RepoMonkey: Using fine-tuning to generate comprehensive codebase documentation

Vedant Singh & Zachary Gelman & Maxwell Chien
10-423/623 Generative AI Course Project

December 14, 2024

1 Introduction

Codebases can be extremely complicated, made up of thousands of different functions potentially designed by many different coders, each with their own documenting ability (or lack thereof). It would therefore be valuable to have a model that could quickly generate documentation in a way that is consistent readable to people other than the person who coded it, both from function to function, and from project to project.

To address these challenges, we propose RepoMonkey, a documentation tool tailored for Python codebases. To build this we fine-tuned CodeT5 and LLaMA to generate meaningful descriptions of code components and their interactions. Using the CodeSearchNet dataset as a benchmark, we preprocess the data and customize the input and output pipelines to align with each model's architectural constraints. Both models are trained under controlled conditions to ensure a fair comparison of their capabilities in the domain of code understanding.

The evaluation of the models' performance was conducted using standard text-generation metrics explained in Section 2.3, yielding the metrics in Table 1. The metrics show that using our method with CodeT5 results in a model that can accurately document functions.

2 Dataset / Task

2.1 Definition of the Task

This task involved creating a tool that:

- Inputs Python code and analyzes it to extract functions, classes, and methods.
- Identifies relationships between these code elements, such as function calls, class inheritances, and method overrides.
- Generates detailed documentation for each function and class, including descriptions of their roles and interactions.
- Compiles the generated information into a coherent and user-friendly document.

2.2 Dataset Overview

- **CodeSearchNet Dataset:** We utilized the Python snippets from the CodeSearchNet dataset, which comprises a substantial collection of code snippets paired with natural language descriptions, such as docstrings and comments. Its extensive size makes it an ideal benchmark for evaluating the effectiveness of code summarization models like CodeT5 and LLaMA.
- **Data Splits:** To ensure consistency and reproducibility, we adopted the standard train/validation/test splits provided by CodeSearchNet. These predefined partitions maintain balanced representation across various code snippet lengths.

2.3 Metrics

- **Cross-Entropy Loss:** Common loss measure used in methods involving text generation
- **BLEU Score:** Measures n-gram overlap between generated and reference texts.
- **ROUGE Score:** Focuses on recall by counting overlapping units such as n-grams.

We chose these metrics as a proxy for human evaluation because it's a lot faster and cheaper to scale with given that we don't have the resources necessary for human evaluation. Out of the metrics out there, we went with BLEU and ROUGE since they're widely used in the language generation community. Their consistent scoring methodology provides a common ground for performance comparisons.

3 Related Work

- **CodeSearchNet Corpus:** Husain et al. [2020]

  - A dataset of 2 milllion (comment, code) pairs from opensource libraries hosted on GitHub.
  - CodeSearchNet corpus was gathered by HuggingFace to explore the problem of code retrieval using natural language.
- **LoRA:** Hu et al. [2021]
  - Method we used for fine-tuning
- **Documentation Generators:**
  - **Doxygen:**
    * Generates documentation from annotated code but does not deeply analyze code relationships.
    * Limited to processing comments and does not leverage advanced language models.
  - **Sphinx:**
    * Widely used in Python projects to generate documentation from docstrings.
    * Focuses on static descriptions without integrating code relationship mappings or LLMs.
  - **RepoAgent:** Luo et al. [2024]
    * This was a paper demonstrating a large language model powered open-source framework aimed at proactively generating, maintaining, and updating code documentation
- **Code-Specific Language Models:**
  - **CodeBERT:** Feng et al. [2020]
    * A bimodal pre-trained model for programming and natural languages.
    * Effective for code understanding tasks but primarily an encoder, making it less suitable for generative tasks without modifications.
  - **CodeT5:** Wang et al. [2021]
    * An encoder-decoder model pre-trained on large-scale code data.
    * Designed for code summarization and generation tasks, making it suitable for generating detailed documentation.
  - **Llama:** Touvron et al. [2023]
    * An open source autoregressive Large Language Model
  - **GraphCodeBERT:** Guo et al. [2021]
    * Extends CodeBERT by incorporating data flow graphs.
    * Aims to better understand code semantics and could enhance the understanding of function relationships.

4 Methods

4.1 Preprocessing Steps

4.1.1 Normalization

- Standardized newline handling and whitespace.
- Stripped trivially uninformative docstrings.
- Applied language-specific preprocessing, such as:
  - Removing trailing semicolons in JavaScript.
  - Normalizing indentation in Python.

4.1.2 Tokenization

- **CodeT5:** Utilized the built-in SentencePiece tokenizer trained on a mixed corpus of code and natural language.

- **LLaMA:** Employed the native tokenizer capable of handling a wide range of tokens, including those specific to code.

4.1.3 Filtering

- Removed code snippets exceeding 1,024 tokens to address computational constraints.
- Excluded docstrings shorter than five words to ensure meaningful training signals.

4.2 Model Setup and Training

4.2.1 Baseline Approach: CodeT5

- With CodeT5, we initiated training from the publicly available `Salesforce/CodeT5-base` checkpoint, leveraging its pre-trained knowledge of programming language syntax and semantics.
- With LLaMA 1B, we adapted the code summarization task into a prompt-based completion problem. This involved careful prompt construction to guide the model in generating accurate summaries.

4.2.2 Task Formulation

Code Summarization: CodeT5: Input the code snippet as the source sequence and train the model to generate the corresponding docstring as the target sequence.
LLaMA: Format the input prompt as follows:

> Below is a code snippet. Summarize its functionality in one sentence:
>
> `<code snippet>`
>
> Summary:

The model is then tasked with generating the summary text following the prompt.

4.2.3 Training Details

Loss Function: Employed cross-entropy loss to minimize the negative log-likelihood of the correct target tokens for the summarization task. We also attempted to fine-tune CodT5 with a custom loss function that included L2 loss, with minimal difference.

Hyperparameters:

- **CodeT5:** Learning rate set to 3×10^{-5}.
- **LLaMA:** Learning rate set to 1×10^{-5}.
- Batch size ranged from 16 to 32 sequences per batch, depending on GPU memory availability.
- Training was conducted for 3 epochs.

4.3 Model Input/Output Handling

4.3.1 Encoder-Decoder Pipeline (CodeT5)

- **Encoding:** The code snippet is processed by the encoder to generate contextualized representations.
- **Decoding:** The decoder generates the docstring token by token, conditioned on the encoder's output.

4.3.2 Decoder-Only Pipeline (LLaMA)

- **Prompt Construction:** Formulated prompts to guide the model in generating summaries.
- **Truncation and Padding:** Ensured all inputs fit within the 2,048-token context window. Truncated overly long code snippets while preserving essential function boundaries and critical lines.

5 Experiments

5.1 Evaluation Metrics and Procedures

5.1.1 Quantitative Metrics

- **BLEU Score:** Assessed the overlap between generated summaries and reference docstrings.
- **ROUGE Scores:** Evaluated the quality of generated summaries using the following variants:
  - ROUGE-1: unigram overlap
  - ROUGE-2: bigram overlap
  - ROUGE-L: LCS comparison

5.1.2 Results

As shown in Table 1, the fine-tuned CodeT5 model with L2 and CE loss achieves the best BLEU and ROUGE scores, outperforming both the base models and the fine-tuned LLaMA model.

5.1.3 Qualitative Analysis

Conducted manual inspections of a representative sample of generated summaries to verify their accuracy and fluency. Observed that visual representations generated by the tool significantly enhanced developers' understanding of complex code interactions, thereby reducing the time required to navigate large codebases.

5.2 Implementation Details and Computational Considerations

5.2.1 Hardware Environment

Training and evaluation were performed on NVIDIA A100 GPUs, each equipped with 40GB of VRAM. Training durations were about several hours long, and runs for LLaMA generally took 2x longer than CodeT5.

5.2.2 Software Stack

- **Frameworks:** Utilized Python with PyTorch for model training and evaluation.
- **Libraries:** Employed Hugging Face Transformers for accessing pre-trained models and tokenizers, and the datasets library for efficient data handling.
- **Tools:** Implemented custom scripts for data preprocessing and pipeline management to ensure integration between components.

5.3 Experimental Controls and Comparisons

To maintain a fair and unbiased comparison between CodeT5 and LLaMA, the following controls were implemented:

- **Consistent Data Splits:** Both models were trained, validated, and tested on identical data partitions.
- **Uniform Task Setup:** Ensured that input and output lengths, as well as token budgets, were comparable across models.
- **Training Regimen:** Applied similar epochs and early stopping criteria based on validation performance for both models.

Any deviations, such as additional adaptation phases required by LLaMA due to its general-purpose pretraining, were documented and their impacts on performance were analyzed separately.

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Base LLaMA	0.28769	0.56739	0.56281	0.56552
Fine-tuned LLaMA	0.40024	0.59953	0.59573	0.59979
Base CodeT5	0.03243	0.14197	0.05112	0.12571
Fine-tuned CodeT5 (CE loss)	0.90473	0.97425	0.90243	**0.93171**
Fine-tuned CodeT5 (L2 + CE loss)	**0.90531**	**0.97505**	**0.90256**	0.93084

Table 1: Post-training evaluation metrics for LLaMA and CodeT5 models.

6 Code Overview

6.1 Preprocessing Pipeline for CodeT5 (`data_preprocessing.py`)

The `data_preprocessing.py` script processes the CodeSearchNet dataset for CodeT5 fine-tuning. Key steps include:

- **Dataset Loading:** Loads the Python subset of CodeSearchNet and selects 100,000 samples for processing.
- **AST Parsing and Analysis:** Extracts detailed information from Python functions using a custom `FunctionAnalyzer` class, including:
  - Function names, parameters, and return values.
  - Control flow constructs and variables.
  - Imports and called functions.
- **Augmented Docstring Creation:** Combines extracted AST data with docstrings to produce enriched documentation, aiding the fine-tuning task.
- **Output Generation:** Saves the processed dataset to a json file for downstream tasks.

6.2 Fine-Tuning Pipeline for CodeT5 (`main.py`)

The `main.py` script fine-tunes CodeT5 using the preprocessed dataset. Key components include:

- **Dataset Preparation:** Loads and tokenizes the augmented dataset, splitting it into training and validation sets.
- **Model Setup:** Initializes the pre-trained CodeT5-base model and tokenizer, adapting them for sequence-to-sequence learning.
- **Training and Evaluation:** Implements fine-tuning via Hugging Face's `Trainer` class and evaluates model performance using BLEU and ROUGE metrics before and after training.

6.3 Preprocessing Pipeline for LLaMA (`data_preprocessing.py`)

The `data_preprocessing.py` script prepares the CodeSearchNet dataset for fine-tuning LLaMA. Key steps include:

- **Dataset Loading:** Loads the Python subset of CodeSearchNet and selects 10,000 samples for processing.
- **AST Parsing and Analysis:** Extracts semantic details from Python functions using the `FunctionAnalyzer` class, similar to the CodeT5 preprocessing pipeline.
- **Prompt and Completion Generation:** Creates prompts by embedding code snippets in a natural language instruction format, while the enriched docstring serves as the completion.
- **Output Generation:** Saves the processed dataset to a json file for fine-tuning.

6.4 Fine-Tuning Pipeline for LLaMA (`main.py`)

The `main.py` script fine-tunes the LLaMA model using the preprocessed dataset. Key components include:

- **Dataset Preparation:** Loads and tokenizes the augmented dataset with specialized prompts and completions.
- **Model Setup:** Initializes the pre-trained LLaMA model (1B variant) and tokenizer. Configures memory allocation and ensures compatibility with GPU.
- **Training and Evaluation:**
  - Implements fine-tuning using the Hugging Face `Trainer` class with gradient accumulation for efficient training.
  - Evaluates model performance using BLEU and ROUGE metrics on a subset of the validation dataset before and after training.
- **Prompt Engineering:** Prompts follow the structure: `<s>Below is a piece of code:\n<code>\n\nGenerate a detailed docstring:\n<s>`.
- **Memory Optimization:** Configures `PYTORCH_CUDA_ALLOC_CONF` to prevent memory fragmentation and leverages Ampere GPU optimizations for faster computation.

7 Timeline

The project began on November 1, 2024, a comprehensive accounting of our time can be found at Table 2.

8 Research Log

8.1 Initial Exploration

After we decided to work on repository documentation, we explored the CodeSearchNet dataset and relevant literature on code-to-text tasks. Early on, understanding the dataset structure and challenges posed by the variability in code snippets and docstrings required significant effort. A notable challenge during this phase was identifying a subset of the dataset that balanced computational feasibility with sufficient complexity for meaningful evaluation.

8.2 Preprocessing Challenges

Preprocessing the dataset for both CodeT5 and LLaMA posed unique challenges. The creation of augmented docstrings required an implementation of Abstract Syntax Tree (AST) parsing to extract relevant function details. Early iterations of the preprocessing scripts had issues with:

- Handling syntax errors in certain code snippets.
- Generating meaningful docstrings when input docstrings were missing or poorly written.
- Balancing prompt length for LLaMA to fit within token limits while preserving semantic integrity.

These challenges were addressed through iterative debugging, implementing fallback mechanisms for syntax errors, and tuning truncation logic for prompts.

8.3 Narrowing Our Vision

We had originally planned to also create a simple UI for our model and to include visual representations alongside our text comments. Taking the advice of our advisor, we narrowed our vision to code commenting specifically, and leave those ideas as fodder for future research.

8.4 Model Training and Fine-Tuning

Fine-tuning the models presented significant computational and algorithmic challenges. For CodeT5, adapting the sequence-to-sequence model required careful management of memory and gradient accumulation strategies. LLaMA, being a causal decoder, necessitated prompt engineering to align the input-output format with its architecture. Initial experiments showed:

- Slow convergence for LLaMA due to its general-purpose pretraining.
- Overfitting tendencies in CodeT5 when using small subsets of the dataset.

These issues were mitigated by tuning hyperparameters, implementing dropout regularization, and increasing dataset size for CodeT5 experiments.

8.5 Evaluation and Results Compilation

Evaluating model performance using BLEU and ROUGE metrics was straightforward but highlighted some discrepancies between quantitative metrics and qualitative assessments. Manual inspections revealed instances where generated summaries were syntactically correct but semantically misleading. To address this:

- Additional qualitative evaluations were conducted to refine the prompt structure for LLaMA.
- Metrics were complemented with anecdotal evidence from specific examples.

8.6 Presentation

Questions brought up at our poster presentation brought up both interesting new avenues of research, and ideas for how we should have improved our model.

8.7 Key Challenges and Lessons Learned

- **Computational Constraints:** Running experiments on large models like LLaMA required careful memory management and batching strategies.
- **Data Quality:** Poorly written or missing docstrings in the dataset highlighted the importance of data cleaning and augmentation.
- **Model-Specific Nuances:** Adapting task setups for encoder-decoder (CodeT5) vs. decoder-only (LLaMA) architectures necessitated different approaches to preprocessing, training, and evaluation.

9 Conclusion

This project set out to explore the application of CodeT5 and LLaMA models for the task of code-to-text generation using the CodeSearchNet dataset, with the purpose of developing a generative model that could document large repositories on a function to function level. Below, we summarize the main findings and considerations for future work.

Activity	Hours Spent
Reading papers, dataset websites, and related resources	8
Reading code documentation (e.g., PyTorch, Hugging Face)	4
Compiling/running existing code	3
Modifying existing code or writing new code from scratch	10
Writing scripts to run experiments	3
Running experiments	10
Compiling and analyzing results	4
Writing this document	4
Total	**46**

Table 2: Approximate time spent on various stages of the project.

9.1 Main Findings

- **Preprocessing Insights:** Developing robust preprocessing pipelines was crucial for extracting meaningful data from the dataset. Abstract Syntax Tree (AST) parsing proved to be an effective method for generating enriched docstrings, but it also highlighted the variability and inconsistencies in real-world code documentation.
- **Model Performance:** CodeT5 demonstrated a very strong performance on the task, perhaps leveraging its encoder-decoder architecture to produce contextually rich summaries. LLaMA did not perform as well, perhaps due to being a larger model that was less flexible to fine-tuning with LoRA, but showcased flexibility in handling natural language instructions for code summarization.
- **Evaluation Metrics:** BLEU and ROUGE scores provided valuable quantitative insights but were complemented by qualitative evaluations to ensure the generated summaries were both accurate and human-readable.

9.2 Key Challenges

- **Computational Constraints:** We needed to use a less advanced LLama model than we intended because of CUDA memory issues
- **Data Quality:** The variability in docstrings showed the need for better curation and augmentation techniques in datasets for code-related tasks.
- **Architectural Adaptations:** The differing architectures of CodeT5 and LLaMA necessitated tailored approaches to preprocessing, training, and evaluation.

9.3 Future Work

This project opens several avenues for future research:

- Developing more sophisticated data augmentation techniques to handle incomplete or poorly written docstrings.
- Extending the evaluation framework to include metrics that better capture semantic understanding and human-centric quality.
- Iteratively evaluating with another model trained to generate functions from RepoMonkey outputs
- Investigating the scalability of these models on larger datasets and more diverse programming languages.

We hope that RepoMonkey serves as a good proof of concept for fine-tuning models to generate documentation in a flexible way.

References

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL https://arxiv.org/abs/2002.08155.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021. URL https://arxiv.org/abs/2009.08366.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Repoagent: An llm-powered open-source framework for repository-level code documentation generation, 2024. URL https://arxiv.org/abs/2402.16667.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. URL https://arxiv.org/abs/2109.00859.

Appendix

Sample Input and Output of RepoMonkey

```
def multihead_graph_attention(query_antecedent,
    memory_antecedent, bias, total_key_depth, total_value_depth,
     output_depth, num_heads, dropout_rate, ...)
    q, k, v = common_attention.compute_qkv(\n
    query_antecedent, memory_antecedent, total_key_depth,
    total_value_depth, vars_3d_num_heads=vars_3d_num_heads)
    q = common_attention.split_heads(q, num_heads)
    k = common_attention.split_heads(k, num_heads)
    v = common_attention.split_heads(v, num_heads)
    key_depth_per_head = total_key_depth // num_heads
    if not vars_3d:\n
      q *= key_depth_per_head**-0.5
    additional_returned_value = None
    if callable(attention_type):  # Generic way to extend
        multihead_attention
      x = attention_type(q, k, v, **kwargs)
      if isinstance(x, tuple):
        x, additional_returned_value = x
    [...]
    if additional_returned_value is not None:
      return x, additional_returned_value
    return x
```

This is a sample snippet of code given to RepoMonkey.

```
Function Name:
multihead_graph_attention
Purpose:
Multihead scaled-dot-product attention with input/output
    transformations.

  Args:
    query_antecedent: a Tensor with shape [batch, length_q,
        channels]
    memory_antecedent: a Tensor with shape [batch, length_m,
        channels] or None
    bias: bias Tensor (see attention_bias())
    total_key_depth: an integer
    total_value_depth: an integer
    output_depth: an integer
    num_heads: an integer dividing total_key_depth and
        total_value_depth
    ...
```

This is a truncated version of the generation of our model on the above snippet of code.