T3Gen Model: Text-to-Testcase generation using CodeLlama with LoRA

Anannya Patra, Jnanasri Manikya Meghana Kalavakuntla,
Pavana Sai Sree Chalamarla, Sai Varnitha Reddy, Sankalp Sethi

Abstract

In the realm of Test-Driven Development (TDD), the generation of test cases from user requirements is a crucial yet challenging task. To address this, we propose T3Gen, a novel approach that utilizes Code Language Models (CodeLLMs) and Low-Rank Adaptation (LoRA) of Large Language Models for automated text-to-test case generation. This method aims to streamline the test case creation process, reduce manual effort, and enhance the overall efficiency and reliability of TDD practices. Our approach leverages the natural language understanding capabilities of the CodeLlama-7B model, enhanced with LoRA techniques to adapt the model to the specific nuances of TDD test case generation. We evaluate our method using the Python-based MBPP dataset, comparing it against traditional CodeLLM baselines. Our evaluation metrics, including Pass@1 accuracy, assess the effectiveness of our model in generating accurate and reliable test cases for TDD. Preliminary results show a promising increase in accuracy by up to 10%, particularly when employing both text and code prompts in the training process. This integration presents a significant advancement in the automation of TDD test case generation.

1. Introduction

Unit testing is a foundational practice in software engineering, where individual sections of code are independently assessed. This process allows developers to detect and rectify errors early in the development cycle, enhancing their understanding of how various code segments integrate and function together as a coherent system.

Fine-tuning LLMs can be resource-intensive. To address this challenge, we implement Low-Rank Adaptation (LoRA), a method that efficiently fine-tunes Transformer models. LoRA achieves this by fixing pre-trained weights and integrating low-rank decomposition matrices into each layer, enabling precise adjustments without imposing extensive computational demands. We tuned LoRA with parameters like lora_r, lora_alpha, and lora_dropout to optimize performance, utilizing an A100 GPU with 40 GB of memory to handle the substantial computational resources required.

Our approach, termed T3Gen, innovatively employs the LoRA technique to fine-tune variants of the CodeLlama model on a specially curated Python dataset containing basic code examples. This setup includes an efficient prompting design that facilitates the direct generation of test cases from textual inputs. Once the CodeLlama model is fine-tuned, it can generate the required test cases in the inference phase when presented with a new task and a corresponding prompt, without requiring the actual implementation of the code. This method diverges from traditional approaches by enabling the direct generation of test cases from Python tasks, bypassing the need for code execution.

Our evaluations, based on the Pass@1 metric of the MBPP dataset, demonstrate a significant increase in accuracy—up to 10%—when both text and code prompts are included. This indicates the model's ability to effectively generate diverse and comprehensive test scenarios.

2. Related Work

Creating unit test cases to attain comprehensive code coverage is both time-intensive and prone to errors. Diverse methodologies for automated test case generation have been introduced to aid developers in the formulation of test cases. These methodologies employ a range of techniques such as randomization[6], evolutionary algorithms, and deep learning[3]. Nevertheless, prevailing approaches for test case generation typically necessitate the provision of source code as input for test case derivation (Code→Test), rendering them incompatible with Test-Driven Development (TDD). Code generation models like Codex and CodeGen[10] demonstrated the capability to generate code solely from text, making them suitable for understanding scenarios to generate test cases.

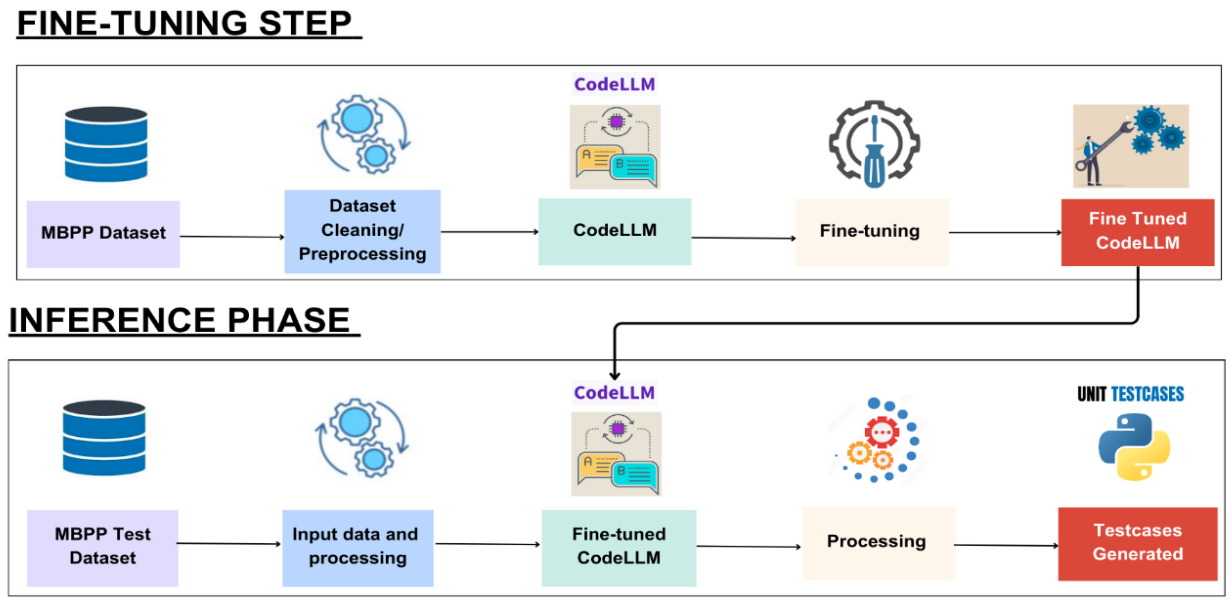

Fig 1: T3Gen Architecture

Studies had been conducted to generate unit test cases in strongly typed languages like Java[5]. However, they had not undertaken any fine-tuning to showcase the generation of more concrete test cases.

A method of text-to-test case generation introduced previously, utilized GPT-3.5 by fine-tuning on curated datasets[1]. They primarily focused on method descriptions, code comments, and test cases in Java. For our experiment with Python, we adapted this approach. Specifically, we aimed to incorporate the task that a function was intended to perform, rather than solely describing its current functionality.

A parameter efficient fine-tuning method like LoRA[6] performs on-par or better than fine- tuning in model quality on GPT-3, despite having fewer trainable parameters, a higher training throughput, and no additional inference latency. Techniques like QLoRA[9] fine tuning on a small high-quality dataset leads to state-of-the-art results. We employed these techniques for fine tuning our data.

3. Methodology

3.1. Problem Statement

The primary objective of this project phase was to develop a model capable of generating comprehensive test cases. To achieve this, we meticulously curated the dataset to align with the specific requirements of our models. We employed three distinct approaches to organize the MBPP dataset effectively-

- Natural language prompts: The Large Language Model (LLM) generates test cases by interpreting prompts written in natural human language.
- Code-based prompts: The LLM generates test cases based on code inputs.
- Integrated Text and Code Prompts: Enabled the LLM to generate test cases based on textual content and corresponding code segments, supplemented by instructional components.

3.2. Effective Prompt Design

```
<s>[INST] <<SYS>>
System prompt
<</SYS>>

User prompt [/INST] Model answer </s>
```

Fig 2: Prompt

To ensure that our three prompts are accurately interpreted by CodeLlama-7b-Instruct, we have structured our input to include specific instructions, example test cases, and unique tags that the model recognizes. Each prompt begins with an instruction block marked by [INST] tags, detailing the task of writing three tests for a given programming problem. Directly above each test assertion, there is a comment specifying the test case number. This number begins at 1 and

increments with each subsequent test. The tests themselves are enclosed within [TESTS] and [/TESTS] tags to clearly delineate them from the rest of the text. This structured approach, complete with example tests, aims to provide a clear, context-rich environment to facilitate accurate test case generation by the model.

3.3. QLoRA

QLoRA (Quantization-aware Learned Optimized Residual vector Arithmetic) is a technique used in machine learning to reduce the computational cost and memory footprint of large language models (LLMs) while maintaining their performance. The main idea behind QLoRA is to quantize (reduce the precision of) the weights and activations of the LLM during inference, effectively compressing the model's size and reducing its computational requirements.

For our project, QLoRA is employed to generate test cases by modifying the model's attention layers to better capture causal relationships and nuances in language. We tuned it with additional parameters like lora_r, lora_alpha, and lora_dropout.

3.4. CodeLlama

We fine-tuned a CodeLLM model, CodeLlama-7b-Instruct, using our Python dataset. CodeLlama-7B-Instruct is a large language model specifically designed for code-related tasks such as code generation, explanation, and translation. The major reasons why we chose this model is-

a) CodeLlama-7b-Instruct contains around 7 billion parameters, making it a relatively large language model capable of handling complex code-related tasks.
b) The model has been fine-tuned on a large corpus of code and natural language instructions, allowing it to understand and follow natural language prompts related to coding tasks.
c) CodeLlama-7B-Instruct can generate code snippets or complete programs in various programming languages based on natural language prompts or specifications.
d) It supports Few Shot Learning, i.e. it can adapt to new tasks with just a few examples, making it versatile and capable of handling a wide range of coding-related scenarios.

3.5. Evaluation

The evaluation metric we used to evaluate our results was- Pass@1Metric. The Pass@1 Metric is useful for evaluating the performance of question answering systems, as well as other information retrieval tasks where the goal is to provide the most relevant result as the top-ranked output. As mentioned earlier, our LLM generates three test cases for each prompt, and if at least one of the generated test cases is correct, then we mark it as a correct output.

4. Experimental Setup

4.1. Dataset

We plan to utilize the MBPP dataset, which consists of approximately 1,000 Python programming challenges tailored for entry-level programmers. Each entry includes a task ID, a textual task description, a code solution, and three automated test cases. The dataset is divided into four splits: train, evaluation, test, and prompt, with the prompt split being used for few-shot prompting rather than training.

Data split	Percentage
Train	40%
Evaluation	50%
Test	9%
Prompt	1%

Fig 3: Dataset

4.2. Fine-tuning

Train Split used for LoRA[4] is 40% and we have used QLoRA[4] to do 4 bit quantization. A 4-bit precision quantization approach is employed, utilizing the "nf4" scheme for efficient model size reduction and computational speedup. Nested quantization is not used to avoid complexity and accuracy loss. Computations are performed using 16-bit floating-point precision (float16), balancing efficiency and precision.

For the project, NVIDIA A100 and V100 GPUs were employed, backed by a system configuration comprising 83.5GB of system RAM, 40GB of GPU RAM, and 78.2GB of disk space, providing a robust computing environment using Google Colab Pro.

4.3. Training arguments and hyperparameters

The training configuration for a machine learning model includes various hyperparameters and arguments tailored to optimize performance. The training is set to run for 5 epochs. For both training and evaluation, the per-device batch size is 4. Gradient accumulation is updated every step, maintaining a delicate balance between computing resource usage and update frequency. The gradients are clipped at a maximum norm value of 0.3 to prevent exploding gradients, a common issue in training deep networks. A modest learning rate of 0.00002 is selected, coupled with a weight decay of 0.01, to regularize and prevent overfitting.

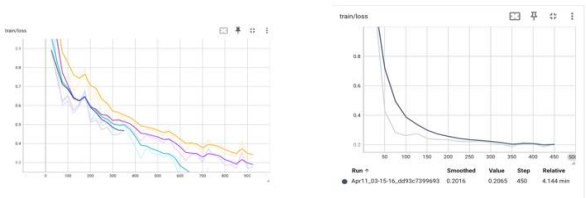

Fig 4: Training Loss Metrics: Evaluating Model Performance Across Hyperparameter Configurations

5. Results and Analysis

Pass@1 metric of MBPP dataset for CodeLllama-7b-Instruct is used as a baseline to compare your results.

This integration allowed for a more comprehensive understanding of the task at hand, leading to enhanced test case generation. The generated test cases exhibited a commendable ability to comprehend model parameters, thereby showcasing the model's effectiveness in capturing the essence of the code's functionality.

Model	Validation @pass1	Test @pass1
CodeLlama Instruct	44.4	44.4
Text input	51.1	50.5
Code input	52.2	52.7
Text and code input	55.8	54.5

Fig 5: Pass@1 metric on validation and test data for various training prompt settings

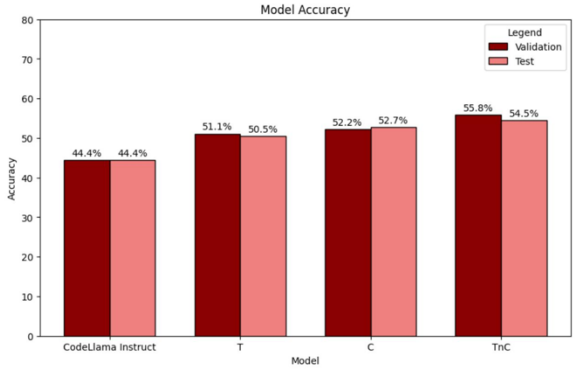

Fig 6: Graphical representation of Pass@1 metric on validation and test data for various training prompt settings

However, despite these strengths, there were instances where the generated test cases displayed limited variability. In some scenarios, the three test cases produced did not deviate significantly from each other, suggesting a potential area for improvement in diversifying the generated test suite. Furthermore, a notable limitation of the model was its occasional failure to cover edge cases, which are crucial for challenging the code's limitations and ensuring its robustness across various scenarios

6. Conclusion and Future Improvements

We developed T3Gen, a model designed to generate unit test cases from natural language input by fine-tuning the CodeLlama-7B-Instruct language model on our Python dataset, consisting of problem statements and corresponding code. This fine-tuning process involved training on Python code and natural language prompts, both individually and combined, resulting in up to 10% improved accuracy. Going forward, several enhancements can augment T3Gen's functionality and performance: expanding input format handling for increased versatility, supporting programming languages beyond Python, accessing more GPU resources for larger batch sizes and improved efficiency, and refining prompt length and token constraint management to mitigate incomplete test cases.

By addressing these areas, T3Gen can evolve into a more robust and versatile automated test case generation tool, catering to diverse software development needs.

7. Individual contribution

Task	Team member
Refining **MBPP** dataset and performing experimentation	Sankalp Sethi, Sai Varnitha Reddy
Model selection (**CodeLLama- instruct**) and prompt engineering.	Pavana Sai Sree Chalamarla, Jnanasri Manikya Meghana Kalavakuntla
Model training and fine-tuning with **QLora.**	Anannya Patra, Pavana Sai Sree Chalamarla
Experimentation with **hyperparameters** and fine-tuning the hyperparameters.	Sai Varnitha Reddy, Jnanasri Manikya Meghana Kalavakuntla,
Creating and executing the **evaluation framework** and refining the results.	Anannya Patra, Sankalp Sethi

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, Chetan Arora, and Aldeida Aleti. 2024. *Enhancing Large Language Models for Text-to-Testcase.* https://doi.org/10.48550/arXiv.2402.11910

[2] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. *LEVER: Learning to Verify Language-to-Code Generation with Execution.* https://doi.org/10.48550/arXiv.2302.08468

[3] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. *Large Language Models for Software Engineering: Survey and Open Problems.* https://doi.org/10.48550/arXiv.2310.03533

[4] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Lu Wang, Weizhu Chen. 2021. *LoRA: Low-Rank Adaptation of Large Language Models.* https://doi.org/10.48550/arXiv.2106.09685

[5] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. *An Empirical Study of Using Large Language Models for Unit Test Generation.* https://doi.org/10.48550/arXiv.2305.00418

[6] Pacheco, C., Ernst, M.D., 2007. *Randoop: feedback-directed random testing for java, in: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion,* pp. 815–816.

[7] Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S.K., and Sundaresan, N. 2020. *Unit test case generation with transformers and focal context.* arXiv preprint arXiv:2009.05617

[8] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, and Q., 2023. *Software testing with large language model: Survey, landscape, and vision.* arXiv preprint arXiv:2307.07221 .

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer. 2023. *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv:2305.14314v1

[10] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.* arXiv:2203.13474v5