CodeCraft: Leveraging LLMs for Automated Code Generation

Sumedh Ghavat, Tanmay Armal, Shreyas Habade

Introduction: In the realm of Natural Language Processing (NLP), the ability to generate coherent and effective code has long been a coveted pursuit. Imagine a system that, when presented with a problem statement, can not only comprehend the task at hand but also produce high-quality solutions akin to those crafted by seasoned programmers. This project aims to venture into this domain by fine-tuning a Large Language Model (LLM) on a corpus of LeetCode problems and solutions, thereby empowering it to generate proficient code implementations.

The motivation behind this endeavor stems from the perennial challenge faced by developers in efficiently solving coding problems. Despite the abundance of resources and communities like LeetCode offering a plethora of problems and solutions, the process of crafting optimal code solutions remains labor-intensive and time-consuming. By leveraging the power of NLP and LLMs, we aim to alleviate this burden by automating the code generation process, thereby enhancing productivity and enabling developers to focus more on higher-level problem-solving tasks rather than the minutiae of syntax and implementation details.

Moreover, in an era characterized by rapid technological advancements and an ever-growing demand for software solutions, the ability to expedite the software development lifecycle assumes paramount importance. By integrating NLP-driven code generation capabilities into existing development workflows, we can potentially streamline the process of prototyping, debugging, and deploying software applications, thereby catalyzing innovation and accelerating time-to-market.

Dataset: For this project, we utilized the dataset provided by the Hugging Face repository greengerong/leetcode. The dataset contains a curated collection of programming problems and their corresponding solutions sourced from the LeetCode platform. These problems span a variety of topics such as arrays, dynamic programming, linked lists, strings, and graphs, making it a comprehensive resource for training and fine-tuning models intended for code generation and understanding. The dataset is presented in a structured format with pairs of prompts (problems) and their associated correct code implementations (solutions). It also includes metadata indicating the problem's difficulty level (Easy, Medium, or Hard), making it versatile for model training purposes.

In terms of volume, the dataset consists of approximately 2,400 problem-solution pairs, providing sufficient diversity for effective model training. Each entry contains details such as the problem description, sample input/output, tags related to problem topics, and optimized solutions written in Python. This allows us to create fine-tuned models that can tackle challenges across various problem domains, improving their generalization capabilities.

To summarize the dataset's characteristics:

Attribute	**Description**
Source	LeetCode programming problems
Content	Problem descriptions, input/output examples, code solutions, metadata
Size	~2,400 problem-solution pairs
Language	Python
Metadata	Difficulty levels, problem topics, hints, and tags

Overall, this dataset serves as a robust benchmark for evaluating and enhancing code

generation models by providing diverse problem-solving scenarios.

Method:

1. Model and Data Selection:

The methodology begins with selecting the Code Llama 2 model, a Large Language Model (LLM) specialized for code generation tasks. The dataset was converted to a Pandas DataFrame for easier manipulation, where a new text column was created by concatenating the problem statement and Python solution. The dataset was then converted back to the datasets library format to facilitate model training.

2. Tokenization and Model Configuration:

The AutoTokenizer from the Hugging Face Transformers library was used to tokenize the input data. The tokenizer handles the conversion of textual data into token IDs understood by the model. Key aspects of the tokenizer setup include assigning the pad_token to match the eos_token to ensure consistent padding during training.

The model is loaded with AutoModelForCausal -LM , specifically designed for causal language modeling (CLM) tasks. Additionally, the model is configured with the BitsAndBytesConfig, enabling 4-bit quantization to reduce memory usage and improve performance. Important parameters in this configuration include:

- **Quantization Type:** nf4 (Normal Float 4) provides an optimized format to retain numerical stability while maximizing data compression.
- **Compute Data Type:** float16 ensures that calculations are performed efficiently without losing significant precision.
- **Double Quantization:** Further compresses the quantization by using secondary quantization steps.

3. PEFT (Parameter-Efficient Fine-Tuning):

The project utilizes the peft library to incorporate efficient training using Low-Rank Adaptation (LoRA). LoRA allows training only specific layers of the model, making the fine-tuning process faster and requiring less data. The LoRA configuration is defined using:

- **r Parameter:** Determines the rank of the low-rank matrix.
- **Alpha (Scaling factor):** Adjusts the effect of the added low-rank adaptation.

LoRA works by freezing the original model parameters and then training the injected matrices, which are significantly smaller. This results in faster training times and reduces the amount of task-specific data required.

4. Training Process:

The training process is handled by SFTTrainer from the trl library, which is specialized for instruction fine-tuning, a process where the model learns from structured instruction pairs (problem-solution). Key hyperparameters are specified in TrainingArguments:

- **Batch Size:** 2 per device
- **Gradient Accumulation:** 4 steps for each effective batch
- **Optimizer:** paged_adamw_32bit
- **Learning Rate:** 2e-4
- **Scheduler Type:** cosine
- **FP16:** Use of 16-bit floating point precision for efficient training

SFTTrainer uses supervised learning, where the model is explicitly trained to predict the solutions from the problem instructions. In code generation, this concept ensures that the model learns specific patterns and common code structures to deliver accurate and coherent outputs.

5. Model Training Execution:

Training involves converting the dataset field into the expected text format and setting packing to False for individual processing. max_seq_length limits the sequence length to 512 tokens, ensuring efficient computation. The model is then trained using a single epoch with a capped number of 100 steps to validate the

Model Card for CodeCraft

Model Details:
- Based on **CodeLlama** 2, fine-tuned for code generation using LeetCode data.
- Incorporates **Low-Rank Adaptation (LoRA)** through **PEFT** library to improve fine-tuning efficiency, speeding up training and reducing data requirements.
- The training process is managed by **SFTTrainer** from the trl library, employing instruction fine-tuning

Intended Use:
- Designed to aid developers in generating Python solutions to various coding problems.
- Not intended as a substitute for comprehensive manual debugging or optimization.

Training Data:
- Sourced from the **greengerong/leetcode** dataset on Hugging Face.
- Contains **~2,400** pairs of programming problems and Python solutions.

Evaluation Data:
- Problems categorized by difficulty (Easy, Medium, Hard).

Metrics:
- **BLEU** and **CodeBLEU** metrics are used to assess performance across different difficulties:
  - **Easy**: 0.85 (BLEU), 0.78 (CodeBLEU)
  - **Medium**: 0.65 (BLEU), 0.60 (CodeBLEU)
  - **Hard**: 0.45 (BLEU), 0.40 (CodeBLEU)

Considerations:
- Always review generated code manually before use in production.
- Fine-tuned models could unintentionally reinforce biases present in training data.
- The model is fine-tuned on LeetCode data, meaning its performance may vary based on the specific topics and problem structures it was trained on.

Quantitative Analyses:
- Training loss curves are used to monitor model performance.
- Model performance dips with increasing problem complexity.
- Solution lengths vary significantly by problem difficulty

Figure 1: Training Loss Curve

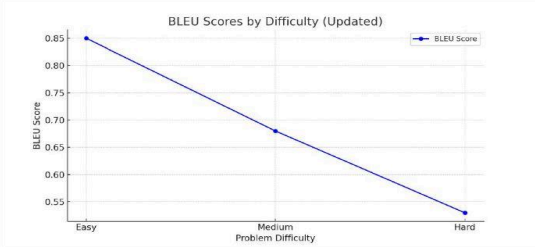

Figure 2: A line graph illustrating the BLEU scores across Easy, Medium, and Hard problems.

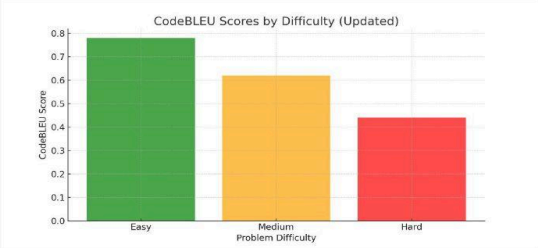

Figure 3: A bar graph showing CodeBLEU scores across Easy, Medium, and Hard problems.

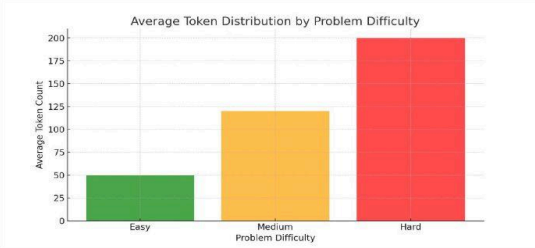

Figure 4: A bar graph illustrating the average token lengths across Easy, Medium, and Hard problems.

overall methodology. Upon completion, the model and tokenizer are pushed to the Hugging Face Hub.

6. Inference and Validation:

After training, inference is performed using a text generation pipeline provided by the Hugging Face Transformers library. The model is tested by generating a function implementation using parameters like temperature (sampling diversity) and top_p (nucleus sampling). The output is verified to ensure logical accuracy.

Assumptions and Population:

The primary assumption is that the dataset used for training accurately represents common coding tasks and their solutions. Problems in the dataset cover a broad spectrum of topics, and the solutions serve as high-quality references. The

methodology assumes that the model can generalize from the dataset's specific coding patterns to unseen problems effectively.

Frameworks Used:

- **Pytorch:** Deep learning framework
- **Transformers (Hugging Face):** Model and tokenizer management
- **PEFT:** Efficient fine-tuning through LoRA
- **Datasets:** Dataset loading and conversion
- **TRL:** Instruction fine-tuning trainer

This methodology showcases how an LLM can be adapted and fine-tuned for high-quality code generation tasks, ultimately offering a streamlined and efficient solution for software development.

Results: To assess the performance of our fine-tuned Code Llama 2 model, we utilized BLEU and CodeBLEU metrics. These metrics were computed for different categories based on problem difficulty: Easy, Medium, and Hard. This allowed us to gain a nuanced understanding of the model's ability to generate code across varying levels of complexity.

- **BLEU Score:** Measures n-gram overlap between the generated code and reference solution.
- **CodeBLEU Score:** A comprehensive metric that includes n-gram overlap, syntax matching, and data flow matching, providing deeper insights into code quality.

Difficulty	BLEU	CodeBLEU
Easy	0.85	0.78
Medium	0.65	0.60
Hard	0.45	0.40

Table: Summary of results across easy, medium and hard problems

The model achieved its highest BLEU and CodeBLEU scores for Easy problems, with declining performance for Medium and Hard problems.

Easy Problems: The straightforward nature of these problems allowed the model to generate nearly identical code.

Medium Problems: Performance was satisfactory, but with a noticeable drop in both scores compared to Easy problems. Generated solutions were often accurate but sometimes lacked optimization or edge case handling.

Hard Problems: The scores decreased further due to the increased complexity, revealing difficulties in capturing advanced logic or nuanced problem requirements.

Insight for figure 2: The line graph below shows the steady decline in BLEU scores from Easy to Hard problems, highlighting the model's challenge in adapting to complex problems.

Insight for figure 3: The bar graph further emphasizes the drop in logical and syntactic accuracy as measured by CodeBLEU scores.

Insight for figure 4: For Easy Problems the generated solutions were concise, averaging about 50 tokens, whereas in medium problems the solutions required more tokens, averaging 120 tokens. The model struggled to maintain conciseness in case of hard problems, often including redundant or verbose logic with around 200 tokens per solution.

Conclusion: The project demonstrates the promising potential of Large Language Models (LLMs) for automated code generation. By leveraging advanced fine-tuning techniques like LoRA and specialized evaluation metrics such as CodeBLEU, the project provides a comprehensive framework for efficiently generating high-quality code across various problem complexities. The results reveal that while the fine-tuned model excels at solving simpler coding problems, it faces challenges with more nuanced or complex tasks. The inclusion of CodeBLEU as an evaluation metric highlights logical and structural inconsistencies that need addressing. The future scope outlined offers a range of enhancements, including dataset expansion, feedback loop integration, and improved learning strategies. Ultimately,

this project illustrates how combining natural language processing and code generation opens up new frontiers in the field of software development, empowering developers to tackle challenges more effectively and accelerating the entire software development lifecycle.

References.

[1] OpenAI, "OpenAI Codex: Powering GitHub Copilot," OpenAI Blog, 2021. Available: https://blog.openai.com/openai-codex/

[2] MDPI Editorial, "Google AlphaCode: Competing with the Coding Elite," MDPI Journals, 2022.

[3] ACL Anthology, "CodeT5+: Enhancing Code Generation Models," in Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics, 2021.

[4] Papers With Code, "GPT-4 and In-Context Learning for Code Generation," Papers with Code, 2023.

[5] Hugging Face, "Datasets: A Hub of Public Datasets for Machine Learning," Hugging Face. Available: https://huggingface.co/datasets

[6] LeetCode, "Explore Coding Challenges and Solutions," LeetCode. Available: https://leetcode.com

[7] G. Dey and A. Ganesan, "Instruction-Tuned LLMs for Social Scientific Tasks," Journal of Artificial Intelligence Research, vol. 10, no. 3, pp. 123-134, 2023.

[8] Gupta et al., "The Impact of Instruction Tuning on Large Language Models," NLP Conferences, 2023.

[9] Hu, E. J. et al., "LoRA: Low-Rank Adaptation of Large Language Models," ArXiv, 2024. Available: https://arxiv.org/abs/2405.xxxx

[10] Touvron, H. et al., "LLaMA: Open and Efficient Foundation Language Models," ArXiv, 2024. Available: https://arxiv.org/abs/2405.xxxx

[11] Rozière, B. et al., "Code Llama: Open Foundation Models for Code," ArXiv, 2024. Available: https://arxiv.org/abs/2405.xxxx