CodeInstruct: Empowering Language Models to Edit Code

Qisheng Hu[1*] **Kaixin Li**[1*] **Xu Zhao**[1] **Yuxi Xie**[1] **Tiedong Liu**[1] **Hui Chen**[2]
Qizhe Xie[1†] **Junxian He**[3†]
[1]National University of Singapore [2]Singapore University of Technology and Design
[3]Shanghai Jiao Tong University
{qishenghu,likaixin,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 utilizing large language models (LLMs) to edit code based on user instructions, covering a broad range of implicit tasks such as comment insertion, code optimization, and code refactoring. To facilitate this, we introduce CodeInstruct, a dataset designed to adapt LLMs for code editing, containing high-diversity code-editing tasks. It consists of over 114,000 instruction-input-output triplets and covers multiple distinct code editing scenarios. The dataset is systematically expanded through an iterative process that commences with editing data sourced from GitHub commits as seed tasks. Seed and generated tasks are used subsequently to prompt ChatGPT for more task data. Our experiments demonstrate that open-source LLMs fine-tuned on CodeInstruct can edit the code correctly most of the time based on users' instructions, showcasing an on-par performance with ChatGPT. Such results indicate that proficient instruction-finetuning can lead to significant amelioration in code-editing abilities. The dataset is public at https://github.com/qishenghu/CodeInstruct.

1 Introduction

Developers typically engage in a cyclic routine of writing and revising code. As a crucial element, automatic code editing could potentially enhance development efficiency significantly. However, the intricacy of this task has hampered substantial progress by deep learning models. This is attributable to the fact that code editing encapsulates diverse subtasks, such as code optimization, comment insertion, and bug fixing. Each of these diverse subtasks presents distinct challenges and requires unique capabilities to solve, thereby posing considerable hurdles for modeling.

Recent development of large language models (LLMs) has made remarkable progresses in NLP, demonstrating strong few-shot and zero-shot abilities (Brown et al., 2020; Scao et al., 2022; Chowdhery et al., 2022; Ouyang et al., 2022; OpenAI, 2022; Touvron et al., 2023). Beyond text models, code LLMs have also elicited significant interest, highlighting their immense potential in code generation (Nijkamp et al., 2023; Chen et al., 2021; Li et al., 2023). Inspired by these advancements, we explore the proficiency of LLMs in editing code based on user instructions, for instance, "add docstring to the function for clarity", "remove redundant code", or "refactor it into reusable functions".

To this end, we curate a code editing dataset, dubbed CodeInstruct, for improving and evaluating code editing abilities of LLMs. CodeInstruct is an instructional dataset containing diverse code-editing tasks. The dataset is primarily generated by ChatGPT (OpenAI, 2022). Specifically, we first collect and manually scrutinize git commit data from public repositories on GitHub as the seed code editing tasks, then we utilize the seed data to prompt ChatGPT to generate new instructions and input-output pairs respectively, where a scenario (e.g. web development) is randomly sampled from a list of scenarios and specified to ensure diversity of the data. This process resembles the Self-Instruct framework (Wang et al., 2022a).

By innovatively incorporating scenarios during the generation process, our approach ensures that the code-editing instances in the CodeInstruct dataset are diverse and relevant to real-world pro-

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

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

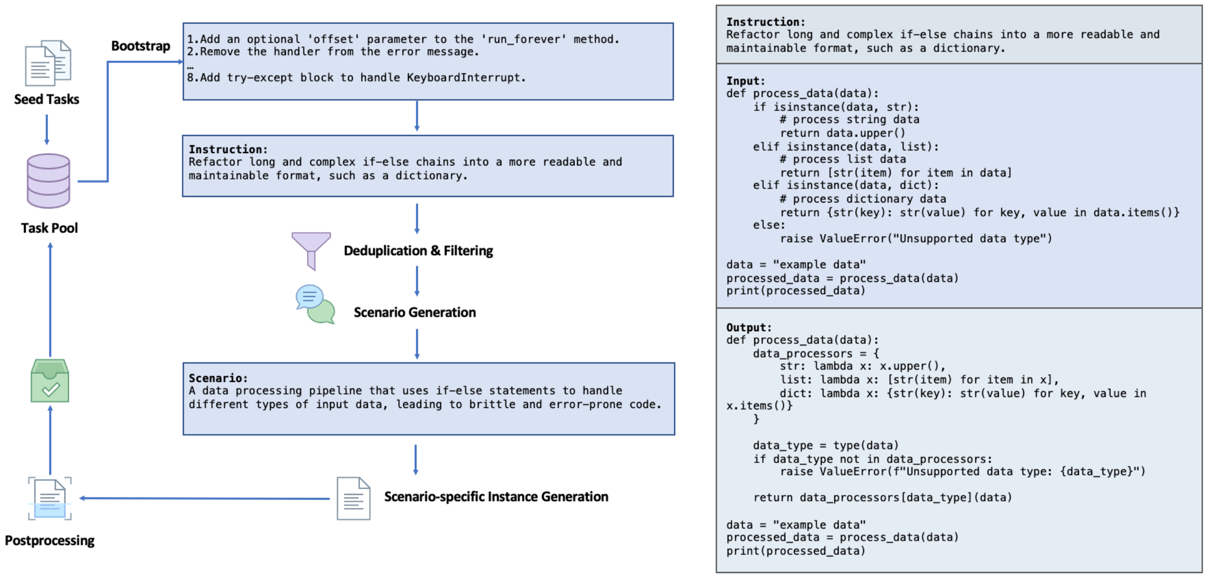


Figure 1: Data collection pipeline of CodeInstruct (left) and an qualitative example from the dataset (right, best viewed with zoom). Initial seed tasks are selected from GitHub commits, and inspire ChatGPT to generate new instructions. Plausible scenarios where the filtered instruction may be used are 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.

gramming situations. This approach enables ChatGPT to synthesize more diverse input-output code snippets in terms of variable naming and functionality given the code-editing instructions and scenarios, 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 post finetuning on CodeInstruct. The largest model used in the experiment, LLaMA-33B, performs on-par with ChatGPT, achieving an edit accuracy of 89.3 and 76.3 as evaluated by GPT-4 and humans respectively. Further findings signify that edit accuracy improves log-linearly with data scale, and the pretraining strategy of the LLMs also has a great impact.

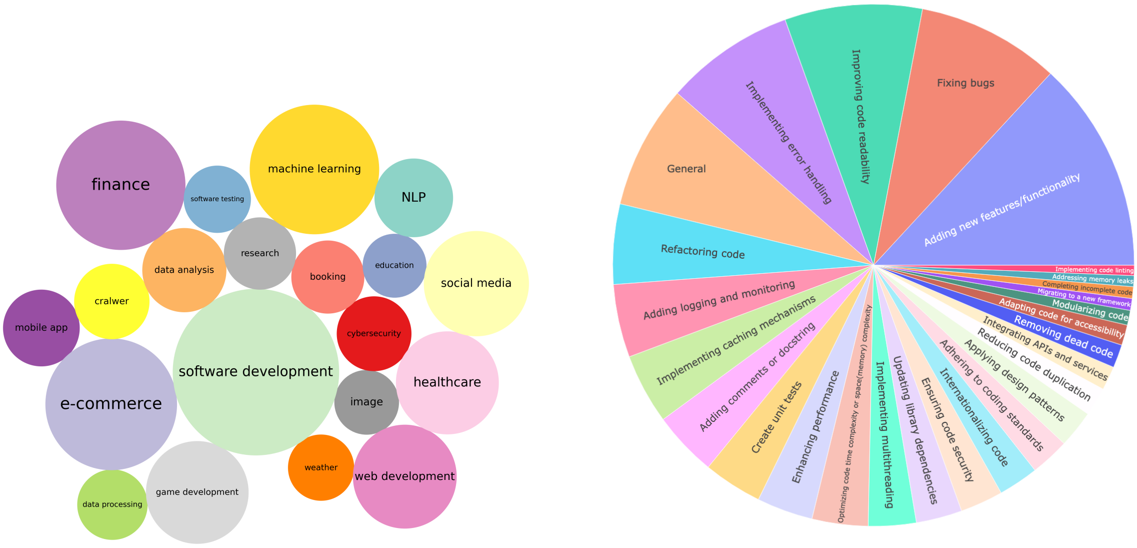
2 Related Work

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., 2023b). 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., 2023), or human-written with crowdsourcing or community joint efforts (Ouyang et al., 2022; Wang et al., 2022b; DataBricks, 2023). 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 is an extensively studied area. Language models pretrained on large collections of code have demonstrated strong abilities in a variety of programming tasks. A number of general LLMs gain code generation abilities due to the mixture of code in the pretraining corpus (e.g. The Pile (Gao



(a) Word cloud of scenarios used in CodeInstruct

(b) Code edit intent distribution

Figure 2: Visualizations of CodeInstruct tasks. Best viewed in zoom.

et al., 2020)), such as GPT-3 (Brown et al., 2020), ChatGPT (OpenAI, 2022), GPT-4 (OpenAI, 2023), LLaMA (Touvron et al., 2023), 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., 2021), CodeGen (Nijkamp et al., 2023), CodeGeeX (Zheng et al., 2023) and StarCoder (Li et al., 2023). These models all adopt the decoder-only transformer architecture, but differ in size and specific model design (e.g. positional embedding, norm layer placement) as well as the selection and preprocessing of pretraining corpus.

On the other hand, relatively fewer 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 performance optimization (Madaan et al., 2023). The PIE (Madaan et al., 2023) dataset is the most relevant to our work, which focuses on speeding up programs. LLMs trained on PIE have demonstrated substantial enhancements in program performance. Nevertheless, datasets particularly tailored for general-purpose code editing are absent. To fill this gap, we introduce CodeInstruct, a novel dataset aimed at further advancing the capabilities of code editing with LLMs.

3 CodeInstruct Dataset Collection

To generate instructional data for code editing, 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 training data with LLMs only requires minimal human-labeled data as seed tasks while still maintaining the quality and relevance of the tasks in the dataset. Through an iterative process of generating new instructions and refining them with proper deduplication, we were able to create a dataset of a wide range of code-editing tasks.

3.1 Seed Data Collection

GitHub[1] is a code hosting platform whose version control service naturally records code edits along with commits, which can be converted to instructions. The extensive repositories available on GitHub provides diverse and ample data with human-generated quality. However, the data from GitHub cannot be directly utilized. 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 functionalities and files,

[1]https://github.com

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.

Initially, raw github commit data were collated through Google BigQuery[2]. The instructions for each task were derived from the commit message, while the input and output corresponded to the code version before/after the commits. While cleaning and annotating manually, we came across many imprecise or emotionally charged commit messages. In order to convert the commit messages to proper instructions, We employed Codex (Chen et al., 2021) to clarify the changes made between versions and improve the commit messages, resulting in more precise and informative instructions for seed tasks. A total of 768 seed tasks were processed from the GitHub commit data through manual efforts. 634 tasks were used for self-instruct purposes while 134 reserved for evaluation.

In addition to using GitHub commit data, we made use of high-quality generated samples for seed tasks. Through manual inspection, we identified 27 types of code edits and compiled a batch of 592 high-quality samples to form additional seed tasks. This set of seed data cover a wide range of code-editing scenarios and forms the very basis on which the CodeInstruct dataset is created, ensuring that the tasks are rooted in plausible real-world code-editing cases.

3.2 Instruction Bootstrapping

Self-Instruct serves 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 in a few-shot manner to prompt ChatGPT for more instructions. To generate more diverse and practically applicable instructions, we also generate tasks across multiple sub-domains by specifying the editing intent in the prompt provided.

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

3.3 Scenario-conditional Generation

During the process of generating task instances for code editing, we found that a number of generated samples share similar codebases and variable names despite the different instructions and few-shot examples provided. Such similarity among generated samples could significantly reduce the dataset's research value. Through empirical analysis, we concluded that this issue could arise due to the fact that the the LLM tends to generate general codebases for both input/output code snippets without further context. To mitigate this, we propose to introduce scenarios to input/output generation. As an illustration of the effects of scenario generation, we present an example in Figure 9 in Appendix F, where we observe that instances generated with the inclusion of a scenario demonstrate higher quality in terms of richer context and code structure compared to those without.

For each generated instruction, we prompt ChatGPT to generate a list of practical events as "real-world" scenarios where the editing instruction could be performed, and randomly selected one for subsequent generation. During instance generation, the LLM is instructed to generate examples that correspond with the operation in the instruction, ensuring the codebases and variable names are appropriate for the given scenario.

By incorporating scenario-conditional generation, the resulting samples also exhibit increased variability in regards to codebases and variable naming, thus augmenting the diversity of CodeInstruct.

3.4 Postprocessing

Following Self-Instruct (Wang et al., 2022a), deduplication is applied on the generated instructions to remove instructions that have a ROUGE-L (Lin, 2004) overlap score larger than 0.7 with the existing instructions. We also employ MinHash with Locality Sensitive Hashing (LSH) indexing using datasketch[3] to remove instances with an input code Jaccard similarity greater than 0.75, in order to deduplicate at the code level. A few more heuristic rules were used to clean up the generated data. Through postprocessing, we achieved a high level of effectiveness in eliminating erroneous and redundant data.

[3]http://ekzhu.com/datasketch/

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 1: Statistic overview of CodeInstruct

We kept 95% of the dataset as the training set and assigned around 5% of the dataset to the validation set. The test set is built with held-out seed samples from real GitHub code to better reflect the real-world edit cases. Since commit messages from GitHub code edits can be noisy, we conducted manual quality filtering and kept 134 tasks.

4 Data Analysis

In this section, we analyze CodeInstruct in terms of 1) diversity, 2) complexity, and 3) correctness. We provide distribution as well as complexity analyses of the tasks contained in the dataset. Finally, we demonstrate through human investigation that our data is reliable, albeit with occasional minor errors.

4.1 Statistic Overview

The CodeInstruct dataset comprises a total of 114,239 code editing instructions, each paired with an input/output instance. The statistics of the dataset are summarized in Table 1. Token length distribution of instruction/input/output can be viewed in Figure 5 and Figure 6 in Appendix.

4.2 Instruction Diversity

To explore the diversity of tasks in CodeInstruct 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 clustered 27 empirical genres through meticulous investigation of the generated samples. Figure 2b shows the distribution of the code edit intent categories in CodeInstruct, which include adding functionalities, optimizing code, improving readability, etc. These objectives underscore the extensive range of CodeInstruct.

Instruction Verbs. The diversity of instruction verbs is also portrayed in Figure 8 in Appendix. We extract the stem of the initial verb in every instruction using spaCy[4], and plot the distribution of the top 20 frequent terms as well as the relatively infrequent verbs merged as "other". While a great portion of the instructions can be roughly clustered as *creation* (e.g. "add", "implement", "creat" and "develop") and *modification* (e.g. "modify", "replace", "change" and "fix"), the data demonstrate a long-tail distribution, with "other" taking up 25.0% percentage. Overall, it is indicated that the dataset contains a wide spectrum of instructions.

4.3 Scenario Distribution

CodeInstruct is designed to cover a wide range of scenarios. In our setting, each instruction is prompted to generate different scenarios where the editing instruction could be performed. This approach ensures that the generated samples exhibit greater diversity in terms of codebases and contexts. We show some of the scenario domains in our dataset, as illustrated in Figure 2a. It was noted that software development, machine learning, finance, and e-commerce are the most represented domains, which highlights their prevalence in these scenarios. In addition to this, the diversity of the dataset is further emphasized by the presence of domains like image processing, web development, cybersecurity and much more.

4.4 Complexity

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

$$n_{diff} = |I \cup O \setminus I \cap O|, \tag{1}$$

$$r_{diff} = \frac{n_{diff}}{|I \cup O|}, \tag{2}$$

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 CodeInstruct is 11.9 and the average ratio is 0.52. These values suggest a fairly acceptable level of complexity, indicating that the dataset is neither too easy nor too hard. *CodeInstruct* strikes a balance in terms of complexity, making it well-suited for finetuning and

[4] https://spacy.io/

[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.

evaluating LLMs in a wide range of code editing tasks. Figure 7 in Appendix illustrates the distribution of the number of differing lines.

4.5 Correctness

To further evaluate the correctness of CodeInstruct, we randomly sample 200 instances and invite three co-authors to evaluate the instances based on two criteria: the validity of the instruction and the correctness of the instances. The validity assessment focuses on deciding if the instructions clearly exhibit editing intent and are appropriate for code editing. The correctness evaluation examines if the input-output pairs reflect the changes specified by the instructions.

The evaluation results in Table 2 indicate that most instructions in the CodeInstruct dataset are valid. A few instances exhibit noise and occasional failure to comply with the instruction, but overall high correctness is achieved. Out of the 200 evaluated instances, 180 cases were successfully solved, showcasing the overall quality and reliability of the CodeInstruct.

5 Experiments

5.1 Instruction Finetuning

We experiment with two families of open-source language models: LLaMA (7B, 13B, 33B) (Touvron et al., 2023) and BLOOM (560M, 3B, 7B) (Scao et al., 2022). LLaMA is a series of large language models with parameter counts ranging from 7B to 65B, and pretrained with an excessive amount of tokens, wherein code takes up approximately 4.5%. BLOOM is a multilingual LLM capable of generating human-like outputs in 46 languages and 13 programming languages.

A full finetuning which updates all the parameters in an LLM can be computationally expensive. Instead, we adopt LoRA (Hu et al., 2022), a parameter-efficient finetuning method which optimizes an approximated low-rank delta matrix of the fully-connected layers. Though the number of parameters updated in LoRA is typically several magnitudes lower than that of the full model, many works have demonstrated its effectiveness comparable to full finetuning (Hu et al., 2022; et. al., 2023; Wang et al., 2023a). In this way we could fit a 33B model in a single A100-80GB GPU card. Across all our experiments, LoRA is applied on the query, key, value and output transform weights of the Transformer architecture (Vaswani et al., 2017). All hyperparameters can be found in Appendix C.

5.2 Baselines

ChatGPT (OpenAI, 2022). ChatGPT serves as a strong baseline for our experiments, which utilize a GPT-3.5-turbo model optimized for chatting and following human intent. We report ChatGPT's zero-shot performance.

LLaMA (Touvron et al., 2023). As we mainly use the LLaMA series for instruction finetuning experiments, we also explored the original LLaMA's code edit capabilities for better comparison. Their zero-shot and one-shot accuracies are evaluated as baselines.

Alpaca (Taori et al., 2023). Alpaca is an instruction-finetuned version of LLaMA. It was finetuned on around 52,000 samples generated with a improved version of Self-Instruct (Wang et al., 2022a). 11 out of 175 used seed tasks are related to coding, resulting in a blend of code instructions in the generated dataset. Since the authors did not release their models, we trained the Alpaca models, using LoRA with the same hyperparameters settings with our other experiments. The prompt template is identical to our CodeInstruct finetuned LLaMA models.

5.3 Metrics

Evaluating the accuracy of code edits presents a complex challenge due to the potential for incomplete code snippets and the existence of multiple valid modifications. Consequently, evaluating correctness using conventional metrics proves arduous, hence our reliance on human evaluation. Each sample is annotated by three examiners, and the

Model	Finetuned	N-shot	Accuracy
ChatGPT	-	0	**90.5**
LLaMA-7B	✗	0	7.7
	✗	1	8.7
LLaMA-13B	✗	0	9.7
	✗	1	14.4
LLaMA-33B	✗	0	14.2
	✗	1	19.9
Alpaca-7B	✗	-	39.3
Alpaca-13B	✗	-	55.2
Alpaca-33B	✗	-	70.6
BLOOM-560M	✓	-	20.9
BLOOM-3B	✓	-	51.2
BLOOM-7B	✓	-	56.2
LLaMA-7B	✓	-	69.2
LLaMA-13B	✓	-	75.9
LLaMA-33B	✓	-	**89.3**

Table 3: Accuracy evaluated by GPT-4. The *Finetuned* column indicates if the model is finetuned on CodeInstruct.

average accuracy is reported. We also endeavored to prompt GPT-4 (OpenAI, 2023) in inspecting the modifications.

Human Scoring We establish a rule indicating three scoring levels: *correct*, *partial* and *wrong*. To ensure the impartiality of the scoring process, output samples from different models are shuffled and each is evaluated by three co-authors using a tool that guarantees the anonymity of the models was used. The edit is assigned *correct* if it correctly reflects the instruction editing demands and *wrong* if it fails to follow the instruction. We also introduce a *partial* class to contain subtle situations where the edit is correct but unexpected modifications are made, such as removal of existing comments and redundant generation after the termination of the input.

GPT-4 (OpenAI, 2023) Evaluation. We leverage GPT-4 as an automatic evaluator, with the purpose of alleviating the need of human effort and ensure fair evaluation. We prompt GPT-4 to evaluate if the code edit is an acceptable response to the input and instruction. The prompts can be found in Appendix D.4.

Model	Correct	Partial	Wrong
ChatGPT	79.3	10.4	10.3
LLaMA-7B	54.1	8.1	37.8
LLaMA-13B	69.6	5.2	25.2
LLaMA-33B	76.3	8.1	15.6

Table 4: Human evaluation results on the real test set. *Correct* means the edit correctly reflect the task demand. *Partial* means the edit is mostly correct, but with minor unexpected modifications. *Wrong* indicates non-acceptable edit.

6 Results

6.1 Finetuning Efficacy

Table 3 provides a comprehensive comparison across models finetuned with CodeInstruct and the baselines. We leave the discussion of the validity of using GPT-4 as an evaluator in Appendix A. The average of three runs was taken for each score. We also showcase human-evaluated model performance finetuned with CodeInstruct in Table 4. While low accuracy are observed in plain LLaMA models and only marginal improvement is achieved through few-shot prompting, finetuning with CodeInstruct significantly boost the accuracy by over 60% absolutely. Overall, finetuning with CodeInstruct leads to great performance improvement over the few-show baselines, suggesting the effectiveness of efficient instruction finetuning with machine-generatied code edit pairs.

It is noteworthy that our largest finetuned LLaMA-33B exhibits a performance comparable with the strong baseline ChatGPT on the test set. Despite the noise present in the data points collected through git-diff,[6] which might entail incomplete contextual information and some disparity in code structure, the finetuned LLaMA-33B achieves an accuracy of 89.3% under GPT-4 evaluation, with a 69% increase over its plain counterpart.

6.2 Dataset Scaling

CodeInstruct has a scale considerably smaller than what LLMs are typically pretrained on. In order to ascertain the sufficiency of this scale, we conducted an experiment wherein we fine-tuned the LLaMA

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

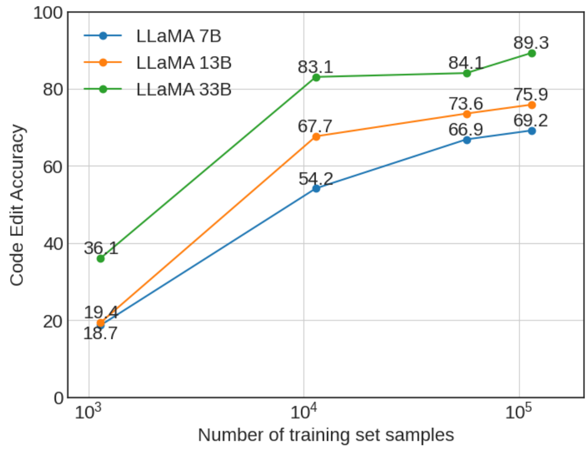


Figure 3: Data scaling performance of CodeInstruct on LLaMA evaluated by GPT-4, using 1%, 10%, 50% and 100% training data.

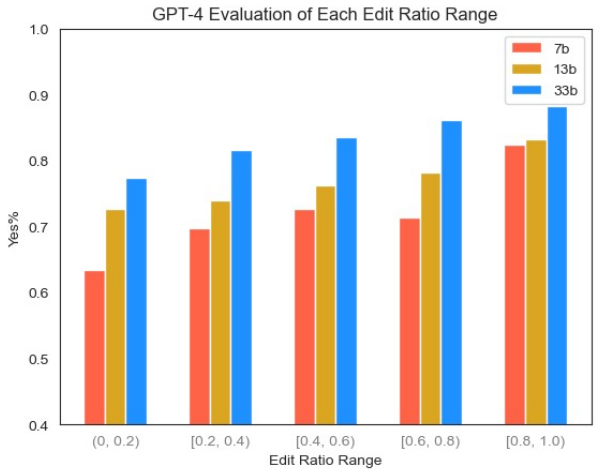


Figure 4: GPT-4 evaluation results at different edit ratios on 2000 validation samples.

family of models using varying proportions (1%, 10%, 50%, and 100%) of the dataset. The data subsets corresponding to smaller proportions are guaranteed to be encompassed within the larger data subsets.

The identified trend demonstrates a positive correlation between the model's accuracy and the scale of the training set. However, this relationship exhibits diminishing returns as the dataset size continues to expand. Utilizing merely 1% of the data for fine-tuning, the models experience very limited number of parameter updates. However, they still surpass the one-shot accuracy benchmarks by considerable margins, demonstrating the necessity of instruction-finetuning. We also found that with over 10% training data, larger models demonstrates superior performance than smaller models trained with full data, except for LLaMA-13B@10% and LLaMA-7B@100%. While we empirically observed that the training time grows approximate linearly with parameter count in our experiments, the results reveals that larger models should be preferred with limited training compute budget.

6.3 Edit Ratios

The accuracy of finetuned LLaMA models across varying levels of edit ratio are illustrated in Figure 4. The edit ratios are equally divided into five bins, and it has been observed that larger models consistently outperforms smaller ones within each bin. Interestingly, the accuracy of models' edit is generally lower as the edit ratio decreases. One plausible reason is that, as the fine-tuning loss is the average of cross-entropy on the label tokens, a shortcut of copying the inputs is easily learnt by the model to achieve a fairly low loss value, especially when the absolute number of modifications is small. Our observations indicate that this issue can be alleviated by scaling up the models. Larger models perform better in capturing subtle differences in low edit ratio cases.

7 Conclusion

We introduce CodeInstruct, the first instruction tuning dataset for general-purpose code-editing tasks. The dataset comprises generations of Large Language Models, 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 show that with computationally lightweight parameter-efficient finetuning, open-source models can gain huge improvements and even yield ChatGPT-like performance. We also reveal that the pretraining strategy and the scale of finetuning 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

While we chose genuine github commits as the source of our seed tasks, the data produced may still exhibit biases that deviate from real-world application. Moreover, our approach did not encompass code changes involving cross-files contexts, which might be the common case in development. We hope to explore these aspects further and incorporate additional programming languages in our future research.

Ethics Statement

This research paper adheres to the ethical guidelines and principles set forth by the Conference on Empirical Methods in Natural Language Processing (EMNLP) and the wider scientific community. All real-world data were collected only from public repositories from GitHub.

Acknowledgements

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.

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.

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

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.

DataBricks. 2023. Databricks' dolly, a large language model trained on the databricks machine learning platform. `https://github.com/databrickslabs/dolly`.

Eric J. Wang et. al. 2023. Alpaca-lora. `https://github.com/tloen/alpaca-lora`.

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*.

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*.

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*.

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

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

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), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 3470–3487. Association for Computational Linguistics.

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

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

OpenAI. 2023. Gpt-4 technical report.

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

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`.

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

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.

Haochun Wang, Chi Liu, Nuwa Xi, Zewen Qiang, Sendong Zhao, Bing Qin, and Ting Liu. 2023a. Huatuo: Tuning llama model with chinese medical knowledge.

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, et al. 2022b. Supernaturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109.

Zhen Wang, Rameswar Panda, Leonid Karlinsky, Rogerio Feris, Huan Sun, and Yoon Kim. 2023b. 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.

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`.

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.

A Analysis on Alignment of GPT-4 and Human Evaluation

Using the scoring standard described in 5.3, we conducted human evaluation on the test set code edits sampled from ChatGPT and LLaMA fine-tuned on CodeInstruct. The results is compared to the evaluation results of GPT-4. When considering the partial type of human scoring as correct, our observations reveal an average consistency ratio of 68.2%, and on the largest evaluated model, LLaMA-33B, the value rises to 78.4%. This renders GPT-4 evaluation as an acceptable method for evaluating the correctness of code edit tasks.

B Additional statistics of CodeInstruct

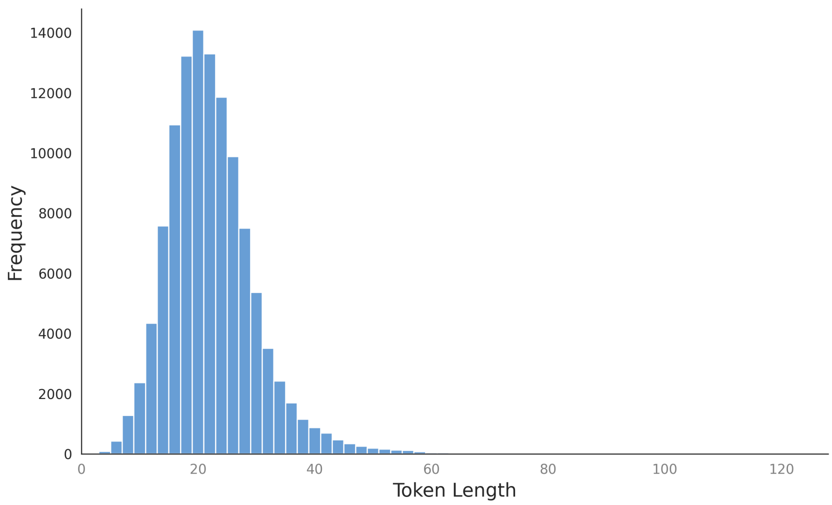


Figure 5: Instruction length distribution of CodeInstruct

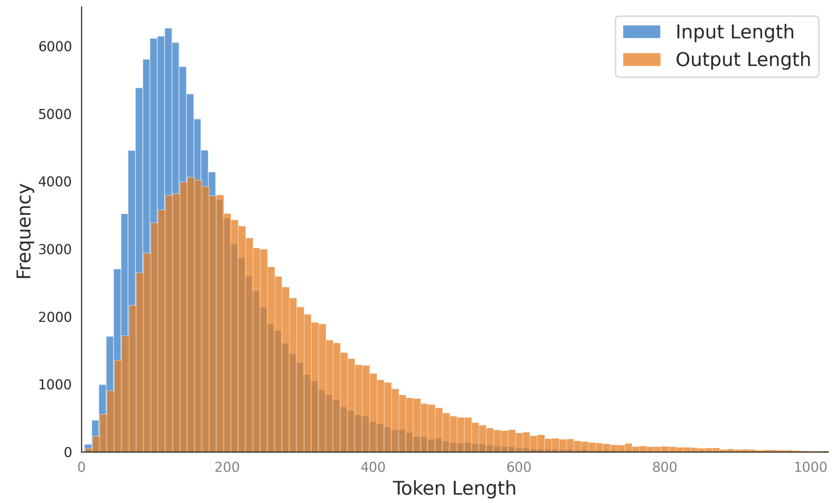


Figure 6: Input length distribution of CodeInstruct

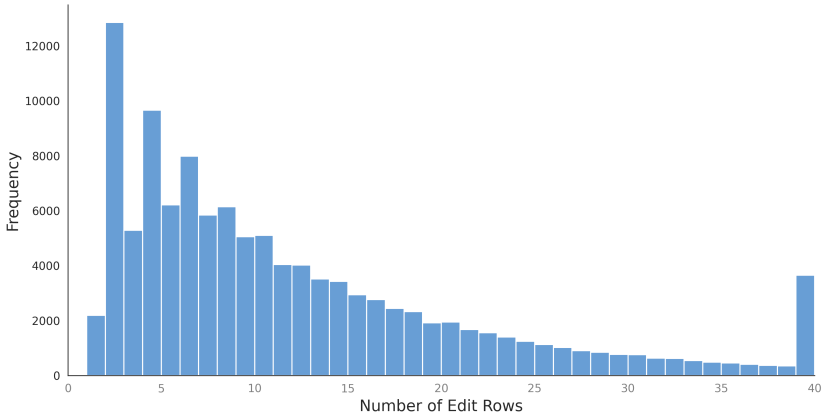


Figure 7: Edit rows distribution of CodeInstruct (Number greater than 40 are aggregated.)

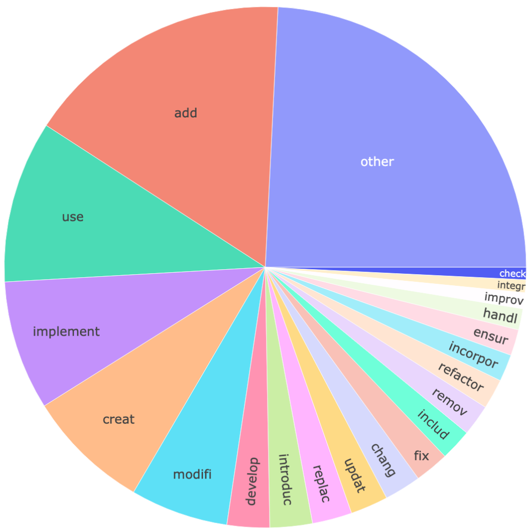


Figure 8: Verb stem distribution

C Hyperparameters

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 5: Hyperparameters used for finetuning language models.

D Prompts

We showcase the prompts we used in our data collection and experiments.

D.1 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 instruction.

D.2 Scenario Generation

Given a python code editing task, please come up with 10 diverse scenarios concise description where this python code editing task could be performed or come from.

D.3 Input and Output Code 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 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 input code.

D.4 GPT-4 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".

E Qualitative Examples Generated by Finetuned LLaMA-33B

```
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:
Add a function to check the file size after saving and raise an error if it exceeds
     the maximum allowed size of 1MB.

### Input:
import os

def save_file(file_path, data):
    with open(file_path, 'wb') as file:
        file.write(data)

file_path = "example.txt"
data = b"Hello, world!"
save_file(file_path, data)
file_size = os.path.getsize(file_path)
print(f"File size: {file_size} bytes")

### Response:
import os

MAX_FILE_SIZE = 1000000 # 1MB

def save_file(file_path, data):
    with open(file_path, 'wb') as file:
        file.write(data)
    file_size = os.path.getsize(file_path)
    if file_size > MAX_FILE_SIZE:
        raise ValueError("File size exceeds maximum allowed size of 1MB.")

file_path = "example.txt"
```

```python
data = b"Hello, world!"
save_file(file_path, data)
file_size = os.path.getsize(file_path)
print(f"File size: {file_size} bytes")
```

F Qualitative Examples of Scenario-Conditional Generation

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

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

Output:
```python
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:
```python
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:
```python
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 instances generated without scenario (Left) and with scenario (Right)