Code Documentation Generation with Fine-tuned Transformers

Kaushik Mellacheruvu, Srirama Bulusu, Saumya Parag Phadkar

New York University

Introduction

Code documentation in Software Engineering or any programming field has been and will be a very important aspect of code readability and maintainability. There has been a lot of legacy code which needs to be manually documented while also having a need for the new code being written or AI- generated to be well documented. Effective code documentation is important for software development, improving code readability, maintainability, and collaborative efforts. However, manual documentation can be a time-consuming and often neglected task, particularly for large codebases and legacy systems. This project investigates the potential of advanced transformer-based models, specifically CodeBERT, DistilBERT, and BART, to generate high-quality and context-aware code comments.

By leveraging the strengths of these pre-trained models in understanding both code and natural language, we aim to develop models that can accurately generate concise and informative comments for code snippets and functions. This research seeks to address the challenges associated with inadequate documentation, improve code maintainability, and ultimately enhance developer productivity by streamlining the documentation process.

Literature Survey

Current models for code documentation, while effective, often lack specificity and context awareness in generating comments tailored to diverse codebases. Several key research efforts have explored different approaches to address these challenges, primarily focusing on the use of transformer-based models like GPT-3, CodeBERT, DistilBERT, and other recent advancements in code-to-text generation. This section reviews some of the most relevant works in the field.

Khan, J. Y., and Uddin, G. (2022) introduced ”Automatic Code Documentation Generation Using GPT-3”. This research investigates the potential of GPT-3 in generating accurate and meaningful documentation from code. The study highlights that GPT-3, despite its advanced natural language generation capabilities, still struggles with producing domain-specific comments that require a deep understanding of the code context. The authors suggest that fine-tuning GPT-3 on specialized datasets could potentially improve its performance, especially for complex codebases.

CodeBERT, introduced by Feng et al. (2020), is a bimodal pre-trained transformer model designed specifically for programming-related tasks. CodeBERT integrates both natural language and source code representations, which makes it particularly suited for tasks like code summarization, comment generation, and code translation. However, while CodeBERT is effective at generating simple comments, its performance can still be limited when dealing with complex, domain-specific codebases.

”A Comparative Study on Code Generation with Transformers” and ”Transformer-Based Models for Computer Code Generation” delves into the use of transformer models, including GPT-2, BERT, and CodeBERT, for code generation tasks. The studies compares the ability of these models to generate code from scratch, as well as to generate documentation from existing code.

Several other works have explored the use of pre-trained models, such as GPT-2, T5, and BERT, for tasks like code summarization and documentation generation. These models, though not originally designed for code, can be fine-tuned on programming-specific datasets to improve their performance. However, the complexity of code semantics remains a challenge, and these models often require substantial domain-specific fine-tuning to produce high-quality documentation.

We aim to use CodeBERT, BART and DistilBERT to generate documentation with context and help address the lack of granularity in code comments and understand context more.

Dataset

For the fine-tuning of our transformer models, we primarily utilize the CodeSearchNet dataset (1), which comprises code snippets from six different programming languages. This dataset is particularly well-suited for our study because it pairs natural language documentation with corresponding code snippets, facilitating the training of models on code summarization and documentation tasks.

Data Structure and Content: The CodeSearchNet dataset provides richly annotated code snippets from six programming languages, each entry featuring fields such as `func_name`, `whole_func_string`, `language`, `func_code_string`, `func_code_tokens`, `func_documentation_string`, and `func_documentation_tokens`. This structure not only details the functionality of the code but also includes both the complete code and its segmented tokens alongside the natural language documentation. Such comprehensive data enables our models to capture essential syntactic and semantic patterns across diverse programming languages, enhancing their ability to generate precise and contextually relevant documentation.

Dataset Splits and Language Distribution: The CodeSearchNet dataset is divided into training, testing, and validation splits to ensure robust model training and evaluation. As illustrated in Figure 1, the dataset features a balanced distribution of six programming languages across all splits. This balanced distribution is critical for training the models to generalize well across different types of code and documentation styles. For instance, Python and Java, which are widely used in the industry, have a strong presence across all splits, ensuring that the models encounter a variety of coding patterns and documentation practices during training. Other languages like PHP, JavaScript, Go, Ruby, and Java also show consistent distributions, which helps in minimizing overfitting to specific language features and enhances the model's adaptability to unseen data in real-world applications.

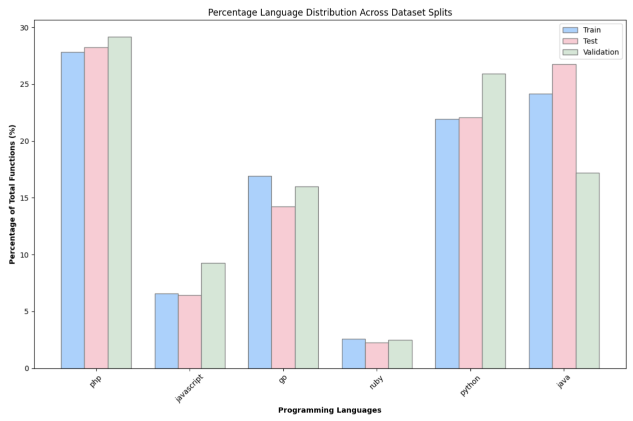

Figure 1: Dataset Split

Models and Training Details

DistilBERT

DistilBERT (2) is a smaller, faster version of BERT, created using knowledge distillation. It retains 97% of BERT's performance while being 60% faster and 40% smaller. DistilBERT is primarily used for tasks such as text classification, named entity recognition, and question answering.

Model Architecture DistilBERT is built on the Transformer architecture, similar to BERT, with a reduced number of layers. The architecture combines bidirectional attention (like BERT) with the efficiency of a smaller model, making it faster to fine-tune.

In our implementation, we use the pre-trained `DistilBERTForMaskedLM` model, which is specifically designed for masked language modeling (MLM). MLM is a pretraining objective where some percentage of input tokens are randomly masked, and the model is trained to predict these masked tokens based on the surrounding context. This is particularly useful for tasks like code documentation generation where the model needs to understand both the structure and meaning of code.

Loss Function DistilBERT is fine-tuned using the cross-entropy loss function for classification tasks:

$$\mathcal{L} = -\sum_{i} \log P(y_i|x_i)$$

where:

- x_i: Input token at position i.
- y_i: Ground-truth token at position i.
- $P(y_i|x_i)$: Probability of the output token predicted by the model.

Hyperparameters DistilBERT's fine-tuning involves the following hyperparameters:

- Batch size: 16.
- Learning rate: $2e-5$.
- Epochs: 1.
- Optimizer: AdamW with weight decay of 0.03.
- Sequence length: 256.

Selection Process: The hyperparameters were chosen through a grid search process, optimizing for a balance between training speed and model performance on the validation dataset.

Data Preprocessing

- **Customized Handling of [MASK] Tokens:** Implemented tailored masking of documentation comments for each programming language in the dataset. This involves the addition of characters that indicate comments in the `func_code_string`, which are then removed from the `func_documentation_string`.
- **Other Preprocessing Steps:** Additional preprocessing includes the normalization of code strings and the removal of non-essential whitespace and special characters to simplify the input for the model.

The preprocessing stage utilizes key fields from the dataset, which are essential for training our models:

- **Repository Name and Path:** This metadata provides context about the storage location of the code, offering insights into project-specific conventions or coding styles.
- **Function Name (`func_name`):** Typically hints at the function's purpose or behavior, helping the model understand the intended functionality.
- **Function Code (`func_code_string`):** The executable code snippet that is central to our analysis and the generation of documentation.

Expected Output The output from the preprocessing is structured to aid in model training:

- **Documentation (`func_documentation_string`):** Provides natural language descriptions of what the code functions perform. These descriptions are used as targets in the documentation generation process, allowing the model to learn to summarize and explain code functionalities accurately.

Training Process

- **Initial Learning:** The training loss decreases sharply at the outset, indicating rapid learning from the training data.
- **Loss Stabilization:** Both training and validation losses stabilize quickly, showing minor fluctuations. This suggests that the model has effectively captured the underlying patterns of the data.
- **Generalization:** The close proximity of training and validation losses throughout the process indicates good generalization, with no significant overfitting observed.
- **Training Efficiency:** The stable loss values after the initial decrease suggest that extending training beyond one epoch might not significantly enhance performance without further adjustments in model complexity or training strategy.

CodeBERT

CodeBERT (3) is a pre-trained model designed specifically for programming language-related tasks, built on the BERT architecture. It is pre-trained on a large corpus of source code and natural language pairs, making it effective for tasks such as code summarization, code generation, and code search.

Model Architecture CodeBERT is based on the BERT architecture and it is trained on a multi-modal corpus containing both code snippets and natural language descriptions, making it suitable for tasks that involve code and text.

Loss Function CodeBERT is fine-tuned using the masked language model loss function:

$$\mathcal{L} = -\sum_{i} \log P(y_i|x_i)$$

where:

- x_i: Input token at position i.
- y_i: Ground-truth token at position i.
- $P(y_i|x_i)$: Probability of the output token predicted by the model.

Hyperparameters CodeBERT's fine-tuning uses the following hyperparameters:

- Batch size: 16.
- Gradient Accumulation: 2
- Warm Up Steps: 200
- Eval Steps: 500
- Learning rate: $2e-5$.
- Epochs: 1.
- Optimizer: AdamW with weight decay of 0.03.
- Sequence length: 256.

BART

BART (Bidirectional and Auto-Regressive Transformer) (4) is a transformer-based sequence-to-sequence (Seq2Seq) model. It combines the bidirectional encoding capabilities of BERT and the autoregressive decoding capabilities of GPT, making it highly effective for tasks like text summarization, translation, and code-to-documentation generation.

Model Architecture BART consists of a fully bidirectional encoder (like BERT) and an autoregressive decoder (like GPT). Both encoder and decoder are built using transformer layers comprising multi-head attention and feed-forward networks.

Loss Function The training objective of BART is to minimize the cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t|x_{<t}, y)$$

where:

- x_t: Ground-truth token at time step t.
- y: Corrupted input sequence.
- $P(x_t|x_{<t}, y)$: Probability of the token predicted by the decoder, conditioned on previous tokens.

Hyperparameters BART's fine-tuning involve the following hyperparameters:

- Batch size: 32.
- Learning rate: $5e-5$.
- Epochs: 1
- Optimizer: AdamW with a weight decay of 0.01.
- Sequence length: 256.

Fine-Tuning

For code documentation generation (using the CodeSearchNet dataset):

- **Input:** Code snippets (e.g., Python, Java).
- **Output:** Natural language documentation.
- **Model Output:** The encoder processes code snippets, and the decoder generates corresponding documentation.

Evaluation Metrics

To evaluate performance on code-to-documentation tasks, the following metrics are used:

- **BLEU:** Measures n-gram overlap between generated and reference documentation.
- **ROUGE-1:** Measures recall and precision of overlaps in generated text.

Tracking and Plotting

We use Weights Biases(wandb) to plot the training and validation losses for each model. We also track the GPU usage and the loss w.r.t hyperparameters such as learning rate and steps.

Results

In this section, we present the training and evaluation results of all 3 models. The training and validation losses, as well as the performance metrics (ROUGE and BLEU), are shown.

DistilBERT

After fine-tuning DistilBERT for one epoch on the CodeSearchNet dataset, specifically for the Python language, we observed the following performance metrics and trends in training loss. The model's learning curve and generalization capabilities are illustrated through the figures and summarized performance metrics provided below.

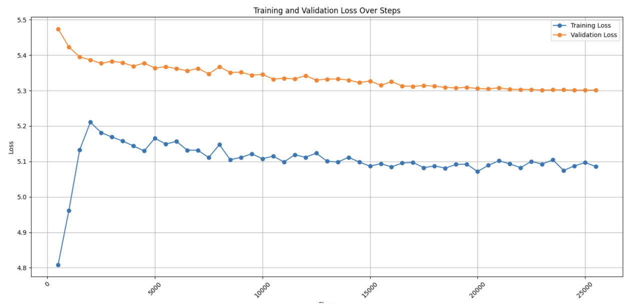

Figure 2: DistilBERT Loss Curve

The following table summarizes the performance metrics for DistilBERT, specifically focusing on the ROUGE-1 F Score and BLEU score:

Scoring Metric	Mean	Min	Max
BLEU	0.0885	0	1
ROUGE-1 F Score	0.3226	0	1

Table 1: DistilBERT Performance Metrics

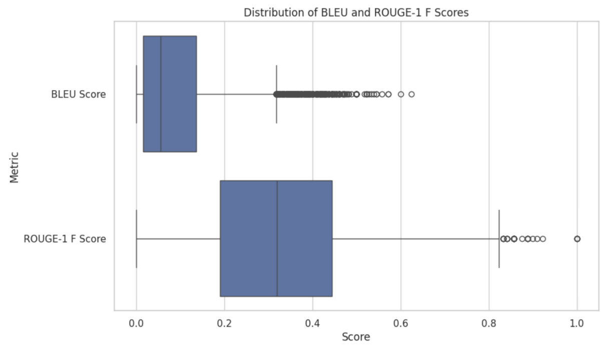

Figure 3: DistilBERT ROUGE and BLEU Scores

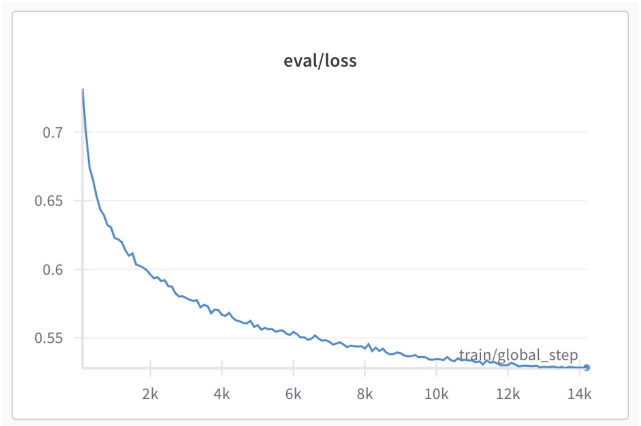

Figure 4: BART-Validation Loss

Rationale for Metric Choice: The focus on 1-gram ROUGE and BLEU scores was driven by the project's initial goal to assess lexical accuracy of generated documentation text against the references. This metric choice reflects an introductory analysis, aimed at evaluating the model's capability to replicate basic terms and phrases accurately, which is crucial in the initial stages of testing documentation generation systems.

BART

After fine-tuning the BART Model for 1 epoch over the entire training set of CodeSearchNet, we can observe the following. Figure 5 illustrates the training loss over approximately 14,000 steps. The training loss starts at 0.8 and saturates at 0.4 at the end of 1 epoch , and Figure 4 shows the validation loss. These graphs help to evaluate how well the model is learning over time and how it generalizes to unseen data.

The ROUGE and BLEU scores provide insight into the quality of the generated documentation compared to the reference documentation. Higher scores indicate better quality. The results for both ROUGE and BLEU scores are summarized in Table 1.

Scoring Metric	Mean	Min	Max
BLEU	0.1622	0	1
ROUGE-1 F Score	0.3008	0	1

Table 2: BART-Performance

CodeBERT

Fine-tuning CodeBERT on the entire CodeSearchNet training set gives us the following information:

- Figure 7 shows us the training loss over 12500 steps. It starts at a pretty high loss of 5.8 but over the course of all the steps it drops down to 2.31.
- Figure 8 shows us the validation loss over 12500 steps, starting from 4.35 and ending at 2.51.

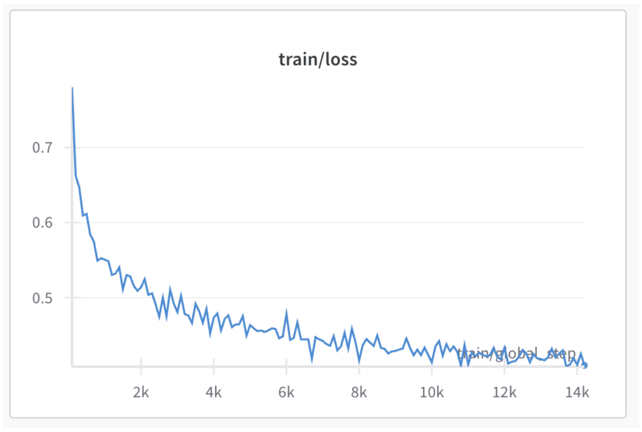

Figure 5: BART-Training Loss

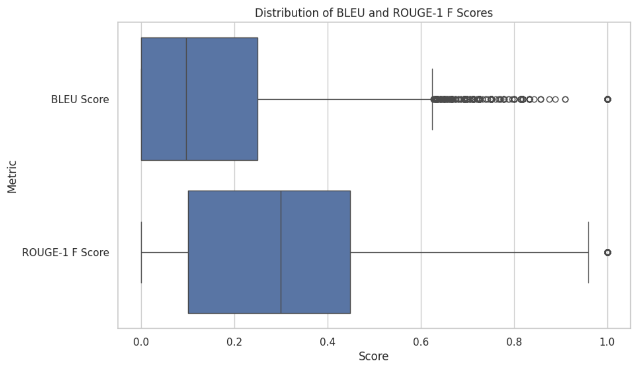

Figure 6: BART-ROUGE-BLEU Scores

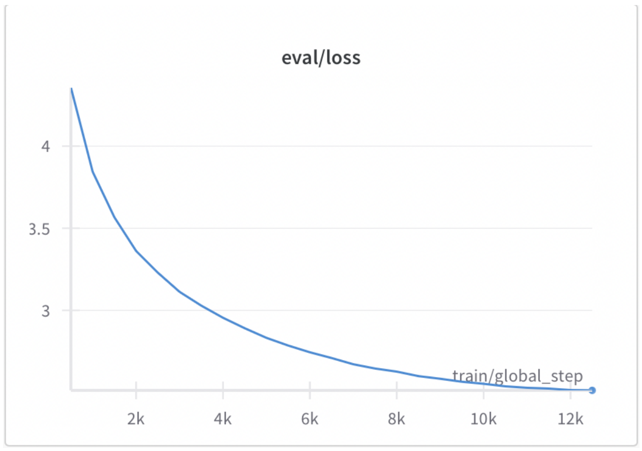

Figure 7: CodeBERT-Validation Loss

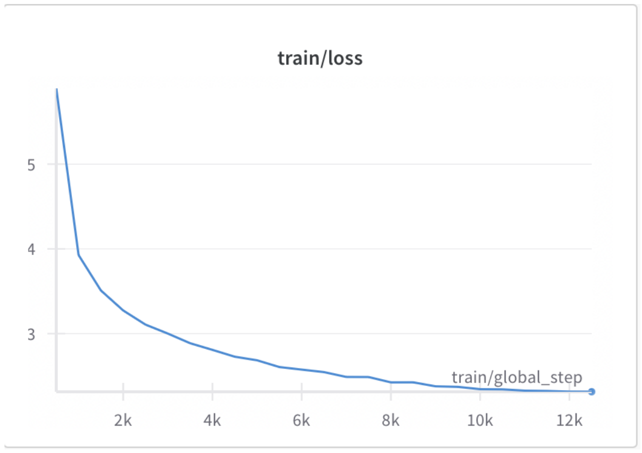

Figure 8: CodeBERT-Training Loss

- The distribution of BLEU and ROUGE scores can be seen in Figure 9.
- This shows us that the model learns pretty well over the 1 training epoch but that there is still room for improvement, which can be seen if trained for more epochs.

The ROUGE and BLEU scores can be seen in Table 2.

Scoring Metric	Mean	Min	Max
BLEU	0.1038	0	1
ROUGE-1 F Score	0.2671	0	1

Table 3: CodeBERT-Performance

Conclusion

This research has demonstrated the potential of transformer-based models like DistilBERT, BART, and CodeBERT in the task of code documentation generation. Each model was fine-tuned on the CodeSearchNet dataset, which comprises diverse programming languages and provides a robust basis

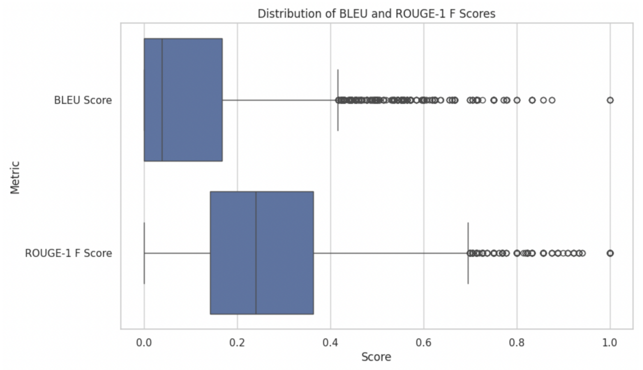

Figure 9: CodeBERT-ROUGE-BLEU Scores

for training and evaluating the models' capabilities in understanding and generating human-like documentation for code.

Key Findings

Our experiments revealed that:

- **DistilBERT** showed promising results by rapidly adapting to the nuances of code documentation with relatively low computational requirements, suggesting its utility in scenarios where resources are limited.
- **BART** excelled in generating coherent and contextually relevant documentation, owing to its sophisticated sequence-to-sequence architecture. Its performance, reflected in both training dynamics and the quality of the generated text, indicates its suitability for comprehensive documentation tasks.
- **CodeBERT**, designed specifically for programming languages, leveraged its bi-modal training on both code and natural language to effectively summarize and translate code into accurate documentation, though with room for improvement in handling complex codebases.

Implications

The application of these models can significantly streamline the process of software documentation, a critical but often neglected aspect of software development. By automating the generation of accurate and insightful documentation, these models can enhance developer productivity, improve code maintainability, and facilitate better understanding and usage of software products.

The advancement in natural language processing and machine learning, as applied to the domain of code documentation, presents a promising frontier for both academic research and practical applications in software engineering. This research underscores the potential of transformer-based models to significantly enhance the automation of code documentation, thus aiding in the maintenance and usability of software systems while boosting developer productivity.

Future Directions

Future work may focus on:

- Enhancing the models' understanding of complex code constructs and advanced programming paradigms through further fine-tuning and more extensive training.
- Expanding the dataset to include a wider array of programming languages and frameworks to enhance the models' applicability.
- Extending the scope to include other kinds of comments, such as inline single-line or multi-line comments, to improve the comprehensiveness of documentation.
- Integrating these models into development environments as plugins, providing real-time, context-aware documentation suggestions to developers.

Code

Here is a clickable link to Code.

References

[1] CodeSearchNet, *https://github.com/github/CodeSearchNet*

[2] Sanh, V., Debut, L., Chaumond, J., Wolf, T. (2019) *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. NeurIPS 2019.

[3] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., & Jiang, D. (2020) *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1536–1547.

[4] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, Luke Zettlemoyer *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*