Focused-DPO: Enhancing Code Generation Through Focused Preference Optimization on Error-Prone Points

Anonymous ACL submission

Abstract

Code generation models have shown significant potential for automating programming tasks. However, the challenge of generating accurate and reliable code persists due to the highly complex and long-reasoning nature of the task. Even state-of-the-art models often fail in code generation due to small errors, which can drastically affect the overall functionality of code. Our study identifies that current models tend to produce errors concentrated at specific error-prone points, which significantly impacts the accuracy of the generated code. To address this issue, we introduce Focused-DPO, a framework that enhances code generation by directing preference optimization towards these critical error-prone areas. This approach builds on Direct Preference Optimization, emphasizing accuracy in parts prone to errors. Additionally, we develop a method called Error-Point Identification, which constructs a dataset that targets these problematic points without requiring costly human annotations. Our experiments on benchmarks such as HumanEval(+), MBPP(+), and LiveCodeBench demonstrate that Focused-DPO significantly improves the precision and reliability of code generation, reducing common errors and enhancing overall code quality. By focusing on error-prone points, Focused-DPO advances the accuracy and functionality of model-generated code.

1 Introduction

Code generation has emerged as a pivotal task in artificial intelligence, enabling models to automate essential software development tasks. Code Models (GPT-4, 2023; Guo et al., 2024; Hui et al., 2024) have demonstrated remarkable capabilities in code generation tasks. These advancements have significantly improved developer's productivity, accelerating software delivery timelines.

Despite their success, generating correct code remains a substantial challenge due to the complex and long-reasoning nature of the task. Writing code necessitates long reasoning, where numerous small decisions about syntax and logic must work together to produce a functional program. Even minor mistakes, such as an incorrect operator, can cause a program to fail. Code generation, therefore, can be viewed as a multi-step long reasoning process. Ensuring the accuracy of every decision in this multi-step process collectively determines the correctness of the resulting output code.

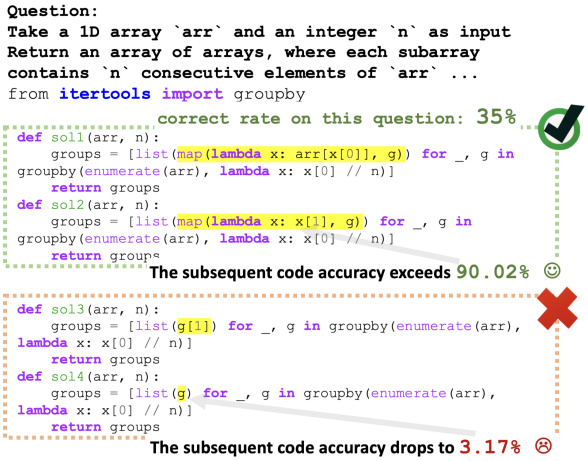

Figure 1: Error-prone points in generated code from Qwen-2.5-Coder-Instruct-7B. We sample 20 outputs for this question. Outputs have common prefixes and suffixes, differing mainly at yellow-highlighted error points. Continuing generation at these points leads to drastically different accuracies (90.02% vs. 3.17%). This disparity is not seen in non-highlighted parts.

When examining the outputs of current code generation models, we find that errors are not evenly spread across the code. Large language models tend to produce errors concentrated in certain error-prone points, even when sampling multiple times with a high temperature. We illustrate this phenomenon in Figure 1, which shows error-prone points highlighted in yellow. Despite the overall code having similar prefixes and suffixes, differences at these highlighted error points significantly impact the final code accuracy. Generating code

[1]Corresponding authors.

from correct outputs at these error-prone points can achieve a final accuracy of up to 90.02%, whereas starting from incorrect outputs reduces accuracy to 3.17%. Parts of the code, such as function headers (usually at the prefix) or return statements (usually at the suffix), often follow familiar patterns. However, some middle parts of the code, which involve more complex reasoning, are more prone to errors. Errors in these parts can disrupt and affect the entire program's reliability.

It is crucial to address these error-prone points for code generation. However, existing studies on code generation overlook this problem. While standard training approaches such as Supervised Fine-Tuning (SFT) (Wang et al., 2022) help improve overall output quality, they do not specifically focus on the crucial parts necessary for correctness. Methods like Direct Preference Optimization (DPO) (Rafailov et al., 2024) aim to align outputs with preferences (e.g., "chosen" vs. "rejected"), but often overlook fine-grained error-prone points of the code. As a result, these trained models might generate code that appears correct initially but contains critical issues at the error-prone points, ultimately affecting overall accuracy.

To tackle these issues, we introduce **Focused-DPO**, a framework designed to enhance code generation through focusing preference optimization on error-prone points. Focused-DPO builds on Direct Preference Optimization by emphasizing accuracy improvement in areas where errors are most likely to occur. Unlike traditional methods that treat all parts of the code equally, Focused-DPO specifically targets those error-prone points, which are essential for the overall correctness of the program.

Focused-DPO is a data-driven preference optimization method that relies on a specially constructed dataset with identified error-prone points. We propose a dataset construction method named **Error-Point Identification**, which includes an automated pipeline to construct paired code preference datasets. This method extracts concepts from real code repositories and synthesizes programming problems. By concurrently generating code and tests, and using a page-rank-inspired algorithm for ranking, we determine the relative performance of all generated code. Error-Point Identification employs common prefix and suffix matching to precisely locate error-prone points. Additionally, our method automatically identifies error-prone code parts, eliminating the need for costly human input, making it scalable and efficient for a variety of programming tasks.

We evaluate Focused-DPO using standard benchmarks such as HumanEval(+) (Liu et al., 2024), MBPP(+), and LiveCodeBench (Jain et al., 2024), and observe significant improvements over existing methods. Even for models like *Qwen2.5-Coder*, which already have undergone large-scale alignment training, Focused-DPO still achieves a 42.86% relative improvement on extremely hard competition-level problems in LiveCodeBench. The results show notable increases in the generation quality on error-prone points, highlighting Focused-DPO's effectiveness in enhancing the accuracy of code generation.

Our contributions are summarized as follows:

- We propose Focused-DPO , a novel framework that enhances code generation by focusing preference optimization on error-prone points, resulting in more accurate codes.
- We introduce a dataset construction method that automatically identifies error-prone points by generating both code and corresponding tests for fine-grained self-verification.
- Experiments on widely-used benchmarks show that Focused-DPO improves the generation quality of code models, even for those that have already undergone extensive post-training on million-level datasets.

2 Related Work

Large language models (LLMs) have made significant progress in generating code from natural language descriptions, showing great potential for automating software development tasks. Models(GPT-4, 2023; Li et al., 2023; Hui et al., 2024; Guo et al., 2024; Jiang et al., 2024) have demonstrated strong performance, thanks to extensive training on diverse datasets. To further enhance their capabilities, posting training methods like Supervised Fine-Tuning (SFT) (Luo et al., 2023; Wei et al., 2023b) and Direct Preference Optimization (Hui et al., 2024; Zhang et al., 2024b; Dou et al., 2024; Gee et al., 2024; Zhang et al., 2024a) are commonly applied. Preference optimization approaches focus on aligning model outputs with desired outcomes by prioritizing more favorable responses over less favorable ones. However, existing DPO approaches fail to address one important

issue: they do not directly target the most error-prone points in generated code. Errors in these high-impact parts can lead to significant quality and reliability issues in the final output. We aim to address this issue by focusing the preference optimization learning on these error-prone points in the generated code.

Some fine-grained preference optimization methods (Rafailov et al., 2024; Lai et al., 2024; Lu et al., 2024; Zeng et al., 2024; Lin et al., 2024) have shown strong potential in domains like mathematics, which rely heavily on natural language reasoning. Step-DPO (Lai et al., 2024) and Step-Controlled DPO (Lu et al., 2024) propose generating step-wise preference datasets to enable optimization learning based on the standard DPO loss. TDPO (Zeng et al., 2024) enhances the DPO loss by incorporating forward KL divergence constraints at the token level, achieving fine-grained alignment for each token. cDPO (Lin et al., 2024) proposes a tricky method to find the critical token in the thought chain that affects overall accuracy. However, the identified tokens are typical in natural language and the method does not apply to code, which features similar overall patterns but relies on specific key elements in long reasoning processes. However, in the context of code generation, where a small error-prone point can lead to major functional errors, these exisiting methods often struggle to construct adequate datasets or fail to achieve ideal improvements due to weak fine-grained reward signals. To address this, we propose Focused-DPO, a framework that improves code generation by focusing on optimizing these high-impact parts. Our dataset construction method employs a self-generation and validation process to construct datasets that explicitly identify error-prone points, ensuring the optimization learning process directly enhances the parts of the code that matter most for overall correctness.

3 Focused-DPO

Our proposed Focused-DPO framework aims to enhance code generation by concentrating on error-prone points through focused preference optimization. Building on Direct Preference Optimization, our Focused-DPO specifically targets those high-impact parts of the source code, rather than treating all code parts equally. As illustrated in Figure 2, our method involves three main steps: ❶ **Synthetic Data Generation with Real-World Source**

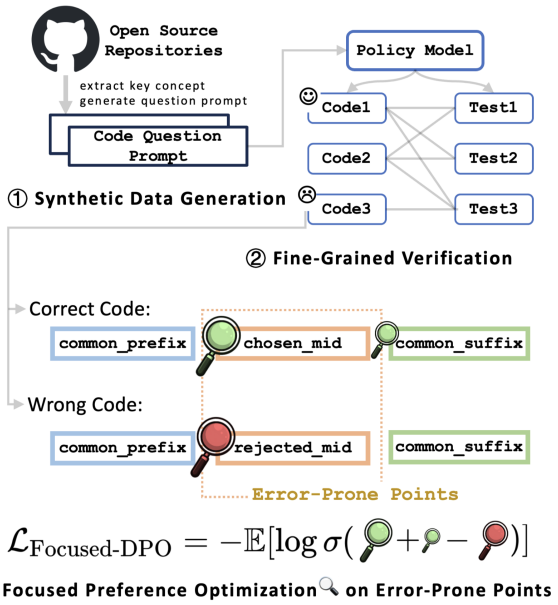

Figure 2: Overview of the Focused-DPO framework. Focused-DPO consists of three key stages: ❶ Generating synthetic question prompts from real-world code repositories. ❷ Using a policy model to simultaneously generate code and test cases, applying a page-rank algorithm to identify correct and incorrect samples and locate error-prone points using common prefixes and suffixes. ❸ Applying Focused-DPO, which pays special attention on error-prone points as if applying a magnifying glass for focused optimization.

Code : We initiate by collecting a seed dataset from open-source code repositories and generate programming task prompts. ❷ **Fine-Grained Verification to Identify Error-Prone Points** : We generate both code and tests simultaneously using a self-generation-and-validation loop. We apply a PageRank algorithm to iteratively update scores and rank the outputs, identifying correct and incorrect code samples. By distinguishing between similar versions of correct code and incorrect code, we locate significant parts that highly affect the final correctness and identify these parts as error-prone points, allowing for further fine-grained optimization learning. ❸ **Focused Preference Optimization Learning** : We design a learning optimization algorithm specifically for these critical error-prone points. Using the constructed dataset, our novel training loss helps the model develop a preference for these focused parts within the code, thus optimizing performance more effectively.

3.1 Synthetic Data Generation with Real-World Source Code

The first step in our approach is the construction of a synthetic dataset. We collect a diverse set of pro-

gramming snippets from open-source repositories to create a seed dataset. Similar to OSS-instruct (Wei et al., 2023a), we use the seed dataset to extract key programming concepts, such as algorithm design and data structure utilization. Then based on these concepts we generate the final prompts. This construction strategy allows the model to explore a broad range of scenarios. The generated question prompts are used in the following stages.

3.2 Fine-Grained Verification to Identify Error-Prone Points

To identify error-prone points, we propose a dataset construction method named **Error-Point Identification**. Firstly, we use the policy model to simultaneously generate k output codes and test cases based on the question prompts using a higher-temperature setting. In our experiment, we set $k = 10$. Using their execution relationships, we then adopt the ranking method from CodeDPO (Zhang et al., 2024b), a page-rank algorithm to iteratively update scores and rank the outputs:

$$\begin{aligned}\text{Score}_t(c_i) &= (1-d) \times \text{Score}_{t-1}(c_i) \\ &\quad + d \times \sum_{t_j} \text{Score}_{t-1}(t_j) \times \text{Link}(t_j, c_i) \\ \text{Score}_t(t_j) &= (1-d) \times \text{Score}_{t-1}(t_j) \\ &\quad + d \times \sum_{c_i} \text{Score}_{t-1}(c_i) \times \text{Link}(c_i, t_j)\end{aligned} \tag{1}$$

Where d is the damping factor, and $\text{Link}(t_j, c_i)$ indicates whether a code snippet c_i passes the test case t_j. The ranking score is updated iteratively until the ranking of the code stabilizes.

We consider the test case that the highest-ranked code correctly passes as the ground truth test case for this question. Subsequently, we split all generated codes into two categories: correct code that passes all ground truth test cases and incorrect code that does not. For each pair consisting of a correct code sample and an incorrect code sample, we match their common prefix and suffix to decompose each code snippet into three parts: `common_prefix`, `mid_chosen` (or `mid_rej`), and `common_suffix`. We then define a *Diff* function as follows:

$$\begin{aligned}\textit{Rank}(\text{mid}) &= \text{Score}(\text{common\_prefix}, \text{mid}, \text{common\_suffix}), \\ \textit{Diff} &= \text{Rank}(\text{mid\_chosen}) - \text{Rank}(\text{mid\_rej}) \\ &\quad + \lambda * (\text{length}(\text{common\_prefix}) + \text{length}(\text{common\_suffix})).\end{aligned} \tag{2}$$

Our constructed *Diff* function includes two components: ❶ the difference in rank between the correct and incorrect code, and ❷ the sum of the lengths of the common prefix and suffix, which ensures that the error-prone points are more concentrated. We maximize *Diff* to choose the `mid_chosen` and `mid_rej` parts that significantly impact the code's correctness, and identify these as the error-prone points. By focusing on error-prone points, we create training samples that directly address the parts of the code that have significantly impact on correctness. For each policy model, we apply necessary filtering to the generated data, resulting in a final dataset containing 5,000 training samples and 1,000 validation samples. Table 9 presents an example of data statistics.

3.3 Focused Preference Optimization Learning

The core of our method lies in modifying the Direct Preference Optimization (DPO) framework to better enhance code generation by focusing on error-prone points of the code. Given a pairwise preference dataset $\mathcal{D} = \{(x_i, y_i^{chosen}, y_i^{rej})\}_{i=1}^{M}$, the standard DPO loss (Rafailov et al., 2024) is expressed as:

$$\ell_{\text{DPO}} = -\mathbb{E}_{(x,y^{chosen},y^{rej})\sim\mathcal{D}} \left[\log \sigma \left(\phi(x, y^{chosen}) - \phi(x, y^{rej})\right)\right], \tag{3}$$

where $\phi(x, y)$ is an implicit reward function. The reward function is defined as:

$$\phi(x, y) = \beta \cdot \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} + \underbrace{\beta \cdot \log Z(x)}_{\text{this term can ultimately be reduced}} \tag{4}$$

where $\pi_\theta(y|x)$ represents the probability of a generated response y under the policy model, and $\pi_{\text{ref}}(y|x)$ is the probability under a reference model, typically the SFT baseline. The goal of DPO loss is to maximize reward difference between the preferred and non-preferred samples.

Reward Function Modification In its original form, the DPO reward $\phi(x, y)$ is calculated over the entirety of the sample y, treating all parts of the code equally. However, in the context of code generation, not all parts of the code contribute equally to correctness. Building on our observation that the middle part (`mid`) of code—the error-prone point we identify in Section 3.2—should receive more attention, we restructure the reward to reflect the relative importance of different code parts. The reward function is modified to weight the `mid` part more heavily, reflecting its critical contribution to the correctness of the code. For the preferred sample, the reward function becomes:

$$\phi_{\text{chosen}}(x, y) = \beta \cdot \Big(\log \frac{\pi_\theta(\texttt{prefix}|x)}{\pi_{\text{ref}}(\texttt{prefix}|x)} + w_{\text{focused}} \cdot \log \frac{\pi_\theta(\texttt{mid}|x, \texttt{prefix})}{\pi_{\text{ref}}(\texttt{mid}|x, \texttt{prefix})} + \log \frac{\pi_\theta(\texttt{suffix}|x, \texttt{prefix}, \texttt{mid})}{\pi_{\text{ref}}(\texttt{suffix}|x, \texttt{prefix}, \texttt{mid})} \Big) \tag{5}$$

Where w_{focused} is a weight that amplifies the importance of the `mid` part.

For the non-preferred sample, we adopt a similar structure but introduce an adjustment to further downweight the contribution of the `suffix`. This adjustment is based on our observation that regardless of whether the `mid` part contains errors, the content of the `suffix` is often the same or similar. Our results in Section 5.1 show that the correlation between the `suffix` and the overall accuracy of the final code is low, making it less significant in the reward calculation. The reward becomes:

$$\phi_{\text{rej}}(x, y) = \gamma \cdot \Big(\log \frac{\pi_\theta(\texttt{prefix}|x)}{\pi_{\text{ref}}(\texttt{prefix}|x)} + w_{\text{focused}} \cdot \log \frac{\pi_\theta(\texttt{mid}|x, \texttt{prefix})}{\pi_{\text{ref}}(\texttt{mid}|x, \texttt{prefix})} \Big) \tag{6}$$

Final Loss Function Substituting the modified rewards for the preferred (y^{chosen}) and non-preferred (y^{rej}) examples into the original DPO loss and simplifying by canceling common terms, we can obtain that:

$$\begin{aligned}
&\Delta\texttt{reward} = \phi_{\text{chosen}}(x, y^{chosen}) - \phi_{\text{rej}}(x, y^{rej}) \\
&= \sum_{j=k+1}^{m} \beta \cdot w_{\text{focused}} \cdot \log \frac{\pi_\theta(t_j^{(\text{mid_chosen})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:j-1}^{(\text{mid_chosen})})}{\pi_{\text{SFT}}(t_j^{(\text{mid_chosen})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:j-1}^{(\text{mid_chosen})})} \\
&\underbrace{-\textstyle\sum_{j=k+1}^{n} \beta \cdot w_{\text{focused}} \cdot \log \frac{\pi_\theta(t_j^{(\text{mid_rej})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:j-1}^{(\text{mid_rej})})}{\pi_{\text{SFT}}(t_j^{(\text{mid_rej})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:j-1}^{(\text{mid_rej})})}}_{\Delta_{\texttt{mid}}} \\
&+ \underbrace{\sum_{j=m+1}^{L_1} \beta \cdot \log \frac{\pi_\theta(t_j^{(\text{suffix})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:m}^{(\text{mid_chosen})}, t_{m+1:j-1}^{(\text{suffix})})}{\pi_{\text{SFT}}(t_j^{(\text{suffix})}|x, t_{0:k}^{(\text{prefix})}, t_{k+1:m}^{(\text{mid_chosen})}, t_{m+1:j-1}^{(\text{suffix})})}}_{\Delta_{\texttt{suffix}}} \\
&= \Delta_{\texttt{mid}} + \Delta_{\texttt{suffix}}
\end{aligned} \tag{7}$$

So the final loss function for Focused-DPO is expressed as:

$$\mathcal{L}_{\text{Focused-DPO}}(\pi_\theta; \pi_{\text{SFT}}) = - \mathbb{E}_{(x, y^{chosen}, y^{rej}) \sim \mathcal{D}} \left[\log \sigma \left(\Delta_{\texttt{mid}} + \Delta_{\texttt{suffix}} \right) \right], \tag{8}$$

The terms $\Delta_{\texttt{mid}}$ and $\Delta_{\texttt{suffix}}$ capture the weighted differences in the probabilities of critical parts between the preferred and non-preferred samples, with greater emphasis focused on the `mid` parts, which is the error-prone point.

Through this modification, Focused-DPO shifts the focus of optimization toward the error-prone point in the code. By increasing the weight of these parts in the reward calculation, our framework ensures that the model prioritizes improvements where they matter most, leading to higher-quality and more reliable code generation.

4 Experiment Setup

We aim to answer the following research questions:

RQ1: Are there error-prone points in generated code that significantly affect the correctness of the output? This question addresses the core motivation behind Focused-DPO. To investigate this, we construct the validation dataset following Section 3.2. This setup provides empirical evidence supporting the theoretical underpinnings of our Focused-DPO.

RQ2: Can Focused-DPO improve the generation quality of code models, even those that have already been heavily post-trained with alignment techniques such as standard DPO? To explore this, we evaluate Focused-DPO on several widely-used code generation benchmarks, including HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2024), MBPP (Austin et al., 2021), MBPP+, and LiveCodeBench (Jain et al., 2024).

RQ3: How do different components of the Focused-DPO loss formulation affect model performance? Ablation studies include evaluating our dataset construction method, as well as key components in our loss formulation.

4.1 Baselines

We evaluate several widely used large language models (LLMs) in the code generation domain. For ***base models***, we apply Focused-DPO to **DeepSeekCoder-base-6.7B)** (Guo et al., 2024) and **Qwen2.5-Coder-7B** (Hui et al., 2024). For ***instruct models*** , we evaluate on **Magicoder-S-DS-6.7B** (Wei et al., 2023b) and **DeepSeekCoder-instruct-6.7B**, which are post-trained from *DeepSeekCoder-base-6.7B* with large-scale SFT. We further evaluate **Qwen-2.5-Coder-Instruct-7B**, which is post-trained from *Qwen2.5-Coder-7B* on million-level datasets with SFT and DPO.

We compare against several widely used training techniques, including: **SFT**, **standard DPO**, **Step-DPO** (Lai et al., 2024), **TDPO** (Zeng et al., 2024). SFT trains models only with positive samples, while the other methods utilize a pairwise dataset of preferred and rejected samples.

4.2 Training and Inference Settings

For each backbone LLM, we sample 10 code candidates and corresponding test cases for each problem prompt using `temperature=1.5`. An example of data statistics is in Table 9. Our analysis shows this configuration results in a stable ranking score and ensures diversity. We focus on Python-based datasets given its widespread use. For training, we train for 10 epochs on 8 NVIDIA V100 GPUs and select the best-performing checkpoint based on the lowest validation loss. We set $w_{focused} = 2$ in our experiments. We use a learning rate of 5×10^{-6} with a linear scheduler and warm-up. We employ greedy search during inference.

5 Results and Analyses

5.1 Exploration of Error-Prone Points in Code (RQ1)

We conduct experiments to validate our motivation: ❶ Correlation analysis confirms that **error-prone points in the code significantly impact correctness**, whereas other code parts have minimal effect. ❷ Generation experiments show that **continuing at these points with different content leads to significant differences in overall correctness**. ❸ Observations reveal that **existing code models perform suboptimally at these points**.

Correlation Between Different Code Parts and Final Correctness Utilizing the dataset construction pipeline described in Section 3.2, we evaluate the validation dataset based on *Qwen2.5-Coder-Instruct-7B*. We analyze the relationship between `prefix`, `suffix`, two types of `mid` parts, and the final code correctness, as presented in Table 1.

Segment	Correct	Incorrect	Phi Coefficient
Common Prefix	0.7907	0.7325	0.0683
Common Suffix	0.8479	0.7864	0.0796
Common Prefix + Chosen Mid	0.6367	0.0911	0.5651
Common Prefix + Reject Mid	0.0116	0.5575	-0.6085

Table 1: Relationships between the `prefix`, `suffix`, and the two types of `mid` parts with the final code correctness. The table includes the frequency of each part in correct and incorrect code, as well as their correlation coefficients with overall code correctness.

Results in Table 1 show that `common_prefix + chosen_mid` appears much more frequently in correct solutions, while `common_prefix + rej_mid` is prevalent in incorrect solutions. This confirms the critical influence of the `mid` part, with strong positive and negative correlations respectively, affirming the existence of error-prone points in generated code. In contrast, we find that the prefix and suffix parts have little relation to the correctness of the final answer. It is important to note that in incorrect code, despite the errors in the `mid` section, the following suffix is not a significant cause of the errors. This observation justifies our decision to exclude the suffix in the reward modification in Section 3.3. These findings provide empirical evidence supporting our hypothesis that focusing on these error-prone points is essential to enhance model performance, which is the core motivation behind our Focused-DPO framework.

Accuracy of Continuation at Error-Prone Points We further generate 20 code solutions based on different contents at error-prone points, to explore the correctness of the final code generated under different conditions in Table 2.

Based on Input	pass@1	pass@3	pass@5	pass@10
Common Prefix + Chosen Mid	0.9002	0.9532	0.9688	0.9871
Common Prefix + Reject Mid	0.0317	0.0633	0.0810	0.1159

Table 2: Pass rates based on different content at error-prone points.

The pass rates shown in Table 2 highlight a striking contrast: using `chosen_mid` at error-prone points results in significantly higher pass rates, reaching around 90% at pass@1, compared to just over 3% for the `rej_mid` version. This demonstrates the critical importance of accurate content in the error-prone points for determining the correctness of the final generated code.

Based on the above results, we have noticed that **the generated content at the error-prone points significantly affects the final outcomes**. This leads to a question: *how do current code generation models behave at these error-prone points?*

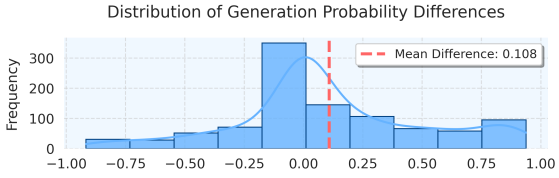

Figure 3: Generation probability difference (p(chosen_mid) − p(rej_mid)) with input.

Generation Preferences at Error-Prone Points in Code Models We further analyze the *Qwen-2.5-Coder-Instruct-7B*, which has been post-trained on million-level datasets using SFT and DPO. We

examine the generation preferences of this heavily post-trained model at error-prone points. Specifically, we calculate the probability difference between generating `chosen_mid` and `rej_mid` when given the `common_prefix` as input. The distribution of the difference is shown in Figure 3. The model exhibits little to no clear preference, indicating that existing code generation models lack effective generation capability at these error-prone points. Through this exploration, we confirm that focused preference optimization of error-prone points is crucial for improving the accuracy of code models, addressing RQ1.

5.2 Main Results (RQ2)

Results on benchmarks Tables 3 and 4 summarize the performance of Focused-DPO compared to various baselines, including standard DPO, Step-DPO, TDPO, and SFT. Note that the formulas for standard DPO and Step-DPO are identical, making them equivalent. The relative improvements (*Rel*) are reported for a clearer comparison.

Model	HumanEval	HumanEval+	MBPP	MBPP+
Instruct Model				
Qwen2.5-coder-instruct-7B	0.915	0.841	0.828	0.714
+ Our Focused-DPO	**0.927**	**0.878**	**0.847**	**0.762**
Relative Improvement	1.29%	4.41%	2.24%	6.71%
DPO / Step-DPO	0.921	0.854	0.841	0.743
Token-DPO	0.927	0.872	0.833	0.751
SFT	0.927	0.872	0.833	0.717
DeepSeekCoder-instruct-6.7B	0.774	0.701	0.751	0.659
+ Our Focused-DPO	**0.823**	**0.732**	**0.765**	**0.669**
Relative Improvement	6.35%	4.38%	1.80%	1.56%
DPO / Step-DPO	0.787	0.713	0.751	0.661
Token-DPO	0.799	0.726	0.751	0.661
SFT	0.787	0.726	0.759	0.667
MagiCoder-S-DS-6.7B	0.732	0.683	0.767	0.667
+ Our Focused-DPO	**0.823**	**0.744**	**0.794**	**0.698**
Relative Improvement	12.50%	8.93%	3.45%	4.76%
DPO / Step-DPO	0.762	0.701	0.772	0.675
Token-DPO	0.811	0.732	0.780	0.680
SFT	0.738	0.701	0.762	0.653
Base Model				
Qwen2.5-coder-base	0.835	0.787	0.794	0.683
+ Our Focused-DPO	**0.884**	**0.829**	**0.817**	**0.704**
Relative Improvement	5.89%	5.37%	2.95%	3.03%
DPO / Step-DPO	0.848	0.799	0.802	0.688
Token-DPO	0.866	0.799	0.815	0.690
SFT	0.848	0.805	0.802	0.688
DeepSeekCoder-base-6.7B	0.476	0.396	0.702	0.566
+ Our Focused-DPO	**0.518**	**0.427**	**0.717**	**0.574**
Relative Improvement	8.89%	7.79%	2.13%	1.43%
DPO / Step-DPO	0.488	0.396	0.709	0.569
Token-DPO	0.500	0.421	0.717	0.574
SFT	0.488	0.396	0.704	0.566

Table 3: Pass Rate on HumanEval(+), MBPP(+)

As shown in Table 3, Focused-DPO consistently outperforms the baseline models across all benchmarks. On the HumanEval(+) and MBPP(+) benchmarks, Focused-DPO improves relative accuracy by 4.79% on average over the baseline. We also evaluate on LiveCodeBench, a challenging benchmark that features iteratively updated, competition-level programming problems sourced from platforms such as LeetCode. The benchmark is divided into three levels of difficulty: Easy, Medium, and Hard. Focused-DPO achieves consistent improvements across all difficulty levels of LiveCodeBench. Notably, on the hardest category (*Hard*), Focused-DPO can achieve huge relative performance. Focused-DPO entirely outperforms other advanced preference optimization baselines such as Step-DPO and TDPO. These findings highlight the effectiveness of Focused-DPO in challenging code generation scenarios, where optimization on error-prone points of code plays a crucial role in determining final correctness.

Model	Easy	Medium	Hard	Average
Instruct Model				
Qwen2.5-coder-instruct-7B	0.692	0.220	0.034	0.312
+ Our Focused-DPO	**0.735**	**0.242**	**0.048**	**0.339**
Relative Improvement	6.22%	10.04%	42.86%	8.44%
DPO / Step-DPO	0.685	0.233	0.019	0.310
Token-DPO	0.706	0.239	0.037	0.325
SFT	0.670	0.208	0.015	0.295
DeepSeekCoder-instruct-6.7B	0.453	0.091	0.009	0.181
+ Our Focused-DPO	**0.477**	**0.106**	**0.019**	**0.197**
Relative Improvement	5.30%	15.89%	108.33%	8.87%
DPO / Step-DPO	0.462	0.094	0.007	0.184
Token-DPO	0.470	0.100	0.019	0.192
SFT	0.462	0.094	0.004	0.183
MagiCoder-S-DS-6.7B	0.481	0.107	0.001	0.193
+ Our Focused-DPO	**0.513**	**0.118**	**0.019**	**0.213**
Relative Improvement	6.56%	10.12%	1751.85%	10.10%
DPO / Step-DPO	0.491	0.109	0.004	0.198
Token-DPO	0.505	0.118	0.015	0.209
SFT	0.498	0.112	0.004	0.201
Base Model				
Qwen2.5-coder-base-7B	0.567	0.150	0.017	0.241
+ Our Focused-DPO	**0.595**	**0.175**	**0.030**	**0.264**
Relative Improvement	5.00%	16.47%	77.78%	9.23%
DPO / Step-DPO	0.577	0.151	0.015	0.244
Token-DPO	0.584	0.163	0.022	0.253
SFT	0.584	0.157	0.022	0.251
DeepSeekCoder-base-6.7B	0.399	0.074	0.004	0.155
+ Our Focused-DPO	**0.423**	**0.085**	**0.011**	**0.169**
Relative Improvement	6.00%	14.31%	177.78%	9.24%
DPO / Step-DPO	0.412	0.079	0.004	0.161
Token-DPO	0.419	0.079	0.004	0.164
SFT	0.419	0.082	0.007	0.166

Table 4: Pass Rate on LiveCodeBench

Enhancing Heavily Post-trained Models Focused-DPO can significantly enhance the performance of code models that have already undergone extensive post-training. As demonstrated in Table 5, models like *Qwen2.5-Coder-instruct*, which have been meticulously optimized using millions of data points from SFT and DPO processes, still exhibit substantial improvements with our Focused-DPO framework. To further illustrate Focused-DPO's benefits on heavily post-trained

models, we conducted an extensive initial DPO training phase. Following the methodology from CodeDPO, we used the model *DeepSeekCoder-base-6.7* and a large-scale dataset with 93k samples for DPO training, continued until full convergence. We then apply Focused-DPO for further experiments. This allows us to explore the extent to which Focused-DPO could drive additional improvements, even in models already trained by intensive post-training processes.

Model	HumanEval	HumanEval+	MBPP	MBPP+
DeepSeekCoder-base-6.7B	0.4760	0.3960	0.7020	0.5660
+ SFT Stage *(with MagiCoder-OSS-instruct)*	0.7317	0.6829	0.7672	0.6667
+ First DPO Stage *(with CodeDPO-OSS-instruct)*	0.8354	0.7622	0.8070	0.7093
+ Focused-DPO	**0.8719**	**0.7926**	**0.8227**	**0.7275**

Table 5: Performance of DeepSeekCoder-6.7B at different training stages. The stages include base model, SFT with MagiCoder, first DPO with CodeDPO, and our Focused-DPO. Focused-DPO achieves additional improvements even after high-quality post-training.

As shown in Table 5, we start from the base model and progressively incorporate the SFT stage (Wei et al., 2023b) and the first DPO stage (Zhang et al., 2024b). Finally, applying our Focused-DPO leads to the highest pass rates achieved. These results demonstrate that Focused-DPO effectively boosts the performance of models that have already been extensively post-trained and optimized through previous stages. We further evaluate how Focused-DPO enhances the quality of error-prone points in Appendix C.

5.3 Ablation Study (RQ3)

Dataset Construction Ablations Focused-DPO includes an automated data construction and Error-Prone Identification process. We perform ablation experiments on the dataset construction methods in Table 6. We design two alternative approaches: ❶ The *Step-DPO strategy* (Lai et al., 2024) constructs datasets by considering only the common prefix parts, with the rest treated as Error-Prone Points for training. ❷ Using a *git-diff tool* [1], we construct datasets where the differences covered by the diff were treated as Error-Prone Points, with the parts following the final diff difference treated as the suffix. Note that Step-DPO dataset construction method is closely tied to the formulation of the Step-DPO loss function, leading to consistent outcomes between the two. However, we observe that Step-DPO performs suboptimally on code generation tasks. In contrast, the current dataset construction method used in Focused-DPO, which employs a simple yet effective Error-Prone Identification strategy, achieves the best experimental results.

Loss Function Ablations Our Focused-DPO has made appropriate modifications to the calculation of positive and negative rewards. We carry out ablation experiments in Table 6, including trying different values of $w_{focused}$ and various treatments of the suffix in the reward function. Our findings indicate that increasing or decreasing $w_{focused}$ leads to a decline in model performance, suggesting that the current value of $w_{focused}$ is optimal. Additionally, we observe that including the suffix part in the reward function results in degraded performance. Through detailed analysis in Section 5.1, the suffix in incorrect code does not exhibit strong correlations with the overall accuracy. These experiments validate the practical advantages of the design choices in our loss function.

Dataset Construction	HumanEval / HumanEval+	MBPP / MBPP+
Focused-DPO Error Prone Identification	**0.9268 / 0.8780**	**0.8466 / 0.7619**
Step-DPO Strategy	0.9207 / 0.8537	0.8413 / 0.7434
Diff-based Strategy	0.9268 / 0.8598	0.8439 / 0.7539
Loss Function Setting	**HumanEval / HumanEval+**	**MBPP / MBPP+**
Focused-DPO $w_{focused} = 2$, No Suffix in Reject Reward	**0.9268 / 0.8780**	**0.8466 / 0.7619**
Decrease Weight $w_{focused} = 1$	0.9268 / 0.8720	0.8386 / 0.7487
Increase Weight $w_{focused} = 3$	0.9268 / 0.8720	0.8439 / 0.7566
$w_{focused} = 5$	0.8963 / 0.7683	0.8201 / 0.6878
Suffix in Reject Reward	0.9268 / 0.8659	0.8413 / 0.7487

Table 6: Dataset Construction and Loss Function Ablation Results based on *Qwen2.5-Coder-Instruct-7B*

6 Conclusion

We propose Focused-DPO, a framework that improves code generation by focusing on error-prone points. These critical parts significantly impact overall program correctness. Focused-DPO improves Direct Preference Optimization by prioritizing these points, using our Error-Point Identification method to create datasets without costly human annotations. Evaluations show Focused-DPO reduces errors and improves code quality, even in heavily post-trained models. This research highlights the benefits of focusing on fine-grained preference optimization in AI-driven software development.

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

Limitation

Despite the contributions of our work, there are several limitations that we aim to address in future research:

Comparison with Advanced RL Techniques While our study demonstrates the effectiveness of Focused-DPO, we do not extensively compare it with other advanced reinforcement learning (RL) alignment techniques, such as DeepSeek-R1 (Guo et al., 2025). These online RL alignment techniques typically require substantial training resources, high-quality datasets, and complex reward environments, making their application highly resource-intensive. In contrast, offline alignment methods such as Focused-DPO approximate similar optimization objectives while introducing necessary simplifications and derivations. This allows Focused-DPO to achieve comparable or even equivalent optimization results with significantly lower resource requirements. Moreover, we leverage prior knowledge discovered in this work: the insight that only a small part of the generated code—specifically, the Error-Prone Points—plays a critical role in determining the overall correctness of the output. By incorporating this insight into the training loss design, we further enhance training efficiency and effectiveness. Focused-DPO's low resource requirements and reliable performance make it applicable to a wide range of code generation scenarios. Further exploration of how Focused-DPO compares to these advanced RL techniques in performance and efficiency remains an area for future investigation.

Dataset Construction Strategy In Focused-DPO, we introduce a dataset construction technique named **Error-Prone Identification**, which automatically identifies error-prone points in generated code. The primary focus of this paper is on error-prone points associated with correctness in the final output code. However, other factors in source code, such as efficiency, readability, and security, are equally important for optimization. Exploring whether these factors also reveal "Error-Prone Points" in source code is an intriguing direction for future work. For example, techniques like static code analysis, code smells detection, and identification of common vulnerabilities could help identify and penalize insecure patterns during data construction, leading to safer and more robust code generation.

Additionally, our dataset construction pipeline includes specific design choices, such as the use of a page-rank mechanism and the identification of error-prone points based on common prefixes and suffixes. Our preliminary experiments suggest that these settings effectively support the performance of Focused-DPO. Detailed discussions on these designs are provided in Appendix A.

References

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

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

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4571–4585.

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2024. Code-optimise: Self-generated preference data for correctness and efficiency. *CoRR*, abs/2406.12502.

GPT-4. 2023. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo. *OpenAI*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang, Yihong Dong, Kechi Zhang, and Ge Li. 2024. aixcoder-7b: A lightweight and effective large language model for code completion. *CoRR*, abs/2410.13187.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv preprint arXiv:2406.18629*.

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*.

Zicheng Lin, Tian Liang, Jiahao Xu, Xing Wang, Ruilin Luo, Chufan Shi, Siheng Li, Yujiu Yang, and Zhaopeng Tu. 2024. Critical tokens matter: Token-level contrastive estimation enhances llm's reasoning capability. *CoRR*, abs/2411.19943.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. Step-controlled DPO: leveraging stepwise error for enhanced mathematical reasoning. *CoRR*, abs/2407.00782.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2023a. Starcoder2-instruct: Fully transparent and permissive self-alignment for code generation. *BigCode*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023b. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Yongcheng Zeng, Guoqing Liu, Weiyu Ma, Ning Yang, Haifeng Zhang, and Jun Wang. 2024. Token-level direct preference optimization. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*.

Dylan Zhang, Shizhe Diao, Xueyan Zou, and Hao Peng. 2024a. PLUM: preference learning plus test cases yields better code language models. *CoRR*, abs/2406.06887.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024b. Codedpo: Aligning code models with self generated and verified source code. *CoRR*, abs/2410.05605.

A Discussion

A.1 Error-Prone Points Identification

In our Focused-DPO method, we introduce a dataset construction technique called **Error Prone Identification** to automatically identify error-prone points in generated code. To assess the correctness of the code, we employ a self-generation-and-validation mechanism based on PageRank, which captures the relative quality of different code snippets (Zhang et al., 2024b). We are not like approaches such as Magicoder (Wei et al., 2023b), which directly use all test cases as ground truth. In our experiments we use the policy model to generate datasets. Since the policy model's generation quality is not as robust as that of more powerful models like GPT-4 (used in Magicoder), the PageRank-based method allows us to automatically filter out lower-quality test cases (those with lower scores after iteration), thereby ensuring higher overall dataset quality.

We find that different models exhibit varying levels of accuracy across different problems. Therefore, for each model's training dataset, we performed necessary filtering by removing code problems with excessively high or low accuracy rates, ensuring a consistent number of code problems in the final dataset. Moreover, we observe that models tend to exhibit similarities in error-prone points when solving the same problems. For example, when comparing the error-prone points identified by *DeepSeekCoder-instruct-6.7B* and *Qwen2.5-Coder-instruct-7B* models on the same set of programming problems, we found a 32% overlap. This indicates that there are commonalities in the error-prone points across different models.

In our ablation studies, we compare error-prone points constructed using the *git-diff* method and the *Step-DPO* method, noting slight differences in the final results. Balancing effectiveness and efficiency, we use the method based on *prefix* and *suffix*, which allows us to identify error-prone points in generated code in a simple yet effective manner. We plan to further explore more identification strategies in future work.

A.2 Data Scaling For Focused-DPO

In our experiments, we use the policy model to sample the dataset for training, with the dataset statistics provided in Table 8. We also explore how scaling the training data affects the final performance of Focused-DPO. Specifically, we investigate two additional settings: doubling the original training dataset to 10k samples and halving the dataset to 2.5k samples, to observe how these changes impact the effectiveness of the model after Focused-DPO training. The experimental results are presented in Table 7. The results indicate that fine-grained preference optimization converges efficiently within our given data range, and increasing the dataset size does not significantly improve the results.

Data Scaling	HumanEval / HumanEval+	MBPP / MBPP+
Qwen2.5-coder-instruct-7B	0.915 / 0.841	0.828 / 0.714
Focused-DPO (5k)	0.926 / 0.878	0.846 / 0.761
Decrease to 2.5k	0.926 / 0.847	0.830 / 0.719
Increase to 10k	0.926 / 0.878	0.843 / 0.756

Table 7: Dataset Scaling for Focused-DPO based on *Qwen2.5-Coder-Instruct-7B*

B Dataset Statistics

Dataset	Problems		Avg. Hidden Tests
HumanEval	164		9.57
HumanEval+			748.07
MBPP	378		3.11
MBPP+			105.40
LiveCodeBench	Easy	279	18.07
	Medium	331	21.81
	Hard	270	24.78

Table 8: Statistics of Evaluation Benchmark.

Statistics based on Qwen2.5-Coder-Instruct-7B	
Problems	
Training Set	5000
Validation Set	1000
Average Token Lengths	
Common Prefix	78.17
Common Suffix	33.98
Chosen Mid	57.37
of Total Chosen Code	34%
Rejected Mid	42.63
of Total Rejected Code	28%

Table 9: Training Dataset Statistics based on Qwen2.5-Coder-Instruct-7B

C Improvement in Error-Prone Points

We further evaluate how Focused-DPO enhances the quality in error-prone points. Using our validation dataset (Table 8), we measure the model's performance on these error-prone parts. The generation probability difference between *chosen_mid* and *reject_mid* in error-prone points is illustrated

in Figure 4 for the *Qwen2.5-Coder-Instruct-7B* model.

Compared to pre-Focused-DPO results (Figure 3), Focused-DPO demonstrates a strong preference for generating more accurate code at error-prone points. This improvement is particularly critical in complex coding tasks, where precise decisions in error-prone points directly impact the correctness of the generated code. For instance, on the **LiveCodeBench-Hard** dataset—which consists of challenging, dynamically problems—Focused-DPO achieves a significant improvement of 42.8% in correctness for the *Qwen2.5-Coder-Instruct* model. Notably, on this dataset, Focused-DPO achieves performance on par with **GPT-4o**, highlighting its ability to address difficult code generation tasks effectively.

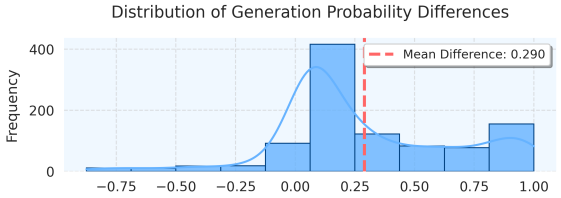

Figure 4: Generation Probability Difference (p(chose_mid) - p(reject_mid)) after Focused-DPO.

D Case Studies for Error-Prone Points

We show some case studies for error-prone points based on *Qwen2.5-Coder-instruct* in the following Figure 5, 6 and 7.

Question Prompt

Write a Python function that takes a string as input, and returns a valid variable name in Python. The function should perform the following steps:
1. Remove all invalid characters from the string, such as spaces, hyphens, and special characters.
2. Split the string into a list of words.
3. Capitalize the first letter of each word.
4. Join the list of words into a single string.
5. Prepend the resulting string with an underscore (_) if it begins with a number.
6. Return the resulting string as a valid variable name.

This function can be useful for converting user-provided input into a valid variable name for use in Python programs.

Correct Solution

```
def sol1(string):
    string = re.sub(r'[^\w\s]', '', string)

    words = string.split()

    words = [word.capitalize() for word in words]

    string = ''.join(words)

    if string[0].isdigit():
        string = '_' + string

    return string
```

InCorrect Solution

```
def sol2(string):
    string = re.sub(r'[\W_]+', '', string)

    words = string.split()

    words = [word.capitalize() for word in words]

    string = ''.join(words)

    if string[0].isdigit():
    string = '_' + string

    return string
```

Figure 5: Case Study: convert to valid variable name

Question Prompt

Write a Python function `parse_version` that takes a string representing a version number and returns a list of integers representing the version number. Each integer should correspond to a number in the version string. Note: This function should handle both a dot-separated and a dash-separated version string.

Correct Solution

```
def sol1(version_string):
    version = version_string.replace('-', '.').split('.')
    return [int(v) for v in version]
```

InCorrect Solution

```
def sol2(version_string):
    version = version_string.split('.')
    if len(version) == 4:
        version[-1], _ = version[-1].split('-')
    return [int(v) for v in version]
```

Figure 6: Case Study: parse version

Question Prompt

Implement a Python function to find the maximal value of the product of elements of a list `array` given a sliding window size `k`. The function should take two arguments: `array` and `k`, where `k` is optional and defaults to 2. The function should return an integer representing the maximal value of the product.

Correct Solution

```
def sol1(array, k = 2):
    if k > len(array):
        raise ValueError("k cannot be greater than the length of the
array")
    products = [1] * len(array)
    product = 1
    for i in range(k):
        product *= array[i]
        products[i] = product
    max_product = product
    for i in range(k, len(array)):
        product = product * array[i] // array[i - k]
        products[i] = product
        max_product = max(max_product, product)
    return max_product
```

InCorrect Solution

```
def sol2 (array, k = 2) -> int:
    if k > len(array):
        raise ValueError("k cannot be greater than the length of the
array")
    products = [1] * len(array)
    for i in range(k):
        products[i] = array[i]
    max_product = max(products[:k])
    for i in range(k, len(array)):
        products[i] = products[i - 1] * array[i] / array[i - k]
        max_product = max(max_product, products[i])
    return max_product
```

Figure 7: Case Study: max product