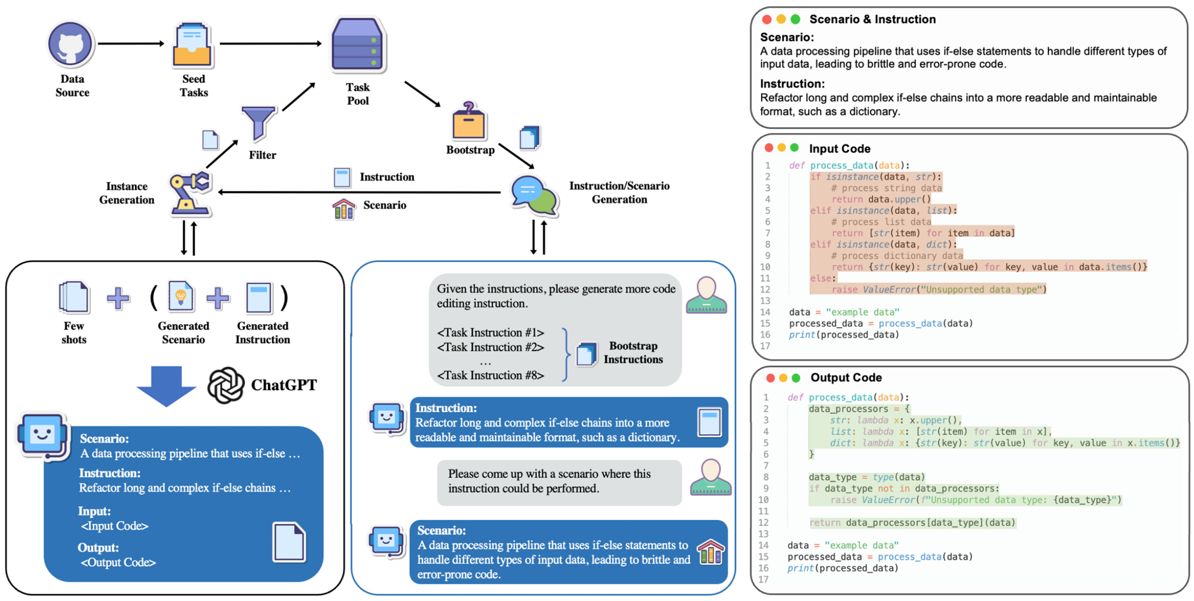
Inspired by the recent advancements in LLMs (Brown et al., 2020; Chowdhery et al., 2022; Ouyang et al., 2022; OpenAI, 2022; Touvron et al., 2023a; OpenAI, 2023) and Code LLMs (Nijkamp et al., 2023a; Chen et al., 2021a; Li et al., 2023a), we explore the proficiency of LLMs in code editing tasks based on user instructions, for instance, "add a docstring to the function for clarity", "remove redundant code", or "refactor it into reusable functions". These tasks are distinctly different from code completion, which involves generating code to complete given code snippets or comments. Code editing requires the model to not only understand the existing code but also execute modifications that are in line with the given instructions, while seamlessly integrating with the context. For example, removing redundant code or refactoring a function should not affect the return value.

To systematically evaluate LLMs for code editing, we created a novel benchmark named **EditEval**. It contains various types of code edits adapted from Github commits and existing datasets. Intriguingly, we found that open-source models yield unsatisfactory results, and even the most advanced proprietary LLMs struggle to solve these tasks.

In addressing this challenge, we present InstructCoder, a diverse dataset for instruction finetuning, particularly designed to improve the code editing

[*] Equal contribution. Ordering is determined by dice rolling.

[†] Equal advising. Ordering is determined by dice rolling.



Given the instructions, please generate more code editing instruction.

<Task Instruction #1>
<Task Instruction #2>
…
<Task Instruction #8>

**Instruction:** Refactor long and complex if-else chains into a more readable and maintainable format, such as a dictionary.

Please come up with a scenario where this instruction could be performed.

**Scenario:** A data processing pipeline that uses if-else statements to handle different types of input data, leading to brittle and error-prone code.

**Scenario:** A data processing pipeline that uses if-else …
**Instruction:** Refactor long and complex if-else chains …
**Input:** <Input Code>
**Output:** <Output Code>

**Scenario:**
A data processing pipeline that uses if-else statements to handle different types of input data, leading to brittle and error-prone code.

**Instruction:**
Refactor long and complex if-else chains into a more readable and maintainable format, such as a dictionary.

Input Code

```python
def process_data(data):
    if isinstance(data, str):
        # process string data
        return data.upper()
    elif isinstance(data, list):
        # process list data
        return [str(item) for item in data]
    elif isinstance(data, dict):
        # process dictionary data
        return {str(key): str(value) for key, value in data.items()}
    else:
        raise ValueError("Unsupported data type")

data = "example data"
processed_data = process_data(data)
print(processed_data)
```

Output Code

```python
def process_data(data):
    data_processors = {
        str: lambda x: x.upper(),
        list: lambda x: [str(item) for item in x],
        dict: lambda x: {str(key): str(value) for key, value in x.items()}
    }

    data_type = type(data)
    if data_type not in data_processors:
        raise ValueError(f"Unsupported data type: {data_type}")

    return data_processors[data_type](data)

data = "example data"
processed_data = process_data(data)
print(processed_data)
```

Figure 1: Data collection pipeline of InstructCoder (left) and a qualitative example from the dataset (right, best viewed with zoom). Initial seed tasks are selected from GitHub commits, and inspire ChatGPT to generate new instructions. Plausible scenarios where the filtered instructions may be used are then generated. Finally, corresponding code input and output are obtained conditioned on both the instruction and scenario. High-quality samples are manually selected and recurrently added to the task pool for further generation.

abilities of LLMs. Specifically, we first collect and manually scrutinize commit data from public repositories on GitHub as the seed code editing tasks. Then, we utilize the seed data to prompt ChatGPT (OpenAI, 2022) to generate new instructions and input-output pairs respectively. This process resembles the Self-Instruct (Wang et al., 2022a) and Alpaca (Taori et al., 2023) frameworks. By innovatively forcing scenarios to guide the generation process, our approach ensures that the tasks in InstructCoder are diverse and relevant to real-world programming situations, resulting in a robust dataset for instruction finetuning in the code editing domain. After proper deduplication and postprocessing, we retain over 114,000 samples in the dataset.

Our empirical studies reveal that LLMs display notable gains in code editing abilities after fine-tuning with InstructCoder. Code LLaMA achieves the best results through fine-tuning, attaining an accuracy of 57.22%, closely matching ChatGPT. Further studies also signify that while the pre-training of the models is fundamental, the code editing performance is highly influenced by the quality and volume of the instruction-tuning data.

In summary, the contributions of this work are (1) **InstructCoder**, the first instruction-tuning dataset featuring a wide range of diverse code editing tasks, and demonstrate the effectiveness of instruction-finetuning with InstructCoder; (2) **EditEval**, a novel human-written execution-based benchmark for the rigorous evaluation of general-purpose code editing; (3) We find that open-source models instruction-tuned with InstructCoder can demonstrate strong code editing performance matching ChatGPT.

## 2 Related Work

### 2.1 Instruction Finetuning Datasets

Previous studies have concluded that instruction finetuning LLMs on a diverse collection of instructional tasks can further improve the ability of LLMs to generalize well on unseen tasks (Ouyang et al., 2022; Mishra et al., 2022; Wei et al., 2022; Chung et al., 2022; Wang et al., 2023c). To support these tasks, datasets consisting of a large number of code snippets with corresponding annotations are necessary. These instruction can be reformulated from existing datasets (Aribandi et al., 2022; Wei et al., 2022; Mishra et al., 2022; Longpre et al.,

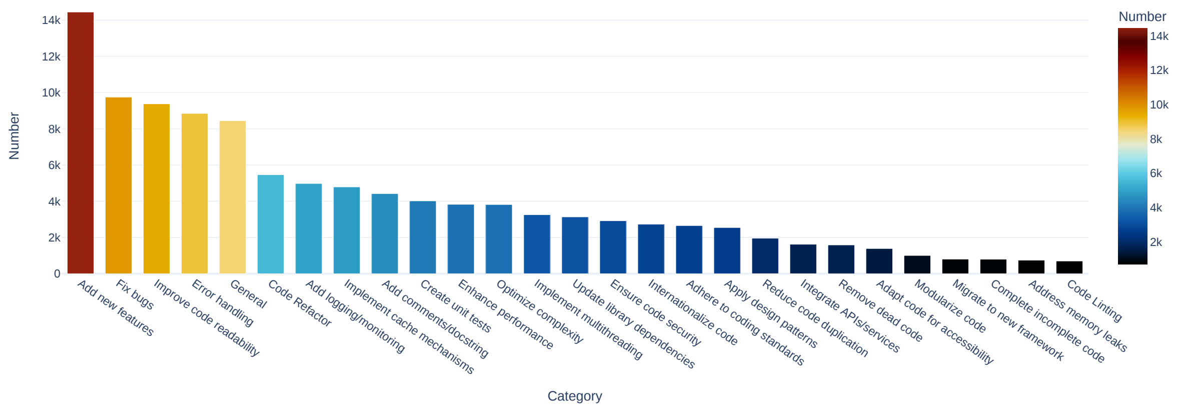
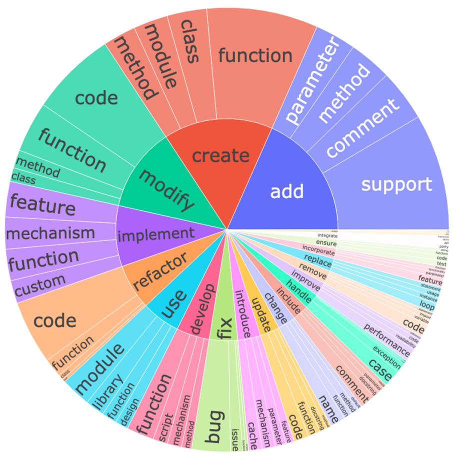
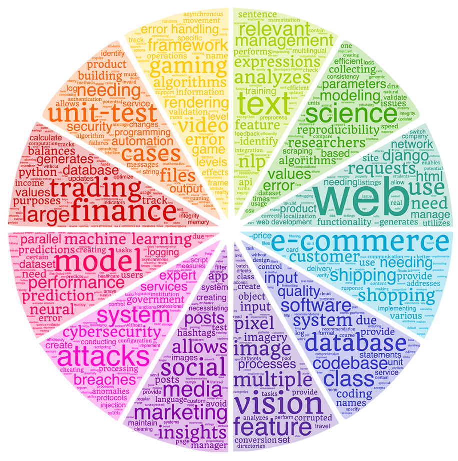


Figure 2: Distribution of code edit intent categories.



(a) The top 20 most common root verbs with each top 4 noun objects in the instructions. Instructions with other infrequent root verbs take up 25%.



(b) Wordcloud of scenario domains. Each sector with a different color corresponds to a different scenario domain. Each domain is a cluster of similar scenarios.

Figure 3: Visualizations of InstructCoder data. Best viewed in zoom.

2023), or human-written with crowd-sourcing efforts (Ouyang et al., 2022; Wang et al., 2022b). Machine generation of instruction data has also been explored to reduce human labour (Wang et al., 2022a; Honovich et al., 2022; Taori et al., 2023; Xue et al., 2023). Despite the presence of elevated noise levels within the data, its effectiveness has been identified.

## 2.2 Code Synthesis

Code generation has been extensively studied (Zhang et al., 2023). Language models pre-trained on large collections of code have demonstrated strong abilities in a variety of programming tasks. Some general LLMs gain code generation abilities due to the mixture of code in the pre-training corpus (e.g. The Pile (Gao et al., 2020)), such as GPT-3 (Brown et al., 2020), ChatGPT, GPT-4 (OpenAI, 2023), LLaMA (Touvron et al., 2023a), BLOOM (Scao et al., 2022), GPT-NeoX (Black et al., 2022), and Pythia (Biderman et al., 2023). LLMs specifically trained on code and optimized for code generation are also studied, e.g. Codex (Chen et al., 2021a), CodeGen (Nijkamp et al., 2023b), CodeGeeX (Zheng et al., 2023) and StarCoder (Li et al., 2023a). These models all adopt the decoder-only transformer architecture but differ in size and specific model design

| Model | | Accuracy (%) | |
|---|---|---|---|
| ChatGPT (gpt-3.5-turbo-0613) | | 57.73 | |
| GPT-4 (gpt-4-0613) | | 68.56 | |
| GPT-4 Turbo (gpt-4-1106-preview) | | 66.49 | |
| | **7B** | **13B** | **33B** |
| Alpaca | 12.37 | 19.59 | 30.93 |
| LLaMA+CodeAlpaca | 18.56 | 18.56 | 35.56 |

Table 1: Results of several instruction-tuned models evaluated on EditEval.

(e.g. positional embedding, norm layer placement) as well as the selection and preprocessing of the pre-training corpus. The study of Code Synthesis has led to exciting applications (Li et al., 2024; Xiao et al., 2024).

On the other hand, relatively little literature addresses the objective of code editing. Previous works focus on a subset of code editing tasks, such as code infilling (Fried et al., 2023) and debugging (Just et al., 2014; Tarlow et al., 2020; Ding et al., 2020; Jimenez et al., 2023). The PIE (Madaan et al., 2023) dataset is a concurrent work most relevant to ours, which focuses on speeding up programs. Other works (Yin et al., 2018; Wei et al., 2023; Chakraborty et al., 2020) can not accept natural language as edit intentions, rendering them less user-friendly.

Nevertheless, datasets particularly tailored for general-purpose code editing are absent. To fill this gap, we introduce InstructCoder, a novel dataset aimed at further advancing the capabilities of code editing with LLMs.

## 3 EditEval: Evaluating Code Editing Models

As aforementioned, code editing is significantly different from code completion. Consequently, widely utilized datasets in the realm of code completion, such as MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021b), fall short in evaluating code editing capabilities. To rigorously evaluate the code editing capabilities, we curated a test set of 194 code editing tasks, derived from three key sources: GitHub commit data, MBPP, and HumanEval. We harness the input code from these sources and create plausible edit instructions. For GitHub sources, we manually create execution contexts so that the code is runnable. Each sample is accompanied by a canonical solution written by humans to ensure the instruction is viable. The generated code edits are strictly assessed using automated test cases to evaluate the correctness of the edits. An edit is considered correct only if it passes all the test cases. This automated method provides a robust and objective evaluation framework, essential for benchmarking the model's performance in diverse code editing situations. Appendix A showcases an example of the test set.

We benchmarked several instruction-tuned models on EditEval, and the results are listed in Table 1. Generally, the results reveal significant potential for improvement in code editing. Alpaca and CodeAlpaca exhibit accuracies below 20% with 7B and 13B sizes, and it only gets better at 33B. At this size, CodeAlpaca beats Alpaca, achieving 35.56% accuracy. Turning to the GPTs, the most advanced proprietary models up to this point, GPT-4 achieves the best performance at 68.56%. Even ChatGPT struggles at this task, scoring only 57.73%. Upon closer examination, we found the challenge of EditEval lies in the high demand for both instruction following and code understanding. The model has to have a grasp of the implicated context of the input code, and then accomplish the edit within its context.

## 4 InstructCoder: Instruction-tuning Empowers Code Editing

In this section, we introduce how we create InstructCoder to boost the code editing abilities of LLMs via instruction finetuning. We employed a method based on Self-Instruct (Wang et al., 2022a), which expands instruction finetuning data by bootstrapping off language model generation. The methodology of generating data with LLMs requires minimal human-labeled data as seed tasks while maintaining the quality and relevance of the tasks in the dataset. Through an iterative process of generating instructions and refining them with deduplication, we create a dataset of a wide range of code-editing tasks. Figure 1 illustrates the data collection pipeline of InstructCoder.

### 4.1 Seed Data Collection

GitHub is a code hosting platform whose version control service naturally records code edits with commits, which can be converted to instructions. The repositories on GitHub provide diverse data with human-generated quality. However, the data is not suitable for direct utilization[1]. First, commit messages are mostly brief and resultant, missing detailed descriptions. Furthermore, they can be imprecise or even absent. Second, commits can be huge involving multiple files, which is beyond the scope of this work. In light of this, we direct our attention towards LLMs as a means to generate data, instead of the direct utilization of collected data.

Raw GitHub commit data were collated using BigQuery[2]. To ensure high quality and address licensing issues, we focused on Python repositories on GitHub with over 100 stars and permissive licenses. Our selection criteria was restricted to commits modifying only one code block within a single Python file. These commits were identified by git-diff[3].

During the collection process, we came across many imprecise or emotionally charged commit messages. Codex (Chen et al., 2021a) was employed in such cases to clarify the changes made between versions and improve the commit messages, resulting in more precise and informative instructions. A total of 634 tasks were processed from the commit data through manual efforts and were used for the self-instruct process.

In addition to GitHub commit data, we also leverage high-quality generated samples as additional seed tasks. With manual inspection, a batch of 592 high-quality samples was compiled as additional seed tasks. This set of seed data covers a wide range of code-editing scenarios and enriches the basis on which InstructCoder is created, ensuring that the tasks are rooted in plausible real-world code-editing cases[4].

### 4.2 Instruction Bootstrapping

Self-Instruct (Wang et al., 2022a) is as an effective automated framework for instruction data generation. It works by iterative bootstrapping off LLM's generation, presenting a way to enrich the instructional dataset while maintaining task quality and relevance from a small set of human-evaluated seed tasks. We leveraged a similar approach to generate diverse code editing instructional data. In each iteration, seven seed task instructions and one ChatGPT-generated task instruction are sampled and combined as a few-shot context to prompt ChatGPT for more instructions. To generate more diverse and practically applicable instructions, we also generated tasks across multiple sub-domains by specifying the editing intent in the prompt provided. Relevant prompts used can be found in Table 4 in Appendix C.

### 4.3 Scenario-conditional Generation

We originally found many generated samples share similar codebases despite different instructions and few-shot examples provided. Such similarity could largely diminish the dataset's value. Empirical analysis suggests the issue could be attributed to LLM generating general codebases for input/output snippets when insufficient context is provided. As a countermeasure, we propose to introduce code editing scenarios for input/output code generation. We present some examples in Figure 9,10,11 in Appendix D, where we generally observe that instances generated with scenario demonstrate higher quality in terms of richer context and code structure compared to those without.

For each generated instruction, we first prompted ChatGPT to generate practical events as "real-world" scenarios where the editing instruction could be performed, and randomly select one for instance generation in the next step. Subsequently, the LLM was instructed to generate samples that correspond with the instruction and scenario, ensuring the codebases and variable names are appropriate. The prompt used can be found in Table 4 in Appendix C.

By incorporating scenario-conditional generation, the resulting samples exhibit increased variability regarding codebases and variable naming, thus augmenting the diversity of InstructCoder.

### 4.4 Postprocessing

Following Self-Instruct (Wang et al., 2022a), deduplication was applied on the generated instructions to remove instructions that have a ROUGE-L (Lin, 2004) overlap score larger than 0.7 with the ex-

[1]Initial attempts to utilize real-world GitHub commit data for model fine-tuning yielded suboptimal results. Please refer to Appendix B for a detailed discussion.

[2]`https://cloud.google.com/bigquery`

[3]`https://git-scm.com/docs/git-diff`

[4]Incorporating additional seeds also allows for modulating the distribution of generated data, facilitating customization for specific requirements.

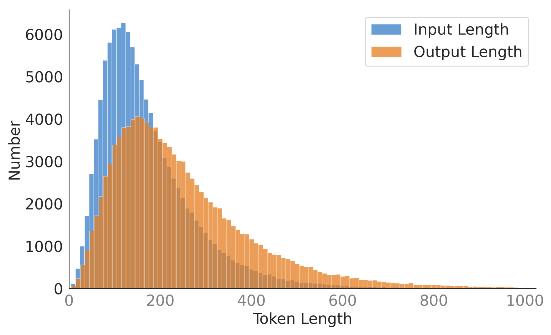


Figure 4: Token length distribution of InstructCoder

isting instructions. For the code, we employed MinHash with Locality Sensitive Hashing (LSH) indexing to remove instances with a Jaccard similarity greater than 0.75. Ultimately, InstructCoder comprises over 114,000 distinct code editing tasks. For experimental purposes, we designated 95% of the tasks for training, while the remaining 5% formed our validation set.

## 5 Data Analysis

We analyze InstructCoder in terms of 1) diversity, 2) complexity, and 3) correctness. We provide distribution and complexity analyses of the task instances. Finally, we demonstrate through human investigation that our data is highly reliable.

### 5.1 Statistic Overview

InstructCoder comprises over 114k code editing instructions, each paired with an input/output instance. The token length distribution of input/output can be viewed in Figure 4 and Table 5 in Appendix E. Most of the data falls within a reasonable range in terms of length, while some extreme values reflect the breadth of our dataset.

### 5.2 Instruction Diversity

To explore the diversity of tasks in InstructCoder and their practical applicability, we present various instruction intents i.e. *what* the code edits intend to accomplish, and instruction verbs, i.e. *how* the code edit is accomplished.

**Instruction Intents.** We asked ChatGPT to classify the types of code edits in our dataset and manually identified 27 empirical genres. Figure 2 shows the distribution of the code edit intent categories in InstructCoder, which include adding functionalities, optimizing code, improving readability, etc. These objectives underscore the extensive range of InstructCoder.

**Instruction Verbs.** The diversity of instruction verbs is also portrayed in Figure 3a. We demonstrate the top 20 root verbs and their top 4 direct nouns both ranked by frequency. While a great portion of the instructions can be roughly clustered as *creation* (e.g. "add", "implement", "creat") and *modification* (e.g. "modify", "replace", "change"), InstructCoder presents a long-tail distribution with less common verbs other than the top-20 taking up 25.0% percentage. This demonstrates that the dataset contains a wide spectrum of instructions.

### 5.3 Scenario Diversity

InstructCoder is designed to cover a wide range of scenarios. As discussed in Section 4.3, each instruction was accompanied by different scenarios where the editing instruction could be performed to improve diversity. A word cloud is provided to show some of the scenario domains in our dataset, as illustrated in Figure 3b, with each sector referring to a different domain. The diversity of the dataset is emphasized by the presence of a wide range of domains such as image processing, web development, and cybersecurity.

### 5.4 Complexity

We reflect the complexity of a code edit task using the number of differing lines and their edit ratio in the input/output pair, which are defined as:

$$n_{diff} = |I \cup O \setminus I \cap O|, \qquad (1)$$

$$r_{diff} = \frac{n_{diff}}{|I \cup O|}, \qquad (2)$$

where $I$ and $O$ are sets of input/output code with single lines as elements.

We measure the differing lines of a code-editing task instance using the Python library *difflib*.[5] We found that the average number of differing lines in InstructCoder is 11.9 and the average edit ratio is 0.52. These values suggest a fairly acceptable level of complexity, indicating that the dataset is neither

[5] https://docs.python.org/3/library/difflib.html

| Question | Pass |
|---|---|
| `Determine if the instruction is valid.` | 97% |
| `Is the output an acceptable edited code response to the instruction and input?` | 90% |

Table 2: Quality check questions and results on a randomly sampled subset with 200 data points.

too easy nor too hard. InstructCoder strikes a balance in terms of complexity, making it well-suited for finetuning and evaluating LLMs in a wide range of code editing tasks. Figure 12 in Appendix E illustrates the distribution of the number of differing lines.

### 5.5 Correctness

We further randomly sampled 200 instances and invite annotators to evaluate the instances based on two criteria: the validity of the instructions and the correctness of the outputs. The validity assessment focused on determining if the instructions exhibit clear and appropriate editing intents. The correctness evaluation examines if the input-output pairs reflect the changes specified by the instructions.

The results in Table 2 indicate that most instructions in the InstructCoder dataset are valid. A few instances exhibited noise and occasional failure to follow the instructions, but high correctness was found overall. Out of the 200 evaluated instances, 180 were successfully solved, showcasing the overall quality and reliability of InstructCoder.

## 6 Experiments

### 6.1 Setup

**Training.** We experiment with two families of open-source language models with various sizes: LLaMA (LLaMA, LLaMA-2 and Code LLaMA) (Touvron et al., 2023a,b; Roziere et al., 2023) and BLOOM (Scao et al., 2022).

LLaMA is a series of LLMs with parameters ranging from 7 to 65 billion. They have been pre-trained on a vast corpus, of which approximately 4.5% comprises code. The LLaMA-2 series extends the family with more intensive pre-training. Additionally, Code LLaMAs are built on LLaMA-2 and specifically trained on 500B tokens of code to enhance its code understanding and generation capabilities. BLOOM is a multilingual LLM capable of generating human-like outputs in 46 languages and 13 programming languages.

A full finetuning updating all the parameters in an LLM can be computationally expensive. Instead, we adopt LoRA (Hu et al., 2022), a parameter-efficient finetuning method that optimizes an approximated low-rank delta matrix of the fully-connected layers. In this way we could fine-tune a 33B model in a single A100-80GB GPU card. In our experiments, LoRA is applied to the query, key, value, and output transform weights of the Transformer architecture (Vaswani et al., 2017). All hyperparameters can be found in Table 6 in Appendix F.

**Baselines.** We select ChatGPT (OpenAI, 2022), GPT-4 (OpenAI, 2023) and GPT-4 Turbo as strong baselines. The aforementioned open-source models along with an instruction-tuned LLaMA model called Alpaca (Taori et al., 2023) are included, and their zero-shot performance is reported.

Concurrent to our work, CodeAlpaca[6] is a popular dataset generated with the pipeline of Alpaca, differing in that its seed data is replaced by handwritten easy instructions with short programs. We fine-tune LLaMA models with CodeAlpaca and Alpaca and compare the results.

| Model | Size | Accuracy (%) w/o ft | Accuracy (%) w/ ft | Δ Acc |
|---|---|---|---|---|
| ChatGPT (gpt-3.5-turbo-0613) | - | **57.73** | - | - |
| BLOOM | 3B | 0.52 | 15.46 | + 14.94 |
| | 7B | 1.03 | 19.59 | + 18.56 |
| LLaMA-1 | 7B | 2.57 | 26.80 | + 24.23 |
| | 13B | 6.19 | 28.35 | + 22.16 |
| | 33B | 6.19 | 41.75 | **+ 35.56** |
| LLaMA-2 | 7B | 4.12 | 27.32 | + 23.20 |
| | 13B | 14.95 | 34.54 | + 19.59 |
| Code LLaMA | 7B | 29.90 | 45.88 | + 15.98 |
| | 13B | 28.86 | **57.22** | + 28.36 |

Table 3: Models finetuned with InstructCoder significantly improve in code edit accuracy on EditEval, regardless of the model family or model size.

## 7 Results

### 7.1 Finetuning Efficacy with InstructCoder

In this section, we demonstrate the value of our InstructCoder dataset. Table 3 presents a detailed comparison of EditEval performance across models fine-tuned with InstructCoder and baseline models. While very low accuracies are observed in

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

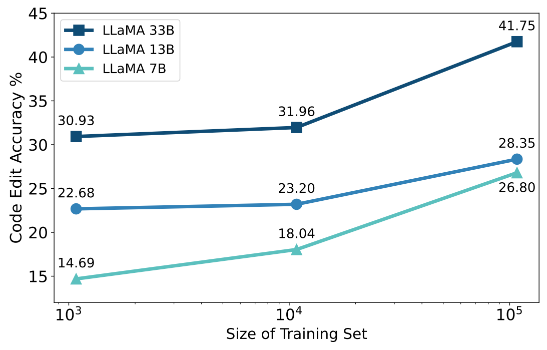


Figure 5: Data scaling performance of InstructCoder on LLaMA evaluated on EditEval, using 1%, 10% and 100% training data.

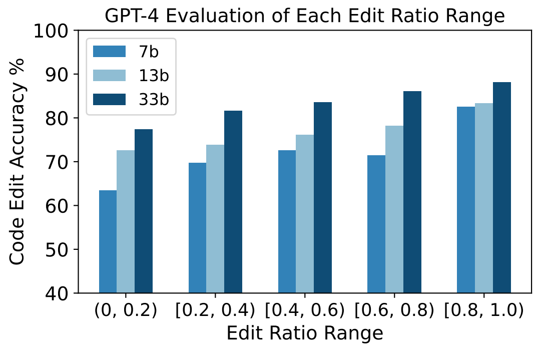


Figure 6: GPT-4 evaluation results at different edit ratios on 2000 validation samples.

open-source plain models, finetuning with InstructCoder significantly boost the accuracy, highlighting the effectiveness of efficient instruction finetuning with machine-generated code edit pairs.

Code LLaMA 13B matches ChatGPT's performance and surpasses other open-source models with a 57.22% accuracy rate. The more substantial LLaMA-33B model shows a notable 35.56% improvement, yet it falls behind Code LLaMA-7B, which benefits from extensive pre-training on code. For qualitative results, see Appendix G.

As expected, the pre-training foundation of LLM significantly influences code-editing efficacy. LLaMA demonstrated higher accuracies than BLOOM models of similar sizes. Among LLaMAs, those pre-trained on more tokens (LLaMA-2 series) outperformed earlier versions. Furthermore, Code LLaMAs exceed LLaMA-2 models as a result of their extensive pre-training specifically on coding data. Despite the varying capabilities of the foundational models, our dataset consistently enhances performance.

### 7.2 Dataset Scaling

InstructCoder has a scale considerably smaller than what LLMs are typically pre-trained on. To ascertain the sufficiency of this scale, we conducted an experiment wherein we fine-tuned the LLaMA models using varying proportions (1%, 10%, and 100%) of the dataset. The smaller subsets are guaranteed to be encompassed within the larger subsets. The results are shown in Figure 5. The identified trend demonstrates a positive correlation between the model's accuracy and the scale of the training set.

Fine-tuned with merely 1% of the data, the models experience a limited number of parameter updates but quickly adapt to the tasks, surpassing their respective zero-shot accuracy scores by significant margins. This underscores the significance of instruction tuning. As the volume of training data increases, we observe consistent improvements in model accuracy, approximately growing linearly with respect to the logarithmic scale of the number of samples. Crucially, our experiment empirically suggests that larger models are more effective with a constrained training compute budget.

### 7.3 Edit Ratio

Figure 6 depicts the accuracy of fine-tuned LLaMA models as evaluated by GPT-4 across five edit ratio levels, using 2000 random samples from the validation set. This evaluation, justified in Appendix H, involves prompting GPT-4 for a quick and general assessment of code edits, offering an alternative perspective to code edit evaluation. In this assessment, larger models consistently outperform their smaller counterparts. Notably, accuracy decreases with lower edit ratios, potentially due to the models adopting the shortcut of copying inputs to minimize loss in scenarios requiring fewer edits. This trend, however, is less pronounced in larger models, which show a greater ability to discern subtle differences in cases of low edit ratios.

## 8 Conclusion

We introduce InstructCoder, the first instruction-tuning dataset for general-purpose code-editing tasks. It comprises generations of LLMs, where real GitHub commits serve as seed tasks to guide the generation process. A scenario-conditional approach is introduced to ensure both diversity and high quality of the data. Our experiments on the novel EditEval benchmark show that open-source models can gain huge improvements and even yield performance matching proprietary models through computationally lightweight parameter-efficient fine-tuning with InstructCoder. We also reveal that the LLM base model and the scale of fine-tuning data are both profound factors of code-editing ability. We hope the dataset can benefit and inspire more research in this area towards building more powerful coding models.

## Limitations

Our approach did not encompass code changes involving multi-file contexts, which might be useful in development. We hope to explore these aspects further and incorporate additional programming languages in our future research.

## References

Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q. Tran, Dara Bahri, Jianmo Ni, Jai Prakash Gupta, Kai Hui, Sebastian Ruder, and Donald Metzler. 2022. 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.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. *arXiv preprint arXiv:2304.01373*.

Sidney Black, Stella Biderman, Eric Hallahan, Quentin Gregory Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Martin Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-neox-20b: An open-source autoregressive language model. In *Challenges & Perspectives in Creating Large Language Models*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399.

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

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint 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. 2022. Scaling instruction-finetuned language models. *CoRR*, abs/2210.11416.

Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 275–286.

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

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

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

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2022. Unnatural instructions: Tuning language models with (almost) no human labor. *arXiv preprint arXiv:2212.09689*.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues?

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, and Jing Ma. 2024. Mmcode: Evaluating multi-modal code large language models with visually rich programming problems. *arXiv preprint arXiv:2404.09486*.

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

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. Gpteval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. 2023. The flan collection: Designing data and methods for effective instruction tuning. *arXiv preprint arXiv:2301.13688*.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2022. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, Dublin, Ireland. Association for Computational Linguistics.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

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

OpenAI. 2022. Introducing ChatGPT. `https://openai.com/blog/chatgpt`.

OpenAI. 2023. Gpt-4 technical report. `https://arxiv.org/pdf/2303.08774`.

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

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

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

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

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pages 19–20.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Jiaan Wang, Yunlong Liang, Fandong Meng, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023a. Is chatgpt a good nlg evaluator? a preliminary study. *arXiv preprint arXiv:2303.04048*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023b. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

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

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. 2022b. Super-NaturalInstructions: Generalization via declarative instructions on 1600+ NLP tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zhen Wang, Rameswar Panda, Leonid Karlinsky, Rogerio Feris, Huan Sun, and Yoon Kim. 2023c. Multitask prompt tuning enables parameter-efficient transfer learning. In *The Eleventh International Conference on Learning Representations*.

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

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Coeditor: Leveraging contextual changes for multi-round code auto-editing. *arXiv preprint arXiv:2305.18584*.

Anxing Xiao, Anshul Gupta, Yuhong Deng, Kaixin Li, and David Hsu. 2024. Robi butler: Multimodal remote interaction with household robotic assistants. In *2nd Workshop on Mobile Manipulation and Embodied Intelligence at ICRA 2024*.

Fuzhao Xue, Kabir Jain, Mahir Hitesh Shah, Zangwei Zheng, and Yang You. 2023. Instruction in the wild: A user-based instruction dataset. `https://github.com/XueFuzhao/InstructionWild`.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to represent edits. *arXiv preprint arXiv:1810.13337*.

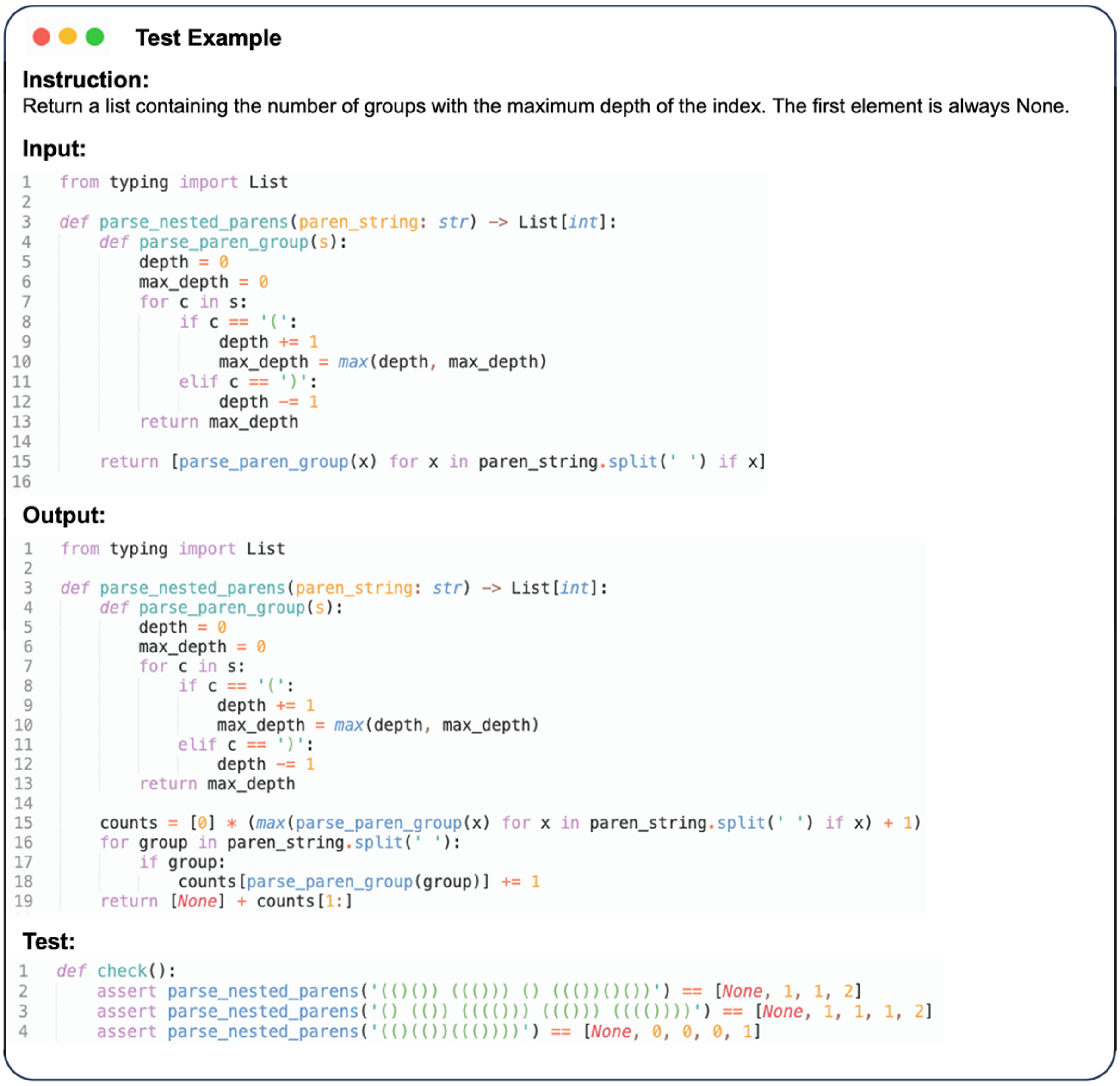
Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

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. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

Terry Yue Zhuo. 2023. Large language models are state-of-the-art evaluators of code generation. *arXiv preprint arXiv:2304.14317*.

## A An Example Test of EditEval

An example of EditEval the test set is showcased below. To accomplish the task, the model must not only adhere to the user's instructions but also comprehend the input code in the context provided.



**Test Example**

**Instruction:**
Return a list containing the number of groups with the maximum depth of the index. The first element is always None.

**Input:**

```python
from typing import List

def parse_nested_parens(paren_string: str) -> List[int]:
    def parse_paren_group(s):
        depth = 0
        max_depth = 0
        for c in s:
            if c == '(':
                depth += 1
                max_depth = max(depth, max_depth)
            elif c == ')':
                depth -= 1
        return max_depth

    return [parse_paren_group(x) for x in paren_string.split(' ') if x]

```

**Output:**

```python
from typing import List

def parse_nested_parens(paren_string: str) -> List[int]:
    def parse_paren_group(s):
        depth = 0
        max_depth = 0
        for c in s:
            if c == '(':
                depth += 1
                max_depth = max(depth, max_depth)
            elif c == ')':
                depth -= 1
        return max_depth

    counts = [0] * (max(parse_paren_group(x) for x in paren_string.split(' ') if x) + 1)
    for group in paren_string.split(' '):
        if group:
            counts[parse_paren_group(group)] += 1
    return [None] + counts[1:]
```

**Test:**

```python
def check():
    assert parse_nested_parens('(()()) ((())) () ((())()())') == [None, 1, 1, 2]
    assert parse_nested_parens('() (()) ((())) (((()))) ((((()))))') == [None, 1, 1, 1, 2]
    assert parse_nested_parens('(()(())((())))') == [None, 0, 0, 0, 1]
```

Figure 7: An example instance of EditEval.

## B Comparing Machine-Generated Data and Real-World Data

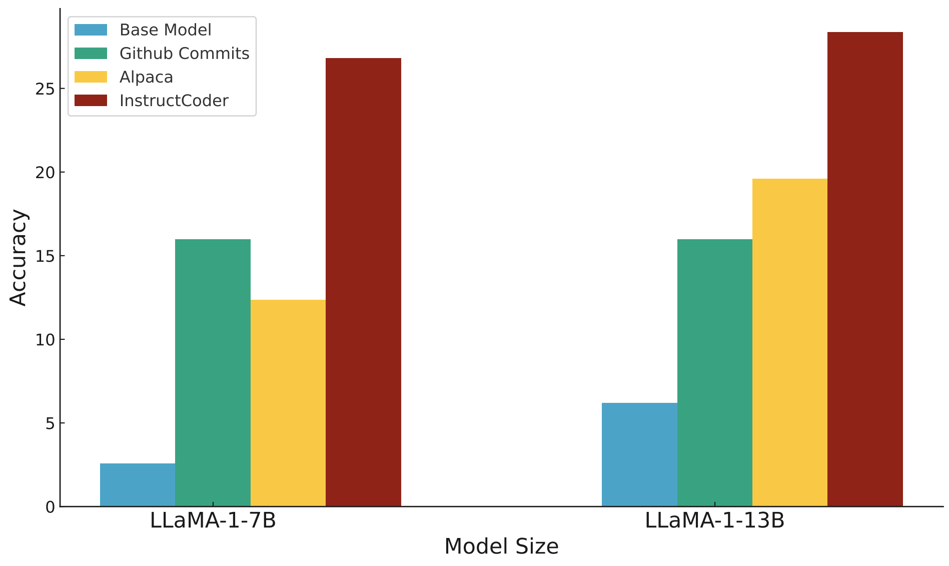


Figure 8: EditEval accuracies of instruction fine-tuned LLaMA-1 models (7B and 13B) with GitHub commits and other datasets. InstructCoder significantly outperformed GitHub commits, and the lead is more pronounced with a larger base model, indicating the effectiveness of InstructCoder. Conversely, fine-tuning with raw GitHub commits yields poor results, and is the worst among all three data sources on LLaMA-1 13B.

Given the substantial repository of code and commit data available on GitHub, a natural idea is to utilize these real-world data to fine-tune a model to perform code editing. However, as discussed in Section 4.1, these data from GitHub can be extremely noisy, especially in the commit messages, rendering them a sub-optimal choice for instruction-tuning. On the other hand, machine-generated data is increasingly recognized for its utility, as evidenced by various studies that achieves enhanced results with this type of data (Gunasekar et al., 2023; Li et al., 2023b; Wang et al., 2023b). This approach provides better controllability over the distribution of the generated contents and facilitates the collection of diverse data, including those under-represented or difficult to mine and clean from real-world data.

The experiment results in Figure 8 corroborate the usage of machine-generated data. We further collected GitHub commits matching the size of InstructCoder, and used the same hyperparameters for instruction fine-tuning. As can be seen in the results, InstructCoder significantly outperformed raw GitHub commits, and the lead is more profound with a larger base model, demonstrating the effectiveness of InstructCoder. On the other hand, fine-tuning with GitHub commits yields poor results, and is the worst among all three data sources on LLaMA-1 13B. The observation suggests that using machine-generated data for instruction fine-tuning is superior in terms of training code editing models.

## C Prompts

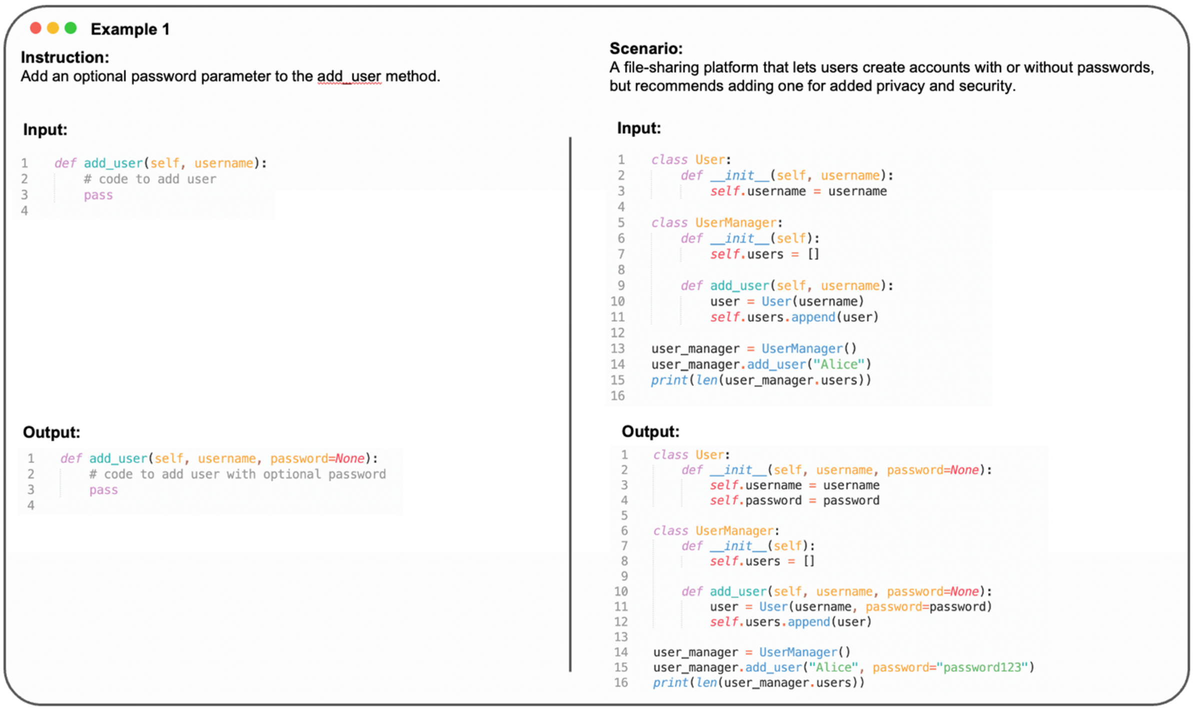
The prompts used in our data collection and experiments are listed in Table 4.

| Stage | Prompt |
|---|---|
| Instruction Generation | `Given the existing instructions, please generate a list of diverse Python code editing instructions. The new instructions should address diverse editing tasks. Please ensure that the instructions are clear and diverse. Include any relevant variable names in the instructions.` |
| Scenario Generation | `Given a Python code editing task, please come up with 10 diverse scenarios with concise descriptions of where this task could be performed or come from.` |
| Instance Generation | `Given Python code editing task instructions and their scenarios where the task instruction could be used, you need to come up with examples for the following code editing tasks. You need to generate an input and output code pair and make sure your variable names are suitable for the scenario. The input code is related to the task instruction, but must NOT meet the task requirements. The output code fulfills the task requirements based on the input code.` |
| GPT4 Evaluation | `Given a code editing instruction, please determine if the output is an acceptable edited code response to the instruction and input. Give "Yes" or "No".` |

Table 4: Prompts used in this work.

## D Qualitative Examples of Scenario-Conditional Generation

Three comparisons are presented, each showing instances that were generated with or without the inclusion of a scenario.



**Example 1**

**Instruction:**
Add an optional password parameter to the add_user method.

**Input:**

```
def add_user(self, username):
    # code to add user
    pass
```

**Output:**

```
def add_user(self, username, password=None):
    # code to add user with optional password
    pass
```

**Scenario:**
A file-sharing platform that lets users create accounts with or without passwords, but recommends adding one for added privacy and security.

**Input:**

```
class User:
    def __init__(self, username):
        self.username = username

class UserManager:
    def __init__(self):
        self.users = []

    def add_user(self, username):
        user = User(username)
        self.users.append(user)

user_manager = UserManager()
user_manager.add_user("Alice")
print(len(user_manager.users))
```

**Output:**

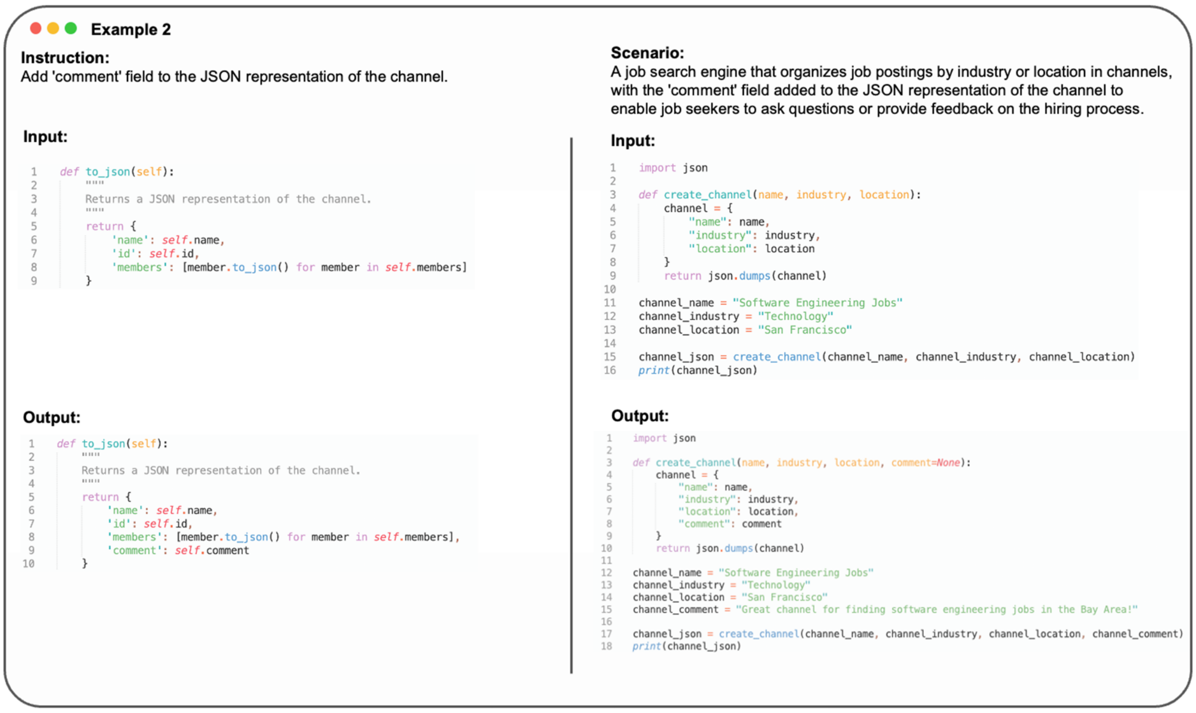
```
class User:
    def __init__(self, username, password=None):
        self.username = username
        self.password = password

class UserManager:
    def __init__(self):
        self.users = []

    def add_user(self, username, password=None):
        user = User(username, password=password)
        self.users.append(user)

user_manager = UserManager()
user_manager.add_user("Alice", password="password123")
print(len(user_manager.users))
```

Figure 9: Example instance #1 generated without scenario (Left) and with scenario (Right)



**Example 2**

**Instruction:**
Add 'comment' field to the JSON representation of the channel.

**Input:**

```
def to_json(self):
    """
    Returns a JSON representation of the channel.
    """
    return {
        'name': self.name,
        'id': self.id,
        'members': [member.to_json() for member in self.members]
    }
```

**Output:**

```
def to_json(self):
    """
    Returns a JSON representation of the channel.
    """
    return {
        'name': self.name,
        'id': self.id,
        'members': [member.to_json() for member in self.members],
        'comment': self.comment
    }
```

**Scenario:**
A job search engine that organizes job postings by industry or location in channels, with the 'comment' field added to the JSON representation of the channel to enable job seekers to ask questions or provide feedback on the hiring process.

**Input:**

```
import json

def create_channel(name, industry, location):
    channel = {
        "name": name,
        "industry": industry,
        "location": location
    }
    return json.dumps(channel)

channel_name = "Software Engineering Jobs"
channel_industry = "Technology"
channel_location = "San Francisco"

channel_json = create_channel(channel_name, channel_industry, channel_location)
print(channel_json)
```

**Output:**

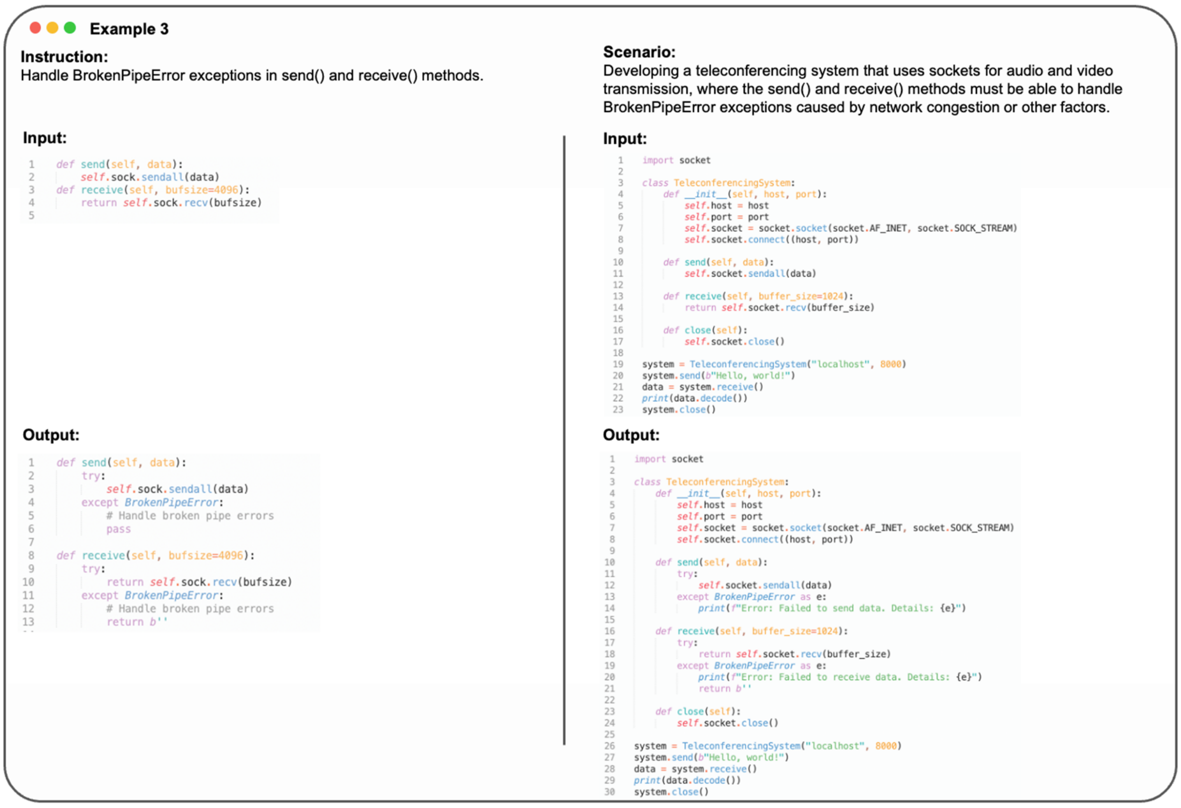
```
import json

def create_channel(name, industry, location, comment=None):
    channel = {
        "name": name,
        "industry": industry,
        "location": location,
        "comment": comment
    }
    return json.dumps(channel)

channel_name = "Software Engineering Jobs"
channel_industry = "Technology"
channel_location = "San Francisco"
channel_comment = "Great channel for finding software engineering jobs in the Bay Area!"

channel_json = create_channel(channel_name, channel_industry, channel_location, channel_comment)
print(channel_json)
```

Figure 10: Example instance #2 generated without scenario (Left) and with scenario (Right)



**Example 3**

**Instruction:**
Handle BrokenPipeError exceptions in send() and receive() methods.

**Input:**

```python
def send(self, data):
    self.sock.sendall(data)
def receive(self, bufsize=4096):
    return self.sock.recv(bufsize)

```

**Output:**

```python
def send(self, data):
    try:
        self.sock.sendall(data)
    except BrokenPipeError:
        # Handle broken pipe errors
        pass

def receive(self, bufsize=4096):
    try:
        return self.sock.recv(bufsize)
    except BrokenPipeError:
        # Handle broken pipe errors
        return b''
```

**Scenario:**
Developing a teleconferencing system that uses sockets for audio and video transmission, where the send() and receive() methods must be able to handle BrokenPipeError exceptions caused by network congestion or other factors.

**Input:**

```python
import socket

class TeleconferencingSystem:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, port))

    def send(self, data):
        self.socket.sendall(data)

    def receive(self, buffer_size=1024):
        return self.socket.recv(buffer_size)

    def close(self):
        self.socket.close()

system = TeleconferencingSystem("localhost", 8000)
system.send(b"Hello, world!")
data = system.receive()
print(data.decode())
system.close()
```

**Output:**

```python
import socket

class TeleconferencingSystem:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, port))

    def send(self, data):
        try:
            self.socket.sendall(data)
        except BrokenPipeError as e:
            print(f"Error: Failed to send data. Details: {e}")

    def receive(self, buffer_size=1024):
        try:
            return self.socket.recv(buffer_size)
        except BrokenPipeError as e:
            print(f"Error: Failed to receive data. Details: {e}")
            return b''

    def close(self):
        self.socket.close()

system = TeleconferencingSystem("localhost", 8000)
system.send(b"Hello, world!")
data = system.receive()
print(data.decode())
system.close()
```

Figure 11: Example instance #3 generated without scenario (Left) and with scenario (Right)

## E Additional statistics of InstructCoder

| Token Length | Instruction | Input | Output |
|---|---|---|---|
| mean | 21.85 | 172.03 | 248.43 |
| 25% | 17 | 99 | 138 |
| 50% | 21 | 147 | 213 |
| 75% | 26 | 218 | 321 |
| min | 3 | 10 | 10 |
| max | 116 | 1019 | 1024 |

Table 5: Token length statistics using the LLaMA (Touvron et al., 2023a) tokenizer.

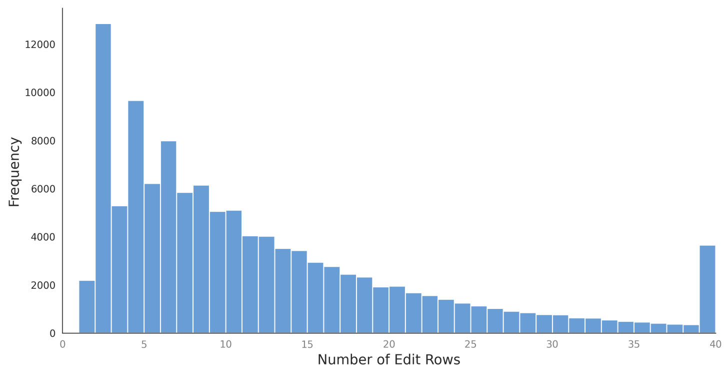


Figure 12: Edit rows distribution of InstructCoder. Numbers greater than 40 are aggregated as the last bin.

## F Hyperparameters

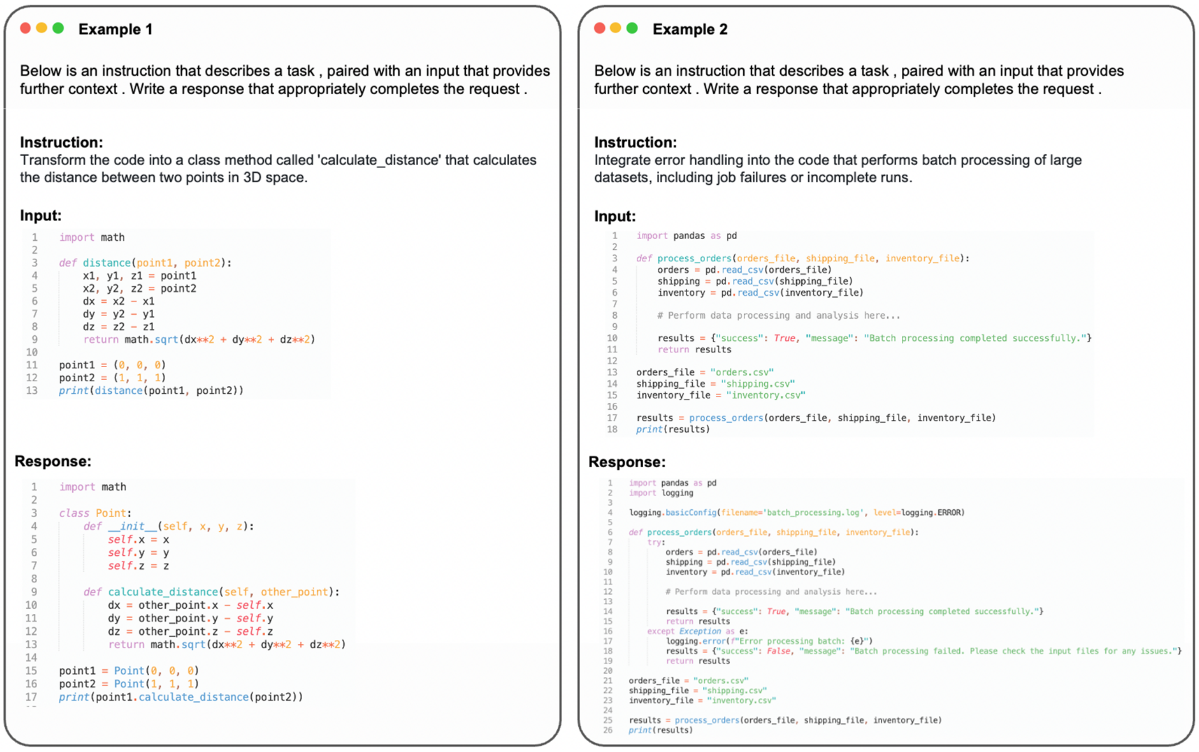
The hyperparameters used in all finetuning experiments are listed in Table 6. For all inferences, we utilize greedy decoding. For OpenAI's GPTs, we achieve this by setting its temperature to 0.

| Hyperparameter | Value |
|---|---|
| learning rate | 0.0003 |
| batch size | 128 |
| epochs | 3 |
| max sentence length | 1024 |
| lora rank | 16 |
| lora dropout | 0.05 |
| lora modules | key, query, value, output |

Table 6: Hyperparameters used for finetuning language models.

## G Qualitative Examples Generated by Finetuned LLaMA-33B

We demonstrate some qualitative example responses generated by finetuned LLaMA-33B.



**Example 1**

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:**
Transform the code into a class method called 'calculate_distance' that calculates the distance between two points in 3D space.

**Input:**

```python
import math

def distance(point1, point2):
    x1, y1, z1 = point1
    x2, y2, z2 = point2
    dx = x2 - x1
    dy = y2 - y1
    dz = z2 - z1
    return math.sqrt(dx**2 + dy**2 + dz**2)

point1 = (0, 0, 0)
point2 = (1, 1, 1)
print(distance(point1, point2))
```

**Response:**

```python
import math

class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def calculate_distance(self, other_point):
        dx = other_point.x - self.x
        dy = other_point.y - self.y
        dz = other_point.z - self.z
        return math.sqrt(dx**2 + dy**2 + dz**2)

point1 = Point(0, 0, 0)
point2 = Point(1, 1, 1)
print(point1.calculate_distance(point2))
```

**Example 2**

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:**
Integrate error handling into the code that performs batch processing of large datasets, including job failures or incomplete runs.

**Input:**

```python
import pandas as pd

def process_orders(orders_file, shipping_file, inventory_file):
    orders = pd.read_csv(orders_file)
    shipping = pd.read_csv(shipping_file)
    inventory = pd.read_csv(inventory_file)

    # Perform data processing and analysis here...

    results = {"success": True, "message": "Batch processing completed successfully."}
    return results

orders_file = "orders.csv"
shipping_file = "shipping.csv"
inventory_file = "inventory.csv"

results = process_orders(orders_file, shipping_file, inventory_file)
print(results)
```

**Response:**

```python
import pandas as pd
import logging

logging.basicConfig(filename='batch_processing.log', level=logging.ERROR)

def process_orders(orders_file, shipping_file, inventory_file):
    try:
        orders = pd.read_csv(orders_file)
        shipping = pd.read_csv(shipping_file)
        inventory = pd.read_csv(inventory_file)

        # Perform data processing and analysis here...

        results = {"success": True, "message": "Batch processing completed successfully."}
        return results
    except Exception as e:
        logging.error(f"Error processing batch: {e}")
        results = {"success": False, "message": "Batch processing failed. Please check the input files for any issues."}
        return results

orders_file = "orders.csv"
shipping_file = "shipping.csv"
inventory_file = "inventory.csv"

results = process_orders(orders_file, shipping_file, inventory_file)
print(results)
```

Figure 13: Qualitative examples generated by finetuned LLaMA-33B

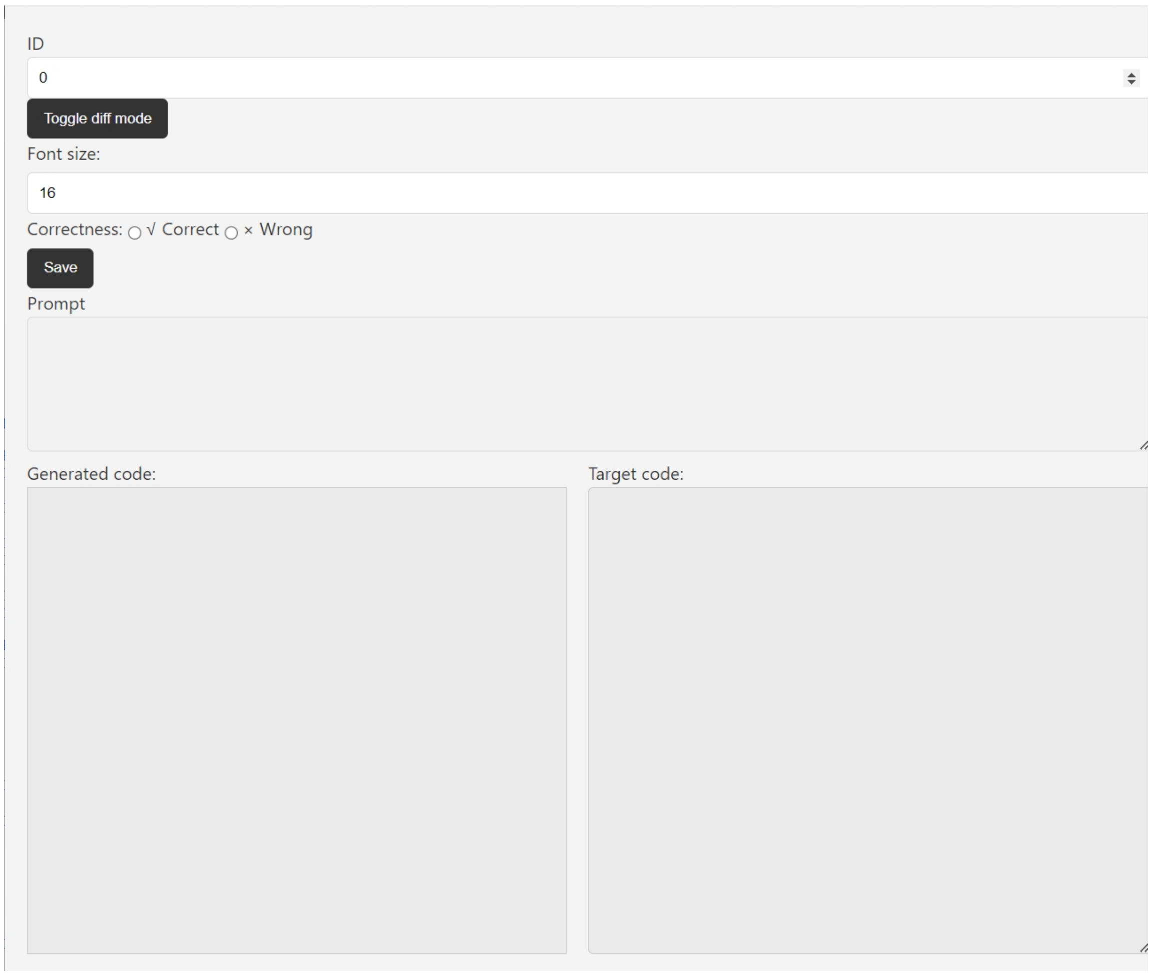
## H Alignment of GPT-4 Evaluation and Human Evaluation

Due to the extremely demanding nature of creating automated tests, we seek to investigate the viability of using GPT-4 as an automatic evaluator to lessen the extensive human effort involved. Using LLMs as generation evaluators has been demonstrated effective in NLG tasks (Liu et al., 2023; Wang et al., 2023a; Fu et al., 2023), and especially in code generation (Zhuo, 2023). To further validate this idea, we collected an additional 134 commits data for testing purposes and processed them in the same manner as the seed tasks. Both GPT-4 evaluation and human evaluation are conducted on this dataset to assess their alignment.

**Human evaluation.** Each sample is annotated by three examiners, and the average accuracy is recorded. We developed an annotation tool to ensure the impartiality of evaluation (see Figure 14 for the user interface). Generations of different models are shuffled and the anonymity of the models is guaranteed. The edit is annotated as *correct* if it correctly reflects the instruction demands and *wrong* if it fails to follow the instruction.

**GPT-4 evaluation.** We ask GPT-4 to evaluate if the code edit is an acceptable response to the input and collect the correct rate. The prompts for GPT-4 evaluation can be found in C.

**Results.** We carry out the experiments on the code edits generated by ChatGPT and LLaMA of three sizes fine-tuned with InstructCoder. While we found that the human annotators are always slightly stricter than the GPT-4 evaluator, the overall Cohen's Kappa value of the GPT-4 evaluations and human evaluations reaches 0.665, which is substantial according to Cohen (1960). This renders GPT-4 evaluation as a convenient and effective method for evaluating the correctness of code edit tasks.


ID

0

Toggle diff mode

Font size:

16

Correctness: ○ √ Correct ○ × Wrong

Save

Prompt

Generated code:

Target code:

Figure 14: A screenshot of our human scoring annotation tool.

## I Data Filtering Process

The detailed process of filtering the dataset is listed below:

- We selected GitHub repos with over 100 stars to ensure the overall quality. We only utilized repos with permissive licenses (MIT, Apache-2.0, GPL-3.0, GPL-2.0, BSD-2.0, BSD-3.0, LGPL-2.1, LGPL-3.0, AGPL-3.0).
- We kept commits in which only one single .py file was changed. Using git-diff, we identified and preserved commits where only one code block was changed.
- We discarded commits with single-word or empty commit messages.
- We removed commits with over 100 edited rows.

Manual:

- We discarded rare commits containing inappropriate language.
- We discarded commits where the change in the source code does not match the commit message.
- We filtered out project-specific adjustments that lack sufficient context.
- We utilized Codex (Chen et al., 2021a) to rewrite ambiguous commit messages, enhancing the clarity of the intended code edits.