Success is in the details: Test and Improve Sensitive to Details of Code LLMs through Counterfactuals

Anonymous ACL submission

Abstract

Code Sensitivity refers to the ability of Code LLMs to recognize and respond to details changes in problem descriptions, serving as an important dimension for evaluating Code LLM's ability. While current code benchmarks and instruction tuning methods primarily focus on difficulty and diversity, sensitivity is overlooked. We first introduce the CTF-Code benchmark, constructed using counterfactual perturbations—minimizing input changes while maximizing output changes. The evaluation shows that many models have a more than 10% performance drop in performance on CTF-Code. To fully utilize sensitivity, we propose the CTF-Instruct incremental instruction fine-tuning framework, which extends on existing data and uses a selection mechanism to meet the three dimensions of difficulty, diversity, and sensitivity. Experiments show that models fine-tuned with CTF-Instruct data achieve over a 2% improvement on CTF-Code, and more than a 10% performance boost on LiveCodeBench, validating the feasibility of enhancing LLMs sensitivity to improve performance.

1 Introduction

Code Large Language Models (LLMs) are LLMs pre-trained on extensive code corpora, demonstrating remarkable capabilities in code generation, debugging, translation and so on (Hui et al., 2024; DeepSeek-AI et al., 2024). Not only do they significantly improve software development efficiency, but also their ability to tackle complex programming tasks is a key indicator of their thinking abilities (OpenAI, 2024). Notably, as a logically complete symbolic system (MacLennan, 1986), code requires LLMs to accurately map between requirements and algorithmic logic during the generation process (Pressman, 2005), a small mismatch will cause the whole task to fail. In Figure 1, changing the description from '**add** one' to '**double** one' alters the underlying algorithmic logic entirely. As

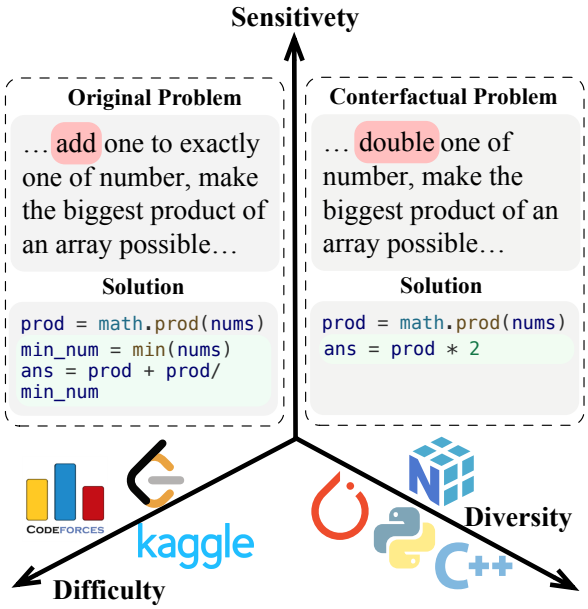

Figure 1: Code LLMs have many studies on diversity and difficulty, but sensitivity to problem details remains underexplored. For example, changing a single word in a problem can completely alter the algorithmic logic. In the original problem, we should increase the smallest number, whereas in the counterfactual version, no matter which number is modified, the result remains the same—double the cumulative sum.

such, the model's sensitivity to detail becomes a crucial measure of its ability. However, the ability of Code LLMs to capture and address such fine-grained differences remains unclear. Currently, code generation benchmarks mainly focus on difficulty and diversity. The difficulty levels range from simple functions to more complex classes and competition-level algorithms (Liu et al., 2023; Du et al., 2024; Jain et al., 2024). Diversity spans domains like data science, system development, and interdisciplinary applications (Zhuo et al., 2024; Lai et al., 2023a; Hu et al., 2024; Liu et al., 2024). However, these benchmarks are often discrete, evaluating the model's ability to solve isolated problems, without assessing its sensitivity to subtle dif-

ferences in requirement details.

This limitation is also present in current approaches to code instruction fine-tuning. Difficulty and diversity are mainstream to import data quality (Wang et al., 2024b): first, by incrementally introducing constraints to create high-difficulty datasets (Luo et al., 2024b; Wang et al., 2024a), though these manually designed iterative rules and inherent biases may cause data distributions to diverge from real-world scenarios (Wei et al., 2024b); second, by rewriting or inspiring from real code to generate diverse data, enhancing domain coverage (Wei et al., 2024b; Luo et al., 2024a; Wei et al., 2024a; Yu et al., 2024). There is a lack of work targeted at improving the model's sensitivity to crucial details.

Take difficulty and diversity as two evaluated and improved dimensions, a critical dimension remains underexplored: sensitivity. Unlike difficulty, changes in details do not necessarily change algorithmic complexity; unlike diversity, little change does not constitute a new task scenario. In NLP, counterfactuals (CTF) involve making minimal changes to inputs to produce outputs that differ substantially (Chen et al., 2023; Sachdeva et al., 2024; Wang et al., 2024c). Inspired by this, we propose the CTF-Code benchmark and CTF-Instruct pipeline. By introducing minimal semantic perturbations such as altering key conditions or replacing critical constraints, we modify the algorithmic logic while maintaining superficial task similarity, thereby assessing the model's sensitivity to requirement details. Experiments reveal that state-of-the-art models like GPT-4o and Qwen2.5-Coder (Hurst et al., 2024; Hui et al., 2024) experience performance drops exceeding 10% on CTF-Code compared to original problems, highlighting significant 'blind spots' in detail sensitivity.

To address this, starting from one existing dimension of data, CTF pairs are generated to cover the sensitivity dimension and then through a selection mechanism to enhance the last dimension. Lastly, by merging selected data and existing one-dimensional data, CTF-Instruct data is three-dimensions-completed. Experiments show that LLMs fine-tuned with CTF-Instruct data achieve a 2.6% improvement on CTF-Code, and also gains on other benchmarks such as HumanEval+ (+4.2%), BigCodeBench-hard (+5.2%), and LiveCodeBench (+11.6%) (Liu et al., 2023; Zhuo et al., 2024; Jain et al., 2024), confirming the help of sensitivity to instruction tuning.

Our contribution is summarized below:

- We propose CTF-Code, the first benchmark focused on sensitivity, and the evaluation results expose the shortcomings of mainstream Code LLMs in understanding requirement details.
- We design a three-dimensional-completed data generation framework, starting from one dimension, completing sensitivity by generation and the last dimension by selection.
- LLMs trained with CTF-Instruct data achieve substantial performance improvements across CTF-Code and other benchmarks compared to existing methods.

2 Related Work

Code Benchmark Existing code generation benchmarks primarily include two dimensions: (1) Difficulty: from function-level like HumanEval and MBPP (Austin et al., 2021; Chen et al., 2021), to class-level like ClassEval (Du et al., 2023), and contest-level like LiveCodeBench (Jain et al., 2024); (2) Diversity: BigCodeBench (Zhuo et al., 2024) focuses on Python package usage, while DS-1000 (Lai et al., 2023b) targets on data science. The existing evaluation paradigm is limited to solving isolated problems. In this work, the first sensitivity benchmark, CTF-Code is introduced.

Code Instruction Tuning Datasets With the development of Self-Instruct (Wang et al., 2023), methods on Code LLMs mainly focus on difficulty enhancement and diversity expansion. Inspired by Evol-Instruct (Xu et al., 2023), Luo et al. (2024b) increases the difficulty of data by adding constraints based on seed data CodeAlpaca (Chaudhary, 2023). Muennighoff et al. (2024) collects nearly 4TB Git commits data and constructs a 2GB instruction tuning dataset based on the commits data. While Evol-Instruct requires seed data, which limits the diversity of generated instruction tuning data, Oss-Instruct (Wei et al., 2024b) and CodeOcean (Yu et al., 2024) rewrite real-world data to better align real distributions, thereby avoiding model bias and enhancing the diversity. While these methods have achieved significant success in their respective dimensions, they overlook the usage and combination of sensitivity.

Counterfactual in NLP Counterfactuals in NLP is to explore the model's output variation patterns

through minimal semantic perturbations (Robeer et al., 2021; Nguyen et al., 2024; Sachdeva et al., 2024; Wang et al., 2024c). Existing studies in the code domain mainly focus on local modifications to code (Hooda et al.), testing the model's ability to differentiate and understand counterfactual code (Gu et al., 2024; Cito et al., 2022). These approaches remain limited to perturbations to code and fail to address the core challenge of code generation tasks—the precise mapping between requirement details and algorithmic implementations. We fill this gap by constructing a new benchmark and dataset based on counterfactuals.

3 CTF-Code Benchmark

3.1 Formal Definition

Evaluation for code tasks is test-driven, with its basic unit formalized as a tuple $\mathcal{P} = (Q, T, S)$, where Q is the problem description (requirement), $T = \{t_i\}_{i=1}^{n}$ is a set of test cases $t_i = (\text{input}_i, \text{output}_i)$, and S is a solution satisfying

$$\forall t_i \in T, S(\text{input}_i) = \text{output}_i.$$

Based on this, the goal of constructing CTF-Code can be formulated as an optimization problem: given the original problem $\mathcal{P}$, generate $\mathcal{P}' = (Q', T', S')$ such that

$$\begin{aligned} \underset{Q', S'}{\text{maximize}} \quad & \mathcal{D}_S(S, S') \\ \text{subject to} \quad & \mathcal{D}_Q(Q, Q') \leq \epsilon \end{aligned} \tag{1}$$

where $\mathcal{D}_Q : Q \times Q' \to [0, 1]$ is the problem similarity function, $\mathcal{D}_S : S \times S' \to \mathbb{N}$ is a code difference function, and ϵ is the similarity threshold. This optimization objective ensures that Q' is highly similar to Q, while S' and S differ significantly. After obtaining Q' and S', T' is generated for evaluation.

3.2 Benchmark Construction

As shown in Figure 2, the construction of CTF-Code follows a three-phase paradigm: First, select problems that have large semantic space as original data. Then, semantic perturbations are performed to generate CTF data and derive CTF pairs based on the optimization objective. Finally, construct the CTF testcases while ensuring no data interference.

Origianl Data Selection The first step is to obtain the original data Q, S, and T. LiveCodeBench (LCB) (Jain et al., 2024), which collects problems designed by algorithmic experts from three online judges—LeetCode, Codeforces, and AtCoder, are selected. First, problem descriptions in LCB are much clearer for the inclusion of input/output format and example explanations. Second, their average problem length is 210.5 words compared to 71.6 words for Humaneval in Table 1. The longer context gives them more semantic space for perturbations. Third, complex problems require the model to process multi-dimensional information while maintaining contextual consistency. Last, algorithmic competition problems require participants to meticulously consider every detail and boundary condition, where even small deviations can lead to wrong answers. This aligns perfectly with our goal of assessing LLMs' sensitivity to details. Since the original benchmark only provides Q and T, we supplement it with S and finally get complete original data.

	Acc.	Problems	Length	Expl
Humaneval	**96.3**	164	71.6	✗
LCB-Easy	95.6	**15**	**210.5**	✓

Table 1: Comparison between HumanEval and LiveCodeBench (LCB) -Easy. **Acc.** represents the Pass@1 score of o1-mini on both benchmarks. **Problems** indicates the number of problems, **Length** represents the average word count per problem, and **Expl** shows whether including example explanations.

CTF Pair Generation This step aims to generate Q' and S'. Given the complexity of solving the bi-objective optimization problem defined in Equation 1 in the discrete semantic spaces of language, a heuristic generation-selection strategy is proposed to approximate the optimal solution. Based on Q, a collection of LLMs parallelly sample K candidates $\{\hat{Q}'_k\}_{k=1}^{K}$. However, $\hat{Q}'$ may be unsolvable or too difficult, so several algorithm experts are invited for annotation, retaining only the candidates that satisfy the requirements. Next, the similarity between each $(\hat{Q}', Q)$ is assessed, and exclude candidates where $\mathcal{D}_Q(Q, \hat{Q}'_k) \geq \epsilon$, ensuring that $\hat{Q}'$ only differs from the original problem in key conditions, i.e., only a single sentence or a few words. After completing $\hat{S}'_k$, we select the perturbation pair that maximizes the objective function:

$$(Q', S') = \underset{(\hat{Q}'_k, \hat{S}'_k)}{\arg\max} \left[\mathcal{D}_S(S, \hat{S}'_k) - \lambda \mathcal{D}_Q(Q, \hat{Q}'_k) \right] \tag{2}$$

where λ is a scaling factor that ensures $\mathcal{D}_S$ and $\mathcal{D}_Q$ can compute. The annotation documents are

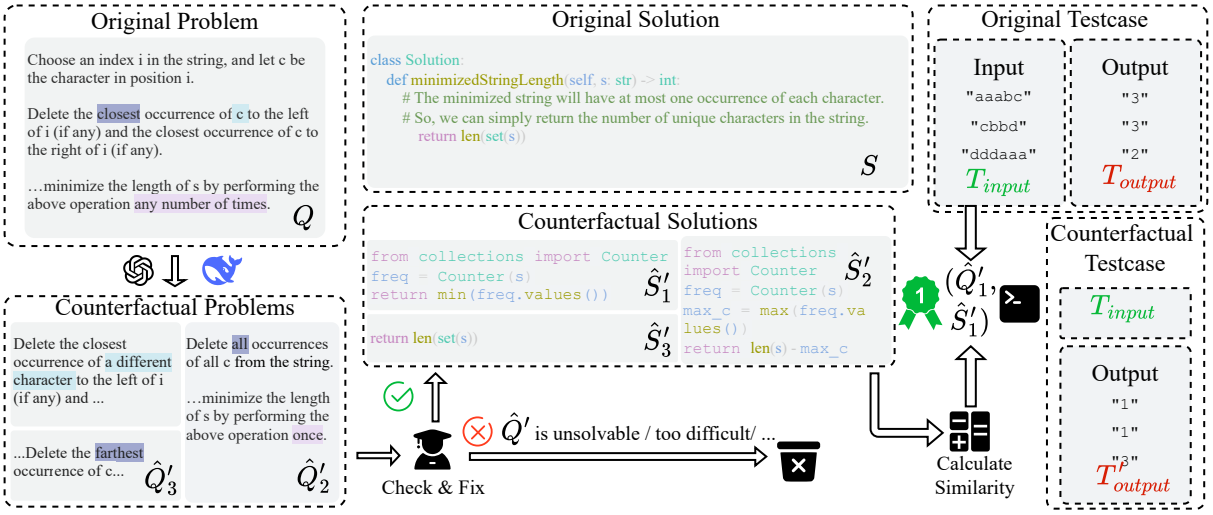

Figure 2: The pipeline of CTF-Code benchmark construction. First, original problems are selected and then sent to LLMs to sample semantic permutations on the problem description. Algorithmic experts will carefully check the CTF problems and decide to drop or fix them to generate CTF solutions. After selecting the most suitable CTF problem, its testcases are constructed by executing its solution on the inputs from the original testcases.

provided in Appendix C. Through this heuristic rule, we obtain an approximate optimal Q' and S'.

CTF Testcase Completion To ensure the performance shake of LLMs latter only from details change between (Q, Q'), a dual-constraint test case generation mechanism is designed to avoid the influence from (T, T'): **Input Space Inheritance**: We retain the original testcases' input distribution, i.e., $T'_{\text{input}} = T_{\text{input}} = \{input_i\}_{i=1}^{n}$. **Output Space Reconstruction**: The expected output is generated based on the new solution S', i.e., for each $\text{input}_i \in T_{\text{input}}$, $\text{output}'_i = S'(\text{input}_i)$. Finally, $T' = \{(\text{input}_i, \text{output}'_i)\}_{i=1}^{n}$ is constructed. The data distribution interference is eliminated by fixing the input variables, and the correctness of the test case is ensured by the correctness of S'. Additionally, fixed inputs enable backtracking when the LLM behavior differs between Q, Q'.

Compared to the traditional code benchmarks that evaluate isolated problems, CTF-Code introduces paired data with only details differences to enable analysis of sensitivity firstly.

4 CTF-Instruct

Existing instruction datasets have effectively addressed the challenges of data difficulty and diversity (Lai et al., 2023b; Jain et al., 2024; Zhuo et al., 2024). However, the detail sensitivity hasn't been explored. We assume that sensitivity, difficulty and diversity are orthogonal and propose an incremental data construction approach. Starting with datasets that satisfy a single dimension, we generate sensitivity data through counterfactual perturbations. Then, a selection algorithm is designed based on the third dimension, ultimately constructing a dataset that satisfies all three dimensions' requirements efficiently. An example of the data distribution of the process is in Figure 7.

4.1 Generation

Sensitivity inherently requires a paired format, as only through comparison can the subtle differences in details be effectively highlighted. Generating paired sensitivity data from scratch is challenging. When using Self-Instruct (Wang et al., 2023), LLMs tend to generate simpler, regular data. Moreover, the inherent limitations of LLM's context window prevent it from recognizing previously generated data, leading to huge repetition. As the assume that sensitivity is orthogonal to other dimensions, incrementally expanding the data becomes a more efficient approach. The existing single-dimensional dataset $\mathcal{D}_{base}$ serves as high-quality seed data. For each sample (Q, S), counterfactual perturbations are applied to generate sensitivity pair (Q', S'). This process generates the sensitivity dimension data $\mathcal{D}_{sens}$.

After generation, some filtering rules are performed, such as removing duplicates from benchmarks to avoid data leakage and eliminate data with generation failures like noise samples.

Algorithm 1 K-Center Greedy Selection

Input: Sensitivity data $\mathcal{D}_{sens}$, needed data amount τ, original data $\mathcal{D}_{base}$
Output: Set $\mathcal{D}_{sub} \subseteq \mathcal{D}_{sens}$ of τ data
$C \leftarrow \mathcal{D}_{base}$ ▷ Initialize centers
for $i = 1$ to k **do**
  $\text{dist}_x \leftarrow \min_{y \in \mathcal{D}_{base} \cup \mathcal{D}_{sub}} ||\phi(x) - \phi(y)||_2$
  $x \leftarrow \arg\max_{x \in \mathcal{D}_{sens}} \text{dist}_x$ ▷ Select the farthest data x
  $\mathcal{D}_{sub} \leftarrow \mathcal{D}_{sub} \cup \{x\}$ ▷ Update centers
  $\mathcal{D}_{sens} \leftarrow \mathcal{D}_{sens} - \{x\}$ ▷ Update data
end for
Return $\mathcal{D}_{sub}$ ▷ Return the set of τ data

4.2 Selection

After obtaining $\mathcal{D}_{base}$ and $\mathcal{D}_{sens}$, the original dimensions and sensitivity are already addressed. Since the three dimensions are orthogonal, directly merging them would amplify the shortcomings of $\mathcal{D}_{base}$ in the third dimension. Conversely, by filtering subsets, the distribution of $\mathcal{D}_{sens}$ can be shifted towards the third dimension, thereby covering all three dimensions. Specifically, the dimension projection function is defined as:

$$\Psi(\mathcal{D}) = [\psi_{\text{diff}}, \psi_{\text{dive}}, \psi_{\text{sens}}] \tag{3}$$

where ψ_{diff} reflects data difficulty, ψ_{dive} measures semantic space coverage, and ψ_{sens} evaluates the quantity of sensitivity data. The filtering process is equivalent to solving:

$$\begin{aligned} \max_{\mathcal{D}_{sub} \subseteq \mathcal{D}_{sens}} \quad & \psi_{\text{target}}(\mathcal{D}_{base} \cup \mathcal{D}_{sub}) \\ \text{s.t.} \quad & \psi_{\text{sens}}(\mathcal{D}_{sub}) \geq \tau. \end{aligned} \tag{4}$$

where ψ_{target} represents the dimension to be completed. This optimization problem ensures that the selected subset $\mathcal{D}_{sub}$ not only fills the target dimension but also retains the original sensitivity advantage. Since $\psi_{\text{sens}}(\mathcal{D}_{sub}) = |\mathcal{D}_{sub}|$, ψ_{sens} is sampled multiple times and the optimal value is selected based on experimental results.

The data is mapped into the semantic space through a small LLM. To make it familiar with the data distribution, it is lightly fine-tuned. Then (Q, S) is fed in the extractor, and the average hidden state vectors of all tokens in S from the final layer are taken as the semantic vector. When $\mathcal{D}_{base}$'s dimension is the difficulty, the diversity needs to be filled. The K-Center Greedy algorithm is used to select the samples most distant from $\mathcal{D}_{base}$ in semantic space $\phi(\cdot)$. The objective function is:

$$\max \psi_{\text{dive}}(\mathcal{D}_{sub}) = \max_{\mathcal{D}_{sub}} \min_{\substack{x \in \mathcal{D}_{sub} \\ y \in \mathcal{D}_{base}}} ||\phi(x) - \phi(y)||_2 \tag{5}$$

The detailed algorithm is shown in Algorithm 1. Since $\psi_{\text{dive}}(\mathcal{D}_{sub})$ is clearly a monotonically non-increasing function with respect to $|\mathcal{D}_{sub}|$, the final $|\mathcal{D}_{sub}| = \tau$, since the sensitivity should be retained. When $\mathcal{D}_{base}$ is dominated by diversity, difficulty needs to be completed. A fine-tuned LLM is used as a scorer. ψ_{diff} is simplified as $\min_{x \in \mathcal{D}_{base}} \varphi(x)$, where $\varphi(\cdot)$ represents the difficulty score of the data. When sorting by $\varphi(\cdot)$, ψ_{diff} is also a monotonically non-increasing function with respect to $|\mathcal{D}_{sub}|$. Thus, the subset $\mathcal{D}_{sub}$ of size τ is selected to satisfy the difficulty dimension. Based on the existing data, the sensitivity data has been expanded; then, filtering algorithms are designed to select sensitivity data that fill the missing dimensions. Regardless of the original dimension, data that satisfies diversity, difficulty, and sensitivity simultaneously are ultimately obtained.

5 Experiment

5.1 CTF-Code Benchmark

Setup During calculating the similarity between the original data (Q, S) and CTF data (Q', S'), $\mathcal{D}_Q$ and $\mathcal{D}_S$ are (1-BLUE) and (1-CodeBLUE), respectively. For generating $\hat{Q}'$, we sample five times from each of the models `gpt-4o-2024-08-06`, `gpt-4-turbo-2024-04-09`, and `o1-mini` (OpenAI, 2024) to enhance diversity. Four competition programmers are invited to annotate considering the task difficulty, each of whom has represented their university in at least one ICPC [1] competition and earned at least a bronze medal. During the construction of the benchmark, problems that fail to meet any of the required criteria are discarded. Ultimately, CTF-Code curated a set of 186 problems. Models evaluated are in Appendix B.

Evaluation As shown in Figure 3, almost all of the mainstream models show some degree of degradation on CTF-Code compared to the original dataset. This suggests that existing models still have some difficulty in understanding the details of the input problem. It is worth noting that only the Claude model still shows some performance gains under the CTF-Code, which is a good proof of it's

[1] International Collegiate Programming Contest

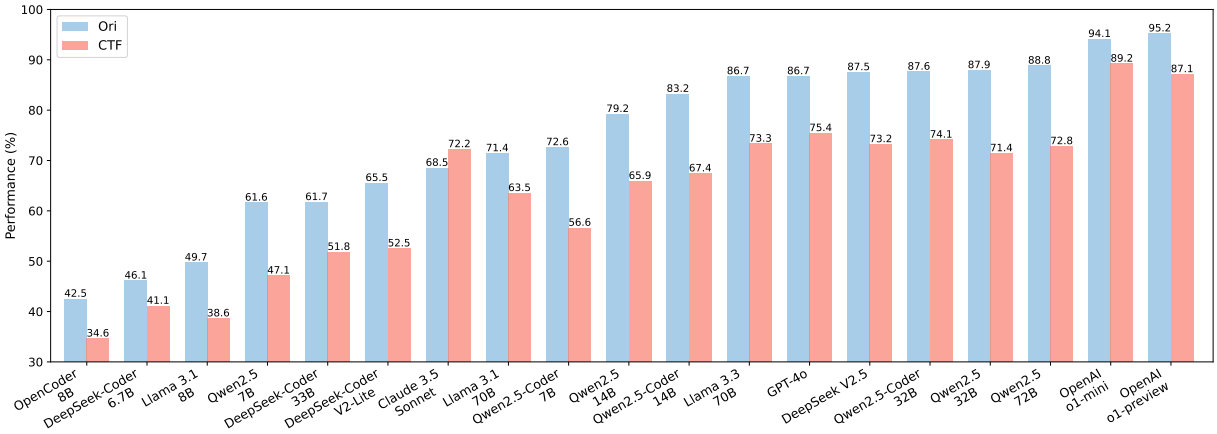

Figure 3: The evaluation results of Code LLMs on CTF-Code.

Base	Model	EvalPlus	LiveCodeBench		BigCodeBench		CTF-Code	
		HumanEval (+)	All	Easy	Full	Hard	Ori	CTF
DeepSeek Coder 6.7B	DC-6.7B-Instruct	74.4 (71.3)	18.9	45.3	35.5	10.1	45.8	38.1
	Wavecoder	75.0 (69.5)	18.9	46.0	33.9	12.8	47.7	39.2
	Inversecoder	76.2 (72.0)	18.1	43.1	35.9	10.8	47.8	39.1
	Magicoder	76.8 (71.3)	19.2	46.6	36.2	13.5	48.8	43.4
	CTFCoder	**78.7** (**75.0**)	**21.4**	**53.3**	**37.6**	**14.2**	**52.8**	**44.5**
	$\text{CTFCoder}_{\text{oss}}$	71.3 (65.9)	18.3	46.6	37.0	12.2	51.4	43.1
Qwen2.5 Coder 14B	Evol	85.4 (79.3)	23.9	71.5	43.7	14.2	76.3	59.9
	CTF	**88.4** (**80.5**)	**24.6**	**74.1**	44.1	**17.6**	**79.5**	**60.8**
	w/o select	85.4 (78.0)	24.1	72.8	**44.2**	16.2	76.4	60.2
	Oss	84.1 (77.4)	20.6	61.8	42.0	12.2	75.5	58.6
	CTF_{oss}	86.0 (79.9)	**22.3**	**67.9**	**42.5**	**18.9**	**78.2**	**60.0**
	w/o select	**86.6** (**80.5**)	20.8	67.2	**42.5**	14.9	76.8	59.2

Table 2: Performance comparison of CTFCoder with other models. To avoid environmental discrepancies, the official leaderboard results are presented. Only when results are missing, local testing are conducted. CTFCoder represents for CTF-Instruct based on Evol-Instruct while $\text{CTFCoder}_{\text{oss}}$ represents based on Oss-Instruct. For Qwen 2.5 Coder 14B, Model represents the instruction datasets they finetuned on. 'w/o select' means original data mix random selected CTF-Instruct without methods in Section 4.2.

capabilities of coding. Specifically, by analyzing the error cases we find that the models are most likely to make mistakes when confronted with details such as modified boundary conditions. This is because detailed modifications to code problems usually directly change the boundary cases of the original problem or add some additional restrictions that lead to incorrect model judgments.

5.2 Instruction Tuning

Setup Evol-Instruct (Luo et al., 2024b) and Oss-Instruct (Wei et al., 2024b) are selected as the original datasets for difficulty (110k samples) and diversity(75k samples), respectively. Sensitivity data are generated using `gpt-4-turbo-2024-04-09`, with failed data removed, resulting in 108k and 73k sensitivity samples. During data filtering, the open-source model XCoder-Complexity-Scorer (Wang et al., 2024b) is used to score difficulty, and semantic vectors are extracted using Deepseek Coder 1.3B Base (Guo et al., 2024) after one epoch of SFT on the code-feedback (Zheng et al., 2024) dataset. After selection, 30k sensitivity samples are added to Evol-Instruct and 10k to Oss-Instruct, named CTF and CTF_{oss}, respectively. After SFT with Deepseek Coder 6.7B base, CTFCoder and $\text{CTFCoder}_{\text{oss}}$ are obtained. Qwen 2.5 Coder 14B Base (Hui et al., 2024) is also tuned. During training, the batch size is 512 and the sequence length is 2048. The initial learning rate is 2e-5 with 10

warmup steps and the learning rate scheduler is cosine. A100-80GB is used to finetune for 3 epochs.

Baseline & Benchmark CTFCoder is compared with other well-known models tuned on Deepseek Coder 6.7B Base, including Deepseek Coder 6.7B Instruct (Guo et al., 2024), Magicoder (Wei et al., 2024b), Wavecoder (Yu et al., 2024), and Inversecoder (Wu et al., 2024). Magicoder first trains for 2 epochs using oss-instruct to enhance diversity, then uses 1 epoch of evol-instruct to improve difficulty; Wavecoder rewrites function code docstrings into instruct format and uses LLMs to increase difficulty; Inversecoder generates instructions based on evol-instruct output and filters them using a fine-tuned model. All models address both difficulty and diversity dimensions. For Qwen 2.5 Coder, the original Evol-Instruct and Oss-Instruct are used for SFT, serving as the baseline.

For the benchmarks, Humaneval, Humaneval (+), and LiveCodeBench are selected to cover various difficulty levels. Humaneval+ from Evalplus (Liu et al., 2023) extends Humaneval by adding a large number of test cases to cover corner cases. LiveCodeBench includes three difficulty levels: easy, medium, and hard (with easy being more challenging than Humaneval). Since GPT-4-turbo's training data ends in December 2023, we test LiveCodeBench questions from January 2024 onwards. For diversity, BigCodeBench, which specializes in Python package usage, is selected to complement the tests. Additionally, BigCodeBench selects high-difficulty sub-data to form a Hard subset.

5.3 Results

Table 2 shows the performance comparison between CTFCoder and other models. CTFCoder demonstrates consistent performance improvements across all benchmarks. Although previous models already cover difficulty and diversity and achieve strong performance, the addition of sensitivity acts like a further 'activation'. CTFCoder shows significant improvements across all three dimensions. It has a nearly 3% improvement on CTF-Code, indicating that CTF indeed helps the model pay more attention to details. On harder benchmarks, Humaneval+, BigCodeBench-Hard, and LiveCodeBench, CTFCoder achieves over 4%, 5%, and 11% performance improvements, respectively. CTF, building upon the difficulty dimension of Evol-Instruct, results in further enhancement. This illustrates that generating sensitivity data using existing data as seeds not only preserves the original data dimensions but can even trigger further improvements.

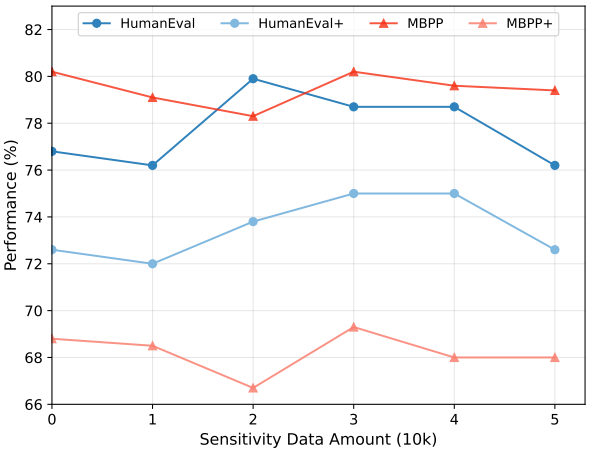

Figure 4: The change in model performance as sensitivity data joined into Evol-Instruct.

Even though $\text{CTFCoder}_{\text{oss}}$ has a relatively small amount of SFT data, CTF_{oss} helps it outperform other models on LiveCodeBench-Easy and BigCodeBench-Full, reflecting the 'activation' effect on diversity works, too. On Qwen 2.5 Coder 14B, compare the baseline, random selection of CTF-Instruct data ('w/o select') and CTF generally shows a progressive performance improvement, highlighting the effectiveness of sensitivity data and the importance of data selection.

6 Discussion

There exists an optimal range for the amount of sensitivity data. Figure 4 shows the performance trend when sensitivity data is gradually mixed into Evol-Instruct (110K), with the performance evolving in three stages: an initial decline, a mid-stage increase, and a final decline. The initial drop indicates that a certain amount of sensitivity data is required to have an effect, which verifies the assumption in Equation 4, where a threshold τ ensures a minimum quantity of sensitivity data. The subsequent rise followed by a decline suggests that there is an upper limit for sensitivity data, confirming our observation that directly merging sensitivity and original data dimensions exacerbates the lack of the third dimension. Figure 8 in Appendix B also shows the results for Oss-Instruct (75k).

Influence of sensitive data mount **The effectiveness of the selection strategy is universal.** Table 2 and Figure 5 compare the performance of different models and data using the selection strategy

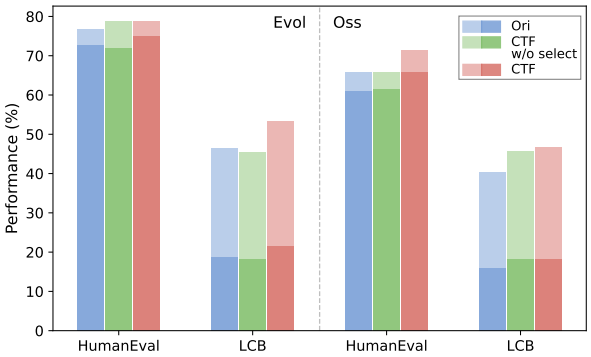

Figure 5: The performance change brought by the selection strategy on Evol-Instruct and Oss-Instruct. The darker shade in Humaneval represents Humaneval+, while in LCB, the darker shade represents LCB-All, and the lighter shade represents LCB-Easy.

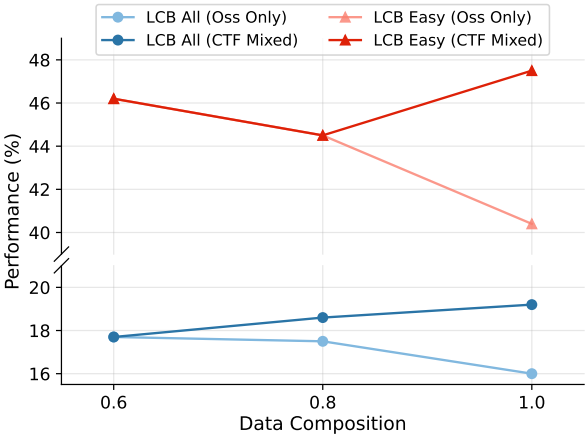

Figure 6: Performance changes with the increase in diversity data (Oss only) and the gradual injection of sensitivity into the diversity data (CTF Mixed).

versus not using it ('w/o select') with the same amount of data. Regardless of the original data or base model, the strategy generally leads to performance improvement. Figure 5 shows that, with Evol-Instruct, performance on LiveCodeBench improved by over 17%, while for OSS-Instruct, performance on Humaneval increased by more than 7% compared to 'w/o select'. This validates our hypothesis that data offset can effectively address the third dimension.

Epoch	Strategy	HE (+)	LCB (Easy)
2	2+0	74.4 (69.5)	18.5 (46.8)
	1+1	**79.9** (**75.0**)	**21.0** (**51.8**)
	2+0	65.9 (61.0)	16.0 (40.4)
	1+1	**66.5** (**61.6**)	**18.8** (**46.9**)
3	3+0	**76.8** (72.6)	18.6 (46.4)
	2+1	**76.8** (**73.2**)	**20.7** (**51.3**)
	3+0	65.2 (59.8)	17.0 (42.5)
	2+1	**68.3** (**62.2**)	**19.0** (**47.2**)

Table 3: The results of continual training with CTF. Epoch is the total number of training epochs, and 'x+y' indicates that the model is first trained for x epochs on the original data, followed by y epochs on CTF-Instruct. HE represents Humaneval, and LCB refers to LiveCodeBench-All, with 'Easy' inside the parentheses.

Since the amount of data used for training the open-source models in the main experiment differs, we designed a controlled experiment to verify the independent gain from the sensitivity dimension. In Figure 6, when the total training data volume is fixed, replacing 40% of the original Oss-instruct data with randomly selected CTF data led to a 20% improvement on LCB-All, whereas simply increasing the Oss-instruct data volume caused a 9.5% performance drop. This suggests that **when the diversity dimension is saturated, and sensitivity can be considered a new 'performance engine'.**

Sensitivity can be directly used for continual training. Inspired by Magicoder (Wei et al., 2024b), in Table 3, after training on the original data for 1 or 2 epochs, an additional epoch of CTF-Instruct is added. Compared to continuing training with the original data alone, this approach shows a significant performance improvement. Particularly on LiveCodeBench, every setup achieves a 10% gain. This further demonstrates the orthogonality of the sensitivity dimension with the other two dimensions, as its benefit does not depend on joint training, allowing for efficient and convenient continual training to achieve gains.

7 Conclusion

Beyond diversity and difficulty, we introduced sensitivity as a key dimension for evaluating and improving Code LLMs. By constructing the CTF-Code benchmark, we revealed the shortcomings of existing Code LLMs in understanding details. To futher utilize sensitivity, we propose the CTF-Instruct framework, which generates sensitivity data based on existing dimensions to cover sensitivity and employs a filtering algorithm to shift towards the third dimension. Experiments show that CTF-Instruct data fine-tuned LLMs improves performance on CTF-Code and outperform existing open-source models on general code generation benchmarks, validating the universal benefits of sensitivity optimization for Code LLMs.

Limitation

Due to constraints in training resources and manpower, our work was limited to constructing a relatively modest set of CTF-Code problems, without exploring the potential for more complex or challenging examples. Additionally, the CTF-Instruct framework was not tested with multi-round generation, nor was it evaluated on larger, more advanced LLMs. While our experiments demonstrate the effectiveness of the proposed approach on the models tested, we acknowledge that the full potential of CTF-Instruct could be realized by scaling up the dataset and conducting more extensive fine-tuning experiments, particularly on models with greater capacity. Furthermore, the impact of training on larger models with more rounds of fine-tuning remains an open question and is a promising direction for future work.

References

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Zeming Chen, Qiyue Gao, Antoine Bosselut, Ashish Sabharwal, and Kyle Richardson. 2023. DISCO: Distilling counterfactuals with large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5514–5528, Toronto, Canada. Association for Computational Linguistics.

Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. 2022. Counterfactual explanations for models of code. In *Proceedings of the 44th international conference on software engineering: software engineering in practice*, pages 125–134.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Do large code models understand programming concepts? counterfactual analysis for code predicates.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023a. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023b. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Xu Wang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024a. Semi-instruct: Bridging natural-instruct and self-instruct for code large language models. *Preprint*, arXiv:2403.00338.

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

Bruce J. MacLennan. 1986. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, USA.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Van Bach Nguyen, Christin Seifert, and Jörg Schlötterer. 2024. CEval: A benchmark for evaluating counterfactual text generation. In *Proceedings of the 17th International Natural Language Generation Conference*, pages 55–69, Tokyo, Japan. Association for Computational Linguistics.

OpenAI. 2024. Openai o1 system card.

Roger S Pressman. 2005. Software engineering: a practitioner's approach. *Pressman and Associates*.

Marcel Robeer, Floris Bex, and Ad Feelders. 2021. Generating realistic natural language counterfactuals. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3611–3625, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Rachneet Sachdeva, Martin Tutek, and Iryna Gurevych. 2024. CATfOOD: Counterfactual augmented training for improving out-of-domain performance and calibration. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1876–1898, St. Julian's, Malta. Association for Computational Linguistics.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Yutao Mou, Mengdi Zhang, Jingang Wang, Xunliang Cai, et al. 2024a. Dolphcoder: Echo-locating code large language models with diverse and multi-objective instruction tuning. *arXiv preprint arXiv:2402.09136*.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma Gongque, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, et al. 2024b. How do your code llms perform? empowering code instruction tuning with really good data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14027–14043.

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

Yongjie Wang, Xiaoqi Qiu, Yu Yue, Xu Guo, Zhiwei Zeng, Yuhong Feng, and Zhiqi Shen. 2024c. A survey on natural language counterfactual generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4798–4818, Miami, Florida, USA. Association for Computational Linguistics.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024a. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and LINGMING ZHANG. 2024b. Magicoder: Empowering code generation with OSS-instruct. In *Forty-first International Conference on Machine Learning*.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024. Inversecoder: Self-improving instruction-tuned code llms with inverse-instruct. *Preprint*, arXiv:2407.05700.

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

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 5140–5153. Association for Computational Linguistics.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12834–12859. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *Preprint*, arXiv:2406.15877.

Appendix

A Example

B Experiment Results

Models A range of LLMs with a size of >7B (mainly focusing on Code LLMs) are selected for testing, including Qwen 2.5 Coder, Deepseek Coder v1 & v2, and OpenCoder. Some general-purpose LLMs with strong code capabilities, such as Qwen 2.5, Llama 3.1 & 3.3 are also tested. All of these models are tested on the Instruct version. Additionally, we tested closed-source models via APIs, including GPT-4o (`gpt-4o-2024-08-06`) and Claude-3.5-Sonnet (`claude-3-5-sonnet-20240620`). Finally, think-capable LLMs, o1-mini, and o1-preview are also tested.

BLEU Score

C Annotation Process

C.1 Mission Background

Today, large language models (LLMs) demonstrate remarkable capabilities in code generation. However, it remains unclear how well LLMs can capture the nuances of programming problem details, such as the distinction between "swapping any two characters" and "swapping two adjacent characters". Can LLMs accurately capture the differences between these two concepts? To investigate this, we propose to modify a set of original problems (LeetCode Easy Level) to construct a new set of **counterfactual** (**CTF**) problems. These CTF problems are designed to have minor textual differences from the original problems while yielding significantly different solutions. To minimize the effort required to construct this benchmark, we aim to ensure that CTF problems can utilize the test cases of the original problems without the need for reconstruction.

C.2 Annotation Content

The annotation process does not involve modifying the problem itself, as this task has already been done by the LLMs. Instead, the annotator's role is simply to evaluate whether the modified problem is correct and aligns with our requirements.

C.3 Annotator Requirements

The annotator requirements are summarized below.

- The annotator is required to have a solid foundation in algorithms and the ability to quickly solve LeetCode Easy-level problems.
- The annotator needs to have one hour of free time per day for one month.
- This mission requires 3 to 6 annotators.

C.4 Construction Workflow

To illustrate the construction workflow in detail, we will supplement it with an example.

1. Read the original problem and briefly explain the meaning of the original problem. As shown in Figure 10, the meaning of the original problem is: "Given a string consisting of three letters `'abc'` in any order, can `'abc'` appear after swapping any two characters at most once?"

2. Read and understand the newly automatically generated problem. If there are errors in the `Sample Input/Output` or in the `Test Cases`, correct them.

3. In comparison with the original problem, classify the new problem into three types (Bad, Robust, CTF) and explain what changes have been made.

    - Bad. The new problem has a significant vulnerability (logical vulnerability or conflict) and can not be a complete problem.
    - Robust. The new problem has only a different wording from the original question, i.e. the algorithm used by the new problem and the answer is exactly the same. As shown in Figure 11, this is a robust version of the original problem. After understand the meaning of the new problem, we can tell that the change is "any substring can be reversed".
      For the new problem, the total length of cards is 3. Reversing a substring of length 3 is equivalent to swapping the letters in positions 1 and 3, and position 2 will not be changed during the reversing process; reversing a substring of length 2 is equivalent to swapping adjacent letters in the original question. The operation of the original problem and the operation of the new problem are exactly the

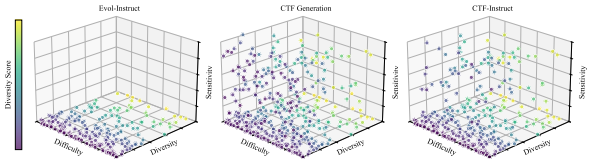

Figure 7: The data distribution change trace during the CTF-Instruct pipeline.

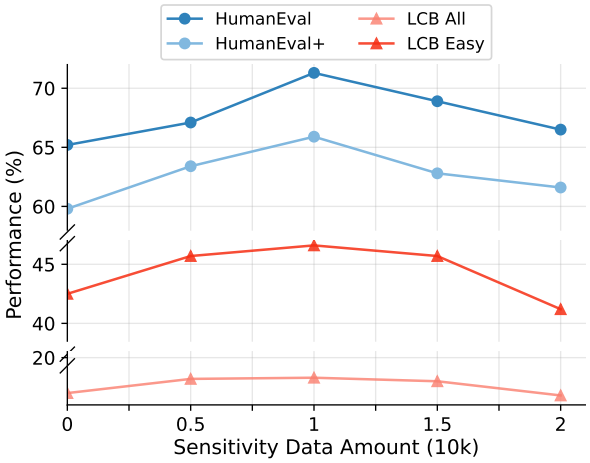

Figure 8: The performance varies with the amount of sensitivity data.

same. Therefore, the answers are completely consistent and do not need to be modified.

- CTF. The new problem has only a small difference from the original problem, but it changes the meaning of the original problem, making the answers not exactly the same as the original problem (With not too much variation in difficulty, the more variation in answers the better). Figure 12 and Figure 13 are two example of CTF problems. The change of the former problem is "only two adjacent characters can be exchanged", and the change of the latter problem is "cards become abcd".

4. Determine whether new test cases need to be added to the CTF problem. For example, the annotator should determine whether the range of data of the new problem is fully consistent with the original problem, and whether the input of test cases of the original problem can be directly executed by the CTF problem. For the first CTF problem, there is no need to add new test cases, while for the second CTF problem, some new test cases should be added.

C.5 Annotation Tabular

As shown in Table 4, we provide an example of the annotation table that the annotator should fill in.

D Prompt

This section shows the prompt used to instruct LLMs to generate desired counterfactual question and instruction tuning data.

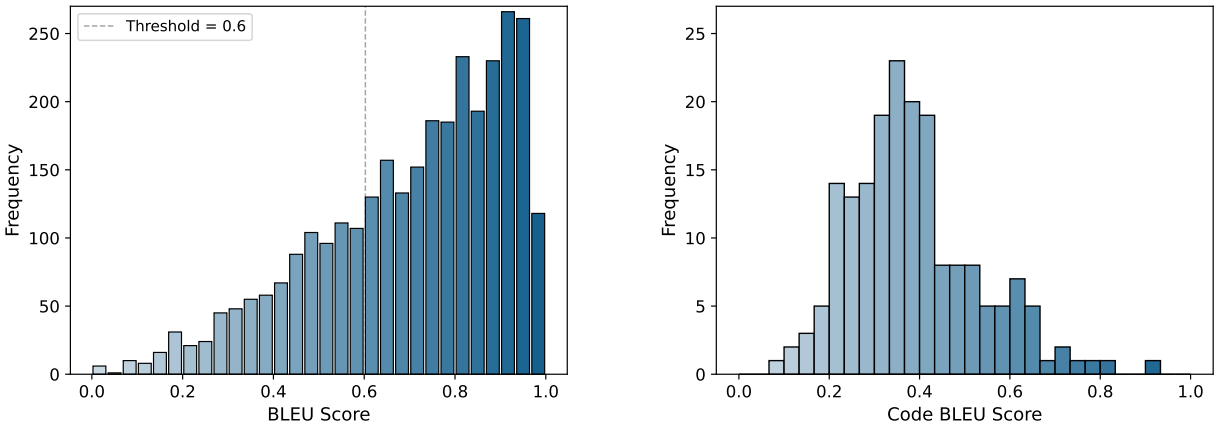

Figure 9: The left frequency histogram shows the BLEU score between the original problems and the CTF problems. The right one shows the Code BLEU score between the solutions corresponding to the original problems and the CTF problems.

Original Problem Index	Original Problem Meaning	Model	New Problem Index	New Problem Statement Error	New Problem Type	Modification	Add New Test Cases
0	Given a string composed of letters 'abc' in any order, exchange any two characters to see if string 'abc' can occur.	o1-mini	0-0		Robust		No
0	Given a string composed of letters 'abc' in any order, exchange any two characters to see if string 'abc' can occur.	o1-mini	0-1		CTF	Only two adjacent characters can be exchanged	No
1	Add 1 to a number in an array of positive numbers, how to maximise the array product	o1-mini	1-1		CTF	Replace a number in an array with a number from 0-9, how to make the array product maximum	No
4	A string with a phone number in front and 2 digits in the middle indicating age. Find those over 60 years old	o1-mini	4-0		CTF	How many people are over 60 years old and have unique phone numbers?	Yes
4	A string with a phone number in front and 2 digits in the middle indicating age. Find those over 60 years old	o1-mini	4-1		CTF	Age is hexadecimal	No

Table 4: An example of the annotation table.

```
## Question Content:

There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can do the following operation at most once:


-  Pick two cards, and swap them.  Is it possible that the row becomes $\texttt{abc}$ after the
    operation? Output "YES" if it is possible, and "NO" otherwise.

Input

The first line contains a single integer $t$ ($1 \leq t \leq 6$)  the number of test cases.

The only line of each test case contains a single string consisting of each of the three
    characters $\texttt{a}$, $\texttt{b}$, and $\texttt{c}$ exactly once, representing the
    cards.

Output

For each test case, output "YES" if you can make the row $\texttt{abc}$ with at most one
    operation, or "NO" otherwise.

You can output the answer in any case (for example, the strings "yEs", "yes", "Yes" and "YES"
    will be recognized as a positive answer).Sample Input 1:
6

abc

acb

bac

bca

cab

cba

Sample Output 1:

YES
YES
YES
NO
NO
YES

Note

In the first test case, we don't need to do any operations, since the row is already
    $\texttt{abc}$.

In the second test case, we can swap $\texttt{c}$ and $\texttt{b}$: $\texttt{acb} \to
    \texttt{abc}$.

In the third test case, we can swap $\texttt{b}$ and $\texttt{a}$: $\texttt{bac} \to
    \texttt{abc}$.

In the fourth test case, it is impossible to make $\texttt{abc}$ using at most one operation.

## Starter Code:

## Test Cases:

"[{\"input\": \"6\\nabc\\nacb\\nbac\\nbca\\ncab\\ncba\\n\", \"output\":
    \"YES\\nYES\\nYES\\nNO\\nNO\\nYES\\n\", \"testtype\": \"stdin\"}]"
```

Figure 10: An example of the original problem.

```
## Question Content:

There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can perform the following operation at most once:

- Choose any substring of the cards and reverse it.

Is it possible that the row becomes $\texttt{abc}$ after the operation? Output "YES" if it is
    possible, and "NO" otherwise.
...
```

Figure 11: An example of the robust version of the original problem.

```
## Question Content:
There are three cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$ placed in a row in
    some order. You can perform the following operation at most once:

- Pick two **adjacent** cards and swap them.

Is it possible that the row becomes $\texttt{abc}$ after the operation? Output "YES" if it is
    possible, and "NO" otherwise.
...
```

Figure 12: The first example of the CTF version of the original problem.

```
## Question Content:

There are four cards with letters $\texttt{a}$, $\texttt{b}$, $\texttt{c}$, $\texttt{d}$ placed
    in a row in some order. You can do the following operation at most once:

- Pick two cards, and swap them.  Is it possible that the row becomes $\texttt{abcd}$ after the
    operation? Output "YES" if it is possible, and "NO" otherwise.
...
```

Figure 13: The second example of the CTF version of the original problem.

```
Please create a **counterfactual** version of the given original python programming problem.
    Your goal is to **make a minimal change to the problem that leads to a significant change
    in the solution**. Follow these detailed steps:
1. Carefully read and comprehend the original problem's context, conditions, constraints, and
    requirements.
2. Identify a critical point in the original problem and think about a modification. **The
    modification should be slight but cause a substantial change in the solution approach**.
3. Consider the influence of the modification. Ask yourself: Would it change data structures or
    algorithms? Explain the influence before output the counterfactual problem. If the
    influence does not impact the solution approach significantly, rethink another critical
    point to modify. Repeat Step 2 and Step 3 until you find a point that satisfies the
    requirement.
4. Modify the original problem based on the most influential point. The modified problem must
    be consistent, clear, and requires a significantly different solution approach. Update the
    sample inputs and outputs to match the new problem condition.
5. Output the counterfactual problem, ensuring the following format:
    - Before the JSON format, include a section marker "###Counterfactual Problem".
    - After the section marker, provide the counterfactual problem in the same JSON format as
    the original, including "question_content", "starter_code", "public_test_cases", and
    "metadata".
### Original Problem
```

Figure 14: The prompt used to generate CTF-Code Problem.

```
Refine a code generation task, initially presented as #Original_Sample#, which is a JSON dict
    including three keys: a task instruction, and the output generated from  the instruction.
Your task is to produce a #Modified_Sample# by altering the original task instruction in a way
    that significantly changes the output, yet with minimal adjustments to the instruction
    itself.

## Requirements:
1. **Minimal Instruction Change**: Achieve the code change with minimal alterations to the
    instruction. The difference will be assessed through evaluated by the Rouge score,
    indicating the high similarity in wording, sentence structure, and length to the original.
2. **No Trival Changes to Instruction**: Ensure the modification to the instruction is
    semantic-relevant. Do not make trivial changes like adding or removing a word, changing the
    order of words, or replacing synonyms.
3. **Maximal Code Change**: Your adjustments should lead to considerable changes in the output,
    impacting aspects like algorithms, data structures, data and control flows, or boundary
    conditions. The difference will be assessed through both the Rouge score and AST score,
    indicating the output's functionality, implementation, and naming should substantially
    diverge from the original.
4. **Encourage Trival Code Change**: The code output should be significantly different. Change
    every aspect of the code, including the function name, variable names.

## Format:
1. Your output should be a #Modified_Sample# dict in **JSON format** as the #Original_Sample#
    is.
2. Using **markdown code snippet syntax** in the instruction and the output.
3. Ensure all characters are **properly escaped** in the JSON string.

## Examples:
{seeds}

## Question:
- Original_Sample:
```

Figure 15: The prompt used to generate CTF-Instruct data.