FINE-TUNE LLM TO BE A PYTHON PROGRAMMER

Sirish Desai
School of Data Science
University of Virginia
Charlottesville, VA 22903
skd3nz@virginia.edu

Sivaranjani Kandasami
School of Data Science
University of Virginia
Charlottesville, VA 22903
nyc2xu@virginia.edu

Stavros Kontzias
School of Data Science
University of Virginia
Charlottesville, VA 22903
szk3tc@virginia.edu

December 3, 2023

ABSTRACT

In the dynamic landscape of software development, the creation of a supportive environment and the assurance of high-quality code delivery are crucial. This study delves into the innovative realm of leveraging synthetic datasets, generated by a larger model, to train a compact counterpart tailored for aiding Python programming tasks. The primary objective is to expedite activities such as code analysis, auto-completion, and debugging within the software development lifecycle.

By exploring the potential of synthetic datasets, derived from a larger model, we aim to optimize the training process for a smaller model specialized in Python programming support. This approach holds promise for enhancing developer productivity by streamlining code-related tasks and facilitating more efficient suggestions in the Python programming domain. The study underscores the significance of adopting novel methodologies to foster a conducive development environment, ultimately contributing to the advancement of software development practices.

1 Motivation

The global software development landscape is poised for remarkable growth, with a projected compound annual growth rate (CAGR) of 11.9% from 2023 to 2030. As the digital era advances, the number of software developers worldwide is anticipated to rise significantly, reaching 27.7 million in 2023 and 28.7 million in 2024.

Software development projects come with diverse price tags, spanning from $3,000 to $120,000. For custom eCommerce systems developed from scratch, costs start at $50,000 for simpler products, escalating to hundreds of thousands for intricate solutions. Similarly, Software as a Service (SaaS) development entails expenses ranging from $25,000 to $100,000, with more complex SaaS products demanding even higher investments. Furthermore, the industry norm dictates that maintaining software typically costs 15-20% of its original development expenditure.

In a field boasting over 700 programming languages, businesses and developers grapple with myriad choices. Python, a globally favored language, claims the second spot with an impressive 9 million active developers. However, making such choices is only one facet of the industry, as software development projects vary in duration, taking anywhere from one to nine months. On average, a custom software development project is estimated to span about 4.5 months.

Despite the burgeoning demand for software developers, there exists a significant shortage. By 2030, estimates from global consulting firm Korn Ferry suggest that the U.S. could face an annual revenue loss of $162 billion due to a scarcity of tech talent, a figure that balloons to $8.5 trillion globally. A staggering 53.8% of companies acknowledge that adapting to evolving client requirements poses the most significant challenge for developer teams.

Becoming a software engineer, while promising, comes at a price. Prospective students may incur tuition costs ranging from $37,000 to $91,000. Interestingly, the journey to becoming a proficient developer often involves online resources, as nearly 60% of software developers admit to learning how to code through such platforms. This underlines the fact that a relevant degree alone does not necessarily equip one to be job-ready in this dynamic and ever-evolving industry.

(a) Software Development Duration

(b) Learning Sources

Figure 1: Software Development - Statistics

This paper focuses on developing a language learning model tailored towards achieving proficiency in Python programming. We are addressing the historical issue where becoming a proficient Python programmer was often financially restrictive for many. Therefore, we're aiming to create a learning model capable of addressing queries related to the 2000 sub-topics commonly encountered in Python coding. The primary motivation is to provide an affordable means for professionals to learn Python and craft their own code effectively. By doing so, we envision enabling greater automation within the workplace.

The application we are developing fundamentally empowers users to integrate their own data, allowing the model to adapt and improve specifically based on the characteristics and intricacies of their individual datasets, rather than relying on generic or standardized data. This approach enables a more personalized and tailored experience, where the model's performance and accuracy are significantly enhanced by the inclusion of data that directly reflects the user's unique circumstances, preferences, or requirements.

By permitting users to bring their own data into the application, we enable a dynamic process where the model is fine-tuned to cater to the specific nuances and complexities inherent in the user-provided information. This approach fosters a more accurate and customized outcome, as the model's learning is driven by the diverse and personalized nature of the individual datasets. Consequently, the resulting model is better equipped to offer more relevant and precise insights, predictions, or solutions, optimizing the user's experience within the application.

The motivation behind this paper extends beyond mere cost reduction; we aspire to democratize Python proficiency. Through our innovative learning model, we aim to make Python programming education accessible to a broader audience, fostering a more inclusive environment for skill development. This inclusivity not only benefits aspiring programmers but also serves the interests of professionals seeking to enhance their Python capabilities.

The project envisions a future where the acquired Python proficiency leads to increased workplace automation. By empowering individuals to develop their own code solutions across the myriad sub-topics, we anticipate a positive impact on overall workplace efficiency. The model's ability to address diverse problem domains within Python contributes to a more versatile skill set, enabling professionals to navigate the evolving demands of the digital landscape.

2 Literature Survey

The dataset can be found here: `https://huggingface.co/datasets/jinaai/code_exercises`.

Originally created by Jina.ai, the collection of 120,000 Python exercises was devised in collaboration with ChatGPT. These exercises were specifically designed to enable the Language Learning Model (LLM) to grasp Python at a significantly reduced cost.

Potential Issues Enhancements are assessed using the Human Eval Benchmark, but it's crucial to note that progress on this benchmark might not necessarily reflect an all-encompassing improvement in programming performance.

Another important aspect to consider is the deficiency in diversity within model-generated synthetic data. This indicates that when presented with the same prompt repeatedly, the produced solutions tend to be highly similar to each other. This lack of variety poses a challenge as it could potentially lead to over fitting on specific types of problems and solutions. Given the vast size of the data set, removing such information would result in a non-negligible issue.

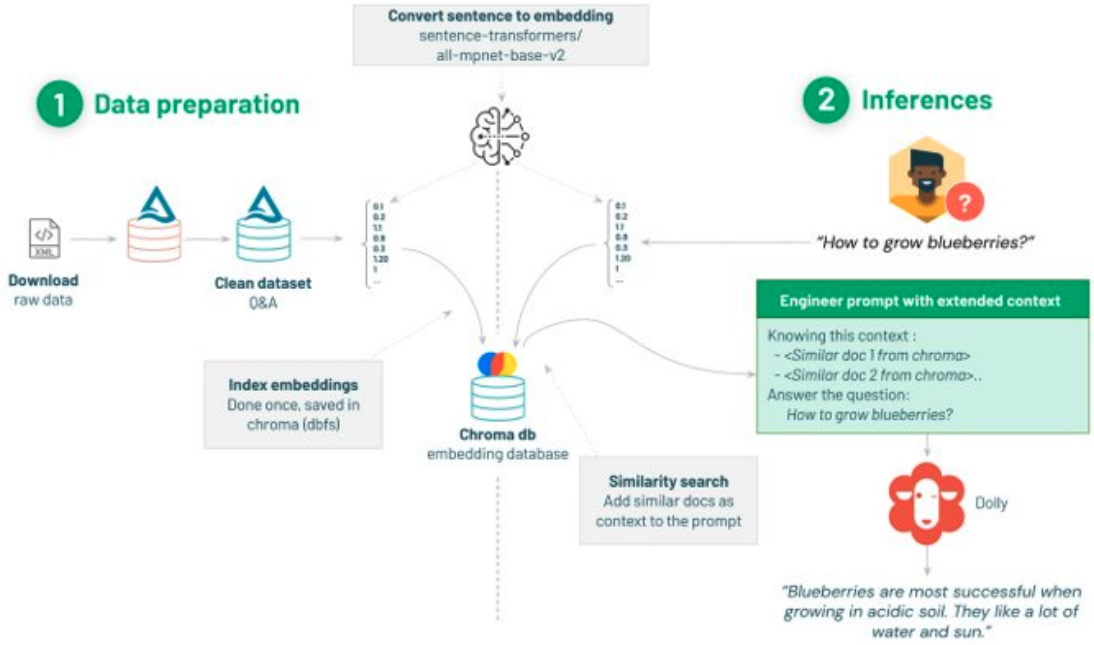

Figure 2: Retrieval Augmented Generation (RAG)

Evaluating Large Language Models Trained on Code by Mark Chen et al. research paper introduced Codex, a GPT-based language model, to the realm of Python coding. The evaluation of accuracy employed a novel method called HumanEval, specifically designed to gauge functional correctness using docstrings. HumanEval comprises a dataset encompassing 164 original programming problems, each equipped with unit tests, covering a wide array of topics. Within this evaluation set, Codex successfully resolves 28.8 percent of the problems. In contrast, GPT-3 fails to solve any, while GPT-J manages to solve 11.4 percent.

To enhance performance in tasks involving the creation of functions from docstrings, Codex underwent fine-tuning specifically focused on accurately implementing functions. A significant breakthrough emerged when, within just 100 samples, Codex was able to generate a correct function for 77 percent of the problems, marking a substantial stride in its capabilities.

The approach was to pre-training and fine-tuning, particularly considering that BERT is primarily pre-trained to predict missing words. An essential consideration involves determining how we select and maintain a consistent level of difficulty across tasks. Additionally, the process of fine-tuning Codex outputs significantly increased the accuracy of individual samples, a method that notably resembles real-world programming solutions. This aspect adds an intriguing layer to the overall process.

An interesting point for consideration is how Codex adapts during training when deliberately provided with incorrect code alongside negative reinforcement. This approach could potentially yield a distinct outcome, addressing the challenges users face when encountering issues in their code. Codex, by more readily discerning what is incorrect, might effectively address the existing issue.

An apparent limitation of Codex is its occasional failure to align perfectly with the user's intentions. For instance, if there's a subtle mistake in the input code, Codex might generate flawed code in response. Additionally, it's noteworthy that Codex sometimes generates code that reflects societal stereotypes related to gender, race, emotions, and class. Ensuring that the outputs during generalization do not exhibit bias from the data presents a significant challenge.

We adopted the Retrieval Augmented Generation (RAG) architecture(Figure 2), featuring a transformer-based LLM and efficient knowledge retrieval with BERT to function as Python Programmer.

The primary objective is to enhance the capabilities of the pre-trained model by fine-tuning it on the NVIDIA dataset. This process involves exposing the model to the python language patterns, context, and domain-specific information found in the question/answer pairs from NVIDIA.

The goal of the experiment is to develop a language model expected to generate human-like text, typically through the utilization of machine learning and natural language processing techniques(Figure 3).

2.1 Use Cases:

Efficient Code Generation: The RAG architecture facilitates the efficient generation of Python code by combining language understanding with a rich knowledge base, leading to contextually relevant and syntactically correct outputs.

Adaptability to Diverse Programming Tasks: Due to its retrieval-augmented nature, the tool can adapt to diverse programming tasks, ranging from basic syntax generation to more complex algorithmic implementations.

Enhanced Productivity for Developers: Developers can leverage this tool to expedite coding tasks, access programming knowledge on-the-fly, and enhance overall productivity.

Reduced Learning Curve for Novice Programmers: The contextual understanding and retrieval capabilities of the RAG architecture can lower the learning curve for novice programmers, providing them with valuable insights and guidance as they develop their coding skills.

3 Model and Preliminary Experiments

The selection of the model sentence-transformersall-mpnet-base-v2 model was based on the performance of the pre-trained models(Figure 4) on available training data (more than 1 billion training pairs) and are designed as general purpose models. Based on the performance data available, all-mpnet-base-v2 model provides the best quality, while all-MiniLM-L6-v2 is 5 times faster and still offers good quality.

The research involves the utilization of the pre-trained sentence-transformers/all-mpnet-base-v2 model with a dataset sourced Nvidia dataset. The idea is to extend LLM knowledge base to be able to solve answers that it wasn't specifically trained on.

This dataset comprises of over 7000 question and answer pairs with over 6000 unique combinations extracted from various NVIDIA websites, including content related to development kits and guides. The primary objective of the research is to refine the pre-trained model by fine-tuning it with dataset, enhancing its ability to understand and respond to questions.

The methodology incorporates the design of a functional system that reads data from a CSV file containing questions and answers. Users interact with the system by posing questions, and the system leverages the SentenceTransformer model to encode queries. The model then identifies the most similar questions in the dataset, providing responses based on semantic similarity. The system includes functionality to split text into chunks using various parsing methods such as paragraphs, sentences, or custom patterns.

One notable feature is the implementation of a mechanism to return the top 'n' responses to the user, where 'n' can be customized. Users are given the flexibility to choose 'n' from a range of 1 to 5, allowing them to receive a tailored number of relevant responses.

The response of this custom model was compared with that of gpt 3.5 and gpt-3.5-turbo. The goals of this experiment is to verify the semantic similarity and response time.

To enhance the conversational aspect of the system, it integrates with OpenAI's GPT models for generating conversational responses. A functional docstring is employed to guide the model in understanding the task requirements, aligning with a broader trend in code translation that emphasizes functional correctness. This approach mimics how human developers assess the accuracy of their code through functional correctness and unit tests in practical development scenarios.

As part of the ongoing development, the model is planned to undergo further training with a specific focus on python-related questions and dataset responses. Users will be given the opportunity to contribute their own data, enabling further tuning and customization of the model. This user-driven approach recognizes the importance of incorporating diverse perspectives and use cases in refining and expanding the capabilities of the system.

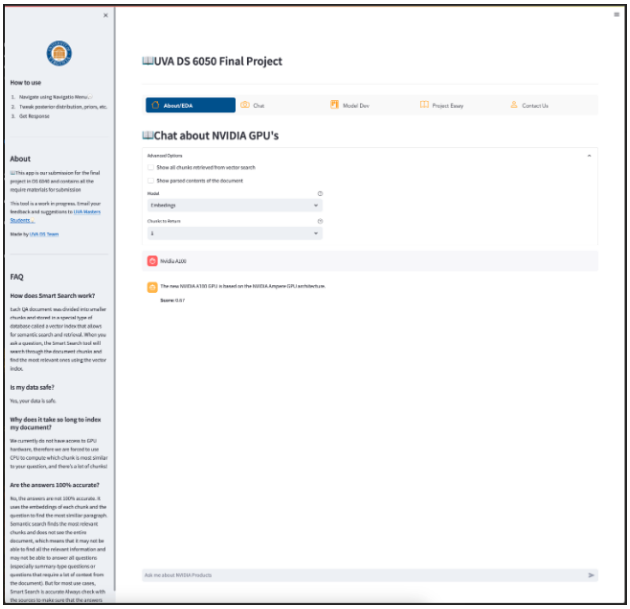

Figure 3: Mobile App UI

4 Mobile App

The trained model is deployed and made accessible to users through a mobile application(Figure 3). The mobile app is developed using Streamlit, a popular Python library for creating interactive web applications. The structure of the mobile app code is organized within the Streamlit framework, utilizing various Streamlit components to design a user-friendly interface.

4.1 Key Features

Streamlit Framework: The app is built on the Streamlit framework, which simplifies the process of creating web applications with Python. Streamlit is known for its ease of use and efficient development capabilities.

User Interface Components: Various Streamlit components are employed to construct the user interface. These components could include text inputs, buttons, sliders, or any other interactive elements that enhance the user experience.

Text Inputs for Interaction: Users can interact with the system through text inputs, allowing them to ask questions or provide prompts to the model. The input functionality is crucial for engaging users in a natural and conversational manner.

Chat-Like Format: The user interface is designed to resemble a chat format, displaying messages in a conversational style. This format enhances the user experience by presenting the interactions in a familiar and intuitive way.

User-System Interaction: The app facilitates user-system interaction, enabling users to input queries or prompts and receive responses in real-time. This interactive feature provides a dynamic and responsive user experience.

Message Display: The chat-like interface displays both user inputs and system responses. This not only allows users to see their queries but also provides transparency in showcasing how the system interprets and responds to their inputs.

Mobile Compatibility: As the app is developed with the intention of being used on mobile devices, it is optimized for mobile compatibility. This ensures that users can seamlessly access and interact with the application on their smartphones or tablets.

4.2 User Workflow:

Text Input: Users enter text-based queries or prompts through the designated input area.

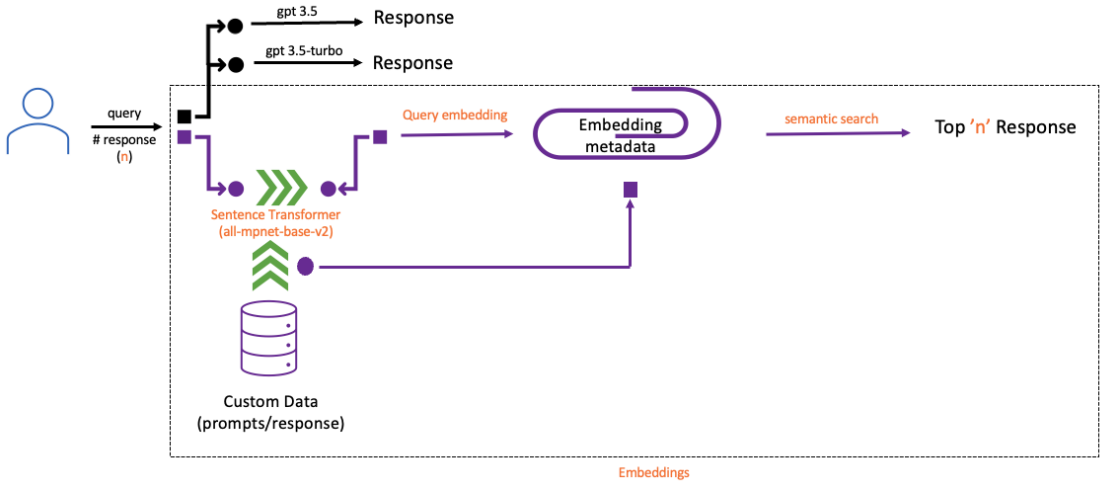

Figure 4: User Workflow

Model Processing: The app utilizes the trained model to process and understand the user input.

Real-Time Responses: The model generates responses based on the user input, and these responses are displayed in real-time in the chat-like interface.

Continuous Interaction: Users can engage in an ongoing conversation by providing additional inputs, creating a fluid and interactive user experience.

5 Code

References

5.1 Statistics:

[1] Pretrained Models

[2] Software Development Research

[3] Software Engineering Statistics

[4] Software Development Statistics

[5] Learning How to Code

5.2 Research Papers:

[6] Gunasekar et al Textbooks-Are-All-You-Need *arXiv:2306.11644*

[7] Mark Chen et al Evaluating Large Language Models Trained on Code *arXiv:2107.03374*

[8] Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Wei Jiang Mftcoder: Boosting Code Llms With Multitask Fine-tuning *arXiv:2311.02303v1*

[9] Jialing Pan1,Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, Sylvain Flamant Stelocoder: A Decoder-only Llm For Multilanguage To Python Code Translation *arXiv:2310.15539v1*

[10] Yaqing Wang, Jialin Wu, Tanmaya Dabral, Jiageng Zhang, Geoff Brown, Chun-Ta Lu, Frederick Liu, Yi Liang, Bo Pang, Michael Bendersky, Radu Soricut Non-intrusive Adaptation: Input-centric Parameter-efficient Fine-tuning for Versatile Multimodal Modeling *arXiv:2310.12100v1*

[11] Martin Weyssow,Xin Zhou , Kisub Kim, David Lo, Houari Sahraoui Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models *arXiv:2308.10462v1*

[12] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, Chun Zuo LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning *arXiv:2308.11148v2*

5.3 Ideas With Sample Code (offline model):

[13] Shawhin Talebi Fine-tuning Large Language Models (LLMs) | w/ Example Code

[14] AI Jason "okay, but I want GPT to perform 10x for my specific use case" - Here is how

[15] Sam Witteveen "Fine-tuning LLMs with PEFT and LoRA

[16] 1littlecoder "Fine-Tune Large LLMs with QLoRA

[17] Databricks "LLM2 Module 2 - Efficient Fine-Tuning | 2.3 PEFT and Soft Prompt

[18] DataTrek "LLAMA-2 Open-Source LLM: Custom Fine-tuning Made Easy on a Single-GPU Colab Instance | PEFT | LORA

[19] Databricks "LLM2 Module 2 - Efficient Fine-Tuning | 2.7 Notebook

[20] Abdul M "transformers meets bitsandbytes for democratzing Large Language Models (LLMs) through 4bit quantization

[21] Ali Abdin "tHow to Finetune LLMs with LoRA

5.4 Videos With Sample Code (using OpenAI API):

[22] Sam Witteveen "Fine Tuning GPT-3.5-Turbo - Comprehensive Guide with Code Walkthrough

[23] AI Jason "How to give GPT my business knowledge?" - Knowledge embedding 101

[24] Prompt Engineering ChatGPT Fine-Tuning: The Next Big Thing!

[25] Better w/AI How to Fine-Tune GPT 3.5-Turbo

5.5 Dataset:

[26] Nvidia Nvidia Documentation Question and Answer pairs

[27] Ashutosh Kumar Digital Marketing QnA for LLM finetuning.

[28] Prompt Engineering How to Create Custom Datasets To Train Llama-2