Code-Optimise: Optimising Code Language Models for Functional Correctness and Efficiency

Anonymous ACL submission

Abstract

Code Language Models are capable of generating solutions that are fairly functionally correct and efficient. However, previous work has focused on improving either functional correctness or efficiency, usually at the expense of the other. To this end, we introduce **Code-Optimise**, a lightweight optimisation for Code Language Models that incorporates learning signals for correctness (pass, fail) as well as code efficiency (fast, slow). During training, Code-Optimise dynamically selects solutions from our self-generated code preference data to reduce overfitting. Code-Optimise achieves significant improvements in $pass@k$ while decreasing average runtime by up to 6% for cheaper code execution. It also reduces the average length of generated solutions by up to 23% for HumanEval and up to 48% for MBPP for faster response / inference times while demonstrating the fastest overall (single) solutions ($best@k$). The data and code will be open-sourced at `www.open-source.link`

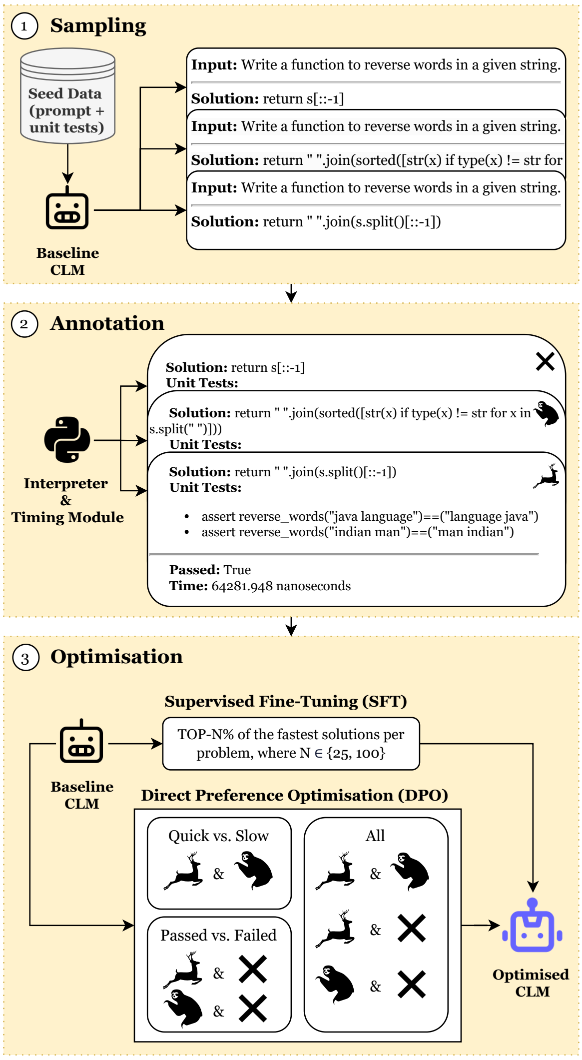


Figure 1: Overview of Code-Optimise. (1) Diverse solutions are sampled per problem. (2) A code interpreter annotates the solutions by functional correctness and runtime. (3) The CLM is optimised using SFT or DPO.

1 Introduction

Pretraining Code Language Models (CLMs) on large code repositories e.g. The Stack (Kocetkov et al., 2022; Lozhkov et al., 2024) gradually increases their understanding of code semantics. This enables CLMs to generate functionally correct and relatively efficient solutions to programming problems (Austin et al., 2021; Chen et al., 2021), among many other code related skills (Li et al., 2023). Subsequent CLM optimisation efforts have focused *either* on advancing code correctness *or* code efficiency, but not both. The most common way to improve functional correctness is distilled supervised fine-tuning (Tunstall et al., 2023; Xu et al., 2023; Luo et al., 2023; Wei et al., 2023) using training data generated by large models such as GPT-4 (Achiam et al., 2023). However, we aim to avoid reliance on proprietary APIs to make our methodology as *self-contained* as possible. Additionally, we seek to overcome the limitations of the standard supervised fine-tuning (SFT) loss, which only optimises for 'positive' examples with no means to *reduce the likelihood* of generating undesirable code (incorrect or slow solutions). This may be addressed by Reinforcement Learning (RL) methods (Le et al., 2022; Wang et al., 2022; Gorinski et al., 2023), however, RL algorithms are often complex and unstable. Beyond code correctness, Shypula et al. (2023) have shown that CLMs can optimise slow-running code to achieve large efficiency gains, however, this comes at a huge cost to functional correctness (down by up to ∼30%). Therefore, to the best of our knowledge, our work is the first to show improvements in code correctness *and* efficiency. We propose **Code-Optimise**, a lightweight optimisation for CLMs that incorporates learning signals for correctness (pass, fail) and efficiency (fast, slow). Code-Optimise, shown in Figure 1, dynamically selects solutions from our self-generated code preference data during training to reduce overfitting. The methodology consists of three steps: 1) *Sampling*; generate N solutions for each problem description, 2) *Annotation*; automatically label each solution with correctness and runtime, 3) *Optimisation*; train the CLM on the self-generated preference data using several lightweight configurations. The main benefits of Code-Optimise are:

- Functional correctness of code is significantly improved, particularly for smaller CLMs and lower $pass@k$. This is further enhanced with our train-time dynamic solution selection.
- Runtimes are reduced by up to 6% for MBPP to decrease costs of code execution. The average length of generated code is significantly reduced (up to 23% for HumanEval and up to 48% for MBPP), accelerating inference.
- The runtimes of single fastest solutions out of k generations improve by up to 6% for MBPP and up to 5% for HumanEval.

2 Background

Self-Optimising Models such as Self-Instruct (Wang et al., 2023) and Self-Rewarding Language Models (Yuan et al., 2024) share some similarities with our methodology as they also generate their own training data. However, their application is to instruction-tuning, which is easier than functional and efficient code generation plus their reliance on very large models deviates from our objectives.

Distilled Supervised Fine-Tuning has been applied to code generation to improve (only) functional correctness e.g. MagiCoder (Wei et al., 2023) and WizardCoder (Luo et al., 2023), however, they rely on large proprietary models to provide the training data (Cui et al., 2023; Xu et al., 2023), which is something we aim to avoid.

Direct Preference Optimisation RL for CLMs (Le et al., 2022; Wang et al., 2022; Gorinski et al., 2023) overcomes the shortcomings of supervised fine-tuning as a negative reward for dysfunctional code can be effectively propagated. However, RL algorithms come with added complexity and instability, which we aim to avoid by using Direct Preference Optimisation (DPO). It was recently proposed (Rafailov et al., 2024) as an alternative to Reinforcement Learning from Human Feedback (RLHF) (Touvron et al., 2023) to align LMs with human preferences (Tunstall et al., 2023). The authors claim that DPO is at least as effective as existing methods such as PPO-based RLHF, for preference learning in sentiment modulation, summarisation and dialogue. DPO does not require a separate reward model, the CLM is directly optimised with a simple classification objective.

Code Efficiency Optimisation was recently investigated by Shypula et al. (2023), aiming to transform slow-running code into a more efficient version with the same semantics. CLMs were trained on synthetic data, augmented by GPT-3.5 from a newly introduced dataset. The code efficiency was indicated on a 1-10 scale. At inference, the model is instructed to produce a 10/10 optimisation. However, the greatly reduced runtimes come at a *significant* cost to functional correctness, reduced by up to 30% in many configurations with the 'smaller' CLMs (7B, 13B) losing out more.

3 Code-Optimise

With our motivations and essential background explained, we can now introduce **Code-Optimise**, a lightweight optimisation for CLMs aimed at improving functional correctness of code as well as reducing its runtime / length. The method consists of the following three steps, shown in Figure 1.

3.1 Sampling

We assume access to $D_{seed} = \{x_i, y_i, ut_i\}_{i=1}^{N}$, a dataset of problem descriptions x_i and the corresponding unit tests ut_i that can be used for sampling and testing new solutions from the CLM, denoted CLM_{base} henceforth. Since fine-tuning the model *only* on the given solutions y_i is not

Model	**Split**	**Problem**			**Solution**			
		Total	**Filtered**	**Ratio**	**Total**	**Filtered**	**Ratio**	**CoV**
StarCoder-1B	Train	384	183	47.66	38400	15472	40.29	0.011
	Validation	90	40	44.44	9000	3533	39.26	0.010
StarCoder-3B	Train	384	211	54.95	38400	17575	45.77	0.007
	Validation	90	45	50.00	9000	3926	43.62	0.014
CodeLlama-7B	Train	384	250	65.10	38400	21350	55.60	0.007
	Validation	90	55	61.11	9000	4962	55.13	0.008
CodeLlama-13B	Train	384	261	67.97	38400	22182	57.77	0.007
	Validation	90	56	62.22	9000	5108	56.76	0.007

Table 1: Statistics of our self-generated data. 1) A **Model** generates 100 solutions per problem out of **Total** problems in each **Split**. 2) Functional correctness and runtime are annotated. 3) Problems are filtered to retain those with at least 2 passing and 1 failing solution (**Filtered**). A low coefficient of variation (**CoV** ≤ 0.1) across 5 runs indicates that runtime measurements are stable. **Ratio** is the percentage of **Filtered / Total** retained code solutions.

effective, we leverage its extensive pretraining to generate a *multitude* of diverse solutions to obtain additional training data. We sample 100 solutions from CLM_{base} for each problem description with multinomial sampling due to its lower computational cost. A temperature of $t = 0.6$ is applied to achieve a balance between functional correctness *and* diversity, resulting in non-uniform runtimes.

3.2 Annotation

The solutions are then automatically evaluated for functional correctness and runtime. While the former can be achieved by simply executing a solution with its corresponding unit tests, the latter requires additional steps for obtaining stable runtime measurements, see Algorithm 1. Each solution s is executed 50 times to determine its functional correctness ($passed$) and $runtime$ in nanoseconds. We obtain μ and σ, then calculate the coefficient of variation CoV. A measurement is deemed stable and accepted if $CoV \leq 0.1$ (usually much lower). Otherwise, we repeat the loop up to 1K times. In the *unlikely* scenario that a stable $runtime$ could not be obtained, we set $passed = False$ (mark solution as failed). In order to further increase the reliability of $runtime$ measurements, we execute Algorithm 1 five times (each in a *separate* process) and average the results. Once the solutions have been labelled, we sort them in ascending order of $runtime$ (fast > slow > failed). Lastly, we remove problems x_i, y_i, ut_i which do not have at least *two* passing and *one* failed solution to ensure that optimisation can be enhanced with our Dynamic Solution Selection (3.4) during training. The statistics of the final dataset D_{train} are shown in Table 3.

Algorithm 1 Timing module algorithm.

```
for $s \in solutions$ do
    $CoV \leftarrow \infty$
    repeat ▷ up to 1K times
        $times \leftarrow [\,]$ ▷ initialise empty list
        for $1, \ldots, 50$ do
            $runtime, passed \leftarrow \text{EXEC}(s)$
            $times.append(runtime)$
        $\mu, \sigma \leftarrow \text{MEAN}(times), \text{STD}(times)$
        $CoV \leftarrow \sigma / \mu$
    until $CoV \leq 0.1$
    if $CoV > 0.1$ then
        ▷ stable runtime was not obtained
        $passed \leftarrow False$
```

3.3 Optimisation

In this step, the model is efficiently fine-tuned on D_{train} to bias CLM_{base} towards generating more functionally correct and runtime-efficient solutions. Although several methods for preference data optimisation exist (Yuan et al., 2023; Liu et al., 2024; Azar et al., 2023; Ethayarajh et al., 2024; Hong et al., 2024; Zhao et al., 2023), we opt for DPO due to its simplicity and rapid adoption. We also use SFT due to its widespread use in related work.

Supervised Fine-Tuning (Equation 1) is commonly used in previous work, therefore, we also fine-tune CLM_{base} on D_{train} with SFT. We utilise the TOP-$N\%$ of fastest solutions where $N \in \{25, 100\}$, which means that the diversity of runtimes grows as N increases. Henceforth, the models optimised with the top 25% of fastest solutions are denoted as SFT_{25} and CLMs trained with all (including the *slowest*) solutions as SFT_{100}.

$$\mathcal{L}_{\mathrm{SFT}}\left(\pi_\theta\right) = -\mathbb{E}_{(x,y)\sim D}\left[\log \pi_\theta\left(y \mid x\right)\right] \tag{1}$$

Direct Preference Optimisation Aiming to avoid the complexity or instability of reinforcement learning, DPO (Rafailov et al., 2024) aligns a model to preference data with a simple classification loss, shown in Equation 3.3.

$$\mathcal{L}_{\mathrm{DPO}}\left(\pi_\theta; \pi_{\mathrm{ref}}\right) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}$$

$$\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\mathrm{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\mathrm{ref}}(y_l|x)}\right)\right] \tag{2}$$

We investigate the effectiveness of the following configurations of code preference pairs:

- **Quick versus Slow:** Choose *quick* & *slow* solutions according to the annotated runtime. We denote such models as DPO_{QvS}.[1]
- **Passed versus Failed:** Choose *passed* & *failed* pairs according to the annotated functional correctness, denoted as DPO_{PvF}.
- **All:** Choose all preference pairs from the *Quick vs. Slow* and *Passed vs. Failed* configurations. We denote such models as DPO_{All}.

3.4 Dynamic Solution Selection

Training data is typically fixed at the start of training and remains *static* throughout (Tunstall et al., 2023; Luo et al., 2023; Xu et al., 2023; Wang et al., 2023; Yuan et al., 2024). Our approach takes advantage of the multitude of code solutions from the sampling step (3.1) to *dynamically* select preference pairs *during training*. To this end, we randomly choose a new preference pair (y_w, y_l) for each problem x_i from D_{train} at the *start of the epoch* for DPO configurations. For SFT, we randomly choose *any working solution* (y_w) at the start of a new epoch for a comparable configuration. This reduces overfitting by presenting prompts with multiple code completions. The ablation of this intervention is presented in Section 5.4. In the main result section (5), all model configurations use our dynamic solution selection by default.

4 Experiments

4.1 Datasets

MBPP The Mostly Basic Programming Problems introduced by Austin et al. (2021) consists of 974 crowd-sourced Python programming challenges. Each problem comprises a description, an example code solution and a few automated test cases. The dataset contains training, validation and test splits. We utilise the training and validation splits for optimisation, while the test split serves as the in-domain test data distribution.

HumanEval (Chen et al., 2021) comprises 164 Python programming challenges. The function signatures, docstrings, example solutions and several unit tests were handwritten for each problem. We leverage HumanEval as our out-of-domain test set as the descriptions in MBPP do not contain any unit tests and the writing style of HumanEval problems does not follow a consistent format. This helps us evaluate robustness to handwritten prompts.

4.2 Implementation Details

We use the StarCoder (Li et al., 2023) and CodeLlama (Rozière et al., 2024) families of models in our experiments. We opt for the pretrained (base) versions with sizes of 1B and 3B for StarCoder and 7B and 13B for CodeLlama, hosted on HuggingFace (Wolf et al., 2020) repositories. During training, we fine-tune each model using a total of 30 epochs and select the best model based on the lowest validation loss. We use a learning rate of $5e^{-7}$ with a linear scheduler, a 10% warm-up and a maximum sequence length of 2048 tokens.

4.3 Evaluation Metrics

Functional Correctness is evaluated by sampling 100 solutions per problem via multinomial sampling and a temperature of $t = 0.6$. Following Chen et al. (2021), we measure functional correctness using $pass@k$, where $k \in \{1, 10, 100\}$.

Code Efficiency improvements are challenging to capture hence we evaluate *runtime* (median of all working solutions) as well as *code length* (median number of characters of all working solutions) to show different aspects of Code-Optimise. Since the runtime of a failed program is *undefined*, for a fair comparison between models, we remove problems for which any model has *no working solutions* to compare CLMs on the *same subset* of solved prob-

---

[1] NB: Only correct code can be assigned a runtime thus functional correctness and code efficiency *cannot be separated* as learning signals. Therefore, any DPO_{QvS} configurations are *implicitly* optimising CLM_{base} for correctness as well.

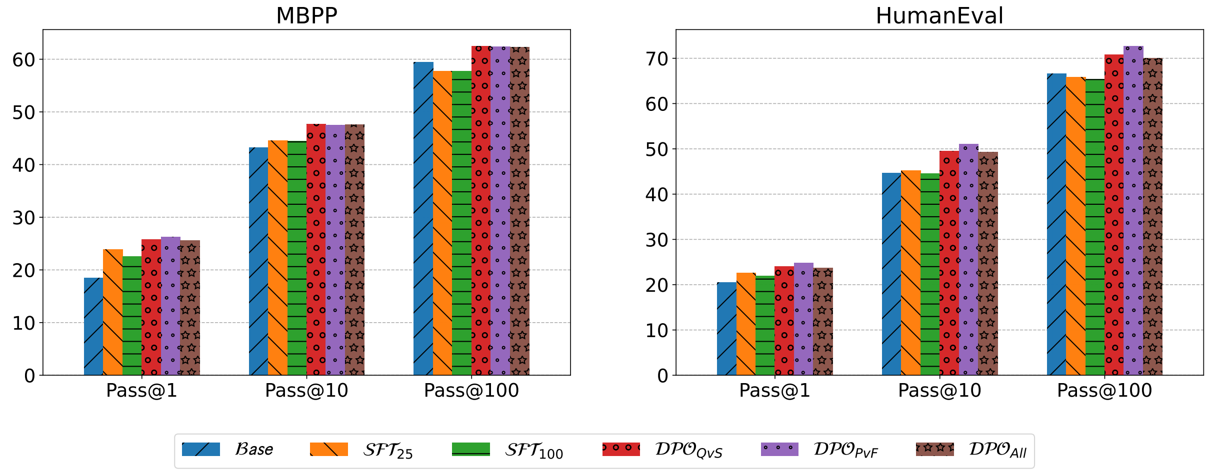


Figure 2: $pass@k$ scores for MBPP and HumanEval **averaged across model sizes** for a 'summary' view. Models optimised via DPO consistently show higher functional correctness compared to Base and SFT for all k.

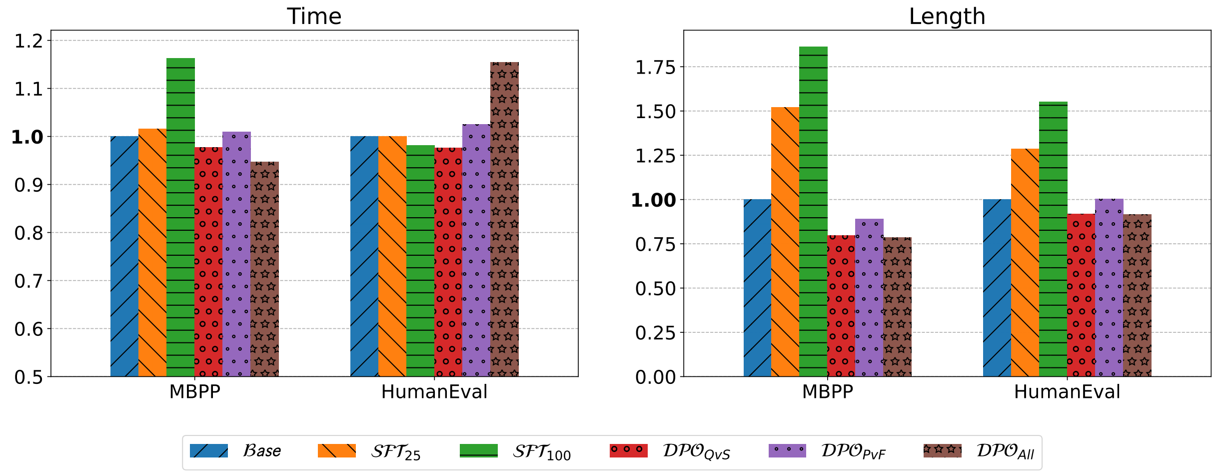


Figure 3: Median runtime and length of solutions for MBPP and HumanEval, **averaged across model sizes** for a 'summary' view. Values shown are *ratios relative to Base*, i.e. **>1** means *slower or longer* than Base by x% and **<1** means *faster or shorter* by x%. The DPO models have a lower runtime compared to Base and SFT models for in-domain but not out-of-domain problems. A significant reduction in code length is seen across both datasets.

lems. Table 2 shows that the intersection increases as CLMs get larger and more 'code-competent'.

5 Results

5.1 Functional Correctness

Figure 2 shows the $pass@k$ scores for MBPP and HumanEval, *averaged* over all model sizes / families for a 'summary' view. The individual $pass@k$ scores are shown in Figure 4. We observe that models optimised via DPO consistently demonstrate higher functional correctness relative to the baseline (Base) and SFT on both datasets. The effect is even larger on in-domain data, particularly with lower k. The different DPO configurations perform similarly on MBPP while DPO_{PvF} (passed vs. failed) is the best overall configuration for HumanEval. The SFT models show a marginal improvement for $k = 1$ but no improvement (or a small decrease) at higher k. We therefore conclude that DPO-based optimisation is a more suitable paradigm for our self-generated code preference data as it is better able to leverage the learning signals (fast, slow, passed, failed) compared to SFT.

5.2 Code Efficiency

The runtimes and lengths of generated programs are plotted in Figure 3 as *ratios relative to the baseline* (values < 1 mean faster or shorter than baseline, > 1 means slower or longer code). Once again, values are *averaged* over model sizes / families for a high-level overview. Individual model

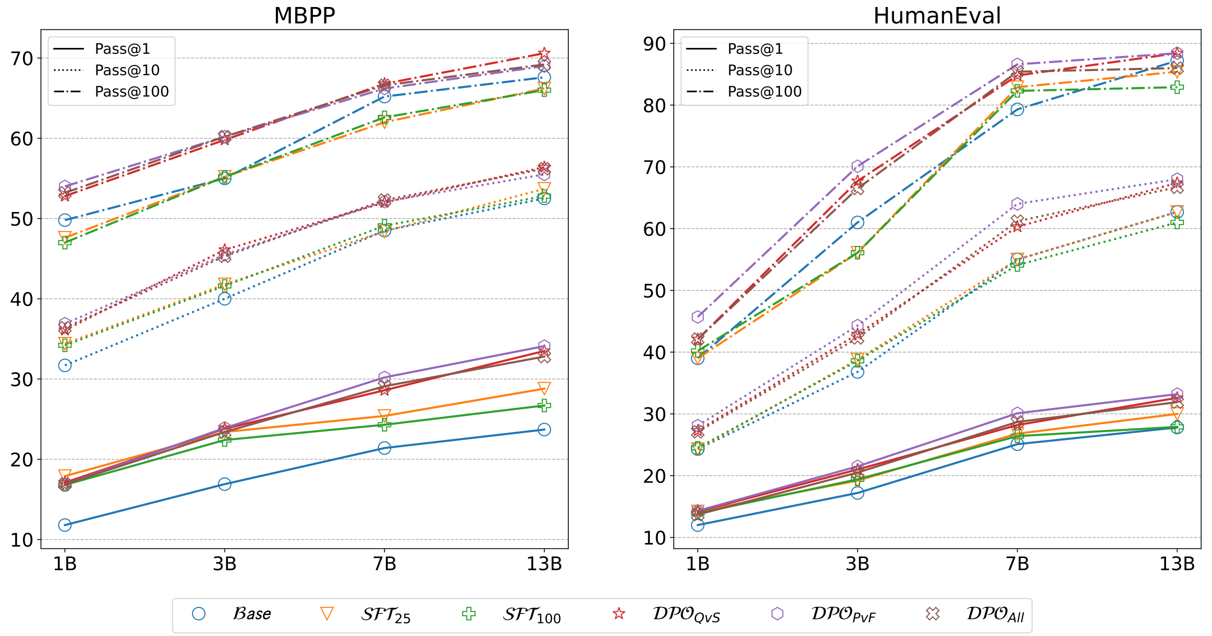


Figure 4: The $pass@1$, $pass@10$ and $pass@100$ scores for MBPP and HumanEval as the number of parameters increases. A significant improvement in scores over Base and SFT can be observed for DPO-based models.

Model	**MBPP**	**HumanEval**
StarCoder-1B	40.60%	30.49%
StarCoder-3B	48.40%	46.95%
CodeLlama-7B	55.60%	73.71%
CodeLlama-13B	60.40%	79.27%

Table 2: Intersection of problems between $Base$, SFT, and DPO models with at least one working solution.

scores are shown in Figures 5 and 6, respectively. In early experimentation, we noted that the Base models are already capable of generating solutions with a reasonably fast runtime. However, the DPO_{QvS} and DPO_{All} models manage to further decrease the runtime on in-domain data by up to 6% although not on the out-of-domain data. The SFT models generally *increase* the runtime across both datasets. In terms of average code length, the DPO models reduce the output by up to 22% on MBPP and up to 9% on HumanEval compared to the baseline. On the other hand, CLMs optimised with SFT tend to generate significantly longer solutions. This is particularly evident with SFT_{100}, which uses *all* code solutions for training, including the *slowest*, *which tend to be longer*. Causal language modelling does not appear to be particularly suitable for optimising runtime or average length of code with self-generated preference data as any inherent biases for generating longer code can be exacerbated.

In summary, Code-Optimise can reduce runtime, which means that the *cost of executing the code* has decreased while also outputting shorter programs, resulting in *faster generation and response times*.

5.3 Model Scaling

Figures 4, 5 and 6 show the evolution of functional correctness, runtimes and lengths of generated solutions as the number of trainable parameters is increased. Analysing $pass@1$ in Figure 4, we can see that *larger* DPO models achieve a more significant improvement over the baseline and SFT, particularly for in-domain problems. Somewhat surprisingly, functional correctness for HumanEval (out-of-domain) improves at a faster rate than MBPP (up to 7B parameters). In Figure 5, we observe that as the DPO_{QvS} and DPO_{All} models increase in size, their runtimes relative to the baseline improve by a *larger* margin. The DPO_{PvF} (passed v failed) configurations tend to show worse runtimes as this setup only optimises for *correctness*, likely at the expense of efficiency. There is no clear pattern for SFT models. On HumanEval, all models have a higher runtime than the baseline. From the analysis, it appears that runtime improvements do not generalise well to out-of-domain data with the limited number of prompts we have used for Code-Optimise. However, the effect on length does generalise well, particularly for larger CLMs (see Figure 6). In fact, we find a clear trend for all

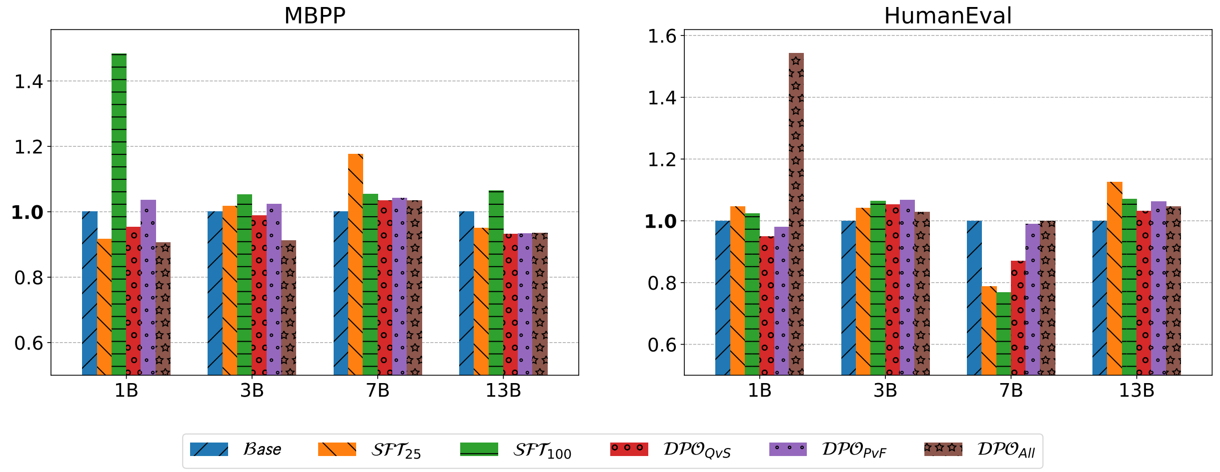


Figure 5: Runtimes for MBPP and HumanEval as model size increases. Values shown are *ratios relative to Base*, i.e. **>1** means *slower or longer* than Base by x% and **<1** means *faster or shorter* by x%. DPO models show a reduced runtime on the in-domain but not out-of-domain distribution. SFT models exhibit inconsistent scaling patterns.

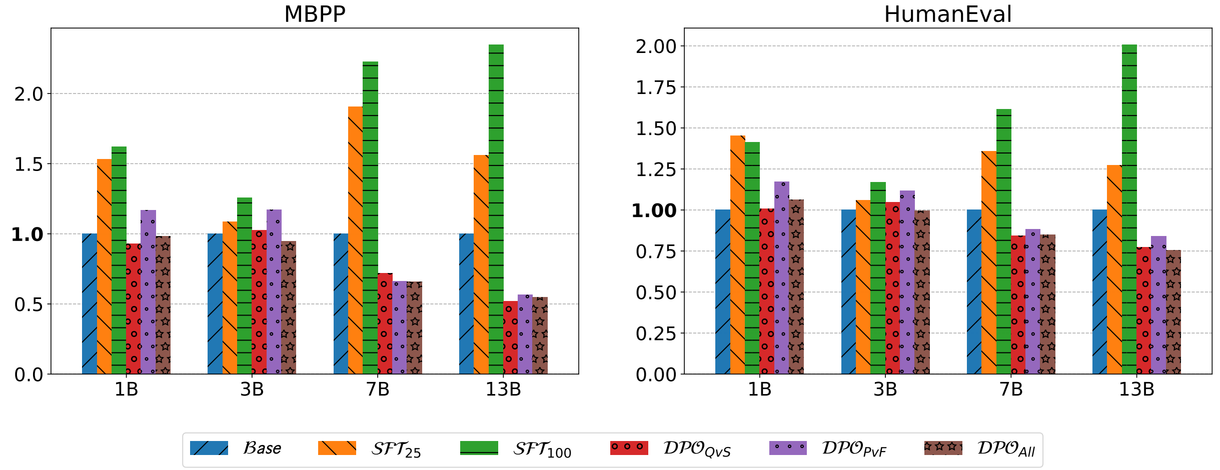


Figure 6: Lengths for MBPP and HumanEval as model sizes increase. Values shown are *ratios relative to Base*, i.e. **>1** means *slower or longer* than Base, **<1** means *faster or shorter*. DPO models consistently produce shorter sequences across both datasets. SFT models generate significantly longer code, particularly the larger CLMs.

DPO models on both datasets showing a reduced code length of up to 48% in-domain and up to 23% out-of-domain. SFT optimised models, however, increase the length in all cases, especially at larger model sizes. Same as the runtime behaviour, this is akin to exacerbating its own biases towards more verbose code as the training data is self-generated.

5.4 Dynamic Solution Selection

Our core methodology for creating high-quality code preference data enables us to dynamically select *unique pairs* for each prompt at the *start of a new epoch*. Since we train all models for 30 epochs, CLMs can potentially be exposed to many unique combinations of code completions. Figure 7 shows $pass@1$ scores for StarCoder-1B improving with dynamic code pair selection compared to static pairs randomly assigned at the *beginning* of training, commonly practiced in related work. The benefits are somewhat more pronounced for DPO, our preferred optimisation method given our code preference data, compared to SFT. However, across $pass@k$ (see Figure 9 in the appendix), all models generally benefit from dynamic solution selection.

5.5 Fastest Solutions Analysis

Following Shypula et al. (2023), we also show the *Best@k* metric, which considers only the *fastest solution* given k samples. We set $k = 100$ (all generated solutions), which is the basis of all our experiments. In Figure 8, we note that DPO models produce faster solutions not only on in-domain

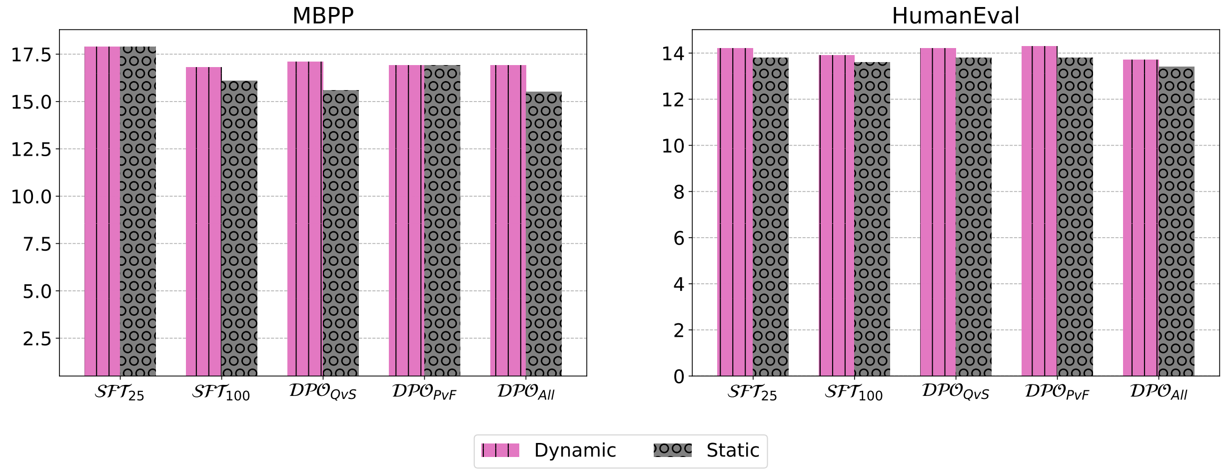


Figure 7: The $pass@1$ scores for StarCoder-1B without (**Static**) and with (**Dynamic**) solution selection (DSS). DSS improves performance for all models, especially DPO. Additional $pass@k$ figures (9) can be found in the appendix.

problems, but also *out-of-domain*, between 2% and 5% faster. DPO_{PvF} once again has the higher runtime as its objective is to optimise only functional correctness. SFT models' fastest solutions are generally slower on MBPP and HumanEval.

6 Conclusions

Code Language Models have demonstrated a strong ability to generate functionally correct and reasonably efficient solutions to programming problems. However, for use cases such as CLMs as Programming Assistants, it is desirable to further increase the pass rates, efficiency and brevity of generated code without relying on proprietary LLMs. To this end, we have introduced Code-Optimise, a computationally simple and efficient method for optimising CLMs using our self-generated code preference data that incorporates learning signals for correctness *and* efficiency (fast, slow, pass, fail). Using ∼200 prompts, our experiments show several benefits of Code-Optimise: 1) functional correctness is significantly improved, particularly for smaller models and lower $pass@k$, 2) dynamic solution selection during training provides an additional improvement in $pass@k$ by reducing overfitting, 3) runtimes are reduced by up to 6% for MBPP, reducing the costs of code execution, 4) average code lengths are significantly shorter, up to 48% for MBPP and up to 23% for HumanEval for the largest models, which reduces the cost of inference hence improving response times, 5) the runtimes of the fastest solutions ($Best@100$) are reduced for in-domain and out-of-domain problems. To the best of our knowledge, Code-Optimise is the first method that optimises CLMs for efficiency *and* correctness. We hope that our insights will stimulate further research in this area.

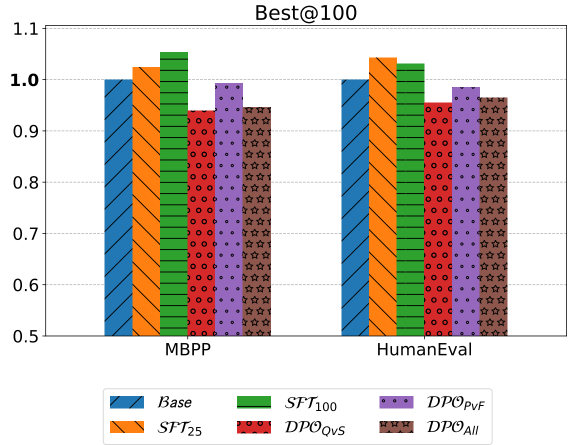


Figure 8: The $best@100$ scores for MBPP and HumanEval, **averaged across model sizes**. Values shown are *ratios relative to Base*, i.e. **>1** means *slower* than Base, **<1** means *faster*. Considering the fastest solution for each problem, DPO models show the best runtimes.

7 Limitations

Timing the execution of short programs accurately is challenging and despite our best efforts, the runtime measurements could probably be improved further with additional software engineering. This would also provide a cleaner and more stable learning signal for Code-Optimise, which could potentially improve results. While our methodology is highly data-efficient, using only ∼200 open-source prompts for data (self-)generation, obtaining additional high-quality problems (free from proprietary models and licensing issues) may potentially yield better results. Other code-related tasks that may be amenable to optimisation for efficient runtime/inference could potentially benefit from our methodology and as such may be investigated outside of the scope of this paper. While we conducted all experiments using Python, we acknowledge that less popular/similar programming languages should also be investigated in follow-up work.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

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*.

Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Rémi Munos. 2023. A general theoretical paradigm to understand learning from human preferences.

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*.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. 2023. Ultrafeedback: Boosting language models with high-quality feedback. *arXiv preprint arXiv:2310.01377*.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Kto: Model alignment as prospect theoretic optimization.

Philip Gorinski, Matthieu Zimmer, Gerasimos Lampouras, Derrick Goh Xin Deik, and Ignacio Iacobacci. 2023. Automatic unit test data generation and actor-critic reinforcement learning for code synthesis. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 370–384.

Jiwoo Hong, Noah Lee, and James Thorne. 2024. Orpo: Monolithic preference optimization without reference model.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

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

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J. Liu, and Jialu Liu. 2024. Statistical rejection sampling improves preference optimization.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation.

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.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

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

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 9–19.

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

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

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Huggingface's transformers: State-of-the-art natural language processing.

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

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, Songfang Huang, and Fei Huang. 2023. Rrhf: Rank responses to align language models with human feedback without tears.

Yao Zhao, Rishabh Joshi, Tianqi Liu, Misha Khalman, Mohammad Saleh, and Peter J Liu. 2023. Slic-hf: Sequence likelihood calibration with human feedback. *arXiv preprint arXiv:2305.10425*.

A Further Details

A.1 Sampling

Functional correctness of the CLMs during sampling is tabulated in Table 3.

A.2 Optimisation

Model performance of the CLMs on the test sets are tabulated in Tables 4, 5, 6, and 7. The CoV is shown beside each runtime.

A.3 Solution Selection

$pass$@10 and $pass$@100 scores for MBPP and HumanEval of StarCoder-1B by ablating the solution selection is shown in Figure 9.

Model	Split	Pass@1	Pass@10	Pass@100
StarCoder-1B	Train	14.00	34.50	55.20
	Validation	12.20	31.70	48.90
StarCoder-3B	Train	19.50	44.30	61.70
	Validation	19.20	42.50	57.80
CodeLlama-7B	Train	25.80	54.00	70.10
	Validation	23.40	50.30	68.90
CodeLlama-13B	Train	28.80	58.20	71.60
	Validation	24.60	52.90	66.70

Table 3: Functional correctness of the CLMs during sampling.

Model	Pass@1	Pass@10	Pass@100	Time	Length
$Base$	11.80	31.70	49.80	114338 ± 0.021	155
SFT_{25}	17.90	34.40	47.60	104690 ± 0.012	238
SFT_{100}	16.80	34.20	47.00	169536 ± 0.017	252
DPO_{QvS}	17.10	36.10	52.80	109051 ± 0.018	144
DPO_{PvF}	16.90	36.90	54.00	118418 ± 0.019	181
DPO_{All}	16.90	36.40	53.20	103588 ± 0.021	152

(a) MBPP

Model	Pass@1	Pass@10	Pass@100	Time	Length
$Base$	12.00	24.30	39.00	150930 ± 0.017	124
SFT_{25}	14.20	24.30	39.00	157975 ± 0.027	180
SFT_{100}	13.90	24.50	40.20	154395 ± 0.020	175
DPO_{QvS}	14.20	27.30	42.10	143259 ± 0.013	125
DPO_{PvF}	14.30	28.10	45.70	147980 ± 0.034	146
DPO_{All}	13.70	27.10	42.10	232759 ± 0.012	132

(b) HumanEval

Table 4: Model performance on MBPP and HumanEval of StarCoder-1B.

Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	16.90	40.00	55.00	113760 ± 0.016	158
SFT_{25}	23.40	41.80	55.20	115834 ± 0.011	171
SFT_{100}	22.40	41.60	55.20	119675 ± 0.035	198
DPO_{QvS}	23.80	46.10	59.80	112395 ± 0.008	162
DPO_{PvF}	23.90	45.50	60.20	116529 ± 0.017	185
DPO_{All}	23.40	45.30	60.20	103726 ± 0.012	149

(a) MBPP

Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	17.20	36.80	61.00	143806 ± 0.012	162
SFT_{25}	19.20	38.80	56.10	149743 ± 0.017	172
SFT_{100}	19.40	38.60	56.10	152948 ± 0.022	190
DPO_{QvS}	21.00	42.90	67.70	151401 ± 0.011	170
DPO_{PvF}	21.50	44.30	70.10	153620 ± 0.013	181
DPO_{All}	20.50	42.30	66.50	147823 ± 0.014	161

(b) HumanEval

Table 5: Model performance on MBPP and HumanEval of StarCoder-3B.

Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	21.40	48.50	65.20	105313 ± 0.012	196
SFT_{25}	25.40	48.40	62.00	124000 ± 0.058	372
SFT_{100}	24.30	49.10	62.60	110982 ± 0.010	435
DPO_{QvS}	28.60	52.00	66.80	108925 ± 0.013	141
DPO_{PvF}	30.20	52.10	66.20	109783 ± 0.006	129
DPO_{All}	29.10	52.30	66.60	108992 ± 0.016	129

(a) MBPP

Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	25.10	55.00	79.30	646547 ± 0.004	188
SFT_{25}	26.80	55.00	82.90	509264 ± 0.004	256
SFT_{100}	26.40	54.10	82.30	496296 ± 0.006	304
DPO_{QvS}	28.20	60.30	84.80	562279 ± 0.005	159
DPO_{PvF}	30.10	64.00	86.60	639553 ± 0.003	166
DPO_{All}	28.70	61.20	85.40	646486 ± 0.002	160

(b) HumanEval

Table 6: Model performance on MBPP and HumanEval of CodeLlama-7B.

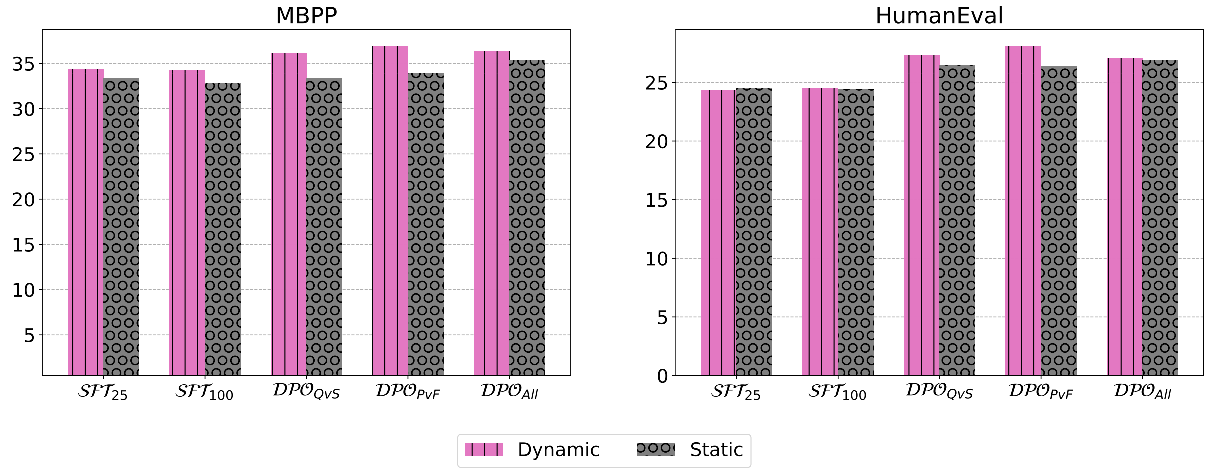
Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	23.70	52.50	67.60	118418 ± 0.009	223
SFT_{25}	28.80	53.70	66.20	112624 ± 0.006	348
SFT_{100}	26.70	52.80	66.00	126165 ± 0.004	523
DPO_{QvS}	33.50	56.40	70.60	110390 ± 0.008	116
DPO_{PvF}	34.10	55.50	69.00	110427 ± 0.018	126
DPO_{All}	32.80	56.20	69.20	110679 ± 0.008	122

(a) MBPP

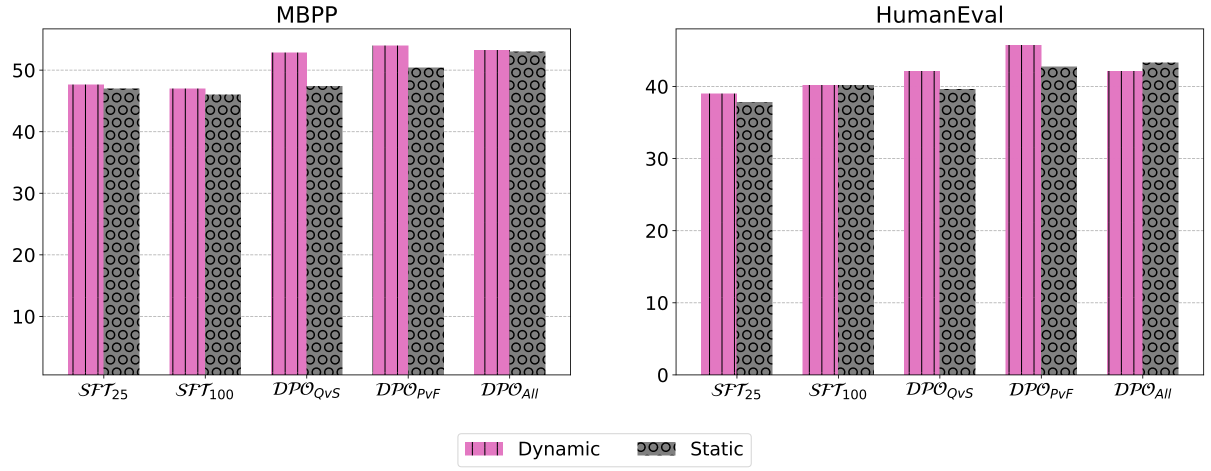
Model	**Pass@1**	**Pass@10**	**Pass@100**	**Time**	**Length**
$Base$	27.80	62.70	87.20	497649 ± 0.015	187
SFT_{25}	30.00	62.70	85.40	560336 ± 0.005	238
SFT_{100}	27.90	61.00	82.90	532856 ± 0.006	375
DPO_{QvS}	32.60	67.40	88.40	513372 ± 0.005	145
DPO_{PvF}	33.20	68.00	88.40	528546 ± 0.008	157
DPO_{All}	31.90	66.70	86.00	520788 ± 0.003	141

(b) HumanEval

Table 7: Model performance on MBPP and HumanEval of CodeLlama-13B.



(a) $pass@10$



(b) $pass@100$

Figure 9: The $pass@10$ and $pass@100$ scores for MBPP and HumanEval of StarCoder-1B with (**Dynamic**) and without (**Static**) solution selection. Performance improves on both metrics and distributions with DSS.