Aligning Crowd-sourced Human Feedback for Code Generation with Bayesian Inference

Man Fai Wong
City University of Hong Kong
mfwong29-c@my.cityu.edu.hk

Chee Wei Tan
Nanyang Technological University
cheewei.tan@ntu.edu.sg

***Abstract*—While large language models (LLMs) excel at code generation, translating abstract descriptions into robust and functional code remains a significant challenge. Despite dedicated efforts, existing works for refining code generation with LLMs have demonstrated limitations, either constrained by the static rules or computational overhead of additional training, ultimately proving insufficient to meet the intricate demands of real-world code quality. This paper proposes a method to improve code generation ability with LLM by combining reinforcement learning from human feedback (RLHF) with crowd-sourced computation, referred to as cRLHF. Our goal is to enhance code quality through diverse end-user feedback. Traditional RLHF, relying on a single evaluator, risks biases and overlooks insights, hampering LLMs' growth. The cRLHF framework, powered by Bayesian inference, ensures objective code evaluation from multiple evaluators. Our experiments exhibit significant improvements in code correctness, showcasing the efficacy of crowd-sourcing with reinforcement learning.**

***Index Terms*—AI alignment, Bayesian analysis, Reinforcement learning, Inductive bias, Code generation**

The advent of Artificial Intelligence (AI) has ignited a transformation in human-machine interaction, notably through the advancement of natural language processing (NLP). Over recent years, the evolution of large language models (LLMs) that are capable of comprehending and producing language has opened up innovative avenues in content comprehension and generation [1], [2]. An especially compelling application of LLMs is in AI-assisted programming, enabling developers to expedite innovation in software development [3]–[7]. However, ensuring that LLMs generate code that meticulously aligns with desired requirements and specifications poses a significant challenge. Beyond basic requirements, capturing the more delicate details of developer preferences and addressing implicit considerations remain major hurdles for LLMs, limiting the ability to generate truly tailored and efficient code.

Inductive bias shapes LLM performance and generalization by embedding assumptions into the model architecture, yet can also impose limitations. By harnessing diverse human inputs, crowd-sourced evaluations or feedback can effectively counteract the inherent biases in LLMs. The collective wisdom of a crowd can enable the identification and correction of biases encoded during training, thereby enhancing adaptability and generalization capabilities [8]–[12]. Integrating human feedback into LLM development and refinement presents a promising approach to mitigating inductive biases and tackling the LLM alignment challenge. This includes reinforcement learning (RL), human-assisted computation, and fine-tuning algorithms. RL from Human Feedback (RLHF) uniquely integrates human insights into training, enabling effective handling of complex tasks [13]–[15]. However, depending solely on raw human feedback may prove counterproductive, especially when RL systems encounter high uncertainty and unreliable input. *Given the constraints of human evaluators, crowd-sourcing aims to extract maximum value from their input, minimizing user involvement while maximizing learning gains for RL agents.*

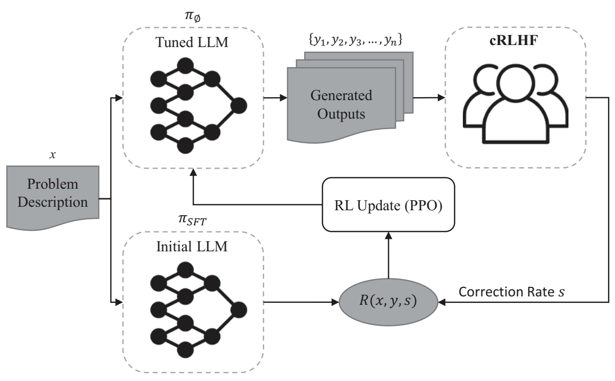


Fig. 1: Integration of the cRLHF framework workflow with the reinforcement learning framework utilizing proximal policy optimization (PPO) for code generation.

This paper introduces cRLHF, a novel framework harnessing crowd-sourced human feedback from diverse sources. By optimizing the final reward score, it simplifies the conventional RLHF feedback process shown in Figure 2 by eliminating the requirement for an additional reward modeling [16]. We establish a crowd-sourced framework rooted in Bayesian inference for self-assessment from multiple sources, targeting explicitly code generation tasks. The open nature of cRLHF empowers a diverse community of participants to collaboratively refine LLM outputs through their annotations. This framework proves particularly valuable when prior knowledge is limited, as it empowers a diverse collective to actively uncover errors. We fine-tune baseline LLMs and evaluate their performance on code generation tasks, both with and without cRLHF integration, to measure the impact of our framework on benchmark scores directly. In summary, the contributions

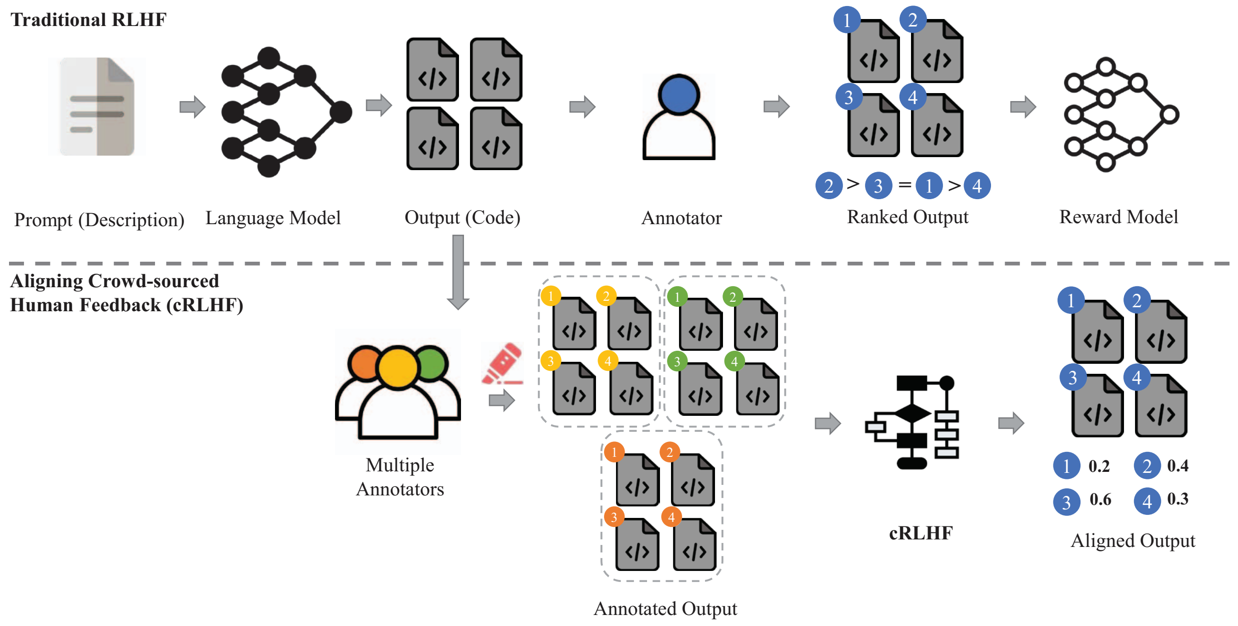


Fig. 2: Comparison between traditional RLHF and crowd-sourced human feedback strategy for code generation. The upper part shows the traditional method with a single annotator ranking outputs forwarded to the reward model. The lower part depicts the crowd-sourced strategy involving multiple annotators, computing consensual outputs based on ranking or reward scores.

of our work in this study are as follows:

- We present cRLHF, a unique framework for integrating crowd-sourced human feedback in RLHF via Bayesian inference. This approach combines input from multiple annotators without requiring extra reward modeling, leading to notable enhancements in code quality.
- We design a systematic computational approach based on Bayesian inference, specifically for code annotation tasks within our cRLHF framework.
- We demonstrate that cRLHF improves the base model on an established benchmark by different experiments, achieving code quality improvement compared for LLMs.

I. Related Works

Code generation encompasses methods like code understanding and generation [6]. More recently, transformer-based LLMs through attention mechanism [17] have demonstrated remarkable prowess in code generation tasks [18]. Code generation can be considered as the sequence generation task, which can be optimized by the RL approaches [19]. Recent progress has been achieved in utilizing human feedback [13]–[15], a paradigm incorporating human insights into the learning process. Unlike solely depending on rewards from the environment, this method uses human preferences as a guiding direction. Human feedback can be gathered and prioritized through interactions with dialogue agents, as demonstrated in previous works [20], [21]. Humans play the role of providing feedback or ranking the generated output in the learning process of the RL system. The input of human feedback enriches the capacity of the system to align its decisions with human expectations. Several recent studies have focused on the aspect of alleviating the process with RLHF. In the study by [22], a new alignment algorithm was introduced within the alignment pipeline for generative models, encompassing not only LLMs. For instance, [23] proposed integrating natural language feedback into the RL process to enhance code generation, while [24] introduced a new policy optimization algorithm as a substitute for policy gradient methods to better align LLMs with human preferences.

II. Methodology

A. Problem Description

To begin, we provide a formal description of the problem in this study. We mainly follow the setting in [25] for utilizing human feedback with LLM in this study. The workflow of our cRLHF framework is illustrated in Figure 1. Given a programming problem description $x \in X$, the supervised fine-tuned model π_{SFT} aims to translate the description from natural language into a code snippet or program y. A reward function $R(x, y, s)$ with a reward score s computed from our crowd-scouring framework based on all human feedback. Our objective is to fine-tune a model π_θ, initially set as π_{SFT} from supervised fine-tuning (SFT), so as to generate responses that return maximal rewards.

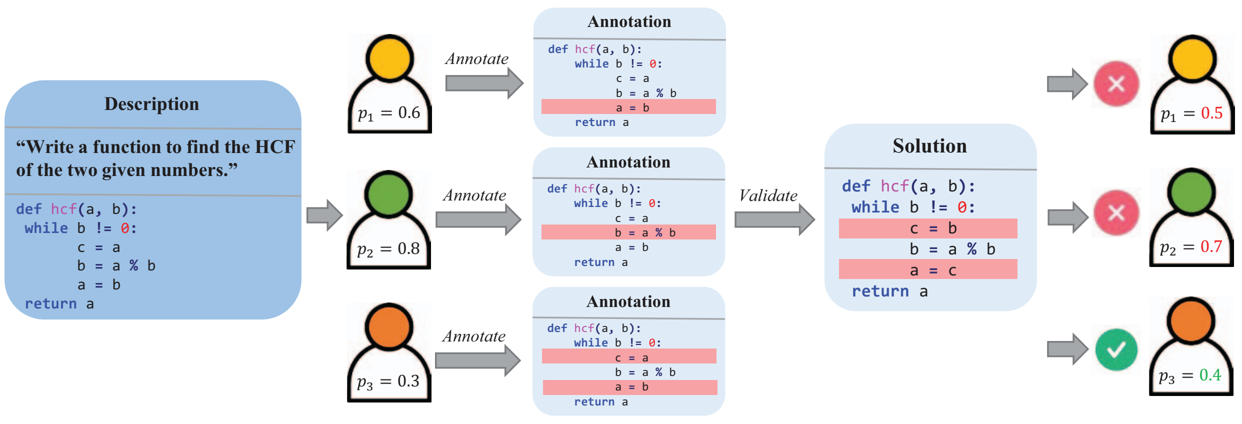


Fig. 3: An overview of the process for evaluating p_i across all annotators. Displayed alongside descriptions and code examples on the left-hand side of the figure, each annotator, along with their corresponding p_i values, is tasked with identifying errors within the code. Subsequently, the system evaluates the accuracy of the annotations and adjusts the p_i values accordingly.

B. Crowdsourced Feedback Alignment for Code Generation

In this section, we describe cRLHF for aligning crowd-sourced human feedback to rank the generated output from LLMs. cRLHF computes the s from multiple annotators as a reward function. For input $\{x_1, x_2, \ldots, x_q, \ldots, x_t\} \in X$, the collection of all generated code snippets from π_θ can be represented as $(y_1, y_2, y_3, \ldots, y_n)$ of $y \in Y$, where n signifies the count of generated outputs. For the input $\{x_1, x_2, \ldots, x_q\}$, we know the exact correctness for each generated output. Our human feedback is procured from a group of m annotators, denoted as $(a_1, a_2, a_3, \ldots, a_m)$ of $a \in A$. These annotators then are tasked with providing annotations for each y. In this work, our annotation methodology revolves around evaluating each line l of the program, wherein $y \in \{l_1, l_2, l_3, \ldots, l_k\}$ with k denoting the number of lines. Workers annotate each line of code using labels such as "correct" for error-free code or "wrong" for buggy code, which are represented by values $\{1, -1\}$. For each line of code, it has a prior probability of being correct and the prior probability is $\frac{1}{2}$. For l_i, the correctness is represented by L_0, where

$$L_0 = \begin{cases} 1 & \text{if the } l_i \text{ is true} \\ -1 & \text{otherwise.} \end{cases} \tag{1}$$

We begin with the assumption that the system has no prior knowledge of the value of L_0, and its task is to compute the likelihood of l_i being true ($P(L_0 = 1)$) based on the input of annotators. Let the prior probability of l_i being true be p_0, i.e.

$$p_0 = P(L_0 = 1) = \frac{1}{2}.$$

Consider the response from an annotator a_1 as L_1, with a corresponding probability of correctness denoted as p_1:

$$p_1 = P(L_1 = 1 \mid L_0 = 1) = P(L_1 = -1 \mid L_0 = -1).$$

From Bayes' Theorem, we have:

$$P(L_0 = 1 \mid L_1 = 1) = \frac{P(L_1 = 1 \mid L_0 = 1)P(L_0 = 1)}{P(L_1 = 1)},$$

which, using p_0 and p_1, can be further rewritten as:

$$P(L_0 = 1 \mid L_1 = 1) = \frac{p_1 p_0}{p_1 p_0 + (1 - p_1)(1 - p_0)}.$$

By introducing the logit function [26], we can further simplify the probability into:

$$P(L_0 = 1 \mid L_1 = 1) = \text{logit}^{-1}(\text{logit}(p_0) + \text{logit}(p_1)), \tag{2}$$

where the logit function is defined as $\text{logit}(\cdot) = \log(\frac{\cdot}{1-\cdot})$, and $\text{logit}^{-1}(\cdot) = \frac{\exp \cdot}{1+\exp \cdot}$. Similarly we have:

$$P(L_0 = 1 \mid L_1 = -1) = \text{logit}^{-1}(\text{logit}(p_0) - \text{logit}(p_1)). \tag{3}$$

Combining (2) and (3), we get:

$$P(L_0 = 1 \mid L_1 = \epsilon_1) = \text{logit}^{-1}(\text{logit}(p_0) + \epsilon_1 \text{logit}(p_1)). \tag{4}$$

Given that the available options for annotators are restricted to $\{1, -1\}$ within this system, it follows that $\text{logit}(p_0) = 0$. With this insight, we can simplify (4) to:

$$P(L_0 = 1 \mid L_1 = \epsilon_1) = \text{logit}^{-1}(\epsilon_1 \text{logit}(p_1)). \tag{5}$$

We then consider another annotator a_2 contributes their response as L_2 with an independent probability p_2 of being correct, and we can regard the posterior probability $P(L_0 = 1 | L_1 = \epsilon_1)$ as our revised prior. Thus, we have:

$$\begin{aligned} &P(L_0 = 1 \mid L_1 = \epsilon_1, L_2 = \epsilon_2) \\ &= \text{logit}^{-1}(\text{logit}(P(L_0 = 1 | L_1 = \epsilon_1) + \epsilon_2 \text{logit}(p_2)) \\ &= \text{logit}^{-1}(\epsilon_1 \text{logit}(p_1) + \epsilon_2 \text{logit}(p_2)). \end{aligned} \tag{6}$$

Building upon the analysis involving two annotators as discussed earlier, we now incorporate feedback obtained from n annotators regarding l_1:

$$P(L_0 = 1 \mid L_1 = \epsilon_1, \cdots, L_n = \epsilon_n) = \text{logit}^{-1}(\sum_{i=1}^{n} \epsilon_i \text{logit}(p_i)). \tag{7}$$

We have thus established a human-assisted computational procedure (e.g., [27]) to evaluate the p_i values as illustrated in Figure 3. To calculate the p_i value for each annotator, we start with a parameter ν as the initial value of p_i. With the initial set of q annotation tasks, the system receives correct answers from the dataset X, which it uses to refine each annotator's p_i values based on the problem sets and annotator responses. In this fine-tuning process, the annotators' p_i values are adjusted incrementally based on the accuracy of their annotations for the q tasks, using a constant step size. The adjustment's magnitude is dictated by the $\bar{p}$ value, acting as a modulating factor for the adjustment scale. This modulation guarantees that the degree of adjustment corresponds to the system's confidence in its initial response. Following the annotation of each program within the set of p tasks, the system then updates each annotator's p_i using the approach:

$$p_i^* = \text{logit}^{-1}(\text{logit}(p_i) + \lambda\mu \ \text{logit}(\bar{p})), \tag{8}$$

where p^* is the updated prior probability value, μ is the correctness of the annotator's response to the particular l_i ($\mu = 1$ for correct annotation, $\mu = -1$ for incorrect response), λ is a hyper-parameter, and $\bar{p}$ is the certainty that the response used by the system for annotator is correct. For the initial questions and the evaluation questions, $\bar{p}$ would be 1, meaning that the system is absolutely sure about the answer. For other questions, $\bar{p}$ would be the value obtained from (6).

For the generated outputs, denoted as y and comprising a set of possible lines of code $\{l_1, l_2, l_3, \ldots, l_k\}$, where k represents the total number of lines, we can evaluate the accuracy of each line using equation (7). The resulting count is labeled as c, serving as the basis for calculating the *correction rate* on a line-to-line basis. We can integrate this score, denoted as $s = c/k$, into the system without relying on reward modeling, similar to existing preference-based RL approaches [28], [29]. The aligned score for each y in the system ranges of $[0, 1]$. We then construct a dataset S, comprising triplets (x, y, s) that capture annotators' feedback for all generated outputs of y to rank score directly, which is to map a given description and its corresponding code to a reward value s. Once the system computes the s for each generated output from the annotators, it follows the approach outlined in [25] to integrate the proximal policy optimization (PPO) algorithm [30] into the RL system. To integrate our framework, as shown in Figure 1, we illustrate the algorithmic procedure in Algorithm 1.

III. Experimental Evaluations

In this section, we describe the experimental setup, the baselines and evaluation metrics for performance evaluation.

Algorithm 1 cRLHF with LLM for Code Generation

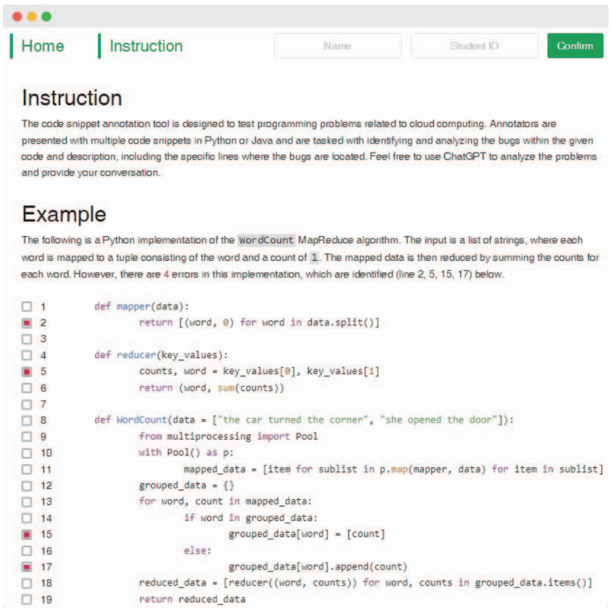
```
Input: Dataset $X$, SFT Model $\pi_{SFT}$
Output: Tuned Model $\pi_\theta$
Parameters: Threshold $\tau$
1:  $\pi_\theta \leftarrow \pi_{SFT}$
2:  for Problem Input $\{x_1, x_2, \ldots, x_q, \ldots, x_k\} \in X$ do
3:      if $\{x_1, x_2, \ldots, x_q\}$ then
4:          for Annotators $a \in A$ do
5:              $\{p\} \leftarrow$ Update by eq. (8)
6:          end for
7:      else
8:          $\{y_1, y_2, y_3, \ldots, y_n \in Y\} \leftarrow \pi_\theta(x)$
9:          for Generated Output $y \in Y$ do
10:             $c, k \leftarrow 0$
11:             for Annotators $a \in A$ do
12:                 $\{\epsilon\} \leftarrow$ Annotate($y$)
13:             end for
14:             for Each Line $l \in y$ do
15:                 $k \leftarrow k + 1$
16:                 if eq. (7) $> \tau$ then
17:                     $c \leftarrow c + 1$
18:                 end if
19:             end for
20:             $s \leftarrow c/k$
21:             for Annotators $a \in A$ do
22:                 $\{p\} \leftarrow$ Update by eq. (8)
23:             end for
24:         end for
25:         $\pi_\theta \leftarrow$ PPO($x, y, s$)
26:     end if
27: end for
```

A. Human Feedback Collection

To generate crowd-sourced human feedback for evaluation purposes, we structure annotation tasks akin to assigning coursework on an online platform designed for code annotation, as illustrated in Figure 4. To obtain diverse feedback for software bug identification, we enlist 30 human annotators to evaluate 20 programming descriptions per baseline model, each containing 10 code outputs with defined correct answers. While annotators are selected to represent varying skill levels, individual annotator accuracy p_i is dynamically adjusted during the experiment using the method in Section II-B for evaluation. To initialize p_i for each annotator, we assign them five tasks with known correct answers, following the approach in [27]. Next, for every problem description, we provide four generated code options. Annotators are then tasked with identifying any errors present within these provided codes.

B. Dataset and Evaluation Metric

We focus mainly on code generation tasks by fine-tuning the baseline models on the code generation dataset from [31]. We evaluate cRLHF on established code generation benchmarks from HumanEval [32] and MBPP [33]. We use



Instruction

The code snippet annotation tool is designed to test programming problems related to cloud computing. Annotators are presented with multiple code snippets in Python or Java and are tasked with identifying and analyzing the bugs within the given code and description, including the specific lines where the bugs are located. Feel free to use ChatGPT to analyze the problems and provide your conversation.

Example

The following is a Python implementation of the `WordCount` MapReduce algorithm. The input is a list of strings, where each word is mapped to a tuple consisting of the word and a count of `1`. The mapped data is then reduced by summing the counts for each word. However, there are 4 errors in this implementation, which are identified (line 2, 5, 15, 17) below.

```
☐ 1    def mapper(data):
☒ 2        return [(word, 0) for word in data.split()]
☐ 3
☐ 4    def reducer(key_values):
☒ 5        counts, word = key_values[0], key_values[1]
☐ 6        return (word, sum(counts))
☐ 7
☐ 8    def WordCount(data = ["the car turned the corner", "she opened the door"]):
☐ 9        from multiprocessing import Pool
☐ 10       with Pool() as p:
☐ 11           mapped_data = [item for sublist in p.map(mapper, data) for item in sublist]
☐ 12       grouped_data = {}
☐ 13       for word, count in mapped_data:
☐ 14           if word in grouped_data:
☒ 15               grouped_data[word] = [count]
☐ 16           else:
☒ 17               grouped_data[word].append(count)
☐ 18       reduced_data = [reducer((word, counts)) for word, counts in grouped_data.items()]
☐ 19       return reduced_data
```

Fig. 4: An online crowdsourcing platform for code annotation tasks. Annotators will receive various code snippets generated by LLMs with the corresponding descriptions. Their task involves annotating lines of the program that contain errors.

the `pass@k` metric in [32] to measure functional correctness, which quantifies the percentage of problems solved.[1] To maintain consistency throughout our experiments, we assess `pass@1`, `pass@10`, and `pass@100` using fixed temperature values of 0.2, 0.6, and 0.8 for all evaluations.

C. Baseline Models

We leverage eight publicly available pre-trained language models for our code generation experiments with cRLHF. The first baseline, POLYCODER [34], utilizes the GPT-2 with parameters set at 400M and 2.7B. The second model, CODEGEN-MULTI [35], is trained on distinct datasets of different scales, featuring parameter sizes of 350M and 2B. Furthermore, CODEPARROT [36], derived from GPT-2, employs parameter sizes of 110M and 1.5B. Finally, GPT-NEO [37] with 1.3B and 2.7B parameters is included in our study.

In our study, ensuring a fair evaluation is paramount, particularly given the potential for certain baseline models to be trained on benchmark data. To mitigate any biases that may arise from this, all models undergo SFT are then integrated as π_{STF} within our cRLHF framework. This step is crucial for maintaining fairness in evaluation.

[1]Pass@k is the unbiased estimator for the probability that the correct answer is within k samples given a total number of n samples and c unit tests, i.e., pass@$k = \mathbb{E}\left[1 - \binom{n-c}{k}/\binom{n}{k}\right]$ [32].

Models	**Baseline**						**w/cRLHF**					
	HumanEval			**MBBP**			**HumanEval**			**MBBP**		
	1	10	100	1	10	100	1	10	100	1	10	100
PolyCoder-400M	1.65	4.57	11.3	0.08	0.71	1.40	1.86	5.12	11.6	0.12	1.33	4.26
PolyCoder-2.7B	5.94	10.0	17.1	1.48	7.80	25.2	6.03	10.2	17.3	1.50	8.71	26.8
CodeGen-350M	5.53	10.9	17.3	2.87	16.0	35.0	6.31	12.1	20.0	3.27	18.3	40.0
CodeGen-2.7B	11.7	25.7	39.8	10.6	32.6	54.1	13.3	29.3	**45.4**	12.1	37.2	**59.7**
CodeParrot-110M	3.66	6.25	11.3	0.13	0.12	6.40	4.13	6.33	12.4	0.22	1.60	8.33
CodeParrot-1.5B	4.42	8.47	15.7	0.40	3.15	13.0	4.91	9.41	17.4	0.66	4.31	16.4
GPT-Neo 1.3B	4.24	6.26	7.85	0.34	2.90	14.2	4.71	6.95	8.72	0.58	4.17	16.8
GPT-Neo 2.7B	5.76	10.8	20.1	1.56	7.72	21.6	5.97	11.2	20.7	1.61	8.00	22.4

TABLE I: Evaluation results on the HumanEval and MBBP benchmark. Each `pass@k[%]` for each model is computed with three sampling temperatures ($t = 0.2, 0.6, 0.8$) and the highest one among the three are displayed. The table is divided into two sections: Baseline models and models with crowd-sourced feedback. Each cell in the table represents the pass rate percentage for a particular model, metric, and k value.

IV. FURTHER DISCUSSIONS

A. Numerical Results

The main numerical results from our experiments with different models are presented in Table I. This table provides a summary of the outcomes obtained from the benchmarks for both the original LLM and the model incorporating cRLHF.

The fine-tuned models with the cRLHF outperformed baselines on established code generation benchmarks, highlighting the effectiveness of crowd-sourced RLHF. The results provide a comprehensive overview of how different-sized LLMs performed in generating code from natural language descriptions, highlighting the improvements achieved by our cRLHF. CodeGen models demonstrated a stronger performance boost in the larger model (45.4% from 39.8% for CodeGen-2.7B vs. 20.0% from 17.3% for CodeGen-350M on HumanEval with $k = 100$), while CodeParrot models exhibited a more pronounced improvement in the smaller model achieving a 9% improvement and the larger model exceeding 10% on HumanEval, emphasizing the conditional nature of cRLHF's influence. For cRLHF, akin to the baseline, models with larger parameters consistently demonstrate better performance across all benchmarks. However, even models with similar parameters, such as model with 2.7B size, can yield significantly varied outcomes dues to differences in their architecture and the pre-trained dataset used. Remarkably, CodeGen-350M manages to achieve best performance for all smaller models on two benchmarks, surpassing even larger models.

B. Discussion

Our findings suggest a significant impact of cRLHF on code generation. The consistent and notable improvements observed when integrating cRLHF into existing baseline models across various benchmarks enhance the quality and relevance of generated code. Through Bayesian analysis, we show that human feedback can affect the generalizability of LLM code generation by encompassing differences in architecture, pre-training or fine-tuning datasets, parameter size, and the inherent randomness of generated output. Our findings offer some initial theoretical insights into bridging the gap between human-in-the-loop frameworks and modern LLM frameworks.

V. Conclusion

This paper introduces cRLHF, a framework that integrates RL and human feedback to enhance code generation in LLMs. Our cRLHF framework uses Bayesian statistics to combine rankings from different assessors, guaranteeing accurate reward scores without the need for complex additional reward modeling. This advancement improves AI-assisted programming, enabling developers to tackle complex system development using LLMs. Our comprehensive evaluation using a widely recognized benchmark with human annotators demonstrates consistent improvements over existing baseline models in various code generation tasks, highlighting the efficacy and practical utility of the cRLHF framework. Future research could extend our framework to enhance software security within code generation by leveraging human feedback to guide the creation of secure and robust software implementations.

Acknowledgement

This work is supported by the Singapore Ministry of Education Academic Research Fund (RG91/22 and NTU startup) with computation partially performed on resources of the National Supercomputing Centre, Singapore (Project 12003619).

References

[1] OpenAI, “ChatGPT: Optimizing language models for dialogue,” Jan 2023. [Online]. Available: https://openai.com/blog/chatgpt/

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond *et al.*, “Competition-level code generation with Alphacode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[3] E. W. Dijkstra, “A preliminary investigation into computer assisted programming,” *E. W. Dijkstra Archive (EWD 237)*, 2007.

[4] R. E. Handsaker, “Code generation in the programmer’s apprentice,” MIT AI Lab, Working Paper 233, May 1982.

[5] E. W. Dijkstra, “The humble programmer,” *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[6] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, “Natural language generation and understanding of big code for AI-assisted programming: A review,” *Entropy*, vol. 25, no. 6, p. 888, 2023.

[7] N. Alshahwan, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, “Assured LLM-based software engineering,” *arXiv:2402.04380*, 2024.

[8] F. Galton, “Vox populi (the wisdom of crowds),” *Nature*, vol. 75, pp. 450–451, 1907.

[9] L. Von Ahn, M. Blum, and J. Langford, “Telling humans and computers apart automatically,” *Communications of the ACM*, vol. 47, no. 2, pp. 56–60, 2004.

[10] L. Von Ahn, R. Liu, and M. Blum, “Peekaboom: a game for locating objects in images,” in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006, pp. 55–64.

[11] N. E. Leonard and S. A. Levin, “Collective intelligence as a public good,” *Collective Intelligence*, vol. 1, no. 1, 2022.

[12] E. Bondi, R. Koster, H. Sheahan, M. Chadwick, Y. Bachrach, T. Cemgil *et al.*, “Role of human-AI interaction in selective prediction,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 5, 2022, pp. 5286–5294.

[13] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[14] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss *et al.*, “Learning to summarize with human feedback,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 3008–3021, 2020.

[15] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin *et al.*, “Training language models to follow instructions with human feedback,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[16] A. Singh, L. Yang, K. Hartikainen, C. Finn, and S. Levine, “End-to-end robotic reinforcement learning without reward engineering,” *Robotics: Science and Systems*, 2019.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[18] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena *et al.*, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[19] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba, “Sequence level training with recurrent neural networks,” in *International Conference on Learning Representations*, 2016.

[20] A. Glaese, N. McAleese, M. Trębacz, J. Aslanides, V. Firoiu, M. Rauh *et al.*, “Improving alignment of dialogue agents via targeted human judgements,” *arXiv:2209.14375*, 2022.

[21] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma *et al.*, “Training a helpful and harmless assistant with reinforcement learning from human feedback,” *arXiv:2204.05862*, 2022.

[22] H. Dong, W. Xiong, D. Goyal, R. Pan, S. Diao, J. Zhang *et al.*, “Raft: Reward ranked finetuning for generative foundation model alignment,” *arXiv:2304.06767*, 2023.

[23] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman *et al.*, “Improving code generation by training with natural language feedback,” *arXiv:2303.16749*, 2023.

[24] R. Ramamurthy, P. Ammanabrolu, K. Brantley, J. Hessel, R. Sifa, C. Bauckhage *et al.*, “Is reinforcement learning (not) for natural language processing: Benchmarks, baselines, and building blocks for natural language policy optimization,” 2023.

[25] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei *et al.*, “Fine-tuning language models from human preferences,” *arXiv:1909.08593*, 2019.

[26] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 02 2008, vol. 2.

[27] L. Ling and C. W. Tan, “Human-assisted computation for auto-grading,” in *IEEE International Conference on Data Mining Workshops*, 2018.

[28] J. Hejna and D. Sadigh, “Inverse preference learning: Preference-based RL without a reward function,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[29] G. An, J. Lee, X. Zuo, N. Kosaka, K.-M. Kim, and H. O. Song, “Direct preference-based policy optimization without reward modeling,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv:1707.06347*, 2017.

[31] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, “XLCoST: A benchmark dataset for cross-lingual code intelligence,” 2022.

[32] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan *et al.*, “Evaluating large language models trained on code,” *arXiv:2107.03374*, 2021.

[33] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan *et al.*, “Program synthesis with large language models,” *arXiv:2108.07732*, 2021.

[34] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Deep Learning for Code Workshop*, 2022.

[35] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou *et al.*, “CodeGen: An open large language model for code with multi-turn program synthesis,” *International Conference on Learning Representations*, 2023.

[36] L. Tunstall, L. Von Werra, and T. Wolf, *Natural Language Processing with Transformers*. " O’Reilly Media, Inc.", 2022.

[37] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, “GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow,” 2021.