This is the Title

Your Name

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Your Name*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, Large Language Models (LLMs) have emerged as powerful tools with applications spanning various domains. Notably, their effectiveness in code intelligence tasks has been remarkable, leveraging the structured and rule-based nature of programming languages. There are three areas that can be considered the main drivers for the success of LLMs: (i) Model architectures: Transformers are capable yet simple models as they rely on the self-attention mechanism. (ii) Learning Algorithms: The Pre-training and Fine-tuning learning algorithms made models more fixable for handling a number of specialised tasks (iii) Scale: better performance can be expected by increasing the model parameters, the data, or the computation according to the scaling laws. These discoveries have resulted in excellent performance in program synthesis and understanding challenges and have been adapted to both commercial [4] and open-source [5] tools.

Accessibility to code language models is one of the important goals of AI development. The Democratization of AI is an important factor in the progress of the field, and AI companies such as Meta [1] and Stability-AI [2] are showing their commitment to it by publicly sharing their models and releasing their weights. However, this may not be enough, as the barrier to tuning and using LLM is higher than access to their weights. For the practitioner, the choices of model architecture and learning algorithm are not obvious, and exploring these options is costly due to the high computation costs. In order to achieve the benefits of democratization of AI use and development [9], these issues need to be resolved.

1.2 Problem Statement

The aim of this project is to address the accessibility gap and further efforts towards democratising the use and development of code LLMs. To make LLMs more usable, we need to first address the high computational cost of large language models. Using smaller models also comes with the cost of being limited to a small number of popular programming languages. Hence, We will train small code language models to perform code completion tasks [6] in a variety of programming languages. Additionally, to address the ambiguity in fine-tuning LLMs, we will provide empirical evidence on the effects different choices have on the training process in terms of time, cost, and performance.

1.3 Contribution

In this work, we fine-tuned a mono-lingual code LLM trained on Python to four new programming languages: Java, Rust, Ruby, and Swift. We used LoRa Parameter Efficient Fine-Tuning to train the four models, and they are all shared publicly along with their respective training datasets and evaluation results. We also did extensive ablation studies, trade-off computations, and error analysis to provide empirical evidence on the effects different choices in the training process have on the efficacy of the process. Finally, we discussed the current state of AI democratisation, the associated risks and benefits, current gaps, and possible ways forward.

1.4 Thesis Structure

This paper will be divided into the following five parts:

- The project's motivation, goals, and outcomes are primarily introduced in the first chapter, which also serves as an introduction. This chapter provides a general overview of the project, highlights its innovation and significance, and discusses its ultimate objective.
- The background primarily introduces prior research and work in the project's pertinent fields. In order for readers to become familiar with the project and better comprehend the works produced by this project.

- The methodology chapter provides a detailed introduction to each step of the project's implementation as well as the project's overall structure. Additionally, it goes into detail about how pipelines are built for data preprocessing, various methods of fine-tuning, and evaluation.
- The result and evaluation are covered in chapter four. The experiments conducted for the project will be covered in this chapter. This will include each experiment's goal, setup, and outcomes, as well as an evaluation of these outcomes.
- The fifth chapter is a summary, which will summarise the contributions of the projects in terms of AI democratization and discuss the benefits, risks, and ways forward for more accessible code language models.

Chapter 2

Background

This will be mostly copied from IPP. Review of:

1. Code Language Models
2. PEFT methods.
3. Code Completion task SOTA

Chapter 3

Methodology

3.1 Dataset

Our training datasets are primarily drawn from TheStack Corpus [**?**]. TheStack is an open-source code dataset with more than 3TB of GitHub data covering 48 different programming languages. We only use a small portion of this dataset because our experiments only consider a small subset of programming languages and because optimising smaller language models only needs a small amount of data.

We had to perform several preprocessing steps in order to prepare our datasets. The first step is choosing our target programming languages. Ruby, Swift, Rust, and Java were ultimately chosen. We selected these languages because they represent the distribution of programming languages on GitHub. As seen in figure 3.1, the most popular language is Java; Ruby and Rust are medium-resource languages, and Swift is a low-resource language. Additionally, Ruby was chosen because it shares syntax similarities with Python and is a dynamic language, whereas Java, Rust, and Swift are all statically typed. The next step is to select a sample of files from TheStack corpus and divide them into train, validation, and test splits. In order to train LLMs effectively, we sampled one million files from each of the four languages. To filter the files, we adhered to best practises [11] and filtered out files with the following characteristics: 1) an average line length of over 100 characters; 2) a maximum line length of over 1000 characters; and 3) a ratio of alphabet less than 25%. Then, we split the remaining files into train, validation, and test splits using ratios of 0.9, 0.05, and 0.05. Figure 3.2 shows the statistics for the final datasets.

The tokenizers used were the same as those for the baseline models [7]. These Byte Pair Encoding (BPE) [10] tokenizers add special tab and white space tokens to

the same GPT-2 vocabulary [8]. The training sequences are produced by joining text from training data to fill the 2048 context length, or just 1024 tokens in the case of full fine-tuning.

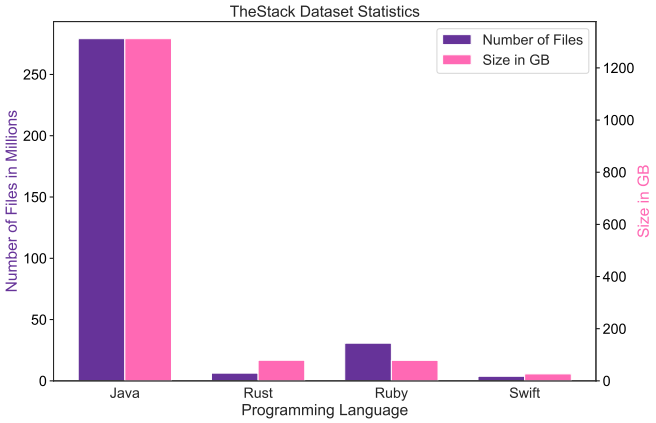

Figure 3.1: TheStack Corpus Statistics for Target Programming languages

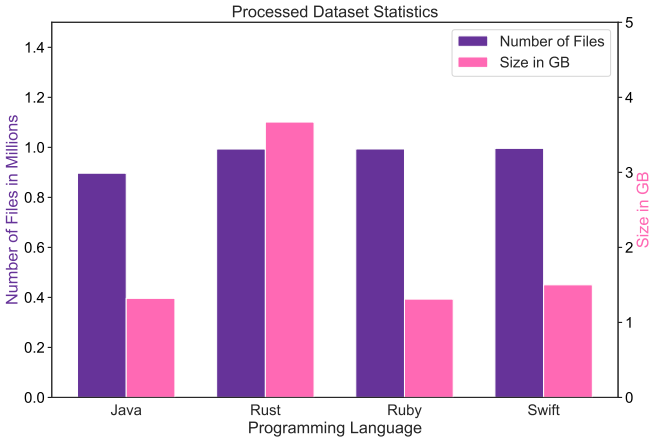

Figure 3.2: Processed Datasets Statistics

3.2 Fine-tuning

3.3 Evaluation

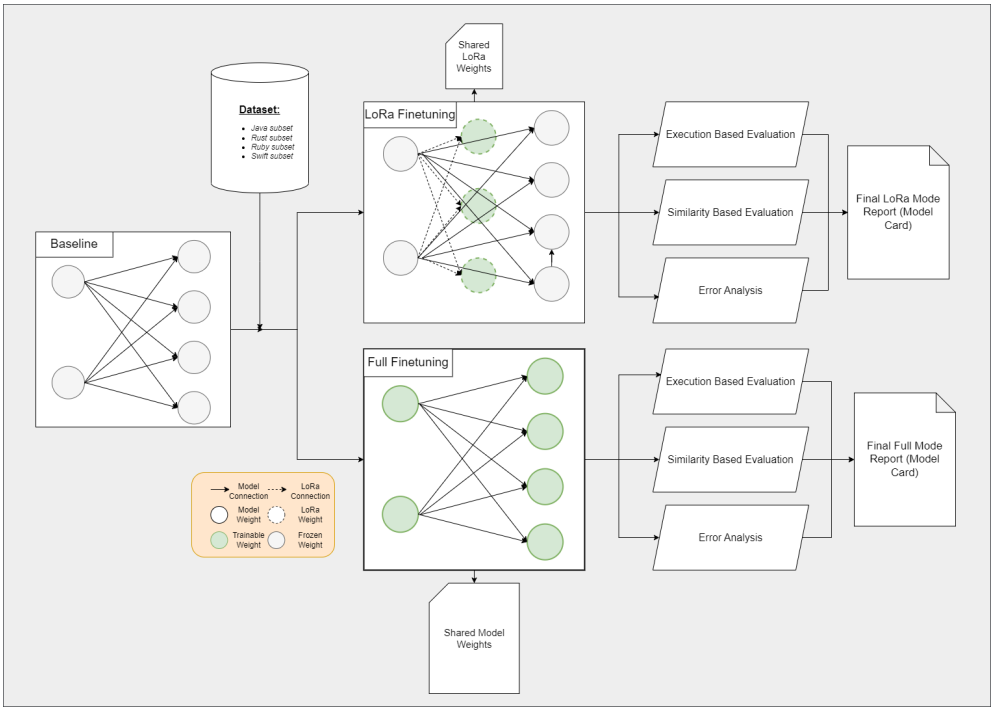

Figure 3.3: Project Diagram

Chapter 4

Results and Evaluation

In this chapter, we will cover all the experiments in the project, discussing the objectives of each experiment , the experimental setup, and then presenting and analysing the outcomes. The chapter will be divided into four sections , where each section builds on the previous results and presents experiments with similar objectives. In the first section, we compare Full fine tuning methods vs. efficient fine-tuning methods and try to find out the difference in performance we can achieve in the Java programming language. Next, we perform ablation studies on the parameters of LoRa fine-tuning to understand the contribution of each factor to the process. In section three we study the trade-off between memory, time, and performance when fine-tuning code language models in an environment with limited resources. Finally, we use the knowledge obtained from previous results to train models on programming languages other than Java, and in the final section, we perform error analysis on the generations of these models.

4.1 Full Fine-Tuning vs. Efficient Fine-Tuning

4.1.1 Objective and Setup

We suggest that a possible solution to the high inference and training costs of Large Language Models is to use smaller, more compact architectures. However, this also comes at a cost, as smaller models are normally trained in only one programming language to achieve good performance with their limited parameters. In fact, work by [3] has empirically shown that training sufficiently large language models benefits the model's overall performance as knowledge can be transferred between different languages. On the other hand, training smaller models in an increasing number of

programming languages hinders learning in all of them.

However, we hypothesise that it is possible to fine-tune small mono-lingul language models to perform satisfactorily well in other programming languages. We argue that one can use full fine-tuning and utilise the trained model weights as initializations with knowledge about general programming knowledge and natural language. Full fine-tuning, however, suffers from the phenomenon known as catastrophic forgetting, which may cause the tuned model to forget the syntax of its original programming language. Using parameter-efficient techniques, we can mitigate the forgetting issue and reduce the computational cost. These gains come from only a small number of low rank matrices, which, on the other hand, may limit the learning ability of the model.

We used the task of code completion, particularly the one outlined by Humaneval, to test our hypothesis. We used the Codegen-350M-mono and Codegen-350M-multi auto-regressive code language models developed and released by Codeforce as baselines. Both models share the same number of parameters (350 million) and structural elements. The difference is that the mono version is trained on Python only (BigPython), and the multi version is trained on a mix of four languages (Python, Java, Javascript, and C) from (TheStack. Then, we fine-tuned the mono model on the Java subset of the preprocessed dataset discussed earlier. We used Full and LoRa fine-tuning using the same masked prediction loss function, Adam optimizer, a 0.0005 Learning rate with cosine decay, and an effective batch size of 8. The maximum sequence length in full fine tuning was 1024, and in LoRa it was 2048 due to memory limitations. A detailed description of these parameters is in the ablation studies section, and the full list of parameters is in the appendix.

4.1.2 Analysis

The results in Figure 1 show that it is possible to fine-tune monolingual models for new programming languages in a reasonable amount of time with limited computational resources. As shown in figure 2 , the training lasted for 10,000 steps, or the equivalent of 80,000 different files, each with 1024 and 2048 Java tokens for Full and LoRa finetuning, respectively. The training was done on a Google Colab T4 15GB GPU and lasted for around 12 hours (details in the appendix).

The Loss curves in figure 1 show that the LoRa fine-tuned model evaluation loss flattened earlier than the full fine-tuned one. But the latter had a lower evaluation loss overall. In Figure 2, we see the differences more clearly The pass@10 rate of the full

fine-tuning is 50% higher than LoRa and, in fact, almost the same as the results from the multilingual baseline, which saw a much larger number of Java tokens (detailed breakdown of token count in the appendix).

We can also see from Figure 2 the effects of catastrophic forgetting, as the Python score for full-finetuning is 0.0 while the LoRa score is 3.75, which is also a significant drop from the 22.36 in the monolingul baseline. Additionally, we see that even though the monolingual baseline was only trained on Python, it still passes some of the Java problems. This is probably due to code spill (Java code inside Python files), and it has been reported in other works (CodeGeeX ciation).

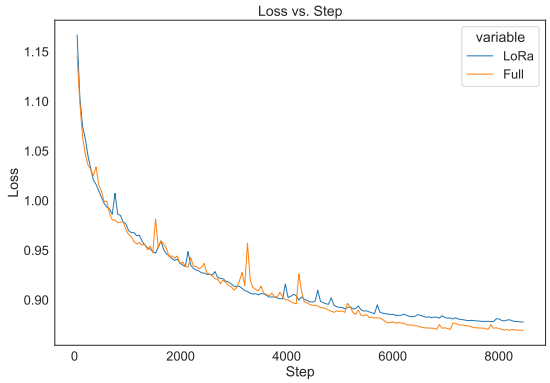

Figure 4.1: Comparison of Loss Curves for LoRa and Full Fine Tuning

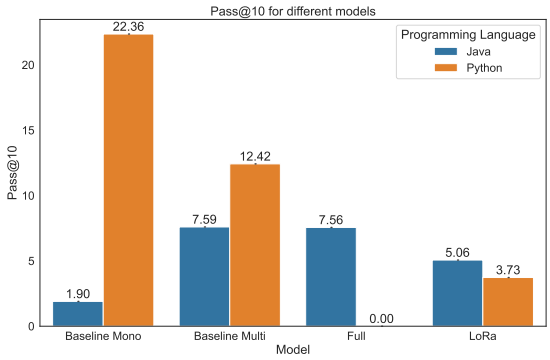

Figure 4.2: Comparison of Pass@10 for LoRa and Full Fine Tuning vs the Baselines

4.2 Ablation Studies

4.3 Trade-off Analysis

Category	Theme	Example
Highest Performing Problems	Single operations	def sum_to_n(n: int) ->int: """sum_to_n is a function that sums numbers from 1 to n. 3 """
	Common type questions	def fib(n: int) ->int: """Return n-th Fibonacci number. """
Lowest Performing Problems	Problems with multiple subproblems	def separate_paren_groups(paren_string: str) ->List[str]: """ Input to this function is a string containing multiple groups of nested parentheses. Your goal is to separate those group into separate strings and return the list of those. Separate groups are balanced (each open brace is properly closed) and not nested within each other Ignore any spaces in the input string. """ (Sub-Problems: Remove Spaces, Identify Parentheses Groups, Balanced Parentheses Checking, Extract Substrings, Combine Results)
	Solving a more common version of the problem	def intersperse(numbers: List[int], delimeter: int) ->List[int]: """ Insert a number 'delimeter' between every two consecutive elements of input list 'numbers' """ return list(map(lambda x: x if x % 2 == 0 else x + delimeter, numbers)) (the model try to insert the delimiter between even elements instead of every two consecutive elements)
	Miscellaneous errors	def parse_music(music_string: str) ->List[int]: """ Input to this function is a string representing musical notes in a special ASCII format. Your task is to parse this string and return list of integers corresponding to how many beats does each not last. Here is a legend: 'o' - whole note, lasts four beats 'o—' - half note, lasts two beats '.—' - quater note, lasts one beat parse_music('o o— .— o— o— .— .— .— .— o o') >>>[4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4] """ (Specialized Problem)

Table 4.1: Different Error Types with examples of each

4.4 Fine-tuning on Low resource languages

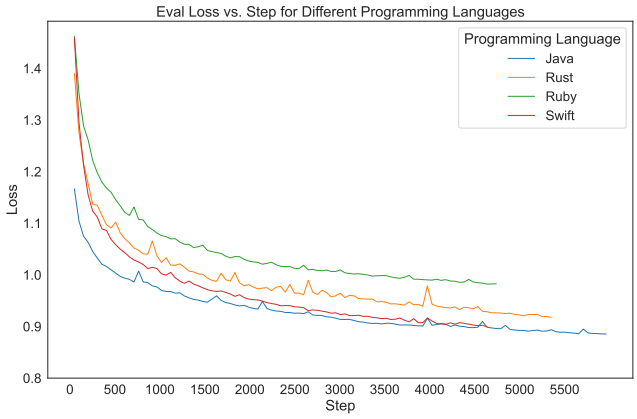

Figure 4.3: Evaluation Loss For Target Programming Languages

4.5 Error Analysis

Chapter 5

Conclusions

Bibliography

[1] Meta AI is sharing OPT-175b, the first 175-billion-parameter language model to be made available to the broader AI research community.

[2] Scale virtual events.

[3] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models, 2023.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

[5] 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 De-

haene, 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, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023.

[6] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[7] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[8] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[9] Elizabeth Seger, Aviv Ovadya, Ben Garfinkel, Divya Siddarth, and Allan Dafoe. Democratising ai: Multiple meanings, goals, and methods, 2023.

[10] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[11] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration.