COFFEE-GYM: An Environment for Evaluating and Improving Natural Language Feedback on Erroneous Code

Hyungjoo Chae[1*]   **Taeyoon Kwon**[1*]   **Seungjun Moon**[1*]

Yongho Song   **Dongjin Kang**[1]   **Kai Tzu-iunn Ong**[1]   **Beong-woo Kwak**[1]

Seonghyeon Bae[1]   **Seung-won Hwang**[2]   **Jinyoung Yeo**[1]

Yonsei University[1]   Seoul National University[2]

{mapoout, kwonconnor101, lune_blue, jinyeo}@yonsei.ac.kr

seungwonh@snu.ac.kr

Abstract

This paper presents COFFEE-GYM, a comprehensive RL environment for training models that provide feedback on code editing. COFFEE-GYM includes two major components: (1) COFFEE, a dataset containing humans' code edit traces for coding questions and machine-written feedback for editing erroneous code; (2) COFFEEEVAL, a reward function that faithfully reflects the helpfulness of feedback by assessing the performance of the revised code in unit tests. With them, COFFEE-GYM addresses the unavailability of high-quality datasets for training feedback models with RL, and provides more accurate rewards than the SOTA reward model (*i.e.*, GPT-4). By applying COFFEE-GYM, we elicit feedback models that outperform baselines in enhancing open-source code LLMs' code editing, making them comparable with closed-source LLMs. We make the dataset and the model checkpoint publicly available.[1]

1 Introduction

Large language models (LLMs) have made great progress in code generation (Li et al., 2023; Rozière et al., 2023), *e.g.*, achieving human-level performances in code generation benchmarks (Chen et al., 2021b). Such success makes them powerful tools for assisting human programmers (Köpf et al., 2023); however, they still produce errors (Guo et al., 2024a; OpenAI, 2023b). Therefore, code editing, *i.e.*, resolving errors in code, remains an important task for code LLMs (Muennighoff et al., 2023).

Studies have utilized natural language (NL) feedback from LLMs as descriptive guidance in editing wrong codes for code LLMs. For instance, Self-Refine (Madaan et al., 2023) largely improves their code editing using GPT-4's feedback. Yet, abilities to generate helpful feedback, as they report, are limited to powerful closed-source LLMs (*e.g.*, GPT-4).

---

*Equal contribution

[1] https://huggingface.co/spaces/Coffee-Gym/Project-Coffee-Gym

Figure 1: A motivating example (Top) and Pass@1 accuracy in HumanEvalFix (Bottom). We compare the feedback from our model and various other models, both paired with DeepSeekCoder-7B as the code editor. SFT denotes the model trained on Code-Feedback (Zheng et al., 2024) using the same backbone model as ours.

This can lead to a heavy reliance on closed-source LLMs that may cause not only high computational (*e.g.*, API) cost but also security risks (Siddiq and Santos, 2023; Greshake et al., 2023), limiting their applicability for confidential codes.

This work aims to foster building open-source feedback models that produce effective feedback for code editing. An intuitive approach is to apply supervised fine-tuning (SFT) on open-source code LLMs using feedback from GPT-4 (generated
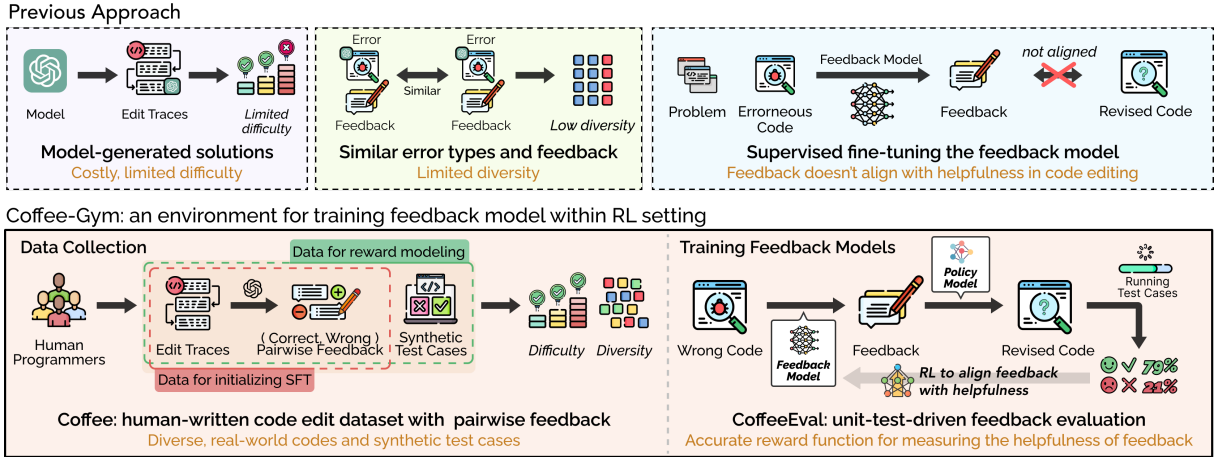
Figure 2: Comparison between COFFEE-GYM and the previous approach.

based on machines' code editing) (Zheng et al., 2024). However, this simplified approach poorly aligns editing performance with the helpfulness of feedback (Bottom of Figure 1) (Liu et al., 2022).

Inspired by the success of RLHF (Ouyang et al., 2022), we reformulate feedback modeling with reinforcement learning (RL), where we align feedback models with the helpfulness of feedback during training. Since the success of RL highly depends on the initial SFT model and a reliable reward function (Lightman et al., 2023; Lambert et al., 2024), we hereby identify 3 main challenges in applying RL to feedback generation for code editing: **(1)** limited scenarios of errors in model-generated code editing datasets for initializing SFT model, **(2)** the lack of pairwise (correct and wrong) feedback to train/test reward functions, **(3)** absence of validated implementation of reward models.

We present **COFFEE-GYM**, a comprehensive RL environment addressing the above challenges in training feedback models for code editing. First, to tackle data scarcity in SFT initialization and reward modeling, we curate ☕ **COFFEE**, a dataset for code fixing with feedback, which consists of code editing traces of human programmers and human annotated feedback. Unlike model-generated data (Figure 2), COFFEE includes (1) problems across various difficulties, including those current LLMs (*e.g.*, GPT-4) cannot solve; (2) pairs of correct and wrong feedback for reward modeling; (3) about 36 test cases per problem to measure the feedback helpfulness in code editing.[2]

Next, to address the absence of validated (*i.e.*, reliable) reward functions, we introduce COFFEEEVAL, a reward function designed to reflect the helpfulness of feedback into reward calculation. Instead of directly assessing feedback quality (Rajakumar Kalarani et al., 2023), we simulate code editing based on generated feedback, conduct unit tests on the edited code, and use the test results to measure feedback helpfulness. With the pairwise feedback from ☕ COFFEE, we train a given code editor to produce edited code that faithfully reflects the helpfulness of the given feedback.

Through experiments, we validate COFFEE-GYM's efficacy in training feedback models. We find that COFFEEEVAL provides more accurate rewards, compared to the current SOTA reward model, *i.e.*, G-Eval (Liu et al., 2023c) with GPT-4. Also, we show that the feedback models trained with COFFEE-GYM generate more helpful feedback, achieving comparable performance to closed-source feedback models in code editing.

2 Task Definition and Problem Statement

2.1 Code Editing with Natural Language Feedback

The task of code editing aims to resolve errors in given codes to produce a correct solution. Formally, given a problem description q and a defective solution y, our goal is to learn a feedback model θ that generates helpful feedback describing the errors in y and provide helpful guidance on code editing: $\hat{c} = \theta(q, y)$. Then, an editor model ϕ that takes q, y, and the generated feedback $\hat{c}$ as input and generates the edited code: $y' = \phi(q, y, \hat{c})$.

In evaluating the edited code y', the functional-
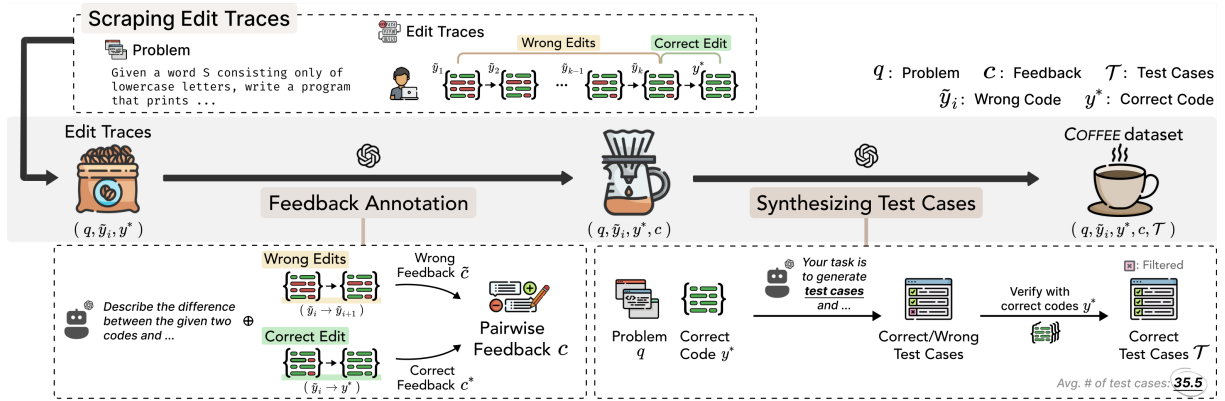
---

[2]This work is a substantially revised and extended version of our preprint (Moon et al., 2023). While both works use the same dataset, this submission presents significant advancements in methodology, analysis, and results.

Figure 3: Overview of the data collection process of ☕ COFFEE.

ity of the edited code is measured with Pass@k, the standard metric that measures the number of passed test cases t_i within the given set $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ (Li et al., 2022, 2023; Muennighoff et al., 2023). Each test case t_i consists of an input x_i and an expected output z_i.

2.2 Learning Feedback Models

In this paper, we consider two widely used learning approaches to build open-source feedback models.

Supervised fine-tuning. A straightforward approach is to fine-tune an open-source code LLM θ on a dataset $D = \{(q_i, y_i, c_i, y_i^*)\}_{i=1}^N$ of problem descriptions, incorrect codes, feedback annotations, and correct codes. The objective is to minimize the negative log-likelihood of the target feedback label y^* given q and y. However, simply training to optimize the probability of the target sequence does not achieve much improvement for code editing, because it does not consider the impact of feedback on code editing (Liu et al., 2022).

Reinforcement learning. Inspired by Ouyang et al. (2022), we adopt reinforcement learning (RL) to further align feedback generation to correct code editing. Specifically, we choose PPO (Schulman et al., 2017) and DPO (Rafailov et al., 2023) as reference RL algorithms and apply them on the feedback model θ initialized via SFT.

The two key factors of RL are (1) **pairwise preference data** and (2) **reward modeling** (Lambert et al., 2024). In our task, we consider a preference dataset where each input q and y comes with a pair of chosen and rejected feedback c^+ and c^-, and their preference ranking $c^+ \succ c^-$. This dataset is then used to model the reward based on the preference ranking. While in PPO a reward model is explicitly trained using c^+ and c^-, DPO relies on implicit reward modeling and directly optimizes the feedback model using the preference dataset.

2.3 Problem Statement

Our goal is to promote rapid development of open-source feedback models by facilitating RL for feedback generation on code editing. Specifically, we aim to provide the two key components in RL for feedback generation:

Dataset. The dataset required for our RL approach covers the following key aspects: (1) **Coverage of difficulty and diversity** (q, y) to initialize a good SFT model. (2) **Pairwise feedback data** $(c^+ \succ c^- \mid q, y)$ to build datasets for training DPO and a reward model for PPO. (3) **Test cases for unit test** $(\mathcal{T})$ are required to implement our R, for directly measuring the impact of c on the correctness of code editing.

Reward model. The current standard of using LLM as a reward model (Lee et al., 2023) to evaluate LLM outputs do not sufficiently models the impact of feedback on code editing outcomes and requires powerful LLMs (*e.g.*, GPT-4) that incur high API costs. Especially, the high computation costs significantly limits the application of online RL algorithms (*e.g.*, PPO) in feedback modeling, which require frequent and continuous API calls for reward calculation.

3 Constructing COFFEE-GYM

We introduce COFFEE-GYM, a comprehensive RL environment for training NL feedback model for code editing. COFFEE-GYM consists of two major components: (1) ☕ COFFEE, a dataset of human-written edit traces with annotated NL feedback, and (2) COFFEEEVAL, an accurate reward model that

Problem Description: q
Given a word S consisting only of lowercase letters, write a program that prints the <u>first occurrence of each letter in the word</u>, or <u>-1 if the letter is not included</u> in the word.

❎ Wrong Code: $\tilde{y}$

```
S = input()
abc = [-1]*26
for c in S:
    abc[ord(c)-ord('a')] = S.index(c)
print(abc)
```

☑️ Correct Code: y^*

```
S = input()
abc = [-1]*26
for c in S:
    abc[ord(c)-ord('a')] = S.index(c)
print(*abc)
```

📝 Correct Feedback: c^*
Your code correctly initializes the list with -1 for each letter, but you need to print the values individually using the operator to unpack the list.

📝 Incorrect Feedback: $\tilde{c}$
The issue is that you need to use a dictionary to store the ...

Synthetic Test Cases: $\mathcal{T}$	
Input (*i.e.*, word S)	Correct Output
zebra	[4, 2 , -1, ..., 0]
⋮	⋮

Dataset Statistics			
# of instance	44,782	Avg. description len.	269.0
# of total prob. sets	742	Avg. # of error lines per code	4.19
Avg. solution len.	674.1	Avg. # of submissions per user	2.7
Avg. wrong code len.	674.1	Avg. # of test cases per prob.	35.5
Avg. feedback len.	649.4		

Figure 4: Example and statistics of ☕ COFFEE.

measures feedback's impact on code editing.

3.1 ☕ COFFEE: Human-written Code Edit Traces with Annotated Pairwise Feedback

We curate **COFFEE**, a dataset of <u>co</u>de <u>f</u>ixing with <u>f</u>eedback, from human-written code edit traces. COFFEE consists of problems of diverse levels of difficulty, including challenging problems that only human programmers can solve, and provides test cases for reward functions (Section 3.2). The overview of constructing COFFEE, data examples, and statistics are in Figure 3 and 4.

3.1.1 Collecting Code Edit Traces from Human Programmers

We collect human-authored code edits from an online competitive programming platform.[3] In this platform, given a problem description q, human programmers keep submitting a new solution y until they reach a correct solution y^* that passes all hidden test cases for q. Formally, for each q and the correct submission y_n^*, we collect the submission history $\{\tilde{y}_1, \tilde{y}_2, ..., y_n^*\}$, where $\{\tilde{y}_k\}_{k=1}^{n-1}$ are incorrect solutions. We then construct $(q, \tilde{y}, y^*)$ triplets

---
[3] https://www.acmicpc.net/

(a) Distribution of average length of edit trace

(b) Diversity analysis on error codes using CodeBERT

(c) Pass@1 of GPT-4-Turbo compared to human

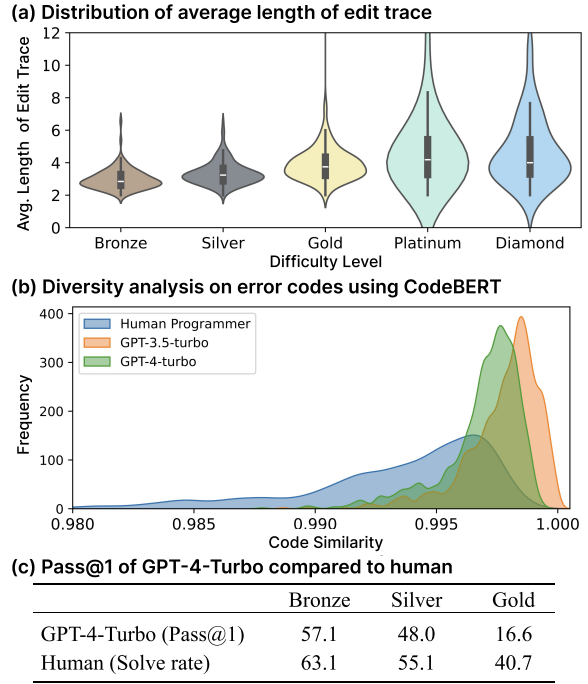|	Bronze	Silver	Gold
GPT-4-Turbo (Pass@1)	57.1	48.0	16.6
Human (Solve rate)	63.1	55.1	40.7

Figure 5: Analysis results of ☕ COFFEE. Experiment details are in Appendix A.1.5.

by pairing each incorrect solution $\tilde{y}_k$ with the correct one y_n^*, *i.e.*, $\{(q, \tilde{y}_k, y_n^*)\}_{k=1}^{n-1}$.

To ensure COFFEE is not biased toward coding problems of a specific difficulty level, we collect an equal number of problems from each of the five difficulty levels in the platforms, ranging from beginner to expert levels. We also ensure that COFFEE includes various solutions to each problem by collecting submission histories from 100 different users. Our analysis in Figure 5 shows that COFFEE (1) includes problems that are challenging for both human and LLMs and (2) covers more diverse error cases than machine-generated codes.

3.1.2 Annotating Pairwise Feedback Data

We additionally annotate NL feedback that provides useful guidance on the necessary edits. For each triplet $(q, \tilde{y}, y^*)$, we prompt GPT-3.5-Turbo (OpenAI, 2023a) to describe how the correct solution y^* differs from the wrong code $\tilde{y}$. The resulting description c^* serves as the correct feedback that describes necessary changes on the wrong code $\tilde{y}$ to obtain the correct code y^*. Along with c^*, we also collect incorrect feedback $\tilde{c}$, which describes the difference between two wrong solutions, $\tilde{y}_{k-1}$ and $\tilde{y}_k$ ($k \neq n$), to provide pairwise labels for both correct and incorrect feedback to a single wrong solution $\tilde{y}$. We discuss details on feedback annotation in Appendix A.1.1, including our prompt used

	mean	std	min	25%	50%	75%	max
Pass ratio	0.342	0.370	0.000	0.000	0.162	0.693	0.985

Table 1: Pass ratio for incorrect code samples in the evaluation set of COFFEE dataset.

for feedback annotation and filtering techniques.

3.1.3 Augmenting Synthetic Test Cases

Finally, we include a set of hidden test cases $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ for each edit instance $(q, \tilde{y}, y^*, c)$ in our dataset to assess whether the edited code is the correct solution to the problem. Each test case t_i consists of an input x_i and an expected output z_i. As the programming platform does not make test cases publicly available, we annotate test cases by prompting GPT-3.5-Turbo to generate inputs x_i for a given q and executing the correct code y^* with x_i to obtain the corresponding outputs z_i. We filter out any invalid test cases with inputs that result in errors during execution. On average, we obtain 35.5 test cases per problem.

A critical question in evaluating our test suite is whether any incorrect solutions manage to pass all the test cases. To address this, we conduct an experiment using the evaluation set of the COFFEE dataset. We randomly sampled 200 wrong code instances and calculated the pass ratios of the wrong codes. We show the statistics of the distribution of pass ratios. As shown in Table 5, the maximum pass ratio is 0.985, which suggests that there are no wrong solutions that passed all the test cases. The mean score is 0.342, indicating that on average, wrong solutions fail the majority of the test cases. We further analyze the COFFEE-TEST and verified that no wrong solutions pass all the test cases.

These test cases are used to measure the correctness of an edited code and estimate the helpfulness of the feedback as the COFFEEEVAL score, which we later use as supervision signals for training feedback models (§3.2) in COFFEE-GYM. We provide details on test case generation in Appendix A.1.3.

3.2 COFFEEEVAL: Unit-test-driven Feedback Evaluation

We present **COFFEEEVAL** as our reliable reward function in COFFEE-GYM. The key idea is to measure the helpfulness of feedback by gauging the correctness of the edited code produced by a small, but cheap editor model that properly aligns editing with feedback. Specifically, given a problem description q, a wrong solution $\tilde{y}$, and feedback

$\hat{c}$ from a feedback model θ, an editor model ϕ generates an edited code y' by grounding on $\hat{c}$, i.e., $y' = \phi(q, \tilde{y}, \hat{c})$. The COFFEEEVAL score is defined as the proportion of test cases for which the edited code y' produces the expected output:

$$\text{COFFEEEVAL}(q, \tilde{y}, \hat{c}, \phi, \mathcal{T})$$
$$= \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}\left(\phi(q, \tilde{y}, \hat{c})(x_i) = z_i\right) \quad (1)$$

where each element $t_i \in \mathcal{T}$ consists of an input x_i and an expected output z_i, and $\mathbb{1}$ is a binary indicator function that returns 1 if the output of y' matches the expected output z_i. By reflecting the correctness of the edited code, the resulting score serves as an accurate measure for the effectiveness of the generated feedback in code editing.

3.2.1 Training a Faithful Code Editor to Align Editing with Feedback

General code LLMs are trained to produce only correct codes, resulting in a bias toward correct editing regardless of feedback quality. To address this, we train a code editor ϕ that aligns its output with the helpfulness of the feedback by training the model to generate both correct edits $(q, y, c^*, y^*) \in \mathcal{D}_{correct}$ and incorrect edits $(q, y, \tilde{c}, \tilde{y}) \in \mathcal{D}_{wrong}$ in COFFEE. The training objective is defined as:

$$\mathcal{L}(\phi) = -\sum_{(q,y,c^*,y^*) \in \mathcal{D}_{correct}} \log p_\phi(y^* \mid q, y, c^*)$$
$$- \sum_{(q,y,\tilde{c},\tilde{y}) \in \mathcal{D}_{wrong}} \log p_\phi(\tilde{y} \mid q, y, \tilde{c}) \quad (2)$$

To prevent confusion during training, we follow Wang et al. (2023a) and indicate the correctness of the target code by prepending the keywords [Correct] and [Wrong] to the code sequence.

By learning from both positive and negative examples, the editor learns to conduct code editing by faithfully following the given feedback. It allows us to use the editor's output as a reliable metric for evaluating feedback generation models in our COFFEE-GYM environment.

4 Validating COFFEEEVAL

4.1 Experimental Setting

Implementation details. We implement COFFEEEVAL with DeepSeekCoder-7B model as the backbone in all our experiments. For further details, please refer to Appendix A.2.1.

Model	Evaluation	Pass@1		Scores			Correlation	Error
		✓ Correct Feedback ↑ (TP)	✗ Wrong Feedback ↓ (FP)	Precision ↑	Recall ↑	F1 ↑	Pearson ↑	MSE ↓
GPT-4-Turbo	G-Eval	-	-	-	-	-	0.135	0.415
GPT-3.5-Turbo	G-Eval	-	-	-	-	-	-0.172	0.575
GPT-4-Turbo	Editing	**53.0**	51.8	50.6	**53.0**	51.8	0.012	0.450
GPT-3.5-Turbo	Editing	43.4	33.6	56.4	43.4	49.0	0.101	0.417
DeepSeek-Coder-7B	Editing	36.0	28.8	55.6	36.0	43.7	0.077	0.428
DeepSeek-COFFEEEVAL (w/o WF)	Editing	36.4	**28.4**	56.2	36.4	44.2	0.085	0.418
DeepSeek-COFFEEEVAL (Ours)	Editing	52.0	**28.4**	64.7	52.0	57.7	**0.149**	**0.408**

Table 2: Performance of our evaluation protocol on the test sets of COFFEE compared to the baselines. Wrong Feedback is abbreviated as WF due to limited space.
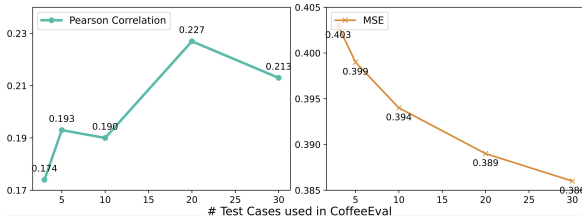

Figure 6: Ablation results on the number of test cases used in COFFEEEVAL. The evaluation performance decreases as the number of test cases declines.

4.2 Reliability of COFFEEEVAL

Baselines. We compare our COFFEEEVAL with two evaluation methods: G-Eval (Liu et al., 2023c) and Editing. For G-Eval, we directly assess feedback quality in Likert-scale (1 - 5) using score rubrics (Kim et al., 2023). Editing baselines follow the same evaluation scheme as COFFEEEVAL but use general code LLMs for the editor ϕ. We consider with three code LLMs, GPT-3.5-Turbo, GPT-4-Turbo, and DeepSeek-Coder-7B. The prompt we use for G-Eval is in Appendix B.3.

Evaluation. To measure the alignment between feedback generation and code editing, we use test set of COFFEE, where each c is annotated with a binary label on its helpfulness. For Editing methods (including ours), we regard the output as positive prediction when the edited code passes all test cases. Also, we provide Pearson correlation coefficients for both Editing and G-Eval methods to analyze the correlation between the predicted score and the ground-truth labels.

4.3 Results and Analysis

COFFEEEVAL faithfully aligns feedback quality with editing performance. As shown in Table 2, DeepSeek-COFFEEEVAL achieves higher Pearson correlation and lower MSE than all G-Eval and Editing baselines. In particular, our approach shows even higher correlation than the G-Eval baseline implemented with GPT-4-Turbo. The strong performance of our COFFEEEVAL validates its effectiveness in assessing the quality of NL feedback in the code editing task.

Code LLMs are skewed toward correct editing, regardless of the feedback quality. While code LLMs have shown promising results in code generation tasks, they do not faithfully reflect the helpfulness of feedback on code editing. Especially, GPT-4-Turbo, the current SOTA code LLM, shows the highest Pass@1 among baselines, but it also tends to generate correct code even with wrong feedback. These results suggest that the training process with our pairwise feedback data is an essential step in building a reliable reward model.

The performance of COFFEEEVAL benefits from the number of test cases. Figure 6 compares the Pearson correlation coefficient and MSE with respect to the number of test cases. We observe that a higher number of test cases leads to more accurate evaluation on the feedback quality, which validates our design choice of ☕ COFFEE.

5 Benchmarking Reference Methods of COFFEE-GYM

In this section, we apply the feedback model trained using COFFEE-GYM on various open-source LLMs and assess its effectiveness in enhance code editing performance. Furthermore, we comprehensively explore a wide range of training strategies available in our COFFEE-GYM to provide insights on building helpful feedback models.

5.1 Effectiveness of COFFEE-GYM in Training Feedback Models

5.1.1 Experimental Setting

Implementation details. We train our feedback model based on DeepSeekCoder-7B using COFFEE-GYM by applying PPO. Further details are in Appendix A.3.

Methods	Params.	Open-source	HumanEvalFix		COFFEE-TEST		Average	
			Pass@1	Δ	Pass@1	Δ	Pass@1	Δ
GPT-4-Turbo (OpenAI, 2023b)	-	✗	83.5	-	43.8	-	63.6	-
GPT-3.5-Turbo (OpenAI, 2023a)	-	✗	75.0	-	32.2	-	53.6	-
DeepSeek-Coder (Guo et al., 2024a)	7B	✓	60.4	-	33.8	-	47.1	-
+ Execution Feedback	-	✓	68.3	+ 7.9	38.3	+ 4.5	53.3	+ 6.2
+ Self-Feedback	7B	✓	67.7	+ 7.3	28.3	- 5.5	48.0	+ 0.9
+ OpenCodeInterpreter-DS-Coder Feedback	7B	✓	64.6	+ 4.2	30.5	- 3.3	47.5	+ 0.5
+ OURS	7B	✓	73.8	+ 13.4	47.2	+ 13.4	60.5	+ 13.4
+ GPT-3.5-Turbo Feedback	-	✗	72.5	+ 12.1	35.5	+ 1.7	54.0	+ 6.9
+ GPT-4-Turbo Feedback	-	✗	74.4	+ 14.0	44.4	+ 10.6	59.4	+ 12.3
CodeGemma (CodeGemma Team et al., 2024)	7B	✓	53.7	-	14.4	-	34.1	-
+ Execution Feedback	-	✓	61.6	+ 7.9	15.0	+ 0.6	38.3	+ 4.2
+ Self-Feedback	7B	✓	53	- 0.7	16.6	+ 2.2	34.8	+ 0.7
+ OpenCodeInterpreter-DS-Coder Feedback	7B	✓	36.5	- 17.2	15	+ 0.6	25.8	- 8.3
+ OURS	7B	✓	59.7	+ 6.0	31.1	+ 16.7	45.4	+ 11.4
+ GPT-3.5-Turbo Feedback	-	✗	57.3	+ 3.6	22.2	+ 7.8	39.8	+ 5.7
+ GPT-4-Turbo Feedback	-	✗	65.8	+ 12.1	22.7	+ 8.3	44.3	+ 10.2
OpenCodeInterpreter-DS-Coder (Zheng et al., 2024)	7B	✓	65.8	-	30.5	-	48.1	-
+ Execution Feedback	-	✓	66.4	+ 0.6	36.6	+ 6.1	51.5	+ 3.4
+ Self-Feedback	7B	✓	62.1	- 3.7	21.1	- 9.4	41.6	- 6.5
+ DeepSeek-Coder Feedback	7B	✓	56.1	- 9.7	28.3	- 2.2	42.2	- 5.9
+ OURS	7B	✓	70.1	+ 4.3	42.7	+ 12.2	56.4	+ 8.3
+ GPT-3.5-Turbo Feedback	-	✗	68.3	+ 2.5	32.7	+ 2.2	50.5	+ 2.4
+ GPT-4-Turbo Feedback	-	✗	72.5	+ 6.7	43.3	+ 12.8	57.9	+ 9.8

Table 3: Code editing results of our feedback model trained with COFFEE-GYM, *i.e.*, PPO-COFFEEEVAL, on HumanEvalFix and COFFEE-TEST. We pair our feedback model with an open-source code LLM as the code editor.

Benchmarks. We test the feedback model trained using COFFEE-GYM on HumanEval-Fix (Muennighoff et al., 2023), a widely used code editing benchmark. The task is to fix the errors in given erroneous code and the correctness of the edited code is measures by running the annotated test cases. Then, if the submitted solution passes all testcases the solution is evaluated as success and pass@1 is calculated as the percentage of the passed solutions for all promplems. We carefully check if there is data leakage in COFFEE and verify there is no overlap between COFFEE and HumanEvalFix (Appendix A.1.6). Additionally, we assess the effectiveness of our approach on a held-out test set named COFFEE-TEST. It consists of 180 instances of $(q, \tilde{y}, y^*, \mathcal{T})$ pairs that are collected following the same process in §3.1 but with no overlapping problems with COFFEE.[4]

Baselines. We compare with the following baselines that provides feedback for code editing: (1)

Execution Feedback (Chen et al., 2023): execution results of the generated code, *e.g.*, error messages, without using any LLMs , (2) Self-Feedback (Madaan et al., 2023): NL feedback generated by the code editor itself, (3) OpenCodeInterpreter Feedback (Zheng et al., 2024): a code LLM especially trained on Code-Feedback dataset. We also provide the results of feedback from closed-source LLMs, GPT-3.5-Turbo and GPT-4-Turbo, but these models are not our main focus as we aim to develop open-source feedback models.

5.1.2 Results

In Table 3, we compare the performance of our best feedback model with other feedback methods using various open-source models. Consistent with the findings from Chen et al. (2023), we observe improvements across all code LLMs when using Execution Feedback. However, we find that open-source code LLMs, despite their capabilities in the code domain, struggle to generate helpful NL feedback for code editing (Self-Feedback), highlighting the complexity of producing effective feedback. Notably, our approach demonstrates comparable performance to GPT-3.5/4-Turbo, signifi-

---

[4]While we have considered other code editing benchmarks, DebugBench (Tian et al., 2024) and CodeEditorBench (Guo et al., 2024b), we find that these benchmarks have a critical issue; even the ground-truth solution cannot pass the unit test. A detailed discussion on this issue is in Appendix B.1.

cantly closing the performance gap between closed-source and open-source models in the task of feedback generation for code editing.

5.2 Comparing Different Training Strategies in COFFEE-GYM

5.2.1 Experimental Setting

Training strategies. For training algorithm, we explore DPO, PPO, and Rejection Sampling (RS). In RS, we sample 10 $\hat{c}$ from SFT model, and collect $\hat{c}$ with top-1 COFFEEEVAL score as labels for the next iteration of SFT. For PPO, we use COFFEEE-VAL as the reward model. We use 3 variants for DPO: (1) DPO-TS: We construct preference pair by selecting the teacher model's feedback (*i.e.*, GPT-3.5-Turbo) as c^+, and the student model's (SFT) response as c^- (Tunstall et al., 2023), (2) DPO-CW: We directly use the labeled feedback pair $(c^*, \tilde{c})$. (3) DPO-COFFEEEVAL: We sample 10 $\hat{c}$, same as RS, and we construct preference pair with $\hat{c}$ of top-1 and bottom-1 COFFEEEVAL score.

5.2.2 Results

COFFEE provides helpful train data for SFT. In Figure 7, we find that SFT-COFFEE provides more helpful feedback than SFT-CODE-FEEDBACK trained on Code-Feedback. This results suggest that COFFEE serves as a valuable resource for fine-tuning feedback models.

COFFEE and COFFEEEVAL allow informative preference pair construction for DPO. DPO-COFFEEEVAL achieves the best results among DPO variants, closely followed by DPO-CW, which utilizes correct-wrong pairs from COFFEE. However, DPO-TS significantly underperforms even with the correct feedback c^+ sampled from the teacher. We conjecture that the teacher's feedback may not always be superior to the student's, leading to suboptimal preference pairs.

PPO is the most effective training algorithm. PPO-COFFEEEVAL outperforms DPO-COFFEEEVAL and RS-COFFEEEVAL, despite using the same reward model. We hypothesize that online RL methods like PPO allow for continuous updates on the reference model and lead to better alignment compared to offline methods like DPO, which learn from a fixed initial model.

5.3 Analysis

Fine-grained analysis by error type. In Figure 8a, we compare the baselines with our approach
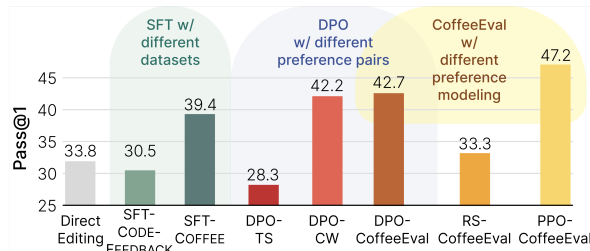

Figure 7: End-to-end validation results of the reference methods in COFFEE-GYM on COFFEE-TEST.
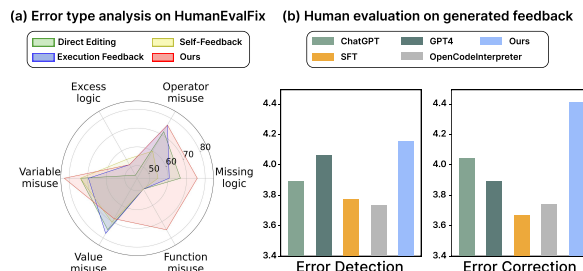

Figure 8: (a) Breakdown of editing performance on HumanEvalFix by different error types. (b) Human evaluation of the feedback generated on HumanEvalFix. See Appendix B.4 for details on human evaluation.

across different error types. Our feedback model is particularly effective at correcting Missing logic and Function misuse errors, which can greatly benefit from NL feedback by providing a detailed explanation for editing. In value misuse, our model shows slightly lower performance. We posit that this is due to the discrepancy between the distribution of errors from human-authored data (*i.e.*, COFFEE) and synthetic data, where our model is tested.

Human evaluation on feedback quality. To provide a more accurate analysis of the feedback quality, we conduct human evaluation using qualified workers from MTurk.[5] The results in Figure 8b show that the feedback from our model is rated as more helpful and informative compared to the baselines, supporting the findings in §5.2.

6 Related Work

Code editing. Code LLMs have shown promising code generation capabilities by training on massive code corpora (Li et al., 2023; Wang et al., 2023b). Despite their promising capabilities, there remains a possibility of errors, making code editing tasks essential for ensuring code quality and correctness (Muennighoff et al., 2023). In response to this necessity, recent studies have focused on as-

---

[5]The details of our human evaluation are in Appendix B.4.

sessing the code editing capabilities of code LLMs, by proposing new benchmarks for the task (Tian et al., 2024; Guo et al., 2024b).

Refining with external feedback. In code editing, two types of widely used external feedback are execution feedback (Gou et al., 2023; Chen et al., 2023) and NL feedback (Madaan et al., 2023; Shinn et al., 2023). Recently, Zheng et al. (2024) explored both types of feedback and demonstrate that NL feedback outperforms execution feedback. Concurrent to our work, Ni et al. (2024) explored building feedback model, but they do not provide the dataset used nor the model checkpoint.

RL in code generation tasks. A line of research has explored improving LLMs' code generation with RL by leveraging the unit test results as reward (Le et al., 2022; Liu et al., 2023a; Shen et al., 2023). While the design of COFFEEEVAL is largely inspired by this line of work, we show that building reward model for feedback learning using unit test results is non-trivial, since code LLMs do not faithfully reflect feedback into editing (Table 2).

7 Conclusion

In this paper, we present a comprehensive study on building open-source feedback models for code editing. We introduce COFFEE-GYM, an environment for training and evaluating feedback models, and share valuable insights from our experiments. We hope our work will encourage researchers to further explore feedback model development using COFFEE-GYM and our findings, advancing the field of code editing with NL feedback.

Limitations

Scope of editing. COFFEE-GYM tackles the task of code editing with a particular focus on correcting errors in codes. This leaves room for improvement in our RL approach to consider the efficiency and readability of the edited codes. Also, we mainly focus on editing incorrect source codes in a competitive programming setting. Some examples from our feedback model (Appendix C.2) suggest that our approach can be further applied to practical programming problems, *e.g.*, those that involve machine learning libraries. In future studies, COFFEE-GYM can be further expanded to real-world software engineering settings with additional training on general code corpora (Li et al., 2023).

Using synthetic test cases for measuring reward. While running synthetic test cases and using the resulting pass rates might be a promising proxy for reward calculation, there might be edge cases where even erroneous codes pass the synthetic test cases. Further research can incorporate Liu et al. (2023b) to make more challenging test cases that can rigorously identify erroneous codes.

Single programming language. Our implementation of COFFEE-GYM is limited to a single programming language, *i.e.*, Python. However, future work might apply a similar strategy as ours to expand our model to a multilingual setting, where the model is capable of understanding and editing diverse programming languages such as Java.

Single parameter size and architecture. Lastly, we implement the feedback models only with one parameter size and architecture. However, future work can apply our method to models with larger parameter sizes (*e.g.*, DeepSeek-Coder 70B), which is expected to perform better in code editing. Our framework can also be further applied to other architectures, as our method is model-agnostic.

Ethical Considerations

While our dataset originates from online competitive programming platforms, we have ensured the exclusion of personal information to maintain privacy standards. Additionally, we are aware of the potential risks associated with texts generated by language models, which can contain harmful, biased, or offensive content. However, based on our assessments, this risk is mostly mitigated in our work. Lastly, there exists a risk of hallucination in the process of feedback generation and code editing, leading to incorrect edits. This emphasizes the need for careful application in our approach.

Acknowledgement

References

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. 2021a. Evaluating large language models trained on code.

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

CodeGemma Team, Ale Jakse Hartman, Andrea Hu, Christopher A. Choquette-Choo, Heri Zhao, Jane Fine, Jeffrey Hui, Jingyue Shen, Joe Kelley, Joshua Howland, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Nam Nguyen, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Siqi Zuo, Tris Warkentin, and Zhitao et al. Gong. 2024. Codegemma: Open code models based on gemma.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *ArXiv*, abs/2401.14196.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, et al. 2024b. Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint arXiv:2404.03543*.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, et al. 2023. Prometheus: Inducing fine-grained evaluation capability in language models. *arXiv preprint arXiv:2310.08491*.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. 2023. Openassistant conversations – democratizing large language model alignment.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, et al. 2024. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*.

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*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Jiacheng Liu, Skyler Hallinan, Ximing Lu, Pengfei He, Sean Welleck, Hannaneh Hajishirzi, and Yejin Choi. 2022. Rainier: Reinforced knowledge introspector for commonsense question answering. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8938–8958.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023b. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023c. G-eval: Nlg evaluation using gpt-4 with better human alignment.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Seungjun Moon, Yongho Song, Hyungjoo Chae, Dongjin Kang, Taeyoon Kwon, Kai Tzu-iunn Ong, Seung-won Hwang, and Jinyoung Yeo. 2023. Coffee: Boost your code llms by fixing bugs with feedback. *arXiv preprint arXiv:2311.07215*.

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

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution.

Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program synthesis with large language models.

OpenAI. 2023a. Chatgpt. https://openai.com/blog/chatgpt.

OpenAI. 2023b. Gpt-4 technical report.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model.

Abisek Rajakumar Kalarani, Pushpak Bhattacharyya, Niyati Chhaya, and Sumit Shekhar. 2023. "let's not quote out of context": Unified vision-language pre-training for context assisted image captioning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 695–706, Toronto, Canada. Association for Computational Linguistics.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of NeurIPS*.

Mohammed Latif Siddiq and Joanna C. S. Santos. 2023. Generate and pray: Using sallms to evaluate the security of llm generated code. *ArXiv*, abs/2311.00889.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*.

Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, et al. 2023. Zephyr: Direct distillation of lm alignment. *arXiv preprint arXiv:2310.16944*.

Peifeng Wang, Zhengyang Wang, Zheng Li, Yifan Gao, Bing Yin, and Xiang Ren. 2023a. SCOTT: Self-consistent chain-of-thought distillation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5546–5558, Toronto, Canada. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*.

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. *arXiv preprint arXiv:2402.14658*.

A Details of COFFEE-GYM

A.1 Details of ☕ COFFEE

A.1.1 Feedback Annotation

We annotate both correct and wrong feedback for our dataset using GPT-3.5-Turbo. We apply top-p sampling and temperature, where $p = 0.95$ and $T = 0.7$. We limit the number of generation tokens to 500. We leave out submission histories where the LLM fails to find any errors. We also filter out submissions from different users whose correct solutions are identical, as these solutions are usually copied from the web without undergoing editing processes. With collected user's submission history $\{\tilde{y}_1, \tilde{y}_2, ..., y_n^*\}$, we sample correct edit pairs $\{\tilde{y}_k, y_n^*\}_{k=1}^{n-1}$ to annotate correct feedback. To annotate the wrong feedback, we use sequential pairs $\{\tilde{y}_k, \tilde{y}_{k+1}\}_{k=1}^{n-2}$ to capture transitions between consecutive incorrect solutions. The prompts used for annotating correct and wrong feedback are demonstrated in Appendix D.1 and Appendix D.2.

A.1.2 Quality Analysis on Annotated Feedback

To thoroughly analyze the quality of the feedback from GPT-3.5-Turbo, we conduct a human evaluation. We ask human raters from Amazon Mechanical Turk (AMT) to score the quality of the feedback on a Likert scale. To ensure proficiency, we filter out human raters who have not passed our qualification test, which assesses their knowledge of programming languages, especially Python. From the test set of COFFEE, we sample 100 instances for the evaluation.

On average, the annotated feedback is scored 3.88 with 0.91 STD, which suggests that the quality of the annotated feedback is generally acceptable by humans. The full distribution of the evaluation results is shown in Table 4.

A.1.3 Synthesizing Test Cases

We prompt GPT-3.5-Turbo to synthesize input test cases given a problem description with three demonstrations. For each test case, we execute the correct code to obtain the corresponding output. If execution was successful, we then pair these inputs and outputs to create sample input-output pairs. On average, we synthesize 35 test cases per problem. We provide the prompt for the test case generation in Appendix D.3.

Correctness Score	Frequency (%)
1	2 (0.6%)
2	21 (7.0%)
3	70 (23.3%)
4	126 (42.0%)
5	81 (27.0%)

Table 4: Distribution of human evaluation scores for GPT-3.5-Turbo feedback quality.

	mean	std	min	25%	50%	75%	max
Pass ratio	0.342	0.370	0.000	0.000	0.162	0.693	0.985

Table 5: Pass ratio for incorrect code samples in the evaluation set of COFFEE dataset.

A.1.4 Analysis on Machine-generated Test Cases

To gain insights into the effectiveness of our machine-generated test cases, we conduct analyses exploring two key aspects: validity and diversity.

Validity of test cases. A critical question in evaluating our test suite is whether any incorrect solutions manage to pass all the test cases. To address this, we conducted an experiment using the evaluation set of the COFFEE dataset. We randomly sampled 200 wrong code instances and calculated the pass ratios of the wrong codes. We show the statistics of the distribution of pass ratios.

As shown in Table 5, the maximum pass ratio is 0.985, which suggests that there are no wrong solutions that passed all the test cases. The mean score is 0.342, indicating that on average, wrong solutions fail the majority of the test cases. We further analyze the COFFEE-TEST and verified that no wrong solutions pass all the test cases.

Diverse difficulty of test cases. To demonstrate that our generated test cases cover a range of difficulties, we analyzed the pass ratio distribution for incorrect code samples annotated in the dataset. We focused on a single problem from the COFFEE evaluation set.

As shown in Figure 9, the results revealed that various incorrect solutions for this problem exhibited different pass ratios, indicating that our test cases encompass diverse difficulty levels.

A.1.5 Data Analysis

We conduct following experiments to explore original features in COFFEE dataset.

Length of edit trace We analyze the distribution of average length of edit trace by problem level. In
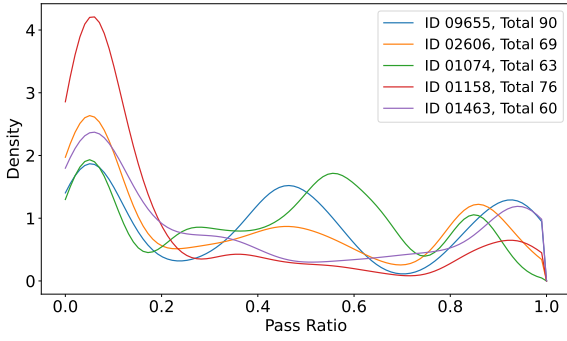
Figure 9: Kernel Density Estimation plot of the pass ratio distribution for incorrect code samples.

Figure 5.a, we observe a steady increase in the average length of edit traces from human programmers with increasing difficulty levels. This suggests that problems in COFFEE are challenging for human programmers, as they tend to make more incorrect submissions for problems with higher difficulty levels.

Code diversity. To assess the diversity of human-written codes compared to machine-generated codes, we conduct a similarity analysis on error codes. Specifically, we sample problems from COFFEE where more than 100 users submitted solutions and collect the wrong code from these users. We also sample an equal number of wrong codes from ChatGPT and GPT-4 with top-p sampling of $p = 0.95$ and temperature $T = 0.6$. For each set of incorrect solutions sampled from user solutions, ChatGPT, and GPT-4, we use CodeBERT (Feng et al., 2020) to compute embeddings for incorrect solutions and measure cosine similarity for all possible pairs in the set.

Figure 5.b shows the histogram of the number of problems by the average embedding similarity of incorrect solution pairs. We find that machine-generated codes (*i.e.*, ChatGPT, GPT4) tend to be more similar to each other than human-generated codes, indicating that collecting human-generated code allows for more diverse set of wrong code samples.

Code complexity To show that problems in COFFEE are challenging for code LLMs, we measure the code generation performance of GPT-4 using Pass@1 and compare it with the solve rate of human programmers. Note that the latter is given as the metadata from the programming platform and computed as the proportion of correct solutions among all solutions submitted for problems

in COFFEE. The results (Figure 5.c) suggest that even the state-of-the-art LLM, *i.e.*, GPT-4, struggles to produce correct solutions for problems in COFFEE and lags behind human programmers.

A.1.6 Analysis on Train-test Overlap

A possible concern is that the training data in COFFEE might overlap with the test data in the code benchmark (*i.e.*, HumanEval). Therefore, we follow Odena et al. (2021) and measure the amount of identical codes (based on the number of repeated lines) between the training and test data. Figure 10 reports both the fraction and the absolution number of line overlaps between COFFEE and HumanEval. We observe that most solutions in COFFEE do not contain lines that appear in the benchmark dataset which we evaluate our models on.

A.2 Details of COFFEEEVAL

A.2.1 Implementation Details

We use DeepSeekCoder-7b[6] as our backbone model using QLoRA (Dettmers et al., 2023), incorporating 4-bit quantization with a learning rate of 5e-5 and a batch size of 4 for 2 epochs. The training is run on 8 NVIDIA GeForce RTX 3090 GPUs. Regarding the LoRA configuration, we specify the dimension of low-rank matrices as 64, and alpha as 16.

A.2.2 Training Details

Following the approach of Wang et al. (2023a), we train the editor in two phases. The initial phase includes the keywords [Correct] and [Wrong] in the code sequence, while the second phase trains the model without these keywords.

Phase I. We finetune our editor model ϕ using pairwise data of correct edits $(q, y, c^*, y^*) \in \mathcal{D}_{correct}$ and incorrect edits $(q, y, \tilde{c}, \tilde{y}) \in \mathcal{D}_{wrong}$ in COFFEE. During this phase, we additionally append keyword tokens t^* and $\tilde{t}$ ([Correct] and [Wrong] respectively) with the target code sequences y^* and $\tilde{y}$. Therefore, the training objective for the initial phase is defined as:

$$
\begin{aligned}
\mathcal{L}(\phi) = \\
- \sum_{(q,y,c^*,y^*) \in \mathcal{D}_{correct}} \log p_\phi(t^*, y^* \mid q, y, c^*) \\
- \sum_{(q,y,\tilde{c},\tilde{y}) \in \mathcal{D}_{wrong}} \log p_\phi(\tilde{t}, \tilde{y} \mid q, y, \tilde{c}) \quad (3)
\end{aligned}
$$

---

[6]https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct

Phase II. After training the editor in Phase I, we continually train the editor model using the same dataset but without the keyword tokens. Thereby, the training object for Phase II is defined as:

$$\mathcal{L}(\phi) = - \sum_{(q,y,c^*,y^*) \in \mathcal{D}_{correct}} \log p_\phi(y^* \mid q, y, c^*)$$
$$- \sum_{(q,y,\tilde{c},\tilde{y}) \in \mathcal{D}_{wrong}} \log p_\phi(\tilde{y} \mid q, y, \tilde{c}) \quad (4)$$

We used the same hyperparameter settings in both phases and the prompt for training the code editor in Appendix D.3.1,

A.3 Details of Reference Methods in COFFEE-GYM

Preference Tuning. Given a problem description, a wrong code, and the corresponding preference set, we apply Direct Preference Optimization (DPO) (Rafailov et al., 2023) to train our critic. That is, we tune critic model to be biased towards helpful feedback.

PPO. PPO optimizes the following objective:

$$\mathcal{L}_{\text{PPO}}(\theta) =$$
$$\hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$$
$$(5)$$

where $r_t(\theta)$ is the probability ratio between the current policy θ and the old policy θ_{old}, $\hat{A}_t$ is an estimator of the advantage function at timestep t, and ϵ is a hyperparameter that controls the clipping range.

DPO. From SFT model we sample 10 feedback strings and score them with COFFEEEVAL. Among the 10 feedback collect feedback with top-1 score and bottom-1 score and construct preference pair, *i.e.*, (c^+, c^-), for DPO training. Using this dataset, we additionally conduct DPO training on SFT model.

Rejection sampling. From SFT model we sample 10 feedback strings and score them with COFFEEEVAL. Among the 10 feedback we only collect feedback with top-1 score and construct dataset for further training. Using this dataset, we additionally conduct SFT.

Terms and License. For our implementation and evaluation, we use Huggingface, TRL and vLLM

library.[7] Both libraries are licensed under Apache License, Version 2.0. We have confirmed that all of the artifacts used in this paper are available for non-commercial scientific use.

B Experimental Details

B.1 Benchmarks

For our experiments, we consider the following benchmarks:

HumanEvalFix HumanEvalFix is a task of HumanEvalPack, manually curated using solutions from HumanEval (Chen et al., 2021a) for the task of code editing. Given an (i) incorrect code function, which contains a subtle bug, and (ii) several unit tests (*i.e.*, test cases), the model is tasked to correct/fix the function. The dataset consists of 164 samples from the HumanEval solutions, and each sample comes with human-authored bugs across six different programming languages, thus covering 984 bugs in total. The bugs are designed in a way that the code is executed without critical failure but fails to produce the correct output for at least one test case.

We have confirmed that the dataset is licensed under the MIT License and made available for non-commercial, scientific use.

Reason for exclusion. We excluded DebugBench and CodeEditorBench for the following reasons:

- **DebugBench** (Tian et al., 2024) is a debugging benchmark consisting of 4253 instances with 4 major categories and 18 minor types of bugs. The metric is based on the test suites provided by LeetCode, requiring API calls for evaluation. Due to the huge amount of API calls, LeetCode blocked the access during the evaluation, which lacked the accurate scoring. Also, some questions were graded incorrectly even though ground-truth solutions were given. Therefore, we decided not to use DebugBench for evaluation.

- **CodeEditorBench** (Guo et al., 2024b) is the framework designed for evaluating the performance of code editing. Code editing is categorized into four scenarios, debugging, translation, polishing, and requirement switching, where our main focus is on debugging. Similar to DebugBench, ground-truth solutions

---

[7] https://huggingface.co/
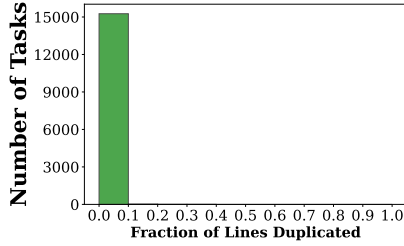
(a) Fraction of line overlaps.
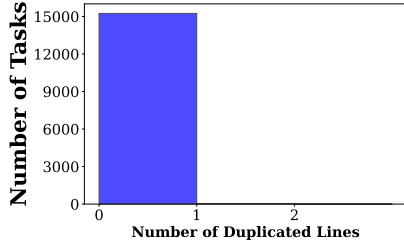
(b) Absolute number of line overlaps.

Figure 10: Analysis on train-test overlap between COF-FEE and HumanEval.

could not pass the unit test for some questions. Also, functions imported from external python files and some specific packages were used in questions without details, which made the question imprecise. So, we sent CodeEditorBench out of our scope.

B.2 Metrics

We use Pass@1 score to measure the code editing performance for all benchmarks. Specifically, Pass@1 is computed as the expected value of the correct rate per problem, when n samples were generated to count the number of correct samples c for each problem.

$$\text{Pass@1} = \mathop{\mathbb{E}}_{\text{Problems}} \left[\frac{c}{n} \right] \times 100 \qquad (6)$$

B.3 Feedback Quality Evaluation

To assess the feedback quality in Likert-scale, we use G-Eval (Liu et al., 2023c) and prompt GPT-4-Turbo to evaluate the feedback quality. Specifically, given problem description, input and output format, wrong code, and the corresponding feedback, we prompt GPT-4 to classify the feedback into one of the following five categories.

- **Completely incorrect**: Feedback has no valid points and is entirely misleading.

- **Mostly incorrect**: Feedback has some valid points but is largely incorrect or misleading.

- **Neutral or somewhat accurate**: Feedback is partially correct but contains significant inaccuracies or omissions.

- **Mostly correct**: Feedback is largely accurate with only minor mistakes or omissions.

- **Completely correct**: Feedback is entirely accurate and provides a correct assessment of the code.

We apply the same top-p sampling and temperature in Table A.1.1 and include the prompt used for the evaluation in Appendix D.3.2.

B.4 Human Evaluation on Quality of Feedback

Task description. The error detection and correction scores were determined by human annotators evaluating feedback on incorrect code using a Likert scale. The error detection score evaluates how accurately the feedback identifies errors in the incorrect code, while the error correction score assesses the correctness and effectiveness of the corrections suggested in the feedback.

Preparing feedback for the evaluation. We aim to analyze the quality of the feedback generated for code editing. We randomly sample 100 codes from COFFEE-TEST to assure the correctness of our evaluation. For generating feedbacks, we use the erroneous codes provided in the dataset.

Details on human evaluation. We conduct human evaluation by using Amazon Mechanical Turk (AMT), which is a popular crowd sourcing platform. As we need workers who have enough experience with Python, we conduct a qualification test to collect a pool of qualified workers. In result, we recruit 186 workers who have passed the test, and task them to evaluate the quality of the feedback on Likert scale, ranging from 1 to 5. Each sample is evaluated by three different raters to ensure the reliability. Based on our estimates of time required per task, we ensure that the effective pay rate is at least $15 per hour. We use the evaluation interface in Figure 12.

C Additional Analysis

C.1 Iterative Editing

Inspired by Zheng et al. (2024), we consider a practical setting where models are tasked with iterative code generation with feedback. We employed
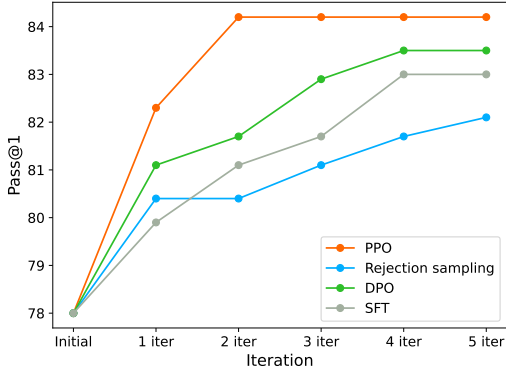
Figure 11: Performance on test cases from HumanEval, measured under the iterative edit setting.

OpenCoderInterpreter-DS-7b as our codeLLM and used our feedback model to provide evaluations on the generated code. Our experiments included comparisons with reference methods in COFFEE-GYM. As shown in Figure 11, using our feedback model consistently enhanced performance over successive iterations. Consistent with our main experiment findings, both PPO and DPO improved feedback quality more effectively than rejection sampling. These results underscore the practical applications of our approach.

C.2 Practical Programming Problems

To further explore the applicability of our feedback model (PPO-COFFEEEVAL) to practical programming problems and assess its robustness across different domains, we conducted experiments using NumpyEval (Zan et al., 2022). This dataset focuses on the general coding domain, specifically involving problems related to the NumPy library. We chose this benchmark to test our model's performance on unseen domains and evaluate its generalizability beyond our initial scope. We utilized OpenCodeInterpreter-DS-Coder-7b as both the generation and editing model, while PPO-CoffeeEval served as the feedback model. To establish a baseline, we compared our approach against a Self-Feedback method, which used OpenCodeInterpreter-DS-Coder-7b for feedback as well.

As shown in Table 6, our PPO-CoffeeEval model outperforms the baseline. These results suggest that our feedback model is not overfitted to Coffee dataset, and did not lost generalization ability to unseen domains.

For further analysis, we conducted a case study to examine the model's performance in more de-

Model	Pass@1
OpenCodeInterpreter-DS-Coder-7b	68.3
+ PPO-COFFEEEVAL	70.3

Table 6: The performance of different feedback models on NumpyEval.

tail. As illustrated in Figure 14 and Figure 15, our model demonstrates the ability to generate helpful feedback even when the problem description is provided in Python comments rather than natural language format. In some instances, the feedback includes the necessary editing code. This capability highlights the potential for using our model in practical scenarios, where users' queries can take various forms and formats, enhancing its applicability in real-world programming environments.

C.3 Case Study on SFT vs. PPO

In Figure 13, we present examples of generated feedback. Although the feedback generated by the SFT model appears plausible, it provides unnecessary feedback which may confuse the editor in feedback-augmented code editing. In contrast, our model (PPO) provides focused and helpful feedback on the incorrect part without unnecessary information. This result aligns with Figure 8, demonstrating that our model generates more accurate and helpful feedback compared to other models.

D Prompts for Our Experiments

D.1 Correct Feedback Annotation Prompt

```
Generate an explanation, analyzation,
and plan to generate code prompt for
the last task considering the example
task instances. Your plan should show
enough intermediate reasoning steps
towards the answer. Construct the plan
as much as you can and describe the
logic specifically. When constructing
the plan for the code prompt, actively
use 'if else statement' to take
different reasoning paths based on the
condition, 'loop' to efficiently
process the repititive instructions, '
dictionary' to keep track of
connections between important variables
.

[Example 1]
Example task instances:
{example_instances_of_task1}

Output format:
{output_format_of_task1}
```

```
Explanation:
{analysis_of_task1}

...

[Example 4]
Example task instances:
{example_instances_of_target_task}

Output format:
{output_format_of_target_task}

Explanation:
```

D.2 Wrong Feedback Annotation Prompt

```
Generate feedback that guides the
refinement from Code before editing to
Code after editing. Assume that the
code after editing is 100% correct and
your feedback should specifically guide
 the editing to the code after editing.
 Please point out only the guidance
from the code before editing to the
code after editing. Do not provide
feedback on the code after editing or
any feedback beyond the code after
editing.

[Example 1]
Problem Description:
{description}

Code before editing:
{wrong_code}

Code after editing:
{next_wrong_code}

Feedback for Refining the Code:
{feedback}

...

[Example 4]
Problem Description:
{description}

Code before editing:
{wrong_code}

Code after editing:
{next_wrong_code}

Feedback for Refining the Code:
```

D.3 Test Case Generation Prompt

```
Given the input format and python code,
 please provide at least 30 challenging
 test input values to evaluate its
functionality.For every start of
```

```
samples, please attach <start> token to
 indicate that the input string has
started. Also, for every end of samples
, please attach <end> token to indicate
 that the input string has ended.

input format:
{input format}

python code:
{python code}

Sample:
```

D.3.1 Code Editor Prompt

```
Provide feedback on the errors in the
given code and suggest the correct code
 to address the described problem.
Description:
{description}
  - output format: {output_format}
  - input format: {input_format}
Incorrect code:
```python
{wrong_code}
```
Feedback:{feedback}

Correct code:
```

D.3.2 G-Eval Prompt

```
You will be provided with feedback on
the given incorrect code. Classify the
accuracy of this feedback using a
Likert scale from 1 to 5, where:

1 (Completely incorrect): This feedback
 has no valid points and is entirely
misleading.
2 (Mostly incorrect): This feedback has
 some valid points but is largely
incorrect or misleading.
3 (Neutral or somewhat accurate): This
feedback is partially correct but
contains significant inaccuracies or
omissions.
4 (Mostly correct): This feedback is
largely accurate with only minor
mistakes or omissions.
5 (Completely correct): This feedback
is entirely accurate and provides a
correct assessment of the code.
Just generate a score from 1 to 5 based
 on the accuracy of the feedback.
Description:
{description}
  - output format: {output_format}
  - input format: {input_format}
Incorrect code:
```python
{wrong_code}
```
```

```
Feedback:{feedback}

Score:
```

We are studying **the quality of feedback** generated by code LLMs.

This evaluation process is designed to assess the quality of feedback generated to solve a given problem description.

Specifically, you will be given a problem description and the code generated to solve it, along with feedback on that code.
You will be asked to check the error detection and correction score of the feedback using a Likert scale, assigning a score between 1 to 5.

Please choose the score that best represents the quality of the feedback.

Guidelines:
Evaluate the quality of feedback based on problem description and the code generated to solve it, choose an appropriate score for the **error detection and correction of the feedback** according to the following scoring guideline.

Problem Description
${description}

Input Format
${input_format}

Output Format
${output_format}

Generated code
`${wrong_code}`

Generated feedback
${prediction}

Error detection Score

This criterion evaluates how accurately the feedback identifies errors in the incorrect code.
A score of 5 indicates that the feedback perfectly identifies all errors.
A score of 1 suggests that the feedback either misses all errors or incorrectly identifies non-issues as errors. A score of 3 is assigned when the feedback only
partially identifies errors or identifies them in a superficial manner, without delving into the underlying issues or complexities.

● 1   ● 2   ● 3   ● 4   ● 5

Error Correction Score

This criterion assesses the correctness and effectiveness of the corrections suggested in the feedback.
A score of 5 signifies that the corrections are entirely correct and well-suited to address the identified issues.
A score of 1 indicates that the corrections are either entirely inappropriate or incorrect.
A score of 3 is given when the suggested corrections are only partially correct or when they address the problem in a rudimentary way,
lacking depth or full understanding of the best practices or more efficient solutions.

● 1   ● 2   ● 3   ● 4   ● 5

Optional feedback?   (expand/collapse)

Submit

Figure 12: The interface used for human evaluation on the feedback.

Problem Description
Input are two strings a and b consisting only of 1s and 0s. Perform binary XOR on these inputs and return result also as a string.

Incorrect Code
| ```python
from typing import List

def string_xor(a: str, b: str) -> str:
    def xor(i, j):
        if i == j:
            return '1'
        else:
            return '0'
    return ''.join(xor(x, y) for x, y in zip(a, b))
``` |

Feedback from SFT model
Your code is almost correct, but you need to convert the input strings to integers before performing the XOR operation, and then convert the result back to a string before returning it.

Feedback from PPO model
Your current logic is incorrect because the xor function should return '1' if the two inputs are different, and '0' if they are the same.

Figure 13: Examples of the feedback from SFT and PPO model in COFFEE-GYM.

NumpyEval
Problem Description
| ```python
import numpy as np
from numpy import newaxis
a = np.array([[1, 2, 3], [3, 4, 5], [5, 6, 7]])
# I have a 2d array with shape (x, y) which I want to convert to a 3d array with shape (x, y, 1).
# Is there a nice Pythonic way to do this?
b =
``` |
| **Feedback from PPO model** |
| Your code is almost correct, but you forgot to add the necessary code to convert the 2d array to a 3d array with shape (x, y, 1).

```python
import numpy as np from numpy
import newaxis
a = np.array([[1, 2, 3], [3, 4, 5], [5, 6, 7]])
b = a[:, :, newaxis]
```

This code will convert the 2d array 'a' to a 3d array 'b' with shape (x, y, 1). |

Figure 14: Examples of the feedback from the PPO model on NumpyEval.

PandasEval
Problem Description

``python
import pandas as pd
import numpy as np
 	
def drop_rows_col_nan(df, col_name):
# How to drop rows of Pandas DataFrame whose value in a certain column is NaN 	
    return
``

Feedback from PPO model

Your code is almost correct, but you need to add a line to drop the rows where the specified column contains NaN values and return the modified DataFrame.
``python
import pandas as pd
import numpy as np
def drop_rows_col_nan(df, col_name):
# How to drop rows of Pandas DataFrame whose value in a certain column is NaN 	
    df = df.dropna(subset=[col_name])
    return df
``
This code will drop the rows where the specified column contains NaN values and return the modified DataFrame.

Figure 15: Examples of the feedback from the PPO model on PandasEval.