COSE474-2024F: Final Project
Prompt Tuning for Rust Code Generation with Llama Model

Korinek Lukas

1. Abstract

In recent years, the demand for automated code generation tools has grown, driven by fast advances in natural language processing, which gained significant popularity following the introduction of ChatGPT. The performance of these tools can vary depending on the prompt effectiveness. This project explores how prompt tuning, a technique to optimize prompts for better performance, can enhance the accuracy of generated Rust code. The project code is available at https://github.com/lkorinek/20242R0136COSE47402.

2. Introduction

2.1. Motivation

Code generation using AI models has developed rapidly in recent years. Many models focus on generating code for popular programming languages, mainly Python. An example is CodeLlama(AI, 2024), which supports many languages but not Rust. With Rust's popularity, the demand for models specializing in Rust will increase.

Rust's memory safety and high performance make it a popular programming language, but its complexity makes it challenging for AI-based code generation. Current models excel in generating code for languages like Python but struggle with Rust. This motivates the exploration of prompt tuning to improve Rust code generation capabilities.

2.2. Problem Definition

This project evaluates the accuracy of Rust code generation using the Llama model. The project compares performance under three conditions: (1) standard prompts, (2) model prompt-tuning, and (3) one-shot tuned prompts where a simple example is provided. Prompt tuning was done using PEFT, an open-source framework for prompt-based tuning. The models were evaluated using a dataset with Rust example codes.

2.3. Contribution

The contributions are summarized as follows:

- Evaluated Rust code generation using the Llama model with standard prompts, prompt tuning, and one-shot tuning.
- Developed a prompt-tuned Llama model and an automated Rust code testing framework.
- Found prompt tuning effective but was limited by GPU constraints (RAM size $<$ 15GB).
- Noted one-shot tuning outperformed prompt tuning in a limited GPU setup.

3. Methods

3.1. Related Work

Many studies are improving the programming language generation models, but only a few focus explicitly on Rust. One such study presents a Rust code-generating model known as RustGen (Wu et al., 2023). This model uses several techniques, including prompt engineering, to improve the quality of the results. It presents the specific prompt template that inspired this project's prompt design.

Furthermore, OctoCoder (Muennighoff et al., 2024), also mentioned in the same paper as HumanEvalPack (dataset used in this project), is another Rust code-generating model. When evaluated on the HumanEvalPack dataset, OctoCoder with 16B parameters performed the worst (among the six languages) on Rust, achieving an accuracy of just 23.4 %. This performance further highlights opportunities for improvement in this language.

3.2. State-of-the-art Models

Current state-of-the-art LLM models for code generation include Codex (Mark Chen, 2021), developed by OpenAI, which powers GitHub Copilot. While Codex is highly effective for generating code, it is not open-source; thus, experiments can be conducted primarily through GitHub Copilot.

Another well-known model is Llama (Touvron et al., 2023), developed by Meta AI, which is open-source and suitable for research. Llama is available in different versions, types, such as CodeLlama, and sizes ranging from 1B to 70B parameters. This project uses Llama due to its availability and good performance.

4. Experiments

4.1. Experimental Setup

The experiments were conducted in Google Colab(Google Research, 2023) using the free T4 GPU with 15GBs of RAM, which is quite limiting but sufficient for a limited testing environment. It is enough RAM to load the 1B-parameter model.

The HumanEvalPack (Muennighoff et al., 2024) was used for code generation and testing. The dataset is an extension of the HumanEval (Chen et al., 2021) dataset developed by OpenAI for their Codex (Mark Chen, 2021) model. The HumanEvalPack extends this dataset to include five additional languages, including Rust. HumanEvalPack contains 164 Rust code samples, each paired with prompts, canonical solutions, and test cases to evaluate the correctness of the generated code.

The generated code was evaluated based on its accuracy in passing test cases. The tests were conducted using an automated tool[1] developed in Rust.

4.2. Results

4.2.1. Llama 3.2 1B

A Llama 3.2 model with 1B parameters generated the Rust code. A sample prompt used to guide the model in generating Rust functions can be seen in Figure 1.

> You are an expert Rust programmer. {**instruction**}
> In your response, use RUST_BEGIN and RUST_END to delimit the rust function.
> RUST_BEGIN

Figure 1. Example Prompt for Rust Code Generation

It illustrates the use of 'RUST BEGIN' and 'RUST END' keywords to delimit the generated code. The code consists of a function that performs the task described in *instruction*. The prompt was inspired by the prompt used in (Wu et al., 2023). The prompt was tweaked and tested, but the original prompt in the paper had the best results.

The main issue observed was the model's inability to understand the expected output format consistently. The RUST_BEGIN keyword was included in the prompt to indicate the start of the result, but the generated outputs varied significantly. To address this, multiple results were generated until one included the RUST_END keyword and was not empty. After the first attempt, the model's temperature was set to 0.5 to increase output diversity. Including multiple attempts slightly improved the number of correct results. If ten attempts were reached without finding the RUST_END keyword, the final attempt was used regardless of the format. Despite these adjustments, almost one-half of the outputs still failed to meet the proper format. Prompt tuning was then used as a potential solution to this issue.

[1]It is recommended to run this tool within a virtual environment, as executing unknown code can potentially cause unintended side effects to your system.

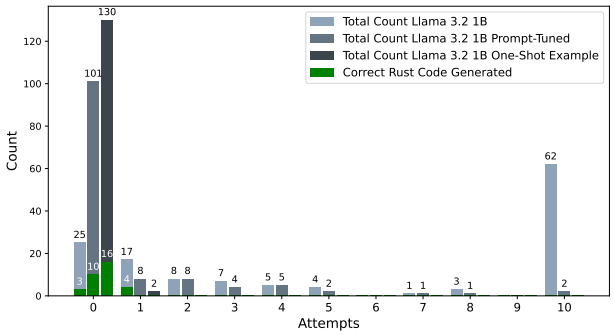


Figure 2. **Distribution of Code Generation Attempts**: The number of attempts required by the model to produce correctly formatted code across different Llama setups. Correct solutions that passed the tests are highlighted in green, while correctly formatted codes are gray. The last attempt was saved regardless of the format.

4.2.2. Llama 3.2 1B Prompt-Tuned

Prompt tuning has emerged as a powerful technique, allowing models to adjust to specific tasks without modifying model weights. This makes it interesting compared to fine-tuning the whole model as it requires fewer training weights, making tuning the model possible in a limited GPU environment.

Prompt-tuning does not change the model weights, making it less computationally expensive than fine-tuning. The Llama 3.2 1B has 1,235,853,312 parameters, and the prompt-tuning has 38,912 trainable parameters, just 0.0031 % of the total parameters. However, changing this small number of parameters can improve the model's output.

For training, the dataset was split into training and testing in a 2:8 ratio, and training was further split into training and validation with the same ratio.

The prompt-tuned model provided consistently well-formatted results (defining the function and ending with RUST_END) compared to the basic prompts.

All the canonical solutions import the same libraries at the beginning, a pattern the prompt-tuned model successfully learned, as seen in the Figure 3. The model-generated code included these libraries in most results.

However, the prompt-tuned model suffered from over-fitting during training, which resulted in bad generalization. Even though the code was in the correct format, the code did not pass the test cases. The problem might have been with insufficient training data and also because of the metric that

Prompt-tuned Model Prompt

Write a Rust function ‘min_sub_array_sum(nums: Vec<i64>) -> i64‘ to solve the following problem:
Given an array of integers nums, find the minimum sum of any non-empty sub-array of nums.
RUST_BEGIN

```
use std::{slice::Iter, ...};
use rand::Rng;
use ...

fn min_sub_array_sum(nums: Vec<i64>)->i64 {
    let mut res = i64::MAX;
    for i in 0..nums.len() {
        let mut sum = 0;
        for j in i..nums.len() {
            sum += nums[j];
            if sum < res {
                res = sum;
            }
        }
    }
    return res;
}
RUST_END
```

Figure 3. Example of Correct Code Generated Using Prompt-Tuned Llama Model

is used during training to compare the results with canonical solutions. Another evaluation metric, such as the CodeBleu(Ren & Ma, 2020) metric, could be used to measure the quality of generated code and improve performance.

The prompt-tuned model achieved an accuracy of 7.58 %, which is a better result than the basic model. The advantage of the fine-tuned model is that we can provide a partial prompt to generate the result. The model only requires the task instruction followed by the RUST_BEGIN keyword to generate the result.

4.2.3. Llama 3.2 1B One-Shot Tuned

Another approach to solving the wrong format generation problem was using one-shot tuning by providing an example of what the result should look like before asking to generate a new one. This example was a simple function returning the sum of two numbers. This simple approach resulted in the best results, generating the highest number of correct formatted codes in the first attempt and achieving the highest number of codes that passed the test cases.

4.3. Summary

Table 1 shows the accuracy achieved by different code generation models, including the CodeLlama model, run on the free Colab TPU v2-8 (which took many hours). One-shot tuning achieved the highest accuracy (12.12 %) compared to prompt tuning (7.58 %) and standard prompts (2.27 %). With more attempts, the standard prompt model's accuracy improved from 2.27 % to 5.30 %. GPU limitations affected prompt-tuning efficiency. It was limited to generating 240 tokens during learning, and the batch size was limited to 8, which might have affected the performance. Also, the size of the learnable tokens (initial prompt) was limited to fit the model in RAM and lower the learning time.

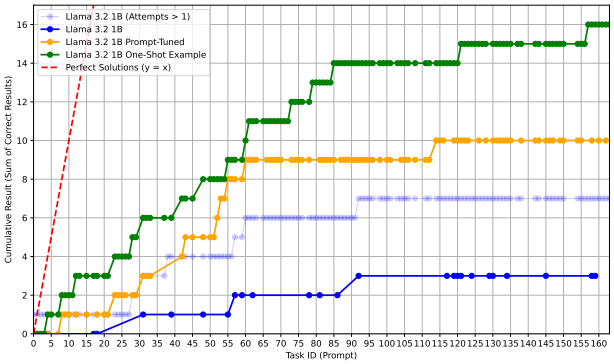


Figure 4. **Cumulative Results of Code Generation**: Cumulative correct results achieved by the different Llama setups for Rust code generation, with a perfect solution line as the baseline.

Table 1. Accuracy of Code Generation Models

Model	Correct Results [%]	
	Attempts=1	Attempts>1
Llama 1B	2.27	5.30
Llama 1B Pr.-Tuned	7.58	7.58
Llama 1B One-Shot	12.12	12.12
CodeLlama 13B	28.79	–

5. Future Directions

This project can be further extended by:

- Utilizing larger models such as Llama 13B, 34B, or CodeLlama, specialized in code generation.
- Expanding the Rust dataset with more diverse functions to improve evaluation.
- Improving prompt-tuning using more powerful GPUs to support larger prompts, code generation length, batch sizes, and longer training sessions.
- Changing the loss function for prompt-tuning to a metric better suited for text-to-code tasks, such as CodeBLEU, could improve performance.

All the code is published online together with the results, dataset, and prompt-tuned model and can be used to replicate the results. There are two notebooks in the repository, one for the general Llama model and one for prompt-tuning.

References

AI, M. Code llama: Large language model for coding. https://ai.meta.com/blog/code-llama-large-language-model-coding/, 2024. Accessed: 2024-12-08.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., and et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Google Research. Google colaboratory. https://colab.research.google.com/, 2023. Last accessed: December 10, 2024.

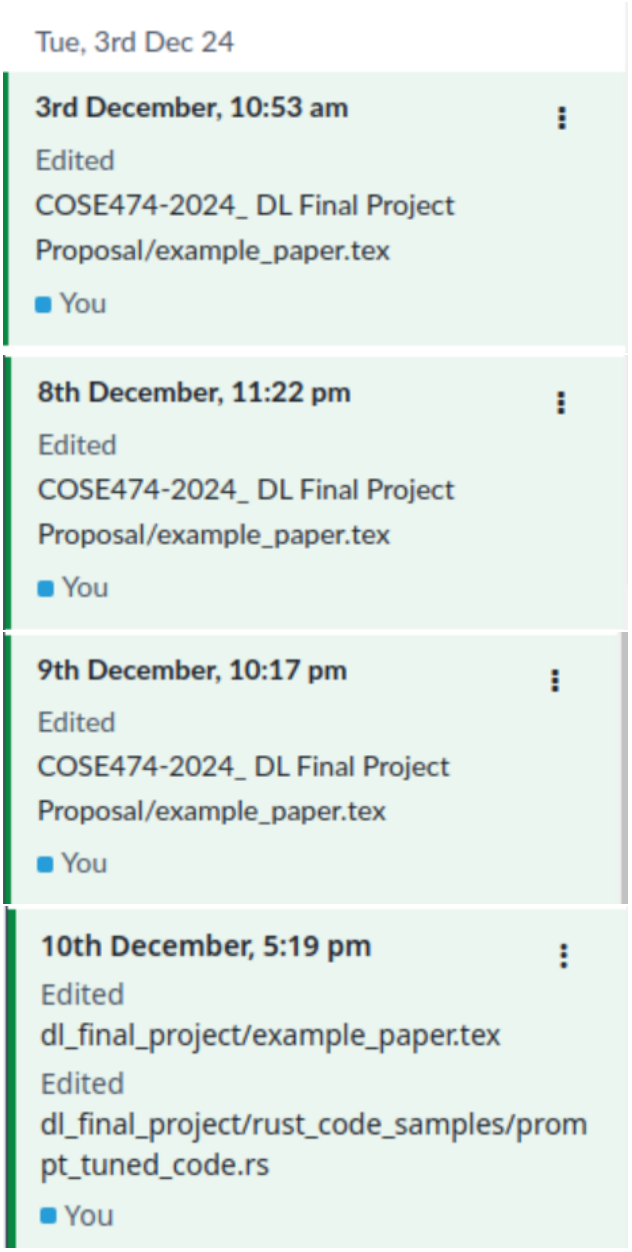
Mark Chen, Jerry Tworek, H. J. Q. Y. H. P. d. O. P. J. K. e. a. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models, 2024. URL https://arxiv.org/abs/2308.07124.

Ren, S., G. D. L. S. Z. L. L. S. T. D. S. N. Z. M. B. A. and Ma, S. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

Touvron, H., Lavril, T., Izacard, G., and Rodriguez, A. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Wu, X., Cheriere, N., Zhang, C., and Narayanan, D. Rustgen: An augmentation approach for generating compilable rust code with large language models. In *ICML 2023 Workshop on Deployable Generative AI*, 2023. URL https://openreview.net/forum?id=y9A0vJ5vuM.



Timeline and Progress Description:

- **3rd December 2024:** Created the project report and defined the general structure.
- **8th December 2024:** Wrote the introduction and included result visualization plots.
- **9th December 2024:** Wrote the Experiments section, summarizing all the model setups.
- **10th December 2024:** Added code and prompt samples, completed formatting, and made final adjustments.

Figure 5. Revision History on Overleaf

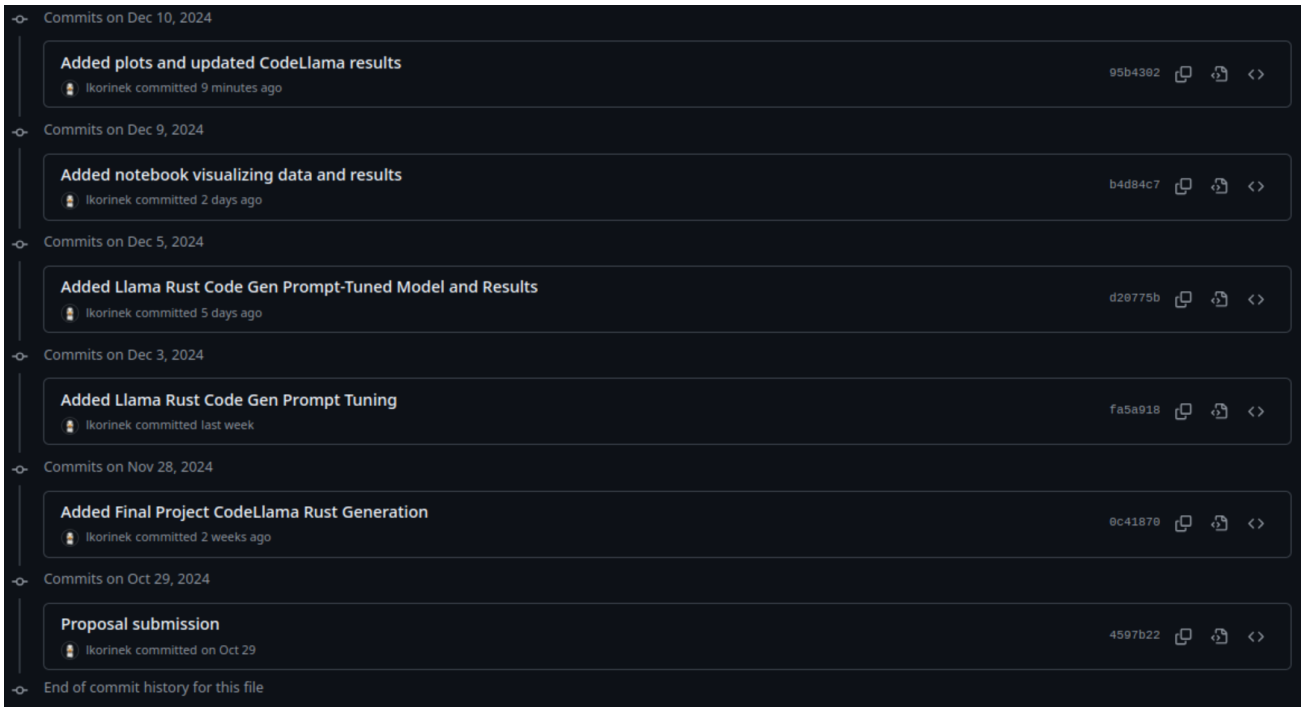


Figure 6. GitHub Commit History