CodeCrafter - AI Chatbot for Code Generation and Debugging
Name - Atharv Hejib

Introduction

CodeCrafter is an AI-powered chatbot designed to assist developers with code generation, debugging, and documentation support. The primary goal of this project is to fine-tune a Large Language Model (LLM) on a curated dataset containing real-world code snippets and debugging examples. By leveraging a specialized model, CodeCrafter aims to improve code quality, streamline debugging, and enhance developer productivity.

Objectives:

- Generate high-quality, syntactically correct code snippets.
- Identify and fix errors in user-provided code.
- Offer explanations and documentation for various programming constructs.
- Integrate seamlessly with developer workflows (e.g., IDE plugins, web-based assistants).

Research & Model Selection

Selecting an appropriate foundational model was crucial for ensuring efficiency, accuracy, and scalability in code-related tasks. Several existing LLMs were evaluated based on factors such as performance in code generation, support for multiple programming languages, fine-tuning capabilities, and computational feasibility.
After careful evaluation, CodeLlama-7B-HF (MetaAI) was selected as the foundational model for CodeCrafter.

Advantages of CodeLlama-7B-HF:

1. Trained specifically for coding tasks – CodeLlama is optimized for multiple programming languages, ensuring better syntax understanding and code structure.
2. Supports fine-tuning – Unlike proprietary models like GPT-4, CodeLlama allows customization based on domain-specific requirements.
3. Computationally efficient – The 7B model balances performance and hardware requirements, making it feasible for fine-tuning on Google Colab or local GPUs.
4. Better Handling of Code Comments & Documentation – CodeLlama can generate explanatory comments, docstrings, and structured code documentation, making it valuable for developer assistance and knowledge transfer.
5. Lower Cost Compared to Proprietary Models – Since CodeLlama is open-source, it offers a cost-effective alternative to OpenAI's GPT models, which require expensive API calls and limit fine-tuning flexibility.

Dataset Selection & Preprocessing

Dataset Source & Description
The dataset used for training CodeCrafter was sourced from a GitHub repository containing open-source codebases. The dataset contains diverse code snippets, including complete functions, algorithm implementations, and debugging examples.

Source: GitHub repositories with (c++, java, Python) programming languages.
Languages Covered: Python, C++, Java
Total Dataset Size: 3.3 MB
Number of Files: 192

Preprocessing Pipeline
The raw code from the dataset underwent several preprocessing steps to ensure quality, consistency, and compatibility before being fed into the tokenizer. These steps aimed to eliminate noise, standardize formatting, and structure the data efficiently, ensuring that the fine-tuning process yields accurate and high-quality results.

Data Cleaning:
Many raw code files contained unnecessary elements such as debugging print statements, temporary logs, and excessive inline comments that were not useful for model training. These elements could bias the learning process by introducing noise and distracting from the core logic of the code.

- Print statements were removed to prevent the model from learning unnecessary debugging behavior.
- Excessive comments were eliminated while retaining essential docstrings and structured explanations that provide meaningful context.
- Metadata elements such as timestamps, author notes, or generated documentation headers were stripped to maintain uniformity across code snippets.

This step helped ensure that the dataset contained pure, functional code rather than extraneous content that could dilute model performance.

Standardization of Indentation and Formatting:
Raw code snippets often exhibited inconsistent indentation, line spacing, and formatting styles, making it challenging for the tokenizer to process them effectively. To address this:

- Auto-formatting tools were applied to enforce a consistent code style across all files.
- Irregular indentation (such as mixed use of spaces and tabs) was corrected to maintain structural coherence.
- Trailing spaces and redundant blank lines were removed to enhance readability and avoid unnecessary token overhead during model training.

This step ensured that all code snippets followed a uniform and standardized structure, enabling the model to learn patterns efficiently without being affected by inconsistencies in coding style.

Data Structuring:

Conversion of Raw Code Snippets into Structured JSONL Format:
To optimize the dataset for efficient storage and processing, all code snippets were converted into JSON Lines (JSONL) format, a widely used standard for handling large-scale text data. Each code snippet was stored as a separate JSON object, ensuring modularity and ease of access.
This format facilitated language-specific filtering, making it possible to train the model on selected programming languages.
Metadata such as file names and language identifiers were retained to allow for better categorization and potential future use in multi-language code assistance.

Format:
{“language”:lang,”code”:code}

Actual record:
{"language": "python", "code": "def isSubtree(self, s, t):\n   stack = [s]\n   while stack:\n node = stack.pop(0)\n   if node.val == t.val:\n   if self.check(node, t): return True\n stack += [child for child in [node.left, node.right] if child]\n   return False\ndef check(self, first, second):\n   if not first and not second:\n   return True\n   if first and second:\n   return first.val == second.val and self.check(first.left, second.left) and self.check(first.right, second.right)\n   return False\n"}

This format facilitated language-specific filtering, making it possible to train the model on selected programming languages.
Metadata such as file names and language identifiers were retained to allow for better categorization and potential future use in multi-language code assistance.
By structuring the dataset in this way, the preprocessing pipeline ensured that the model could be trained on clean, well-formatted, and organized code snippets, ultimately leading to better learning efficiency and improved code generation capabilities.

Tokenization

Tokenization is a crucial step in preparing data for training a language model as it converts raw text into numerical representations that can be processed efficiently. Since CodeLlama is specifically optimized for understanding and generating code, its tokenizer is designed to handle programming languages effectively, preserving syntax and structure. However, certain challenges were encountered during tokenization, particularly with padding tokens, which required additional modifications to ensure compatibility. The steps undertaken to successfully tokenize the dataset are detailed below.

Implementation of Tokenization

Loading CodeLlama’s Tokenizer

The first step was to load the tokenizer that corresponds to CodeLlama's pre-trained model. The tokenizer is responsible for breaking down code snippets into tokenized sequences that the model can interpret.

Using the Hugging Face Transformers library, the CodeLlama tokenizer was initialized with:

from transformers import AutoTokenizer tokenizer = AutoTokenizer.from_pretrained("MetaAI/CodeLlama-7b-hf")

This ensured that the dataset would be tokenized using the same vocabulary and tokenization rules that CodeLlama was pre-trained on. By leveraging this specialized tokenizer, the model benefits from better token efficiency and preservation of code syntax, which is critical for downstream tasks such as code generation and completion.

Setting Padding Token
During the tokenization process, an issue was encountered:

- CodeLlama's tokenizer does not include a default padding token, causing errors when padding was requested.
- Without padding, sequences of different lengths could not be processed together in batches, leading to inconsistencies during training.

The tokenized dataset is now ready for fine-tuning.

Next Steps: Fine-Tuning CodeLlama

The next phase involves fine-tuning CodeLlama-7B-HF on the preprocessed dataset to adapt it to code generation and debugging tasks.

Fine-Tuning Plan

Training on Custom Data
To fine-tune CodeCrafter on the curated dataset, Hugging Face's Trainer API will be utilized, which provides an efficient and scalable way to train transformer-based models. Optimized hyperparameters such as learning rate, batch size, and weight decay will be carefully selected to ensure stable training and prevent overfitting. This step will allow CodeCrafter to specialize in the provided dataset, improving its ability to generate accurate and context-aware code completions.

Evaluation & Testing
Post-training, the model will undergo rigorous evaluation and testing to assess its performance. Metrics such as perplexity (to measure how well the model predicts the next token) and BLEU scores (to evaluate text similarity) will be used to quantify its effectiveness. Additionally, human validation will be conducted by comparing the generated code outputs with expected results to

ensure logical correctness and readability. Debugging scenarios will also be tested to confirm the model's ability to offer relevant fixes and suggestions.

Deployment

Once the model meets the desired accuracy and performance benchmarks, CodeCrafter will be deployed as a web-based AI assistant or integrated into VS Code as a plugin. A web-based interface will allow users to interact with CodeCrafter via API calls, providing real-time code generation and debugging assistance. Alternatively, a VS Code extension will offer in-editor suggestions, enhancing developer productivity. Hosting options such as cloud-based inference servers or local execution will be explored to optimize response time and accessibility.