11-667 Homework 6 Report

Arthur Chien * **Gilbert Wu*** **Yaxuan Mao***
{yuhangch, poyuw, yaxuanm}@andrew.cmu.edu

1 Picking a Task and Dataset

1.A Task Description & Motivation

The primary task for this project is code generation based on natural language descriptions. To be more specific, the model receives a natural language instruction (e.g., "Write a Python function to calculate the Fibonacci sequence") and must generate the corresponding executable code. This task motivates us because it bridges the gap between human intent and machine execution, facilitates software development and increase developer productivity.

1.B Data Description

We utilize the CodeAlpaca-20k dataset for tuning and assessing task performance. This dataset is designed to follow the instruction-tuning format of the original Alpaca dataset but is specialized for code generation. The dataset covers multiple programming languages including **Python**, **JavaScript**, **Java**, and **C++**, and includes diverse tasks such as code generation, debugging, refactoring, and documentation. Below is an example data point:

- **Instruction:** "Create a function in Java that takes an integer as an argument and returns the factorial of that integer."
- **Output:**

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

The total size of the dataset is approximately 20,000 instruction-response pairs, with 18,000 examples in the training split and 2,000 examples in the testing split.

1.C Ethical Considerations

While code generation is generally beneficial, there can still got some ethical risks. First, bias in training data. If the Code Alpaca dataset contains insecure coding patterns (e.g., SQL injection vulnerabilities), the model may propagate these security flaws. Second, automation bias. Users might trust generated code without verification, leading to deployment of buggy or malicious code. And third intellectual property. There are ongoing debates regarding whether models trained on public code repositories infringe on open-source licenses, though Code Alpaca is synthetic, mitigating some direct copyright concerns compared to scraping GitHub directly.

1.D Formulation of Training Data

We formulate this as a sequence-to-sequence generation task. The input we use is a structured prompt template containing the instruction and optional input context, and the target is the code solution. For preprocess, we format the data into a standard instruction-following template to ensure consistency across models. Below is an example of input/target pair.

- **Input Sequence:**

```
### Instruction:
Sort the following list of numbers
in ascending order: [3, 1, 4, 1, 5, 9]

### Output:
```

- **Target Sequence:** `[1, 1, 3, 4, 5, 9]` (with EOS token)

1.E Method for Evaluation

To rigorously assess performance, we use the BigCode Evaluation Harness to evaluate our models and report pass@1 and pass@10 metrics. Unlike n-gram matching metrics (like BLEU) which are unsuitable for code, pass@k measures functional correctness by executing the generated code against

*Everyone Contributed Equally – Alphabetical order

unit tests. Also, we set the random seed of all experiments as 11667 for reproducibility. These are the benchmarks we utilized:

- HumanEval (Chen, 2021): 164 handwritten Python problems to test core logic .
- HumanEval+ (Liu et al., 2023): An extended version with 80x more test cases to test code with edge cases.
- InstructHumanEval: Augments HumanEval with natural language instructions to better test instruction-following capabilities.
- MultiPL-E (Cassano et al., 2022): Evaluates the model's ability to generalize across the multiple languages present in our training set (C++, Java, JavaScript).

Note on prompt selection: We additionally utilized the MBPP (Austin et al., 2021) dataset solely to facilitate the selection of optimal few-shot prompt examples from Code Alpaca for our In-Context Learning experiments, rather than as a evaluation benchmark.

2 Adapting a Language Model to your Task

2.A Method for In-Context Learning

We compare two instruction-tuned models from different families to determine the efficacy of prompting strategies without weight updates.

Models Selected:

1. Qwen2.5-Coder-3B-Instruct: A state-of-the-art small coding model known for high performance. (Hui et al., 2024)

2. Deepseek-Coder-1.3B-Instruct: A lightweight yet competitive code generation model.(Guo et al., 2024)

Prompting Strategy:

To determine the most effective in-context learning configuration, we experimented with two families of prompts that differ in both structure and linguistic content. The first category consisted of natural-language–heavy templates following a `Problem → Solution` format, while the second used minimal, code-only demonstrations containing purely Python function definitions without explanations. Our goal was to assess how much explicit natural language helps or harms small code-focused models when performing MBPP-style tasks.

For `Qwen2.5-Coder-3B-Instruct`, we conducted a systematic prompt selection experiment on the first 100 problems from MBPP, testing five different configurations: baseline (task + signature only), 0-shot (instruction + task + signature), 1-shot, 3-shot, and 5-shot code-only examples. The code-only few-shot examples consisted of pure Python function definitions (e.g., `def factorial(n): ...`) without markdown formatting or additional instructions, following the observation that Qwen2.5-Coder tends to generate markdown code blocks when given instruction-style prompts.

Contrary to initial expectations, adding natural-language descriptions substantially degraded performance for `Deepseek-Coder-1.3B-Instruct`. For example, a prompt beginning with "Below is an example showing how to solve a programming problem..." reduced pass@1 from 0.39 to 0.145. This outcome is consistent with the behavior of small code models whose instruction tuning is largely *code-centric*. Rather than leveraging task descriptions, the model becomes distracted by natural-language text that falls outside its core pretraining distribution, reducing its ability to produce syntactically correct and fully executable code.

Table 1: Pass@1 results on MBPP dataset (first 100 problems). All models are evaluated at temperature 0.2 and top-p 0.95.

Model	Baseline	0-shot	1-shot	3-shot	5-shot
Qwen2.5-Coder-3B-Instruct	20.0	20.0	26.0	32.0	**35.0**
Deepseek-Coder-1.3B-Instruct	**44.6**	42.5	36.3	37.3	40.1

The results reveal a striking contrast between the two model families. For **Qwen2.5-Coder-3B-Instruct**, few-shot examples significantly improve performance, with the 5-shot configuration achieving the best results (35.0%, compared to 20.0% for baseline). The performance increases monotonically with the number of shots: baseline and 0-shot both achieved 20.0%, 1-shot reached 26.0%, 3-shot reached 32.0%, and 5-shot achieved the optimal performance at 35.0%. This represents a 75% relative improvement over the baseline, suggesting that Qwen2.5-Coder benefits from seeing multiple code examples that help it understand the expected output format and coding style.

In contrast, for **Deepseek-Coder-1.3B-Instruct**, we observe a counterintuitive trend: increasing the number of few-shot demonstrations does not improve performance. The best-performing prompting configuration was the **baseline** setting without any prefix (44.6%). Adding 0/1/3/5-shot demonstrations consistently decreased pass@1 performance. This behavior is consistent with known limitations of small code models. Deepseek-Coder-1.3B is optimized primarily for code completion rather than natural-language instruction following or pattern-based ICL. MBPP problem statements already provide a strong natural-language specification; additional few-shot examples introduce distribution shift that small models struggle to integrate. Instead of learning a higher-level pattern, the model tends to treat the prefix as noise, reducing its ability to focus on the new task. Furthermore, our few-shot examples (e.g., factorial, palindrome, Fibonacci) do not closely match the distribution of MBPP problems, which are predominantly short string and list manipulation tasks. The mismatch further reduces effectiveness.

This divergence highlights the importance of model-specific prompt optimization: while Qwen2.5-Coder benefits from few-shot demonstrations, Deepseek-Coder performs best with minimal prompting. Based on these results, we selected the 5-shot configuration for Qwen2.5-Coder-3B-Instruct and the baseline configuration for Deepseek-Coder-1.3B-Instruct for all subsequent evaluations.

2.B Method for Finetuning

We apply Parameter-Efficient Fine-Tuning (PEFT) using LoRA (Low-Rank Adaptation). This allows us to adapt the base models to the specific style of Code Alpaca without the high computational cost of full fine-tuning.

Models Selected:

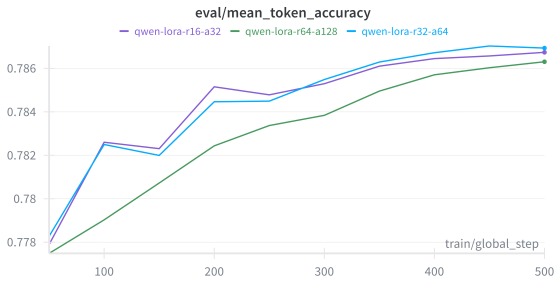
1. Qwen2.5-Coder-3B (Base Model)

2. Deepseek-Coder-1.3B (Base Model)

Hyperparameters & Decisions:

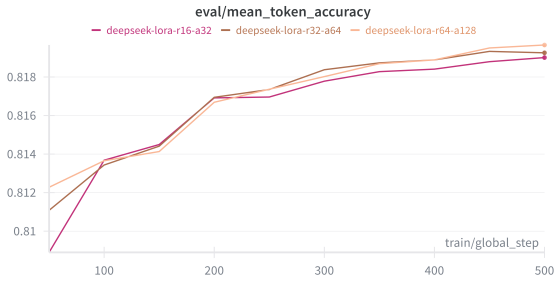
We adopted a systematic approach to hyperparameter optimization to balance model plasticity with stability. For all experiments, we utilized a fixed learning rate of 2×10^{-4} coupled with a cosine learning rate scheduler and 50 warmup steps. The training process was capped at 500 steps, employing a micro-batch size of 4 and gradient accumulation steps of 4 to achieve an effective batch size of 16. To mitigate overfitting, we applied a weight decay of 0.05 and set both LoRA and attention dropout probabilities to 0.1. We investigated the impact of low-rank adaptation capacity by conducting a sweep across three distinct configurations: a high-rank setting ($r = 64, \alpha = 128$), a medium-rank setting ($r = 32, \alpha = 64$), and a low-rank setting ($r = 16, \alpha = 32$).

This comparative analysis allowed us to determine the optimal trade-off between computational efficiency and the model's ability to absorb the new instruction-following distribution. Based on the sweep results, we selected the optimal configuration for each model to maximize evaluation accuracy while minimizing loss. For Deepseek Coder, the high-rank configuration ($r = 64, \alpha = 128$) yielded the best performance with an evaluation accuracy of 81.97% and the lowest evaluation loss of 0.6346. Conversely, the Qwen model benefited from slightly higher capacity, with the medium-rank configuration ($r = 32, \alpha = 64$) achieving the best results: 78.69% accuracy and a loss of 0.7445. Consequently, we proceeded to train these specific best-performing models for an extended duration of 1500 steps.

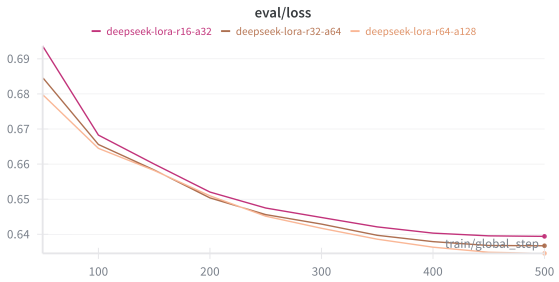


(a) Qwen2.5: Evaluation Accuracy

(b) Qwen2.5: Evaluation Loss



(c) Deepseek Coder: Evaluation Accuracy



(d) Deepseek Coder: Evaluation Loss

Figure 1: Hyperparameter Sweep Results: Comparing Accuracy and Loss across different LoRA configurations (rank=16, rank=32, rank=64).

3 Experiments (required)

3.A Results for In-Context Learning

The ICL evaluation results for both Qwen2.5-Coder-3B-Instruct and Deepseek-Coder-1.3B-Instruct are presented in the following tables. For Qwen2.5-Coder-3B-Instruct, the 5-shot configuration boosts HumanEval performance to **88.41%** pass@1 / **97.56%** pass@10 (vs. 84.1% / 92.35% with the baseline prompt). On HumanEval+, the same prompt reaches **85.71%** pass@1 / **96.43%** pass@10 compared to 80.5% / 91.08% for the baseline.

In addition to the baseline configuration, we also evaluated Deepseek-Coder-1.3B-Instruct using the best-performing MBPP prompt discovered during the ICL prompt search, the 0-shot instruction prefix. Although this prefix provided slight improvements on MBPP relative to other few-shot configurations, its performance on HumanEval and HumanEval+ remained consistently below the baseline. The 0-shot prompt introduces additional natural-language context that provides little value to this small code-focused model and may interfere with its ability to process the tightly structured function signatures used in HumanEval-style tasks. Consequently, we confirm that the baseline (no-prefix) setting remains the most effective configuration for Deepseek-Coder-1.3B-Instruct across all Python code generation benchmarks.

Table 2: Pass@1 results on Python benchmarks. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	HumanEval	HumanEval+
Qwen2.5-Coder-3B-Instruct (baseline)	84.1	80.5
Qwen2.5-Coder-3B-Instruct (ICL)	**88.41**	**85.71**
Deepseek-Coder-1.3B-Instruct (baseline)	59.14	56.10
Deepseek-Coder-1.3B-Instruct (ICL)	45.36	43.04

Table 3: Pass@10 results on Python benchmarks. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	HumanEval	HumanEval+
Qwen2.5-Coder-3B-Instruct (baseline)	92.35	91.08
Qwen2.5-Coder-3B-Instruct (ICL)	**97.56**	**96.43**
Deepseek-Coder-1.3B-Instruct (baseline)	75.00	72.56
Deepseek-Coder-1.3B-Instruct (ICL)	56.70	55.48

On MultiPL-E cross-language benchmarks, Qwen2.5-Coder-3B-Instruct shows strong performance across all languages, with the best results on Java (62.98% pass@1, 72.67% pass@10), followed closely by C++ (62.86% pass@1, 77.02% pass@10), and slightly lower on JavaScript (59.88% pass@1, 80.12% pass@10). Interestingly, while JavaScript has the lowest pass@1 score, it achieves the highest pass@10 score (80.12%), suggesting that the model can eventually generate correct JavaScript solutions with multiple sampling attempts, but may struggle more with the initial generation compared to other languages.

Table 4: Pass@1 results on MultiPL-E. All models are evaluated at temperature 0.2 and top-p 0.95

Model	C++	Java	JavaScript
Qwen2.5-Coder-3B-Instruct (baseline)	63.29	63.98	23.16
Qwen2.5-Coder-3B-Instruct (ICL)	62.86	62.98	59.88
Deepseek-Coder-1.3B-Instruct (baseline)	44.03	41.30	51.73
Deepseek-Coder-1.3B-Instruct (ICL)	41.98	36.33	15.77

Table 5: Pass@10 results on MultiPL-E. All models are evaluated at temperature 0.2 and top-p 0.95

Model	C++	Java	JavaScript
Qwen2.5-Coder-3B-Instruct (baseline)	78.26	73.91	48.45
Qwen2.5-Coder-3B-Instruct (ICL)	77.02	72.67	80.12
Deepseek-Coder-1.3B-Instruct (baseline)	57.14	54.65	65.21
Deepseek-Coder-1.3B-Instruct (ICL)	57.14	53.41	25.46

3.B Results for Finetuning

Based on the experimental results, the effectiveness of supervised fine-tuning varies between the two model families. For Deepseek-Coder, SFT improves performance across all Python benchmarks, with most substantial improvements on InstructHumanEval (23.68→32.01 pass@1). This pattern holds for pass@10 metrics as well, suggesting that SFT successfully enhanced the model's instruction-following capabilities without degrading its core code generation abilities.

In contrast, Qwen2.5-Coder exhibits severe performance degradation after fine-tuning on standard code generation tasks. However, the model shows improvement on InstructHumanEval (34.57→45.06), indicating successful adaptation to instruction-following formats.

Table 6: Pass@1 results on Python benchmarks. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	HumanEval	HumanEval+	InstructHumanEval
Qwen2.5-Coder Base	**52.01**	**41.28**	34.57
Qwen2.5-Coder SFT	26.46	23.78	**45.06**
Deepseek-Coder Base	32.62	28.35	23.68
Deepseek-Coder SFT	**33.35**	**29.27**	**32.01**

Table 7: Pass@10 results on Python benchmarks. All models are evaluated at temperature 0.2 and top-p 0.95

Model	HumanEval	HumanEval+	InstructHumanEval
Qwen2.5-Coder Base	**67.68**	**56.70**	57.93
Qwen2.5-Coder SFT	34.15	31.10	**60.98**
Deepseek-Coder Base	44.51	39.02	38.46
Deepseek-Coder SFT	**46.34**	**42.68**	**43.90**

On MultiPL-E cross-language benchmarks, both models show slight performance degradation after SFT, with Qwen2.5-Coder experiencing more pronounced drops. The dramatic performance collapse of Qwen2.5-Coder on HumanEval benchmarks, combined with high training accuracy (78.69%) and relatively low validation loss (0.7445), suggests overfitting to the CodeAlpaca training distribution. The model appears to have specialized too heavily on the instruction-response format of the training data at the expense of generalizing to different evaluation formats.

3.C Error Analysis

3.C.1 In-Context Learning

The error analysis for ICL experiments reveals several key patterns. First, the model occasionally generates markdown-formatted code blocks despite explicit instructions to avoid them, requiring post-processing to extract executable code. Second, some generated functions fail edge cases, particularly when handling empty inputs or boundary conditions. Third, the model sometimes produces syntactically correct but logically incorrect code, especially for problems requiring complex algorithmic reasoning or multiple-step solutions.

Table 8: Pass@1 results on MultiPL-E. All models are evaluated at temperature 0.2 and top-p 0.95

Model	C++	Java	JavaScript
Qwen2.5-Coder Base	50.87	45.78	53.17
Qwen2.5-Coder SFT	48.82	44.53	50.19
Deepseek-Coder Base	29.50	29.57	28.88
Deepseek-Coder SFT	27.64	28.07	30.99

Table 9: Pass@10 results on MultiPL-E. All models are evaluated at temperature 0.2 and top-p 0.95

Model	C++	Java	JavaScript
Qwen2.5-Coder Base	73.29	68.94	69.57
Qwen2.5-Coder SFT	68.32	63.35	64.60
Deepseek-Coder Base	39.75	41.61	44.72
Deepseek-Coder SFT	36.02	39.75	40.99

The prompt selection phase on MBPP showed that increasing the number of few-shot examples from 0 to 5 consistently improved performance, suggesting that the model benefits from seeing multiple examples of the expected output format.

Language-specific analysis on MultiPL-E shows that the model performs best on Python (as expected, given the few-shot examples are Python code), with varying performance on C++, Java, and JavaScript. The performance gap likely reflects the model's training distribution, which may be more heavily weighted toward Python code.

3.C.2 Finetuning

Both fine-tuned models struggle with problems requiring complex state tracking. For `separate_paren_groups`, which requires separating balanced parenthesis groups, solutions consistently implement incorrect logic. DeepSeek-SFT attempts naive splitting, fundamentally misunderstanding nested structure. Qwen-SFT implements stack-based approaches but with backwards logic—appending groups when the stack is non-empty before popping, rather than tracking when depth returns to zero. These errors persist across all 10 generation samples per model, indicating systematic misunderstanding rather than random variation.

The `intersperse` problem reveals a systematic off-by-one error in DeepSeek-SFT. All incorrect solutions follow the pattern:

```
for i in range(len(numbers)):
    if i % 2 == 0:
        result.append(numbers[i])
        result.append(delimeter)
```

This produces `[1, delim, 2, 3, delim, 4]` instead of `[1, delim, 2, delim, 3, delim, 4]`. Qwen-SFT correctly solves this in most samples using `result = [numbers[0]]` followed by iteratively appending delimiter then element, suggesting stronger algorithmic reasoning on certain problem types. Overall, while both models show improved instruction-following after fine-tuning, they continue to struggle with fundamental algorithmic concepts involving iterative state management and boundary conditions.

3.D Best System for Deployment

For scaled deployment, we will choose Qwen2.5-Coder-3B-Instruct with the 5-shot ICL configuration. This model achieves the best performance across all benchmarks (88.41% pass@1 on HumanEval, 85.71% on HumanEval+) while requiring no training infrastructure, which means we can deploy immediately with just the optimized prompt. At 3B parameters, it offers an ideal balance between accuracy and inference cost for high-volume production use. The fine-tuned variants suffer from overfitting (Qwen-SFT drops to 26.46% pass@1). The ICL approach also demonstrates strong cross-language generalization and avoids the catastrophic failures seen in the 7B model, making it the most reliable choice for real-world deployment.

4 Pick your own Experiment

4.A Investigating Model Size Effects on In-Context Learning

To understand how model scale affects in-context learning performance for code generation, we evaluate larger variants of the Qwen2.5-Coder model family using the same 5-shot prompt configuration that achieved optimal performance on MBPP. This investigation addresses the research question: *How does model size influence ICL performance on code generation benchmarks?*

4.B Experimental Setup

We evaluate Qwen2.5-Coder-7B-Instruct (7B parameters) using the identical experimental setup as the 3B model:

- **Prompt Configuration**: 5-shot prompt (best from MBPP selection)
- **Generation Parameters**: temperature=0.2, top-p=0.95, max_new_tokens=2048
- **Evaluation Metrics**: pass@1 and pass@10 on HumanEval and HumanEval+
- **Sampling**: 10 samples per problem for pass@10 calculation

4.C Results

The evaluation results for Qwen2.5-Coder-7B-Instruct are presented in Table 10 and 11. Surprisingly, the larger 7B model demonstrates lower performance compared to the 3B model on both HumanEval and HumanEval+ benchmarks.

Table 10: ICL pass@1 comparison across model sizes (5-shot prompt, temperature 0.2, top-p 0.95).

Model	HumanEval	HumanEval+
Qwen2.5-Coder-3B-Instruct	**88.41**	**85.71**
Qwen2.5-Coder-7B-Instruct	17.07	17.68

Table 11: ICL pass@10 comparison across model sizes (same settings as Table 10).

Model	HumanEval	HumanEval+
Qwen2.5-Coder-3B-Instruct	**97.56**	**96.43**
Qwen2.5-Coder-7B-Instruct	19.51	21.95

4.D Analysis and Discussion

The dramatic performance difference between the 3B and 7B models (88.41% vs 17.07% pass@1 on HumanEval) is unexpected and suggests several possible explanations:

1. **Model Architecture Differences**: The 7B model may have different architectural choices or training procedures that affect its in-context learning capabilities, despite being from the same model family.
2. **Prompt Sensitivity**: Larger models may be more sensitive to prompt formatting. The 5-shot prompt optimized on the 3B model may not be optimal for the 7B variant, requiring separate prompt engineering.
3. **Evaluation Artifacts**: There may be subtle differences in how the models handle the prompt format or code extraction that affect the evaluation results.

The 7B model's pass@1 score of 17.07% on HumanEval is significantly lower than both the 3B ICL model (88.41%) and the 3B base model (52.01%). This suggests that the performance degradation is not simply due to the lack of fine-tuning, but rather indicates a fundamental difference in how the larger model processes the 5-shot prompt. This finding highlights the importance of model-specific prompt optimization and suggests that scaling model size does not necessarily guarantee improved ICL performance without careful prompt engineering. Future work should investigate optimal prompt configurations for different model sizes and explore whether the performance gap can be bridged through prompt adaptation.

4.E Implications and Future Directions

The unexpected performance gap between model sizes has several important implications:

- **Prompt Engineering is Model-Specific**: The optimal prompt configuration for one model size may not transfer to another, even within the same model family. This suggests that prompt optimization should be performed separately for each model variant.
- **Scaling Does Not Guarantee Improvement**: Simply increasing model size does not automatically improve ICL performance. The relationship between model scale and ICL effectiveness appears to be more nuanced and may depend on factors such as training data, architecture choices, and prompt format.
- **Need for Systematic Evaluation**: This finding underscores the importance of systematically evaluating different model sizes when deploying ICL-based systems, rather than assuming that larger models will perform better.

Future research directions include: (1) conducting prompt selection experiments specifically for the 7B model to determine if a different prompt configuration can improve performance, (2) investigating the architectural and training differences between the 3B and 7B models that might explain the performance gap, and (3) exploring whether the performance pattern holds for even larger models (e.g., 15B parameters).

Team member contributions

Please briefly describe the primary contributions of each teammate.

Arthur Chien

- Led Experiment 3 by designing and executing the Supervised Fine-Tuning pipeline for both Qwen2.5-Coder and Deepseek-Coder base models, including hyperparameter sweeps across different LoRA configurations (r=16, 32, 64) and extended training runs.
- Conducted comprehensive error analysis of fine-tuned models, identifying systematic failures in algorithmic reasoning tasks and diagnosing overfitting patterns in the Qwen2.5-Coder SFT variant.

Yaxuan Mao

- Led Experiment 3 by designing, executing, and analyzing the Qwen2.5-Coder Instruct ICL pipeline (HumanEval, HumanEval+, and MultiPL-E), including the latest baseline re-runs and report revisions.
- Ran Experiment 4 (Pick-your-own study) comparing Qwen2.5-Coder model sizes, executing the 7B evaluations, and writing the associated analysis/discussion sections.

Gilbert Wu

- Led Experiment 3 by designing, executing, and analyzing the DeepSeek-Code-1.3B Instruct ICL pipeline (HumanEval, HumanEval+, and MultiPL-E), and analyzing the prompt performance.
- Conducted error analysis of ICL model, showing that longer few-shot prefixes distracted the small model, explaining the baseline prompt's superior performance..

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*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.