Smaller Models, Smarter Rewards: A Two-Sided Approach to Process and Outcome Rewards

Jan Niklas Groeneveld*
University of California, Irvine
Irvine, CA 92697
jan.groeneveld@outlook.de

Xi Qin
SAP Lab
Palo Alto, CA 94304
grace.qin@sap.com

Alexander Schaefer
SAP Lab
Palo Alto, CA 94304
alexander.schaefer01@sap.com

Yaad Oren
SAP Lab
Palo Alto, CA 94304
yaad.oren@sap.com

Abstract

Generating high-quality code remains a challenge for Large Language Models (LLMs). For the evolution of reasoning models on this task, reward models are a necessary intermediate step. These models judge outcomes or intermediate steps. Decoder-only transformer models can be turned into reward models by introducing a regression layer and supervised fine-tuning. While it is known that reflection capabilities generally increase with the size of a model, we want to investigate whether state-of-the-art small language models like the Phi-4 family can be turned into usable reward models blending the consideration of process rewards and outcome rewards.

Targeting this goal, we construct a dataset of code samples with correctness labels derived from the APPS coding challenge benchmark. We then train a value-head model to estimate the success probability of intermediate outputs. Our evaluation shows that small LLMs are capable of serving as effective reward models or code evaluation critics, successfully identifying correct solutions among multiple candidates. Using this critic, we achieve over a 20% improvement in the search capability of the most accurate code out of multiple generations.

1 Introduction

While several improvements have been achieved on reasoning-related tasks through techniques like chain-of-thought prompting, model improvement through bootstrapping (Zelikman et al. [2022]), tree-of-thought decoding (Yao et al. [2023]), and explicit policy optimization through the tree (Feng et al. [2024]), the first major breakthrough in reasoning capabilities came with OpenAI's o1 series and its successor o3, o4-mini and GPT-5 thinking. The inner workings of these models are not publicly known, which made Deepseek-R1 (DeepSeek-AI and et al. [2025]) the first public model to offer advanced reasoning capabilities. This model was shortly followed by two other Chinese models, Kimi k1.5 (Team and et al. [2025]) and DAPO (et al. [2025]), which gave similar performance and offered more insights into the training and algorithms.

When ChatGPT 3.5 was released in 2022, its success was enabled through a technique called "Reinforcement Learning Through Human Feedback" (RLHF). A dedicated reward model in the

*The work of the paper was performed when he was an research internship under SAP Lab, in collaboration with Stanford Human-Centered AI Institution.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Foundations of Reasoning in Language Models.

decoder-only structure learned human preferences, and the generative model was fine-tuned using this reward signal (Ouyang et al. [2022]). Instead of a classification head to predict the next token, their last layer often predicts just a single signal: the score they assign to the text up to that token (Zhong et al. [2025]).

When looking at the inner workings of reasoning models, all seem to share the same approach of generating a lengthy chain of thought before outputting the final result. This chain of thought helps the models bridge the gap between the prompt and the potentially complex answer, providing helpful intermediate steps that extend the context used by the model to predict the tokens the answer is made of. When there are mistakes or hallucinations in the reasoning trace, or intermediate generations are unstructured, this can easily derail a generation and lead to a wrong output.

Hence, the training process of a reasoning model aims to increase the probability of *good* reasoning traces (those leading to a correct result), and decrease the probability of *bad* traces.

Recent work on reward modeling distinguishes process reward models (PRMs) – which score intermediate steps – from outcome reward models (ORMs) – which score only final outputs (Wang et al. [2023], Lightman et al. [2023b]). ORMs provide a single holistic score at the end of generation, while PRMs offer step-wise feedback, enabling fine-grained supervision and improving credit assignment and sample efficiency in multi-step tasks (Choudhury [2025], Cui et al. [2025]). ORMs are simpler and cheaper but suffer from sparse signals and delayed credit assignment (Cui et al. [2025], Wang et al. [2025]). However, PRMs require fine-grained labels during training—often needing costly human annotations for intermediate steps (Cui et al. [2025]). In contrast, ORMs typically demand many more episodes to learn, which can make RL impractical for complex tasks without auxiliary signals (Wang et al. [2025], Choudhury [2025]).

Recent work blends PRM and ORM signals or adopts hierarchical schemes. Wang et al. [2025] introduce Hierarchical Reward Models (HRM), which combine coarse outcome and fine process scores, allowing later steps to override earlier mistakes. Other work like ThinkPRM (Khalifa et al. [2025]) and PathFinder-PRM (Deep Pala et al. [2025]) focus on data-efficient PRM training and calibration at inference, enhancing the reward quality. In parallel, implicit reward models (LLM-as-judge to collect outcome feedback) like PRIME (Cui et al. [2025]) are explored to sidestep expensive PRM annotation.

Inspired by the strengths and drawbacks in PRMs and ORMs, we aim to build a versatile reward model that can serve both roles. This paper investigates how well fine-tuned decoder-only LLMs can act as reward models, specifically in the context of Python coding applications. Our key contributions in this paper are the following:

- We replace the last layer of Phi-4 models by a single output with sigmoid function. This architecture are successfully finetuned into a reward model in just two episodes, achieving promising results.
- We demonstrate that our trained reward model can serve as an effective lightweight critic and also a confident intermediate step evaluator, improving the correctness of Python code generation by reliably selecting correct rollouts by 20+% (refer to Appendix 7.4 for details).
- We extensively analyze the model's scoring of full rollouts and intermediate reasoning steps, uncovering valuable insights into its evaluation behavior.

2 Related Work

An early work on tree-based decoding strategies was made by Yao et al. [2023]. The authors introduced the terminology "Tree of Thought", and evaluated a tree-based reasoning process on different reasoning tasks. They reported an immense increase in success rates on the benchmarks they evaluated. There are two main differences to our work. First, they used a different level of granularity, where a "thought" is a for example sentence. Our work, in comparison, uses tokens as the level of granularity. Second, for judging intermediate steps, the authors prompted LLMs via a text interface. This model also answered in natural language, and didn't output a single number. Moreover, the team mainly evaluated on non-coding reasoning tasks, while our work is focused on coding challenges.

A follow-up paper on "Tree of Thought" written by Feng et al. [2024] is about decoding with Monte Carlo Tree Search. This paper uses a similar verdict mechanism to estimate the state value of an

intermediate step, and uses this to do Monte Carlo Tree Search. While the authors reported good results in their decoding, they did not discuss the performance of their value estimator. We look into that topic and provide a comprehensive discussion. Additionally, they used sentence-based granularity and focused on non-coding tasks.

Similar work was published by Yu et al. [2024] who also proposed a decoder-based judge. Their paper focused on introducing the concept called "Outcome-supervised Value Model", classifying it into the frameworks of outcome reward models and process reward models, the two most common ways of rewarding reasoning models. They argued the value model approach to be superior over process-based rewarding, because the value estimation contains a forecast into the future, compared to process reward models rewarding correct steps in the past. Like the previously discussed paper, they also used math datasets to assess their performance.

Other works like Lightman et al. [2023a] compare outcome reward models to process reward models and find superior performance for process reward models. However, their approach for ORMs didn't judge intermediate steps, but sampled whole rollouts instead.

Anthropic's work (Kadavath et al. [2022]) sheds light on the self-evaluation capabilities and self-knowledge prediction of language models. By prompting with questions in a specific format, the models in study evaluate the probability "P(True)" that their answers are correct. Then crucially, the models are trained to perform prediction without reference to any specific proposed answer. The study found that larger models exhibit encouraging performance, calibration, and scaling for P(True) across a diverse range of benchmarks. It also reveals that models can effectively predict P(IK) and even partially generalize this prediction across different tasks.

Large Language Models (LLMs) exhibit powerful but opaque behaviors during next-word prediction. Traditional uncertainty estimation approaches typically focus on final outputs, overlooking the intermediate generation process. This work by Bigelow et al. [2024] introduces the Forking Tokens Hypothesis — the idea that certain individual tokens can cause significant shifts in the trajectory of text generation if selected during decoding. To explore this, the authors propose Forking Paths Analysis, a method that tracks uncertainty dynamics across each generated token, rather than only the final ones. Using GPT-3.5 and various tasks, they discover evidence of dramatic shifts in model behavior triggered by specific tokens, including seemingly minor ones like spaces or function words (e.g., "that" or "who"). These findings highlight chaotic uncertainty patterns within LLMs and challenge static views of model confidence.

3 Methods

3.1 Notation

Let $S = (s_1, ..., s_l)$ be a sequence of tokens of length l, where each token comes from a token vocabulary Σ. This sequence contains the prompt in Appendix 7.1 sent to the generation LLM, and an arbitrary number of generated tokens. The LLM uses this context to output a multinomial random distribution that assigns each token $s \in \Sigma$ a probability $p_{(s_1,...,s_l)}(s) \in [0, 1]$ to be the next token.[2] In our situation, tokens are sampled from this distribution. After the whole rollout was generated (either the end token was sampled, or the context window is fully used), the output can be judged and assigned a binary correctness label. This judging happens by executing testcases given in the coding dataset on the generated code in a sandbox environment. In formal notation, the judge calculates the following function:

$$\mathcal{J} : \Sigma^* \to \{0, 1\}. \tag{1}$$

The judge can only give its verdict once the whole code piece is submitted. It can not decide whether intermediate steps are correct, a huge limitation when it comes to tree-based decoding.

3.2 Dataset Generation

Our goal is to estimate a state value v_i for each position i, $1 \leq i \leq s_l$, i.e. the probability of sampling a correct result based on the prefix tokens $(s_1, ..., s_i)$. In formal math, let $\mathcal{G}(s_1, ..., s_i)$ be the generation from the chosen prefix as a random variable. With the binary judge $\mathcal{J}$, the state value

[2] Of course, this probability distribution can have additional hyperparameters such as the temperature.

is defined as

$$v_i = \mathbb{E}\left[J\left(\mathcal{G}(s_1, ..., s_i)\right)\right]. \tag{2}$$

APPS contains both problems that expect input and output via I/O operations, and function completion with a designated function interface. They mostly come with test cases to evaluate the generated code. We use a slightly different prompting framework for these two types of problems, with generation prompts listed in the appendix 7.1. However, calculating an accurate ground truth vector with all exhaustive solutions for each problem is computationally infeasible. Instead, we generate 36 rollouts per problem setup. Later on, we would use this dataset of rollouts labeled by ground-truth correctness for model training and evaluation. To hold these rollouts representative for a tree-based case, these 36 rollouts are not generated independently from the prompt. Instead, we start with one main rollout (generated from Microsoft's Phi-4-mini), select six branching positions, and then generate six rollouts from each branching position, resulting in $6 \times 6 = 36$.

Let p_i be a short notation for $p_{s_1,...,s_{i-1}}(s_i)$, the probability the model assigned to the i-th token with respect to the context $(s_1, ..., s_{i-1})$ in the main rollout. Tokens that are part of the prompt receive probability 1 to have a unified notation for prompt-tokens and generated tokens. When looking at the observed distribution of these p_is in many tasks, the vast amount of probabilities is very close to one, and a few tokens have low probability. For these probabilities to be low, there are two possible explanations. First, these positions could be really ambiguous, with many synonyms as options, or important decisions for the following reasoning. Second, there could be a token with high probability, but just a different path is taken, because sampling is used, and the decision fell for an unlikely token. In both cases, these low-probability tokens are the important branch positions for our reasoning path. Branching here will likely result in different outputs, while branching at high-probability token will likely share a common sequence with the previous rollout. Utilizing this pattern, we select a set $\mathcal{I}$ with $n_b = |\mathcal{I}| = 6$ indices as the set of the positions with the lowest probabilities. For each of these positions $i \in \mathcal{I}$, we generate $k = 6$ rollouts, and evaluate the generated code on the unit tests given in the APPS dataset by Hendrycks et al. [2021].

Therefore, our training dataset is a collection of coding problem description, sample solution, system prompt, and reasoning traces, we provide one relatively short example in Appendix 7.2.

3.3 Training

In our experiments, we train and test value estimations based on the 3.8B Phi-4-mini and 14B Phi-4 model over two episodes. At a high level, we perform fine-tuning on the regression head plus several last layers of these models. Table 1 and Appendix 7.3 provide details of standard hyperparameters and number of last layers.

To estimate the state values as needed, we take the same Phi-4-mini and Phi-4 14B architectures and replace the classifier layer at the network output with a linear regression layer. This modification yields a decoder-only architecture that predicts the value of a partial reasoning trace at a given point, based solely on preceding tokens. The regression layer is designed to estimate the probability of success from each token onward, while maintaining the causal constraint of using only past context. Since the output is probabilistic, we apply a sigmoid transformation to the regression output and train the model using binary cross-entropy loss. For value estimation, we used a batch size of 64 (for the 14B model) and 24 (for the 3.8B model), and fine-tuned the last 12 layers for both models at a learning rate of $1\mathrm{e}{-4}$.

After fine-tuning, our first analysis objective is whether the trained models are capable as an ORM, which are able to distinguish successful rollouts from unsuccessful ones. Also, we take a look into the models' capability to judge intermediate reasoning steps.

3.4 Imbalanced and Balanced Dataset

When Phi-4-mini-instruct generates the rollouts for the problems, the ratio of correct and incorrect rollouts is usually imbalanced. While many problems have more incorrect than correct rollouts, other problems have a higher number of correct rollouts, and the fraction of correct rollouts is typically closely related to the difficulty of the problem for the model. Whenever we train on the raw, imbalanced dataset, we open the door to misleading interpretations and a few biases the model could pick up. First, with an imbalanced dataset the accuracy becomes a less useful metric, especially

Number of problems in test dataset	465
Number of problems in train dataset	3,984
total number of problems	4,449
Train dataset size unbalanced	66,924
Train dataset size balanced	110,016
Batch size Phi-4-mini-instruct (4B)	64
Finetuned Layers Phi-4-mini-instruct (4B)	236-248
Batch size Phi-4 (14B)	24
Finetuned Layers Phi-4 (14B)	180-192
Optimizer	Adam
Learning Rate	1e-4

Table 1: Key data on datasets and training hyperparameters

if the class imbalance is not reported. Therefore, testing on the imbalanced version of the dataset comes with caveats. Second, seeing the distribution of correct and incorrect rollouts per problem can enable the model to just learn the *difficulty of the problem statement*, and only little about the correctness of the reasoning. Suppose there are four correct and 32 incorrect rollouts for a problem. The sequences have the same prefix (the problem statement), and differ later. If the model just learns to predict "false" for this problem statement or this type of problem, it will minimize the loss fairly well. To do so, it is enough to pick up shallow features like complexity, length, or formatting of the problem statement while ignoring the reasoning trace or encoding. So, the reward model can learn to estimate the difficulty of a problem as a proxy for its correctness probability. That enables it to "hack" the accuracy metric without ever learning to judge the actual results.

Therefore, we also train and test our reward models on a balanced dataset. To create this balanced dataset, for each problem we oversample the smaller group (correct/incorrect) to match the number of correct and incorrect rollouts per problem. This is done for both the train and test dataset. With this addition, we take away the option for the model to just learn the difficulty of the problem, and the accuracy numbers are more interpretable.

4 Results & Discussions

Classification Performance For training and evaluation, we use the APPS dataset, a combination of several public coding datasets. APPS contains both problems that expect input and output via IO operations, and function completion with a designated function interface. They mostly come with test cases to evaluate generated code.[3] As explained in section 3.2, we generate one base rollout for each of these samples, select forking tokens (six in our case), and generate also six rollouts from there. Table 2 shows the classification performance at a glance. After training on the same datasets, the 14B model performs substantially better than the 3.8B Phi-4-mini, with the majority of prediction results achieving 60+%. We observe 20+% as the highest improvement base on the calculations in Appendix 7.4, where Pass@1 on best $\binom{3}{1}$ has the baseline of classification accuracy as 45% and the highest as 22.2%; Pass@3 on best $\binom{10}{3}$ has the baseline classification accuracy as 65% and the highest as 20.0%.

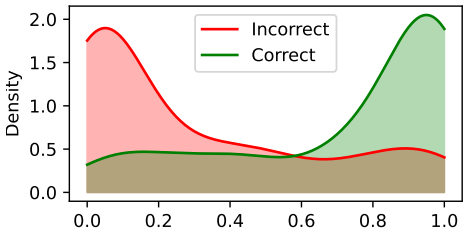
The predictive performance of the 14B model is illustrated in Figure 1a. The distribution shows the approximated probability to visualize how often a certain success probability estimation occurs for that group.

Performance on Balances vs Imbalanced Dataset As Table 2 shows, the accuracies of the models on the imbalanced dataset are similar with minor differences. However, on the balanced test dataset, we do see differences between the two types of model training, with the model trained on the balanced dataset performing substantially better. However, when it comes to determining a relative order of rollouts, when the model needs to select the best or the best three rollouts, we see close accuracies

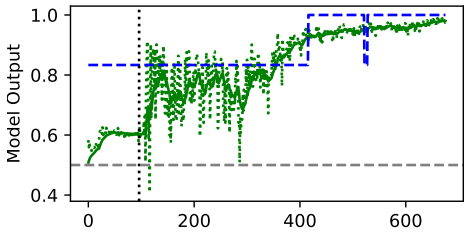
[3]For some problems, only few test cases are given, so the dataset has the unfortunate limitation that false-positives are possible.

	Model	3.8B Phi-4-mini-instruct		14B Phi-4	
	Balanced / Imbalanced Training	**Balanced**	**Imbalanced**	**Balanced**	**Imbalanced**
Imbalanced Test Data	Predicted > 0.5	33.5 %	53.8 %	46.3 %	51.9 %
	Predicted < 0.5	66.5 %	46.2 %	53.7 %	48.1 %
	Accuracy	64.0 %	66.0 %	73.8 %	71.7 %
	If predicted > 0.5, rollout correct	65.2 %	60.0 %	69.8 %	65.7 %
	If predicted < 0.5, rollout incorrect	63.3 %	73.2 %	77.2 %	78.2 %
	If rollout correct, prediction > 0.5	47.2 %	72.2 %	72.5 %	76.5 %
	If rollout incorrect, prediction < 0.5	78.3 %	61.0 %	74.8 %	67.9 %
Balanced Test Data	Predicted > 0.5	31.4 %	53.4 %	45.6 %	51.8 %
	Predicted < 0.5	68.4 %	46.6 %	54.4 %	48.2 %
	Accuracy	55.3 %	51.9 %	65.8 %	60.5 %
	If predicted > 0.5, rollout correct	58.5 %	51.8 %	67.3 %	60.1 %
	If predicted < 0.5, rollout incorrect	53.9 %	52.0 %	64.5 %	60.9 %
	If rollout correct, prediction > 0.5	36.8 %	55.2 %	61.4 %	62.3 %
	If rollout incorrect, prediction < 0.5	73.9 %	48.5 %	70.2 %	58.7 %
	Pass@1 on best $\binom{3}{1}$, baseline 45%	50%	52%	55%	54%
	Pass@3 on best $\binom{3}{10}$, baseline 65%	73%	72%	77%	78%

Table 2: Comparison of the predictive performance of the Phi-4-mini based value prediction and the (regular) Phi-4 performance on a wide variety of metric, measured both on an imbalanced and balanced version of the test data. The baseline passing rates of pass@1, pass@3, and pass@10 are 45%, 65%, and 84% are obtained without interference of any of the reward models, just the generation of Phi-4-mini-instruct.



(a) The trained 14B Phi-4 model's differentiation capability of the correct and incorrect rollouts on the test dataset. The correctness is based on the ground-truth verifier built with unit tests in a safe execution sandbox.



(b) The trained 14B Phi-4 model's predicted belief over number of available tokens in a correct rollout as the green dotted line. The estimated ground truth is the blue dashed line, which is retrieved by the correct fraction in six forking positions.

Figure 1: The CDF plot on the left is generated through kernel density estimation with Gaussian kernels. Then this model is used to generate the success probability estimations on the right.

between the two types of training. In conclusion, these results suggest that balancing a train dataset has impact on the final performance, but it is rather limited.

In what follows, we discuss the key research questions (RQ) we pose and the answers our work offers.

RQ1: Can we use this reward model as a handy critic for code generation? Now with this trained reward model, we want to validate its capability to select correct rollouts out of multiple options for a problem from Table 2. There are two key terminologies involved: 1. Pass@k is the number of passes of k unit tests with the specific rollout; 2. Best $\binom{m}{n}$ is to select the best n rollouts using the reward model out of m rollouts. We observed that before applying the critic, pass@1 is 45%, pass@3 is 65%, and pass@10 is 84%. After applying the critic and measuring best $\binom{m}{n}$, we

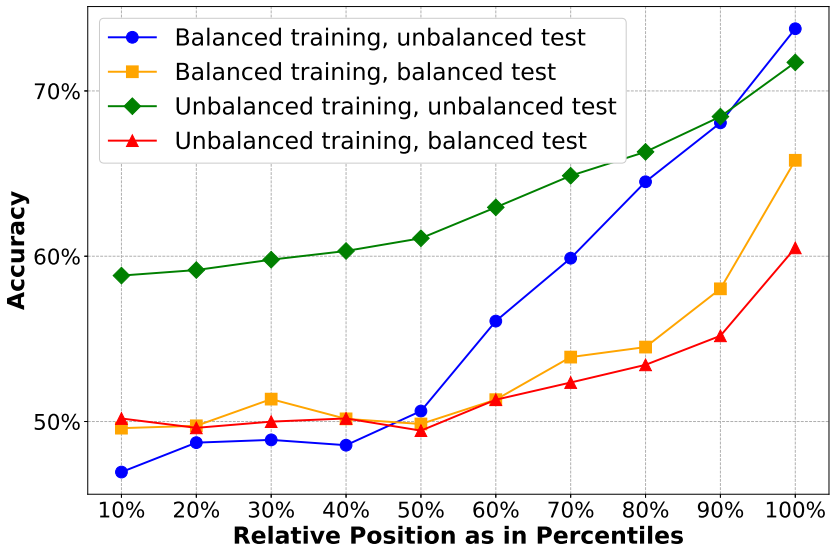


Figure 2: Accuracy of percentiles in rollout generation from four combinations of the 14B model trained and tested on balanced/imbalanced data.

observe significant pass@1 improvement to 50%–55%, and pass@3 to 72%–78%. Therefore, our findings support a confident affirmative answer to the research question: employing our model in a lightweight role as an evaluation critic or data quality filter enhances the correctness of Python code generation.

RQ2: Can this outcome reward model also be capable of judging intermediate generation steps? All the metrics above refer to the model's ability to provide a verdict on entire rollouts, i.e., the outcome reward. To validate how often the reward model succeeds in estimating the probability of intermediate steps, we show percentile-accuracy plots such as Figure 2. We measure the model's accuracy vs. the amount of given partial code, i.e., the accuracy of up to 100% tokens from the first code line equals the accuracy in Table 2. We argue that the earlier the model achieves higher accuracies, the more useful it is for early stopping or rollout guidance. We observe that the model requires at least 50% partial code to judge better than a random guess.

The decoder-only reward model estimates success probability after each token, enabling confidence tracking during rollouts. Figure 1b shows a correct generation where the model's confidence rises after seeing the output, aligning well with ground truth. We observe a warming-up stage before around 100 tokens. This is because our prompt asks the generation model to think in a Chain-of-Thought manner: first elaborate on the problem and describe a solution, and then write the code. Given that the model waits to improve its performance until roughly the 50% percentile, we suspect that the model might be able to catch errors in the code, but lacks the self-correction capability for already incorrect reasoning steps. With Figures 2 and 1b, we also confidently answer RQ2 in the affirmative.

The model begins to show meaningful performance (better than random guesses) as PRMs after approximately 50% of the token generation process, with accuracy improving beyond the initial baseline. This indicates a lower bound of the preceding code tokens required to provide the predictive power.

5 Conclusion

These experiments demonstrate that, large-scale capacity is not a prerequisite for reward modeling: models from the Phi-4 family (14 B parameters) can serve as effective reward models, even when compared with state-of-the-art systems like GPT-5 and Claude 4 Sonnet at vastly larger scales of hundreds to trillions of parameters. These models are capable of truly identifying errors in the rollouts. Moreover, they demo the capability of selecting correct rollouts, substantially increasing the

probability of selecting a correct rollout. However, we see that model size is a primary limitation for this task, as the 3.8B Phi-4-mini-instruct falls significantly behind the 14B model.

6 Limitations

Due to computational resource constraints, there are several experiments that should become our next steps as follows.

- Computing an accurate ground-truth vector for each problem is infeasible for the main solution variants and branching factors. Our current setup of generating 36 rollouts per problem already requires over 48 hours for one pass on a 4×A100 Azure computation instance. This throughput is mainly throttled by the output token throughput of the generation model Phi-4 mini. As a result, tuning the hyperparameters for branching decisions is left to future work.
- During rollout generation, we exclude samples where the Phi model fails to produce any correct rollout. Our current method depends on a ground-truth correct solution such as the tree trunk, from which branching occurs. This cold start problem remains a challenge.
- Preliminary experiments with smaller branching factors (e.g., 6x6) indicate a positive impact from increased branching. However, we were unable to identify the point of diminishing returns, as broader branching was not computationally tractable in our current setup.
- APPS dataset comes with its own train-test split. Specifically, both their training and test datasets contain 5,000 samples. However, as our appendix 7.5 shows, the default test dataset follows an utterly different distribution. Therefore, we performed train-test-split on the original training set.
- Our current setup has the strong assumption that the distribution of the predicted correctness probability does not shift between the base policy and the policy in post-training. And we leave the exploration of post-training dynamics to future work.

7 Acknowledgment

We gratefully acknowledge Kanishk Gandhi of Stanford University Human-Centered Artificial Intelligence Institution, for his guidance in the literature review process and constructive contributions to the development of our methodology.

References

Eric Bigelow, Ari Holtzman, Hidenori Tanaka, and Tomer Ullman. Forking paths in neural text generation. *arXiv preprint arXiv:2412.07961*, 2024.

Sanjiban Choudhury. Process reward models for llm agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, et al. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025.

Tej Deep Pala, Panshul Sharma, Amir Zadeh, Chuan Li, and Soujanya Poria. Error typing for smarter rewards: Improving process reward models with error-aware hierarchical supervision. *arXiv e-prints*, pages arXiv–2505, 2025.

DeepSeek-AI and Daya et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Qiying Yu et al. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL `https://arxiv.org/abs/2503.14476`.

Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training, 2024. URL `https://arxiv.org/abs/2309.17179`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Muhammad Khalifa, Rishabh Agarwal, Lajanugen Logeswaran, Jaekyeom Kim, Hao Peng, Moontae Lee, Honglak Lee, and Lu Wang. Process reward models that think. *arXiv preprint arXiv:2504.16828*, 2025.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023a. URL https://arxiv.org/abs/2305.20050.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023b.

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 Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

Kimi Team and Angang et al. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL https://arxiv.org/abs/2501.12599.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*, 2023.

Teng Wang, Zhangyi Jiang, Zhenqi He, Shenyang Tong, Wenhan Yang, Yanan Zheng, Zeyu Li, Zifan He, and Hailei Gong. Towards hierarchical multi-step reward models for enhanced reasoning in large language models. *arXiv preprint arXiv:2503.13551*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

Fei Yu, Anningzhe Gao, and Benyou Wang. Ovm, outcome-supervised value models for planning in mathematical reasoning, 2024. URL https://arxiv.org/abs/2311.09724.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Jialun Zhong, Wei Shen, Yanzeng Li, Songyang Gao, Hua Lu, Yicheng Chen, Yang Zhang, Wei Zhou, Jinjie Gu, and Lei Zou. A comprehensive survey of reward models: Taxonomy, applications, challenges, and future, 2025. URL https://arxiv.org/abs/2504.12328.

Appendix

7.1 Prompts

<|system|>You are a Python programmer in a programming competition. In the following, you are given a coding challenge.
Please write code to solve the coding challenge. Follow the formatting instructions for input and output exactly.
Get inputs via input(), and write outputs via print().
Before writing the code, do the following:
1. Elaborate on the problem. Highlight the key challenges
2. Write the core algorithm idea in at most three sentences
3. Elaborate on the idea in more detail.
The code must be a directly executable script, not a function.
NEVER include example usage or hard-coded data!
ALWAYS generate the full code!
Here is an example for a very simple problem:

```python
number_of_testcases = int(input())
for _ in range(number_of_testcases):
  a, b, c = input().split()
  b = b[::-1]
  if a in b and b in c:
    print("TRIPLE TWIST")
  else:
    print("NO")
```

<|end|><|user|>{task}<|end|><|assistant|>

Figure 3: The prompt we use for problems that require I/O operations. We use the chat template which the model was trained with, laying out both system and user prompts. In the system prompt, we give instructions and give an piece of example code. The example code is for a straightforward string problem that is neither in the test nor train dataset. The model is supposed to continue with its thoughts and its code after the assistant tag.

<|system|>You are a Python programmer in a programming competition. In the following, you are asked to write code for a function.
At the beginning of the function, do the following in the docstring:
1. Elaborate on the problem. Highlight the key challenges
2. Write the core algorithm idea in at most three sentences
3. Elaborate on the idea in more detail.
NEVER include example usage!
ALWAYS generate the full code!
<|end|><|user|>{task}<|end|><|assistant|> ‘‘‘python
def {function_signature}
"""

Figure 4: The prompt we use for problems that require a complete function as the solution. In the assistant part of the message, we directly give the function signature to force the model to produce a valid function, and also not deviate from the given signature. Moreover, we force the model to start with a docstring. Using this prompt, we significantly reduced the number of malformed generations.

7.2 Sample Data

Please refer to Figure 5 as one example of training data sample.

7.3 Hyperparameters & Hardware

In the stabilized experiment, we used a batch size of 64 for the 14B model and 24 for the 3.8B model, at a learning rate of 1×10^{-4}. During the ablation, we probed batch sizes around 8, 12, 24, 64 to test the memory boundary in our Azure instance. We also tried learning rate at 1×10^{-5} and 1×10^{-6}.

Our computation instance type is Azure's Standard_NC96ads_A100_v4 with 96 CPU cores (AMD EPYC 7V13), 880 GB RAM, 256 GB disk, and 4 Nvidia A100 GPUs in 320GB GPU memory. The total GPU hours are around 95.

7.4 Performance Improvements

From Table 2:

- Pass@1 on best $\binom{3}{1}$
  - Baseline classification accuracy is 45%.
  - The ceiling of improvement is $\frac{55-45}{45} \simeq 22.2\%$
- Pass@3 on best $\binom{10}{3}$
  - Baseline classification accuracy is 65%.
  - The ceiling of improvement is $\frac{78-65}{65} \simeq 20\%$

7.5 APPS Data Distribution Study

We use the APPS Dataset, which is distributed under the MIT License. Our use complies with the license terms, and we will include the original copyright notice in our future artifacts to be released. In the paper where the APPS dataset was released, Hendrycks et al. [2021] highlighted differences in data quality, particularly in terms of unit test coverage, between the "train" and "test" splits. In addition to this observation, we want to emphasize that the intrinsic distribution of problem formats and complexity levels also varies significantly between these splits, as illustrated in Figure 6a and Figure 6b. These differences suggest that the train and test sets may not be fully aligned in terms of problem representation. This will substantially affect our results, as our method is designed to operate on the distribution of contextual tokens rather than the surface-level structure or complexity of the problem descriptions. Therefore, our method is more sensitive to such mis-alignment in data composition. We'll take the deep-dive of the performance impact from task formulation in APPS dataset to future work.

<|system|>You are a Python programmer in a programming competition.
In the following, you are given a coding challenge. Please write code to solve the coding challenge. Follow the formatting instructions for input and output exactly. Get inputs via input(), and write outputs via print(). Before writing the code, do the following: 1. Elaborate on the problem. Highlight the key challenges 2. Write the core algorithm idea in at most three sentences 3. Elaborate on the idea in more detail. The code must be a directly executable script, not a function. NEVER include example usage or hard-coded data! ALWAYS generate the full code! Here is an example for a very simple problem:

```python
number_of_testcases = int(input()) for _ in range(number_of_testcases):
  a, b, c = input().split()
  b = b[::-1]
  if a in b and b in c:
    print("TRIPLE TWIST")
  else:
    print("NO")
```

<|end|><|user|>You are given a permutation $p_1, p_2, \ldots, p_n$. Recall that sequence of n integers is called a permutation if it contains all integers from 1 to n exactly once. Find three indices i, j and k such that: $1 \leq i < j < k \leq n$; $p_i < p_j$ and $p_j > p_k$. Or say that there are no such indices.

——Input——

The first line contains a single integer T ($1 \leq T \leq 200$) — the number of test cases. Next $2T$ lines contain test cases — two lines per test case. The first line of each test case contains the single integer n ($3 \leq n \leq 1000$) — the length of the permutation p. The second line contains n integers $p_1, p_2, \ldots, p_n$ ($1 \leq p_i \leq n$; $p_i \neq p_j$ if $i \neq j$) — the permutation p.

——Output——

For each test case: if there are such indices i, j and k, print YES (case insensitive) and the indices themselves; if there are no such indices, print NO (case insensitive). If there are multiple valid triples of indices, print any of them.

——Example——

Input 3 4 2 1 4 3 6 4 6 1 2 5 3 5 5 3 1 2 4

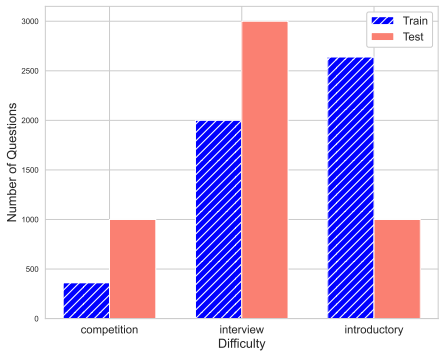
Output YES 2 3 4 YES 3 5 6 NO

<|end|><|assistant|>

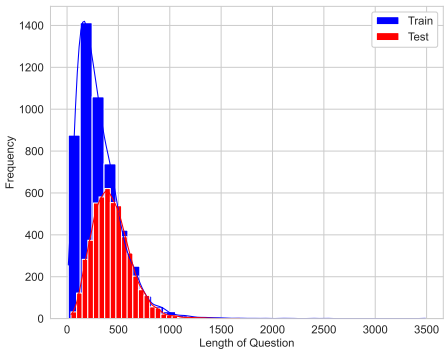
The problem requires us to find a triplet of indices in a given permutation such that the values at those indices first increase and then decrease. The key challenges are to efficiently find such a triplet in each permutation, which can be done by iterating through the permutation and checking for the increasing and then decreasing condition. The core algorithm idea is to iterate through the permutation and maintain a candidate for the second element of the triplet (the peak). When we find a new maximum, we check if the previous element is less than the current one (forming the increasing part) and if the next element is less than the current one (forming the decreasing part). If both conditions are met, we have found our triplet. The idea in more detail is to iterate through the permutation once, keeping track of the current maximum and its index. If we find a new maximum, we check the elements before and after it to see if they form the increasing and decreasing parts of the triplet. If they do, we print the indices and break out of the loop. If we finish the loop without finding such a triplet, we print "NO". Here is the code:

```python
T = int(input()) for _ in range(T):
  n = int(input())
  p = list(map(int, input().split()))
  found = False
  for i in range(1, n - 1):
    if p[i - 1] < p[i] > p[i + 1]:
      print("YES")
      print(i, i + 1, i + 2)
      found = True
      break
  if not found:
    print("NO")
```


Figure 5: Example of one piece of training sample.



(a) The difficulty levels, which are from APPS dataset original definition.



(b) The length of question, which counts the token length per problem's question.