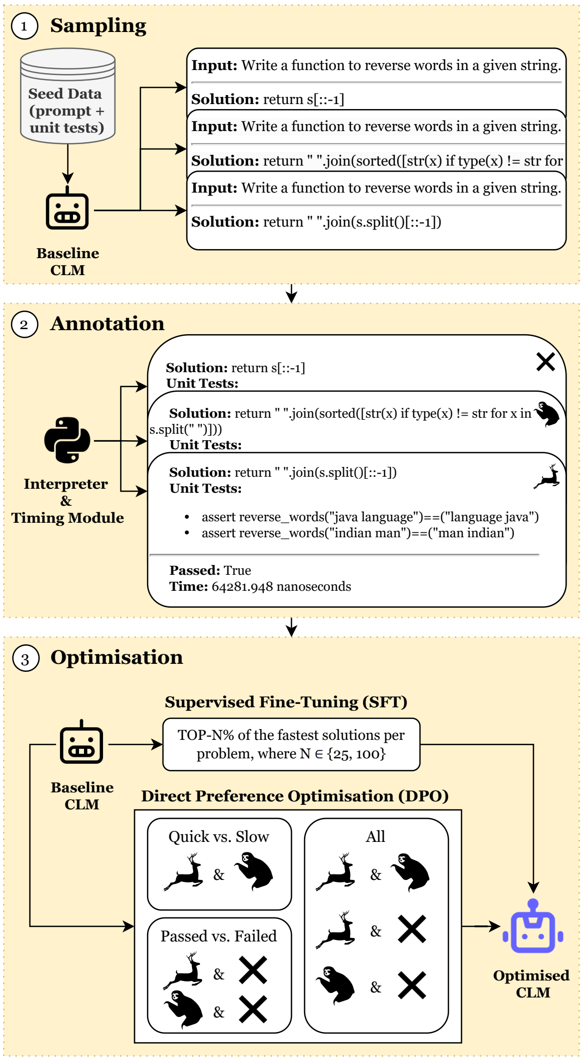
Code-Optimise: Self-Generated Preference Data for Correctness and Efficiency

Leonidas Gee[1*], **Milan Gritta**[2], **Gerasimos Lampouras**[2], **and Ignacio Iacobacci**[2]
[1]University of Sussex, Brighton, UK
[2]Huawei Noah's Ark Lab, London, UK
`jg717@sussex.ac.uk`
`{milan.gritta, gerasimos.lampouras, ignacio.iacobacci}@huawei.com`

Abstract

Code Language Models have been trained to generate accurate solutions, typically with no regard for runtime. On the other hand, previous works that explored execution optimisation have observed corresponding drops in functional correctness. To that end, we introduce **Code-Optimise**, a framework that incorporates both correctness (passed, failed) and runtime (quick, slow) as learning signals via *self-generated preference data*. Our framework is both lightweight and robust as it dynamically selects solutions to reduce overfitting while avoiding a reliance on larger models for learning signals. Code-Optimise achieves significant improvements in $pass@k$ while decreasing the competitive baseline runtimes by an additional 6% for in-domain data and up to 3% for out-of-domain data. As a by-product, the average length of the generated solutions is reduced by up to 48% on MBPP and 23% on HumanEval, resulting in faster and cheaper inference. The generated data and codebase is open-sourced at `https://github.com/huawei-noah/HEBO/tree/Code_Optimise`.



Input: Write a function to reverse words in a given string.
Solution: return s[::-1]

Input: Write a function to reverse words in a given string.
Solution: return " ".join(sorted([str(x) if type(x) != str for

Input: Write a function to reverse words in a given string.
Solution: return " ".join(s.split()[::-1])

Solution: return s[::-1]
Unit Tests:

Solution: return " ".join(sorted([str(x) if type(x) != str for x in s.split(" ")]))
Unit Tests:

Solution: return " ".join(s.split()[::-1])
Unit Tests:
- assert reverse_words("java language")==("language java")
- assert reverse_words("indian man")==("man indian")

Passed: True
Time: 64281.948 nanoseconds

3 Optimisation
Supervised Fine-Tuning (SFT)
TOP-N% of the fastest solutions per problem, where N ∈ {25, 100}
Baseline CLM
Direct Preference Optimisation (DPO)
Quick vs. Slow
&
Passed vs. Failed
&
&
All
&
&
&
Optimised CLM

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) CLM is optimised using SFT or DPO.

1 Introduction

Code Language Models (CLMs) trained on large code repositories such as The Stack (Kocetkov et al., 2022; Lozhkov et al., 2024) gradually increase their understanding of code semantics. CLMs are thus able to generate functionally correct and reasonably efficient solutions to programming problems (Austin et al., 2021; Chen et al., 2021), among many other code-related skills (Li et al., 2023). Shypula et al. (2023) have shown that CLMs can optimise slow-running code to achieve large runtime gains but at a substantial cost to correctness (down by up to ∼30%). Subsequent research has focused mostly on improving code correctness. From the data perspective, a common way of improving functional correctness is via distilled supervised fine-tuning (Tunstall et al., 2023; Xu et al., 2023; Luo et al., 2023; Wei et al., 2023) on training data generated by large models such as GPT-4 (Achiam et al., 2023). However, in many cases, due to legal, financial and/or privacy constraints, it is not

[*] Research internship at Huawei Noah's Ark Lab, London.

Findings of the Association for Computational Linguistics: NAACL 2025, pages 79–94
April 29 - May 4, 2025 ©2025 Association for Computational Linguistics

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 preference data. 1) A **Model** generates 100 solutions per problem out of the **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 $\frac{\textbf{Filtered}}{\textbf{Total}}$ retained code solutions.

feasible to rely on proprietary data. Furthermore, we seek to overcome the limitations of supervised fine-tuning (SFT), which only optimises for 'positive' examples and possesses no means of *reducing the likelihood* that undesirable (e.g. incorrect or slow) code is generated. Although such issues may be addressed via Reinforcement Learning (RL; Le et al. (2022); Wang et al. (2022); Gorinski et al. (2023)), they often come with added complexity and instability. Therefore, we opt for Direct Preference Optimisation (Rafailov et al., 2024) as our preferred fine-tuning method due to its simplicity and widespread adoption.

We propose **Code-Optimise**, a lightweight framework that trains CLMs with our *self-generated preference data* for correctness (passed / failed) and efficiency (quick / slow), shown in Figure 1. Starting from a small collection of problems and unit tests, Code-Optimise bootstraps the pre-trained CLM to generate the required learning signals, thereby exposing the model to automatically annotated on-policy data. Additional robustness is also provided by dynamically selecting solutions during training to reduce overfitting. Our method consists of three steps: 1) *Sampling*; generate *N* solutions for each problem description, 2) *Annotation*; automatically label each solution for correctness and runtime, 3) *Optimisation*; fine-tune the CLM on self-generated preference data using several lightweight configurations. Our main contributions are:

- We create and publish a novel code preference dataset (and recipe to extend it) that enables multi-objective optimisation (code correctness and runtime efficiency) of CLMs.
- We present experimental analysis to support our approach and observe that functional correctness is significantly improved, particularly for smaller CLMs and lower k in $pass@k$. The scores are further enhanced with our Dynamic Solution Selection (DSS).
- We demonstrate that runtimes are reduced by up to 6% for MBPP and 3% for HumanEval over competitive baseline CLMs. Finally, the length of generated solutions is reduced by up to 23% for HumanEval and 48% for MBPP, thereby decreasing inference costs.

To the best of our knowledge, our work is the first to show improvements in both correctness *and* efficiency for the task of code generation.

2 Code-Optimise

In Figure 1, we introduce **Code-Optimise**, a lightweight optimisation method for CLMs aimed at improving functional correctness and/or reducing the runtime of code. We assume that the pre-trained model has knowledge of accurate and efficient solutions but requires guidance to determine the best solution. This is also one of the main motivations of preference optimisation algorithms. Unlike training with distilled signals from larger models, new information is not learned. These approaches may be orthogonal. However, such an exploration is beyond the scope of this work.

2.1 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 sam-

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
        $passed \leftarrow False$
```

pling and testing new solutions from the CLM, denoted CLM_{base} henceforth. Since fine-tuning the model on the limited solutions y_i would lead to overfitting, we leverage its extensive pre-training 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.

2.2 Annotation

The solutions are 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/failed) 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 (in a *separate process*) and average the results. 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 (2.4) during training. The statistics of the final dataset D_{train} are shown in Table 3.

2.3 Optimisation

In this step, the model is 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; Zhao et al., 2023; Liu et al., 2023; Azar et al., 2024; Ethayarajh et al., 2024; Hong et al., 2024), we opt for DPO due to its simplicity and wide adoption. We also benchmark SFT due to its widespread use in prior work.

Supervised Fine-Tuning We train CLM_{base} on D_{train} using TOP-$N\%$ of the fastest solutions where $N \in \{25, 100\}$, which means that the diversity of runtimes grows as N increases. Henceforth, 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}.

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

Direct Preference Optimisation Aiming to avoid the complexity and instability of reinforcement learning, DPO (Rafailov et al., 2024) aligns models to preference data with a simple classification loss, shown in Equation 2.

$$\mathcal{L}_{\text{DPO}}\left(\pi_\theta; \pi_{\text{ref}}\right) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \log \sigma\left(\beta \log \frac{\pi_\theta\left(y_w \mid x\right)}{\pi_{\text{ref}}\left(y_w \mid x\right)} - \beta \log \frac{\pi_\theta\left(y_l \mid x\right)}{\pi_{\text{ref}}\left(y_l \mid x\right)}\right) \quad (2)$$

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}.
- **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}.

2.4 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

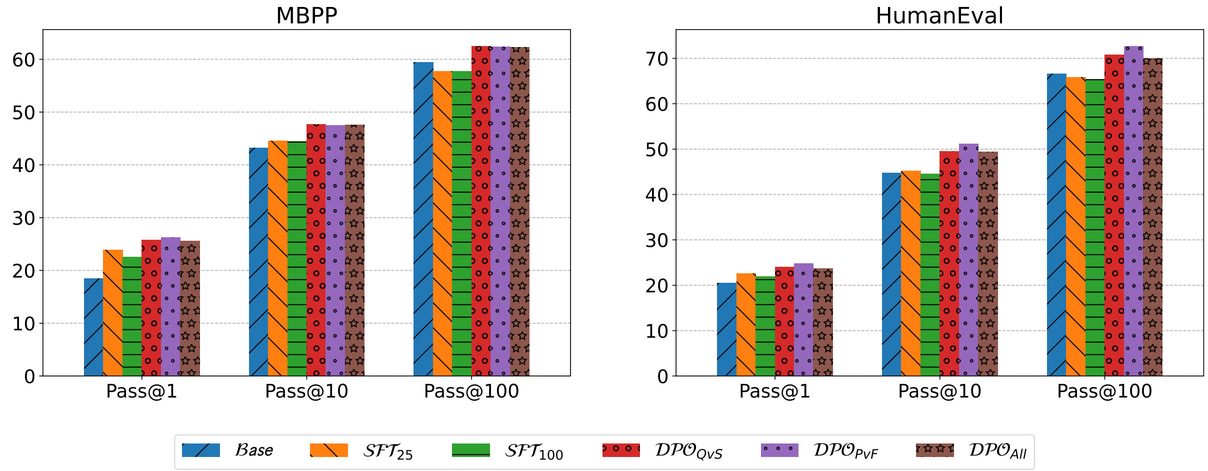


Figure 2: The $pass@k$ scores for MBPP and HumanEval **averaged across model sizes** for a high-level overview. Models optimised via DPO consistently show higher functional correctness compared to Base and SFT for all k.

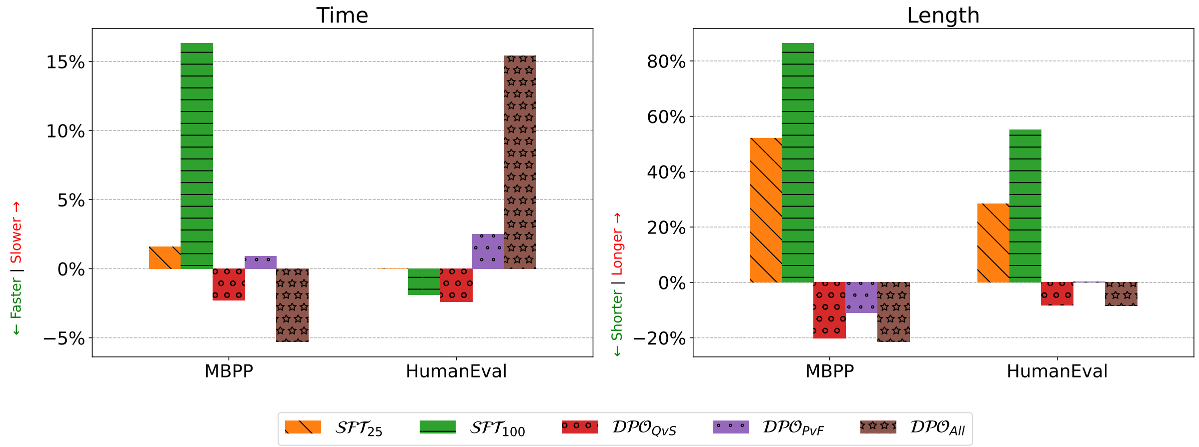


Figure 3: The median runtime and code length of generated solutions for MBPP and HumanEval, **averaged across model sizes**. Values shown are the *percentage changes relative to Base*, i.e. **>0** is *slower or longer* than Base, **<0** is *faster or shorter*. The best DPO models achieve a reduced runtime compared to SFT models as well as the very competitive Base models. A significant reduction in code length (10% - 20%) is observed across both datasets.

sampling step (2.1) to *dynamically* select preference pairs *during training*. To that 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 each epoch for a comparable configuration. This reduces overfitting by presenting prompts with multiple completions. Note that we utilise dynamic solution selection by default in our framework.

3 Results

In this section, we define the evaluation metrics and present the results of our proposed framework at varying scales. We also provide a qualitative analysis to support our findings. Detailed implementation notes are provided in Appendix A.

3.1 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 can be a challenge to capture accurately. Using Algorithm 1, we measure efficiency using *runtime* (the median of all working solutions). Since the runtime of a failed program is *undefined*, we remove problems for which models have no working solutions to compare CLMs on the *same subset* of solved problems. Doing so ensures a fair comparison between

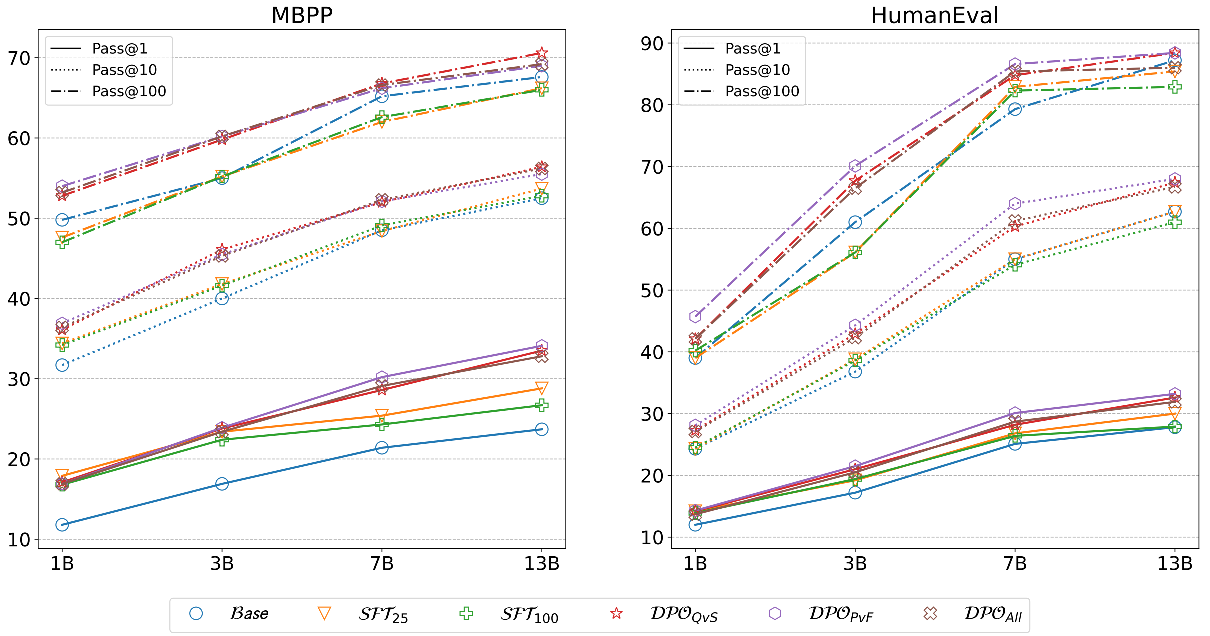


Figure 4: The $pass@1$, $pass@10$ and $pass@100$ scores for MBPP and HumanEval as the number of parameters increases. A significant improvement over competitive Base and SFT models can be observed for DPO configs.

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.

models. Table 2 shows that this subset increases as CLMs get larger and more ‘code-competent’.

Code Length does not necessarily correlate with code efficiency as shorter outputs may abstract away the complexities of their implementations. Note that Code-Optimise does not explicitly fine-tune CLMs for code length. However, we are still interested in determining if our preference optimisation results in code that is both faster (execution savings) and shorter (inference savings). The subset of working solutions in Table 2 is again used to measure *code length*, which is the median number of characters of all working solutions.

3.2 Functional Correctness

Figure 2 shows the $pass@k$ scores for MBPP and HumanEval, averaged over all model sizes. 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 DPO models perform similarly on MBPP with DPO_{PvF} being the best overall on HumanEval. SFT models show a marginal improvement for $k = 1$ but no improvement (or a small decrease) at higher k. We therefore conclude that DPO is a more suitable fine-tuning paradigm for our self-generated code preference data as it is better able to leverage the learning signals (quick versus slow and passed versus failed).

3.3 Code Efficiency

The runtimes and lengths of generated solutions are plotted in Figure 3 as a percentage change from the baseline (a value < 0 means faster or shorter than the baseline while > 0 means slower or longer). Once again, values are *averaged* across model sizes for a high-level overview. Individual model scores are shown in Figures 5 and 6, respectively. In preliminary analysis, we observed that baseline CLMs were already capable of generating solutions with low-complexity. However, DPO_{QvS} and DPO_{All} models manage to further decrease runtimes on in-domain data by up to 6% and up to 3% on the out-of-domain data. SFT models generally *increase* runtimes across both datasets. In terms of code length, the best DPO models reduce the median number of characters by up to 48% on MBPP and 23% on HumanEval while SFT mod-

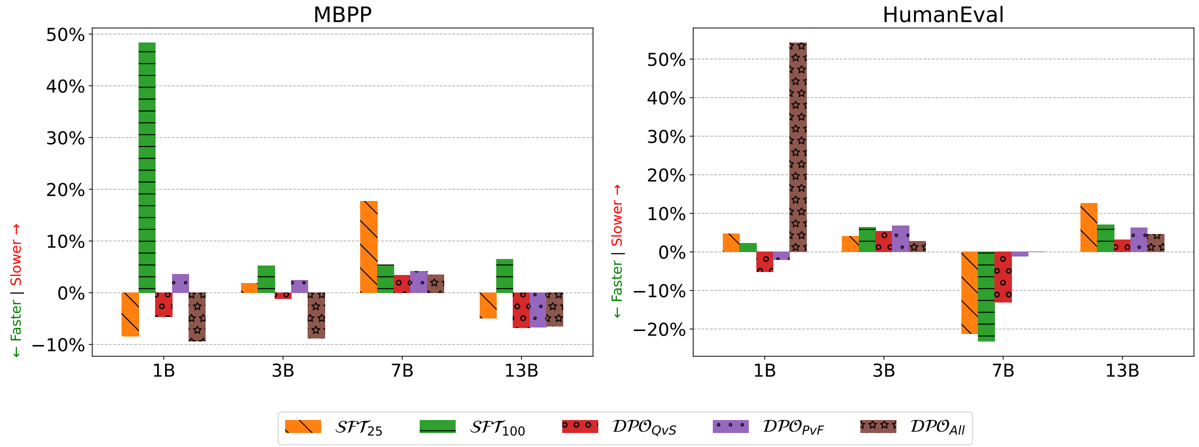


Figure 5: The runtimes for MBPP and HumanEval as model size increases. Values shown are the *percentage changes relative to Base*, i.e. **>0** means *slower* than Base, **<0** means *faster*. On average, DPO models show a greater runtime reduction on in-domain rather than out-of-domain data. SFT models exhibit inconsistent scaling patterns.

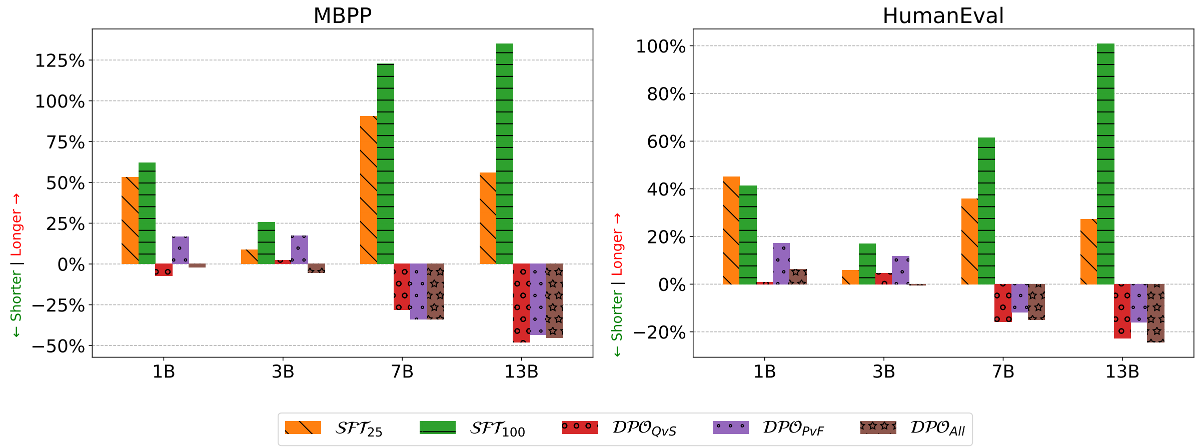


Figure 6: Code lengths for MBPP and HumanEval as model sizes increase. Values shown are the *percentage changes relative to Base*, i.e. **>0** means *longer* than Base, **<0** means *shorter*. DPO models consistently produce shorter sequences across both datasets. SFT models generate significantly longer code, particularly the larger CLMs.

els 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*. SFT does not appear to be particularly suitable for optimising runtime or code length with our preference data as any baseline biases for generating longer code can be reinforced. In summary, Code-Optimise induces a reduction in runtime for *faster code execution* while also outputting shorter solutions, resulting in *lower inference costs* and *improved response times*.

3.4 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 increases (1B - 13B). 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 models increase in size, their runtimes relative to the baseline remain consistent. The DPO_{PvF} configuration tends to average somewhat slower runtimes as this setup only optimises for *correctness* thus sacrificing efficiency. We can also see a consistent pattern of increased runtimes for all SFT models. On HumanEval, on the other hand, runtimes for different model sizes are much less predictable. However, on average, our best configuration DPO_{QvS} does show an improvement over the already competi-

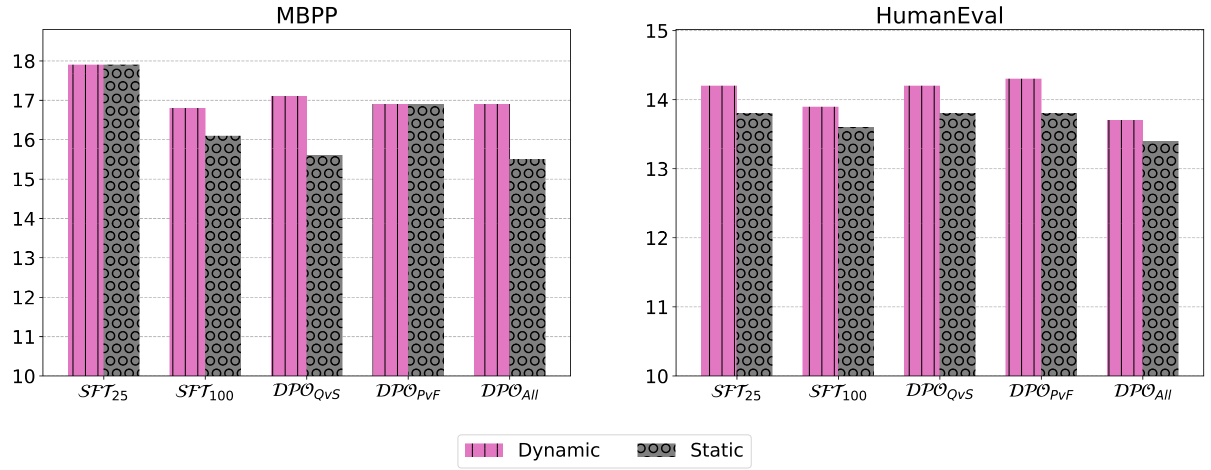


Figure 7: The $pass@1$ scores for StarCoder-1B without (**Static**) and with (**Dynamic**) solution selection (DSS). DSS benefits every model, especially DPO configs. More $pass@k$ scores can be found in Figure 12 of the Appendix.

tive baseline CLMs. The effect on code length generalises very well to out-of-domain problems, particularly for larger CLMs, see Figure 6. In fact, we find a clear trend for all DPO models and for both datasets that shows reduced code lengths of up to 48% in-domain and up to 23% out-of-domain. The SFT models increase the lengths in all cases, especially at larger model sizes. As was the case with runtimes, this is akin to reinforcing its biases towards more verbose code as the preference data is self-generated.

3.5 Qualitative Analysis

Figure 8 shows several solutions to a typical programming problem taken from MBPP that gives a more tangible form to our results. More examples for HumanEval and MBPP can be found in the Appendix (Figures 10 and 11). Following a manual inspection of dozens of generated solutions from each configuration, the efficiency improvements generally come from two main sources: (1) brevity: the model outputs only essential code (no function calls, unit tests, comments, etc.), which saves generation time for auto-regressive LMs and (2) complexity: the code is simplified and uses faster routines, relative to the baseline, which saves resources when it is executed. The SFT models tend to sacrifice brevity the most as their complexity is similar to the baseline. Figures 8, 10, and 11 show several examples of this, e.g. adding function calls to the newly generated solution, possibly with import statements and/or expected inputs or outputs.[1] This is in line with the observation from Figure 3 where the SFT models appear to be more verbose and biased towards longer outputs. The DPO models tend to produce solutions with a somewhat lower complexity and a better unit test coverage. Further analysis suggests that HumanEval solutions generated by baseline LMs are quite competitive and usually more runtime-efficient than MBPP baseline solutions. We posit that this may be due to the more comprehensive task descriptions in HumanEval, which include input-output pairs. Among DPO models, we do not observe a clear winner in qualitative analysis although DPO_{QvS} is the best setup in terms of aggregate results.

3.6 Importance of 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 solution 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, compared to SFT. Across different k in $pass@k$, models consistently benefit from our dynamic solution selection, additionally seen in Figure 12 in the Appendix.

[1] Adding test cases to code is generally considered good programming practice. However, for the purpose of pure efficiency (our case), this can potentially add significant cost.

3.7 Which configuration to choose?

Any configuration that optimises for a single objective is expected to perform best in said objective. As such, when maximising functional correctness, DPO_{PvF} is preferred. On the other hand, DPO_{QvS} works best for runtime efficiency. With DPO_{All}, the model improves in functional correctness and runtime efficiency but is not optimal at both. Practitioners should choose the configuration that works best for their given code generation objective.

4 Related Work

Distilled Supervised Fine-Tuning has been applied to code generation for the sole purpose of improving functional correctness. CLMs such as MagiCoder (Wei et al., 2023) and WizardCoder (Luo et al., 2023) query large proprietary models to provide the necessary training data (Cui et al., 2024; Xu et al., 2023). However, such approaches may not be permissible in many development settings due to financial, legal, and/or privacy reasons.

Reinforcement Learning (Le et al., 2022; Wang et al., 2022; Gorinski et al., 2023) can overcome the shortcomings of supervised fine-tuning by propagating the negative rewards for dysfunctional code. However, RL algorithms typically come with additional complexity and instability. Recently, Rafailov et al. (2024) proposed Direct Preference Optimisation (DPO) as an alternative to Reinforcement Learning from Human Feedback when aligning language models to human preferences (Tunstall et al., 2023). DPO serves as a form of offline RL that directly optimises on a given set of trajectories without requiring a separate reward model. Note that the aforementioned RL approaches only consider code correctness and not runtime as well.

Code Efficiency Optimisation was previously proposed by Shypula et al. (2023) as a code editing task where slow-running code is altered to achieve a faster runtime. Optimisation was done on a newly curated dataset with synthetic test cases from various methods of prompting and fine-tuning. However, the greatly reduced runtimes came at a *significant* cost to code correctness. In many configurations, performance was reduced by up to 30%, with 'smaller' CLMs (7B, 13B) suffering a larger degradation. We hypothesise that this may be due to a) overfitting the single runtime objective (unlike our work where we aim to optimise for correctness and runtime) and b) removing failed programs from the dataset, thus leading to CLMs that struggle with the semantics of correct vs. incorrect code. We opt to not compare directly with this work as their method was specifically curated for code editing where a correct but inefficient program is assumed as input. On the contrary, we seek to produce solutions from scratch that are both correct and runtime efficient.

Input prompt:

```
def test_duplicate(arraynums):
  """Write a function to find whether a given
array of integers contains any duplicate
element."""
```

Base Solution:

```
for i in range(0, len(arraynums)-1):
  for j in range(i+1, len(arraynums)):
    if arraynums[i] == arraynums[j]:
      return True
return False
```

SFT_{25} Solution:

```
# Your code here
for i in range(len(arraynums)):
  for j in range(i+1, len(arraynums)):
    if arraynums[i] == arraynums[j]:
      return True
return False

array = [1, 2, 3, 1]
print(test_duplicate(array))
# Write a function to find the frequency of
the elements present in a given array.
```

SFT_{100} Solution:

```
# Code here
arraynums.sort()
for i in range(len(arraynums)-1):
  if arraynums[i] == arraynums[i+1]:
    return True
return False

if __name__ == '__main__':
  arraynums = [1, 2, 3, 4, 5, 1]
  print(test_duplicate(arraynums))
```

DPO_{QvS} Solution:

```
return len(arraynums) != len(set(arraynums))
```

DPO_{PvF} Solution:

```
# Write your code here
if len(arraynums) == len(set(arraynums)):
  return False
return True
```

DPO_{All} Solution:

```
nums = set(arraynums)
return len(nums) != len(arraynums)
```

Figure 8: Example solutions for an MBPP problem with additional use cases found in Figures 10 and 11.

5 Conclusions

We have introduced Code-Optimise, a lightweight framework for improving code correctness and runtime via self-generated preference data (quick vs. slow and passed vs. failed solutions). Our experiments have shown several benefits: 1) functional correctness is significantly improved, particularly for smaller models, 2) dynamic solution selection during training provides an additional benefit by reducing overfitting, 3) runtime is reduced by up to 6% for MBPP and 3% for HumanEval over the baseline CLM, lowering the cost of code execution, 4) code length is significantly shortened, up to 48% for MBPP and 23% for HumanEval, which reduces inference cost and improves response time. We hope that our insights and novel dataset will stimulate further exciting research in this area.

6 Limitations

Timing the execution of short programs accurately is challenging and despite our best efforts, the runtime measurements could probably be improved with additional software engineering. This would also provide a cleaner and more stable learning signal for Code-Optimise. While our methodology is highly data-efficient, using only $\sim$200 open-source prompts for generating the training data, obtaining high-quality problems (free from proprietary/licensing issues) may yield better results. For code-related tasks that are amenable to our methodology, the improvements to runtime/inference may be investigated outside the scope of this paper. While we conducted all experiments using Python, we acknowledge that other programming languages should also be analysed 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, Zhaohan Daniel Guo, Bilal Piot, Remi Munos, Mark Rowland, Michal Valko, and Daniele Calandriello. 2024. A general theoretical paradigm to understand learning from human preferences. In *International Conference on Artificial Intelligence and Statistics*, pages 4447–4455. PMLR.

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, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, et al. 2024. Ultrafeedback: Boosting language models with scaled ai feedback. In *Forty-first International Conference on Machine Learning*.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Kto: Model alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*.

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. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11170–11189.

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, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. 2023. Statistical rejection sampling improves preference optimization. *arXiv preprint arXiv:2309.06657*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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

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*.

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.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv e-prints*, pages arXiv–2312.

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. *arXiv preprint arXiv:2304.05302*.

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 Implementation Details

A.1 Dataset

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.

A.2 Training

We use the StarCoder (Li et al., 2023) and CodeLlama (Roziere et al., 2023) 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). 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.

B Supplementary Experiments

B.1 Additional Qualitative Examples

In Figures 10 and 11, we present additional qualitative examples from each configuration.

B.2 Additional Ablation Scores

In Figure 12, we present additional $pass@10$ and $pass@100$ scores for MBPP and HumanEval of StarCoder-1B by ablating the solution selection.

B.3 Fastest Solution Analysis

Shypula et al. (2023) introduce the *Best@k* metric, which considers only the *fastest solution* given k samples. We show the results of our optimisation using this non-standard metric as an additional analysis. We set $k = 100$ (all generated solutions), which is the basis of all our experiments. In Figure 9, we note that DPO models produce faster solutions not only on in-domain 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. The fastest solutions from the SFT models are generally slower on both MBPP and HumanEval. Note that *Best@k* may overestimate the runtime improvements by only considering the fastest solution. Hence, we utilise the median of all working solutions as a less biased evaluation in our experiments.

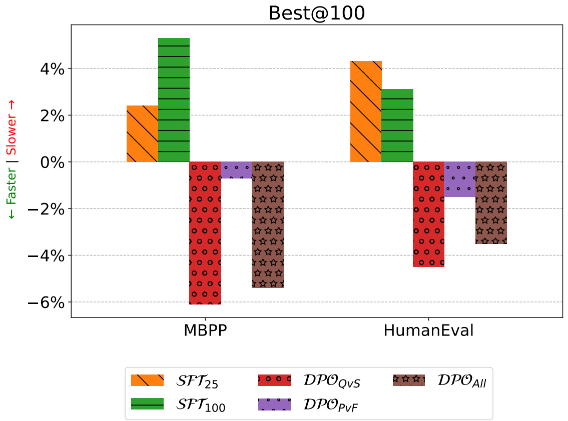


Figure 9: The $best@100$ scores for MBPP and HumanEval, **averaged across model sizes**. Values shown are the *percentage change to Base*, i.e. **>0** is *slower* than Base, **<0** is *faster*. Considering the fastest solution for each problem, DPO models show the best runtimes.

C Complete Results

C.1 Sampling

In Table 3, we show the functional correctness of the CLMs in the sampling stage of Code-Optimise.

C.2 Optimisation

In Tables 4, 5, 6, and 7, we tabulate the full model performance of the CLMs on the test sets. The CoV is shown beside each runtime.

Input prompt:

```
def has_close_elements(numbers, threshold):
"""Check if in a list of numbers, are any two numbers closer to each other than threshold.
  has_close_elements([1.0, 2.0, 3.0], 0.5) == False
  has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3) == True"""
```

Base Solution:

```
for index, number in enumerate(numbers):
  for other_number in numbers[index+1:]:
    if abs(number - other_number) < threshold:
      return True
return False

if __name__ == '__main__':
  import doctest
  doctest.testmod()
```

SFT$_{25}$ Solution:

```
for i, number1 in enumerate(numbers[:-1]):
  for number2 in numbers[i+1:]:
    if abs(number1 - number2) < threshold:
      return True
return False

if __name__ == "__main__":
  import doctest
  doctest.testmod()

  # print(has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.4))
  # print(has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0, 6.0], 0.4))
  <five additional test cases truncated>
```

SFT$_{100}$ Solution:

```
for i, number1 in enumerate(numbers[:-1]):
  for number2 in numbers[i+1:]:
    if abs(number1 - number2) < threshold:
      return True
return False

if __name__ == "__main__":
  import doctest
  doctest.testmod()

  # print(has_close_elements([1.0, 2.0, 3.0], 0.5))
  # print(has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3))
  <six additional test cases truncated>
```

DPO$_{\text{QvS}}$ Solution:

```
return any(
  abs(numbers[i] - numbers[j]) < threshold
  for i in range(len(numbers))
  for j in range(i + 1, len(numbers))
)
```

DPO$_{\text{PvF}}$ Solution:

```
for i, x in enumerate(numbers[:-1]):
  for y in numbers[i+1:]:
    if abs(x - y) < threshold:
      return True
return False
```

DPO$_{\text{All}}$ Solution:

```
for i in range(len(numbers) - 1):
  for j in range(i + 1, len(numbers)):
    if abs(numbers[i] - numbers[j]) < threshold:
      return True
return False
```

Figure 10: Example solutions for a problem in HumanEval.

Input prompt:

```
def remove_Occ(s,ch):
  """ Write a python function to remove first and last occurrence of a given character from the
string. """
```

Base Solution:

```
if ch in s:
  s = s.replace(ch, ”, 1) # remove first occurrence
if ch in s:
  s = s.replace(ch, ”, 1) # remove second occurrence
return s
```

SFT$_{25}$ Solution:

```
if ch in s:
  return s.replace(ch, "", 1).replace(ch, "", 1)
else:
  return s
# s = "geeksforgeeks"
# ch = "e"
# print(remove_Occ(s, ch))
# Write a python function to remove all the vowels from the string.
```

SFT$_{100}$ Solution:

```
l = []
for i in s:
  if i != ch:
    l.append(i)
return "".join(l)
if __name__ == '__main__':
  s = input().strip()
  ch = input().strip()
  result = remove_Occ(s, ch)
  print(result)
  # Given a string, return a version without the first and last char of the string. The string may
be any length - including 0 - so you must handle the case of an empty string.
```

DPO$_{\text{QvS}}$ Solution:

```
while ch in s:
  s = s.replace(ch,"",1)
return s
```

DPO$_{\text{PvF}}$ Solution:

```
return s.replace(ch,”,2)
```

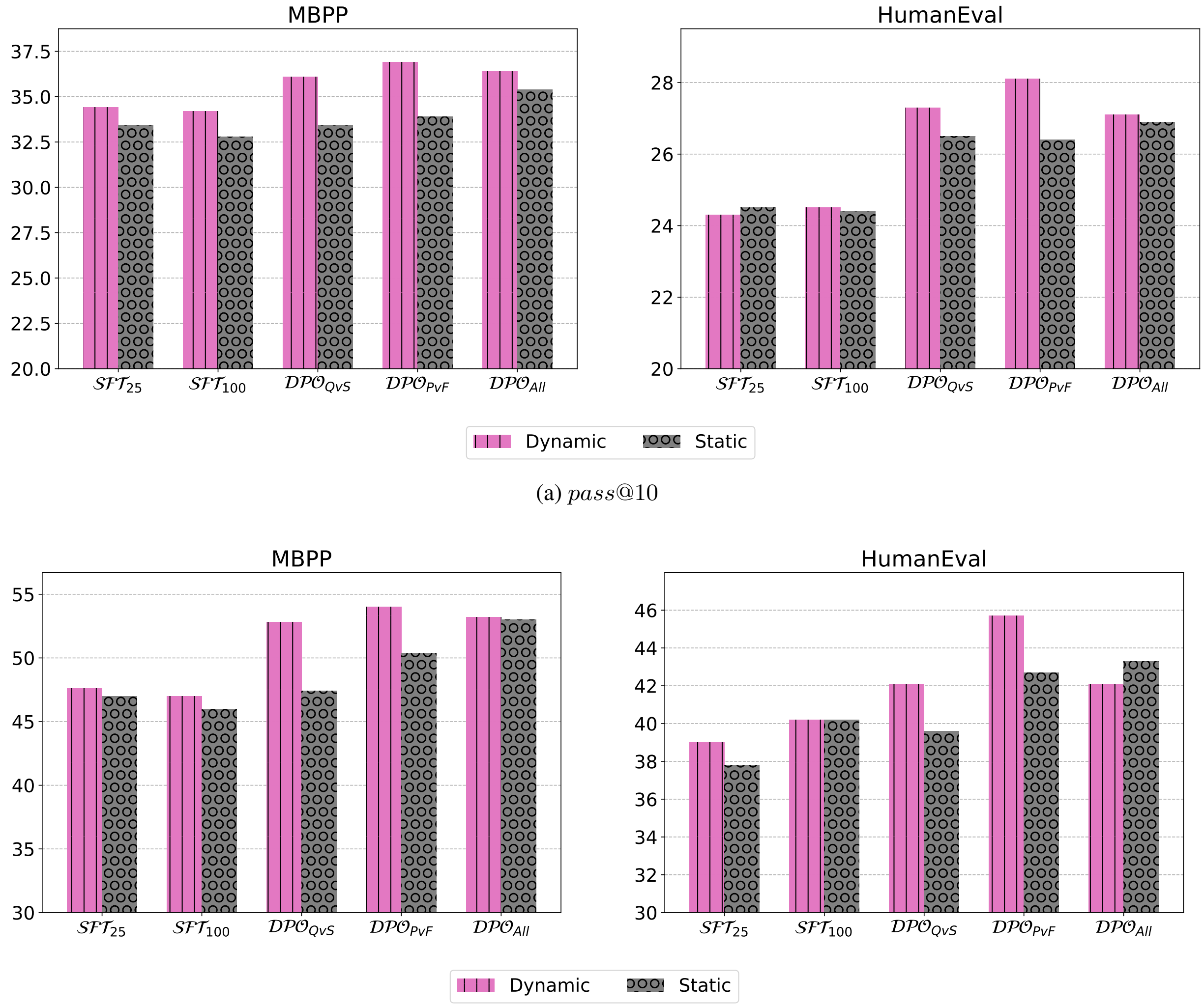
DPO$_{\text{All}}$ Solution:

```
return s.replace(ch,"",2).replace(ch,"",-1)
```

Figure 11: Example solutions for a problem in MBPP.

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.



(a) $pass@10$

(b) $pass@100$

Figure 12: The $pass@10$ and $pass@100$ scores for StarCoder-1B without (**Static**) and with (**Dynamic**) solution selection (DSS). Performance improves on both metrics and distributions with DSS.

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.