CSE 842 Project: Context-aware Code Documentation Generator

Abhilasha Jagtap[1, 2, 3], Huy Pham[1, 2, 3], Sania Sinha[1, 4], Kyllan Wunder[1, 2, 3]

[1] Department of Computer Science and Engineering, Michigan State University
[2] Department of Computational Mathematics, Michigan State University
[3] Department of Statistics, Michigan State University
[4] Department of Mathematics, Michigan State University

Abstract

This project is focused on developing an automatic code documentation generator, specifically creating header comments for code functions similar to Python Docstrings. Utilizing the CodeSearchNet dataset and others, we will compare the performance of a baseline model against simpler architectures, such as BERT, and more advanced models, like CodeBERT, which handle the inverse task of generating code from comments. We will also explore a multi-agent system to decompose the task into subtasks, including variable extraction and high-level function summarization. The goal is to evaluate and improve the accuracy and efficiency of documentation generation across multiple programming languages using both traditional and experimental techniques.

1 Introduction

Automatic code documentation generation has recently gained considerable attention within software engineering and natural language processing. As modern codebases become more intricate and the demand for streamlined developer collaboration increases, accurate and context-aware documentation is more critical than ever. Though essential, manual documentation is time-consuming and prone to human error, making automation a compelling alternative. Initial approaches to this problem were grounded in template-based and information retrieval techniques, which provided some relief but lacked flexibility and adaptability. The advent of deep learning and large language models (LLMs) has introduced more sophisticated models capable of generating high-quality documentation with a deeper semantic understanding of code.

The motivation for this work stems from the ongoing challenges associated with efficiently generating high-quality, accurate, and context-aware code documentation. Code documentation is essential for maintaining codebases, enhancing collaboration, and reducing technical debt, yet manual documentation remains labor-intensive and error-prone. One newly emerging technique, that we will focus on, involves decomposing the complex task of documentation generation into simpler, more manageable components. We propose a multi-agent system where each agent focuses on specific process aspects, such as identifying variable roles, summarizing code logic, or generating high-level function descriptions. By evaluating traditional and experimental methods, we aim to determine the most effective strategy for producing clear, comprehensive, and useful documentation across various programming languages. Ultimately, this work aims to improve the quality of generated documentation while streamlining the process, making it more accessible for developers working on open-source and large-scale professional projects alike.

2 Related Work

Automatic code documentation generation has seen growing interest due to its potential to reduce the burden of manual documentation. Early approaches primarily relied on template-based methods (Chambers and Jurafsky, 2011) and information retrieval (IR) strategies (Singhal et al., 2001). Text similarity-based approaches (Quan et al., 2010), such as TF-IDF (Joachims et al., 1997), BM25 (Robertson et al., 2004), bag-of-words (Zhang et al., 2010), and NNGen (Liu et al., 2018), rely on code tokens for summarization. Conversely, semantic similarity-based methods (Iosif and Potamianos, 2009), such as RNN-based seq2seq models, encode code snippets into semantic vectors using a trained Bi-LSTM encoder, allowing retrieval based on cosine similarity.

More recent approaches leverage deep learning (DL) techniques, including models like CodeBERT (Feng et al., 2020), a bi-modal pre-trained encoder-

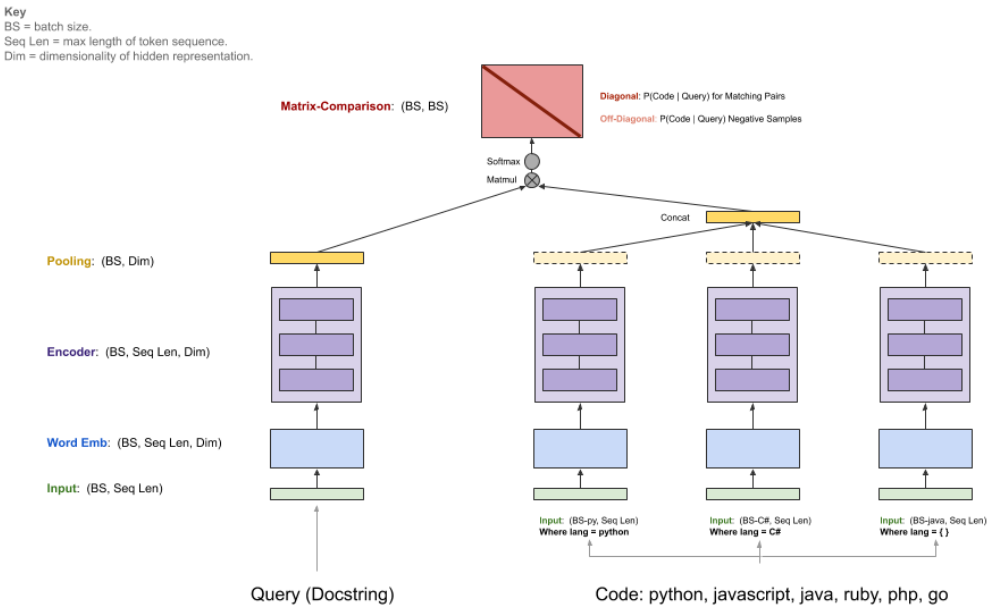

Figure 1: Baseline Model architecture from CodeSearchNet challenge

only Transformer model that learns representations of both natural language and programming language. Codex (Khan and Uddin, 2022), a GPT-3-based large language model, demonstrates strong performance on code documentation tasks without requiring fine-tuning, outperforming models like CodeBERT. Additionally, CodeT5+ (Wang et al., 2023), a unified encoder-decoder Transformer model, offers flexibility by allowing its components to be used in various downstream tasks, such as code summarization and documentation generation.

A recent empirical study (Zhu et al., 2024) comparing IR and DL approaches found that DL models generally outperform traditional methods for code summarization and documentation generation. However, the performance of these models is influenced by factors such as dataset characteristics and evaluation metrics (discussed in the Methodology section).

3 Data

For the training and testing of our models, we selected the Python subset of the CodeSearchNet dataset (Husain et al., 2019). This decision was based on several key factors that align closely with the objectives of our project.

The CodeSearchNet dataset is a comprehensive collection of code functions sourced from open-source GitHub repositories. Specifically, it includes projects that are used by at least one other project, ensuring that the code is not only functional but also has practical relevance within the developer community. By focusing on widely adopted code, we increase the likelihood of encountering well-structured and meaningful examples, which are essential for training effective documentation generation models.

To adhere to legal and ethical standards, any projects without a license or with licenses that do not explicitly permit the redistribution of parts of the project have been excluded from the dataset. This careful curation ensures that our use of the data respects the rights of the original authors and complies with open-source licensing agreements.

The Python subset of the dataset is particularly extensive, containing approximately 1,156,085 functions, of which 503,502 are accompanied by documentation. This substantial volume of documented functions provides a robust foundation for supervised learning approaches, offering ample examples to train the model. By focusing on Python, we reduce the complexity associated with handling multiple programming languages, allowing us to tailor our models more effectively and achieve more precise results.

Furthermore, the dataset's pairing of code functions with their corresponding documentation aligns with our goal of generating header comments similar to Python docstrings (Goodger and van Rossum, 2001). This direct correlation enables us to train models that can learn the intricate relationships be-

tween code and its documentation, facilitating the generation of accurate and contextually appropriate comments.

By utilizing the Python subset of the CodeSearchNet dataset, we ensure that our models are trained and tested on high-quality, ethically sourced data that is directly relevant to our project's objectives. This strategic choice enhances the potential for our automatic documentation generator to produce accurate and meaningful header comments across a wide range of Python code functions.

4 Methodology

This study seeks to build on the existing approaches in automatic code documentation generation by evaluating the performance of state-of-the-art models such as CodeBERT, CodeBERTa, UniXcoder (Guo et al., 2022), Codex, CodeT5-base (Wang et al., 2021), and other code-centric open-source language models on automatic code documentation generation tasks. Performance of the models is measured using the BLEU metric. Additionally, the model performance is compared with the baselines provided. The model described in the CodeSearchNet challenge performs an inverse task, where the model is trained on (code, comment) pairs and is tested on retrieval of appropriate code for a given natural language query which acts as a comment. As shown in Figure 1, they have an encoder for the query/comment itself, and each language gets its own encoder. This is followed by pooling and matrix multiplication of the result from the query and the concatenated result from the language-specific encoders. This is followed by a softmax and matrix comparison to get the final result on whether the predicted (code, comment) is a match or not. Additionally, we propose a novel approach involving multi-agent systems to break down the task of documentation generation into smaller subtasks, such as variable extraction, abstraction, and high-level function description generation. Through these efforts, we aim to further automate the code documentation process, making it more scalable and effective across various programming environments.

Furthermore, we propose the creation of a multi-agent system for generating documentation by decomposing the task into subtasks such as variable extraction, abstraction, and high-level function description generation.

Initial experimentation with CodeBERT was performed by providing a Python function along with three possible descriptions. For example, given the function:

```
def findSynonyms(self, word, num):
    if not isinstance(word, basestring):
        word = _convert_to_vector(word)
    words, similarity = self.call("
        findSynonyms", word, num)
    return zip(words, similarity)
```

the model assigned a higher similarity measure score (dot product) to the correct description, *"Find synonyms of a word"* (a score of 0.486), compared to the two incorrect alternatives, *"Find a word in a list"* (score: 0.127) and *"Find antonyms of a word"* (score: 0.341). Meanwhile, UniXcoder generated the following top three summaries for the same function: `['find synonyms', 'Find synonyms']`. Codet5-base, on the other hand, produced the summary: *"Find synonyms for a given word."* Overall, all models seem to assign or output a reasonable description for the given testing Python function.

Fine-Tuning Process

We fine-tuned some pre-trained models on a code summarization task. The fine-tuning process involved adjusting the model's parameters to improve performance on our specific task of generating summaries or docstrings for code functions.

- **Dataset Preprocessing:** We began by filtering the dataset to include only those rows where the length of the `func_documentation_tokens` (i.e., the function's docstring) was below a predefined threshold. This step reduced the dataset size, making it more manageable given our computational constraints. After filtering, we tokenized the function code and its corresponding docstring using a suitable tokenizer, ensuring that the input and output sequences were padded and truncated to appropriate lengths for the model.

- **Training Process:** Each model was fine-tuned on the preprocessed dataset for a fixed number of epochs. During training, we used the `Trainer` class from the Hugging Face `transformers` library to handle the optimization and evaluation processes. Dynamic padding was applied to minimize memory usage, allowing for variable-length sequences

in each batch. The training setup also included periodic model saving and logging for progress tracking and checkpointing.

- **Computational Challenges:** The fine-tuning process was computationally intensive, requiring substantial GPU power and memory. Despite leveraging high-performance GPUs, we faced several challenges, including frequent crashes during training due to resource constraints. This led us to experiment with different server configurations and training setups. Ultimately, we reduced the number of training epochs and further subset the dataset to alleviate the strain on the system. These adjustments allowed us to complete the fine-tuning process within the available resources, though at the cost of using less training data and fewer epochs.

- **Final Model:** After successfully fine-tuning the models, we saved both the model and tokenizer for subsequent evaluation. The fine-tuned models were then used to generate code summaries, which were evaluated using standard metrics, such as BLEU, to assess their performance.

5 Model Evaluation

Automatic evaluation metrics are essential for machine translation (MT) systems, providing a cost-effective alternative to human evaluations. The selection of an appropriate metric depends on factors such as the dataset used for training and testing, the target language, and the models employed for code generation and summarization. In this study, we evaluate our models using the BLEU metric.
BLEU (Papineni et al., 2002) is one of the oldest and most widely used corpus-level metric for evaluating MT systems. It uses a modified n-gram precision measure, where shorter sentences are penalized when compared to reference sentences. A sentence-level BLEU score calculation was proposed by Chen and Cherry (2014). The BLEU score is defined as:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right);$$

$$\text{BP} = \min\left(1, e^{1-\frac{r}{c}}\right)$$

where *BP* is the brevity penalty, *r* is the length of the reference string, *c* is the candidate translation length. p_n is the weighted overlap between the bag of n-grams for reference S^n_{ref} and candidate S^n_{can}, i.e.,

$$p_n = \frac{\left|S_n^{\text{ref}} \cap S_n^{\text{can}}\right|}{\left|S_n^{\text{ref}}\right|}$$

and w_n are the weights for various n-gram contributions. The BLEU scores range between 0 and 1 where higher scores represent better n-gram precision.

6 Result

We focused on BLEU due to its historical relevance and simplicity in capturing n-gram precision for the code summarization and MT task. We compared the performance of the pre-trained model and the fine-tuned model on the test dataset.
The BLEU scoring script used in our evaluations is adapted from (Koehn et al., 2007), providing an efficient framework for implementing the BLEU metric. This script systematically processes key components of BLEU computation, including text normalization (e.g., tokenization and punctuation handling), n-gram counting, handling multiple reference sentences, candidate sentence evaluation, and final score aggregation. The modular design allows for precise and consistent metric computation, ensuring alignment with the core principles of BLEU while supporting a streamlined workflow to evaluate model outputs.
Due to computational constraints, we opted to evaluate our models on a filtered subset of 4,000 test samples instead of the original test dataset of 22,176 samples. This subset consists of examples with docstring token lengths of 10 or fewer tokens. We believe that this reduced dataset is sufficiently representative for assessing model performance while maintaining computational feasibility.

Modeling Pipeline

- **Dataset Preparation:** We used the Python subset of the CodeSearchNet dataset, loading both the training and test splits. From the test dataset, we applied a filtering criterion to select examples where the length of the `func_documentation_tokens` (tokenized docstring) was less than or equal to 10. This filtering resulted in a subset of over 11,000 samples for evaluation. To create a more manageable test set, we further reduced this subset to 4,000 samples for final testing.

- **Model Setup:** We evaluated two versions of each chosen model:
  - A *base pre-trained model*.
  - A *fine-tuned model*, trained on our specific code summarization task, loaded from a custom model directory.
- **Evaluation Procedure:** For each test sample, the model was given a function's code (with any existing docstring removed) as input. The model generated a predicted docstring, which was compared to the actual docstring using the BLEU metric. The BLEU score was calculated for each test example using the *n-gram precision* approach. We separated the function code with its corresponding docstring by implementing a custom function to process each row of the test dataset. Inputs were tokenized using the appropriate tokenizer for the respective model, and predictions were generated using the model's `generate()` method. BLEU scores were computed using a script adapted from (Koehn et al., 2007).
- **Results Recording:** The generated docstring, BLEU score, and other relevant details were stored in the dataset for both the base pre-trained and fine-tuned models. Summary statistics (e.g., average, median, and standard deviation of BLEU scores) were calculated to provide an overview of the model's performance.
- **Findings:** The evaluation results highlight differences in the BLEU scores between the base and fine-tuned models, offering insights into the benefits of task-specific fine-tuning. The use of a representative 4,000-sample subset allowed us to achieve a meaningful comparison while managing computational resources effectively.

This structured approach ensures the reliability and reproducibility of our evaluation while aligning with the computational constraints of the project.

CodeT5-base Fine-tuned Result:

At first glance, our fine-tuned CodeT5 model appears to show a slight decrease in BLEU score performance, with an average score of 22.4 compared to 22.83 for the base model. Additionally, the median BLEU score for the fine-tuned model (15.35) is lower than that of the base model (17.86), indicating that performance improvements may not be consistent across all test cases. This finding is further supported by the paired t-test, which shows that the difference in BLEU scores between the two models is not statistically significant (p-value = 0.21). Moreover, the fine-tuned model exhibits a higher standard deviation (26.85) than the base model (18.68), suggesting greater variability in performance. This variability implies that while the fine-tuned model may excel on certain examples, it may struggle with others, potentially due to overfitting or instability.

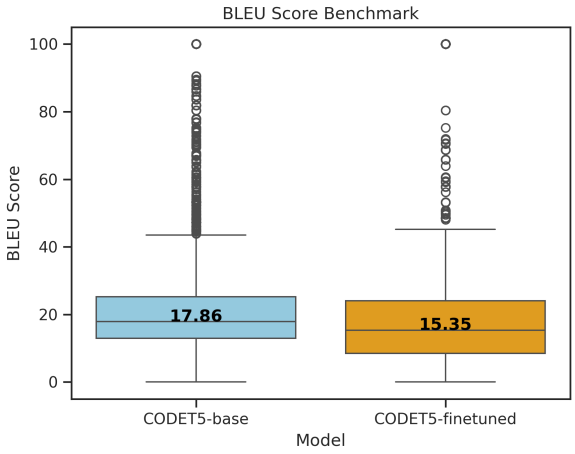

Figure 2: BLEU score benchmark between the base and the fine-tuned CodeT5 model.

Analyzing the test dataset and the generated outputs from both models reveals additional challenges. A key issue stems from the subjective nature of the docstring labels, which serve as the reference for computing the BLEU score. These docstrings often reflect brief mental notes or shorthand comments from function authors, which may not always align with the function's actual purpose. Examples such as "Instance depends on the API version:", "Deprecated after 0.8", or "unary : id '(' expression ')'" highlight this ambiguity, as they can be unrelated to the function's true behavior.

What makes this machine translation task particularly challenging is that it requires mapping between two distinct languages: machine code and human natural language. This mapping is difficult due to the lack of standardized rules, grammar, or post-processing methods for user-generated docstrings. Function authors may use abbreviations or personal notations in their descriptions, such as "viz" for "visualization" or "rtype" for "return type," which further complicates the task. As a

result, the docstring sets cannot be considered as a definitive "translation rulebook" to evaluate model performance against, making it difficult to assess the model's output through traditional metrics like BLEU.
An interesting observation is the difference in the length and conciseness of the outputs generated by the base and fine-tuned models. This discrepancy is a direct result of the fine-tuning process, where the model was specifically trained on Python functions with docstrings of 10 tokens or fewer. As a result, the fine-tuned model generates shorter and more concise outputs, in contrast to the base model's longer, more descriptive phrases.

Base Output	Fine-Tuned Output
Add a help section to the command line parser.	Add help section
Set the public key of the certificate.	set pubkey
Queue a message for sending.	Queue message
Create a queue with the specified parameters.	Create queue
Delete a queue from the service bus.	Delete queue
This function returns the action description of a given action in a given thing .	Get a thing
Add an event to the list of events .	add event

Table 1: Comparison of Base and Fine-Tuned Model Outputs

This table provides a side-by-side comparison of the outputs from the base and fine-tuned models, highlighting how the fine-tuned model tends to produce shorter and more concise text, reflecting the training data on which it was fine-tuned.

SEBIS Code Trans T5 Base Code Documentation Generation

We developed a custom model based on the T5 architecture, utilizing the `T5ForConditionalGeneration` class from the Hugging Face Transformers library. The model features an encoder-decoder structure with shared embeddings and comprises multiple layers of self-attention and feed-forward networks in both the encoder and decoder stacks.
With 222,903,552 trainable parameters the encoder consists of 12 layers (`T5Block`) with self-attention mechanisms and feed-forward networks, each layer containing a `T5LayerSelfAttention` and a `T5LayerFF` module. The decoder mirrors this structure but includes cross-attention layers (`T5LayerCrossAttention`) to attend to the encoder's output.
We trained the model on the Python subset of the CodeSearchNet dataset, focusing on generating accurate and contextually appropriate docstrings for Python functions. The training process involved optimizing the model to minimize the difference between the generated docstrings and the reference documentation provided in the dataset.
After training, this model achieved a BLEU score of 16.84

7 Discussion

The task of machine translation for code summarization, where models are trained to generate human-readable descriptions (docstrings) of code, presents several unique challenges. While the task shares some similarities with traditional machine translation, such as mapping one language (machine code) to another (natural language), it also introduces additional complexities that make it distinctly difficult.

- **Subjectivity and Ambiguity in Docstring Labels:** Docstrings are often concise and informal, varying in style, detail, and format. This inconsistency complicates the model's ability to learn accurate summarization patterns. The lack of standardized conventions and grammar for writing docstrings introduces ambiguity, making it difficult to assess model output using traditional metrics like BLEU.

- **Main Challenges in Code Summarization as a Machine Translation Task:**
  - *Lack of Defined Grammar:* Code lacks natural linguistic structures, making it difficult to map programming constructs to natural language phrases. This requires both syntactical understanding and contextual comprehension.
  - *Code Context and Intent:* Grasping the programmer's intent is challenging as functions may have subtle behaviors or side effects, necessitating an understanding of the broader code context.
  - *Diversity in Docstring Style and Detail:* Docstrings vary in length, style, and detail, creating inconsistency in the training data and making it harder for models to generalize effectively.

- **Intensive Computational Resources for Fine-Tuning and Evaluation:** Fine-tuning LLMs on large datasets requires substantial computational resources, including GPUs/TPUs for parallel processing and large memory

capacities. Evaluating performance using metrics like BLEU on extensive datasets adds to the resource load. Additionally, hyperparameter tuning and batch evaluation increase the computational cost, making this process expensive, especially for smaller research labs.

Evaluation techniques. We evaluated our models using the BLEU score and conducted a paired t-test to measure the difference between the performance of the base model with the fine-tuned model. The results revealed that the fine-tu ned model achieved a lower BLEU score than the base model, and the paired t-test revealed that the difference was not statistically significant. This outcome suggests that the BLEU metric may lack the sensitivity needed to effectively evaluate and differentiate between the two models in code generation and summarization tasks (Wei et al., 2019; Tran et al., 2019; Mathur et al., 2020a; Roy et al., 2021; Evtikhiev et al., 2023).

To address the limitations of BLEU, alternative evaluation metrics have been proposed. One such alternative is ROUGE-L, a metric from the ROUGE family (Lin, 2004), originally developed for evaluating headline-like summaries and later adapted for other summarization tasks (Wei et al., 2019; Roy et al., 2021; Evtikhiev et al., 2023). ROUGE-L measures precision and recall based on the longest common subsequence (LCS) between the candidate and reference strings. Various studies (Wei et al., 2019; Roy et al., 2021; Evtikhiev et al., 2023) have highlighted ROUGE-L as a superior alternative to BLEU for code-related tasks.

Another alternative is METEOR (Metric for Evaluation of Translation with Explicit Ordering) (Banerjee and Lavie, 2005), which, like ROUGE-L, incorporates both precision and recall but balances them using a harmonic mean. METEOR has been cited in several studies (Wei et al., 2019; Evtikhiev et al., 2023; Roy et al., 2021) as being closer to human evaluation than BLEU. Additionally, it is capable of calculating both corpus-level and summary-level scores, making it a versatile evaluation tool.

The last alternative to BLEU scores is chrF (Popović, 2015) which stands for character n-gram F-scores and is a character-based metric that doesn't depend on tokenization rules. It was developed by Popovic, 2015 and takes into consideration- each character in the candidate and reference set except for spaces. Studies comparing the evaluation metrics for various NLP tasks indicate that chrF is a better metric than BLEU for code generation and summarization tasks within the MT system (**?**Roy et al., 2021; Evtikhiev et al., 2023).

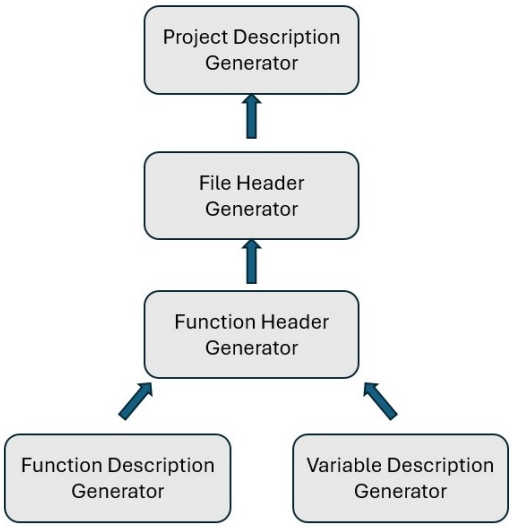

Figure 3: Example Multi-agent structure

Multi-agent system. A custom model for generating function headers can be integrated into a multi-agent framework for a more complete application. In our experiment, we used LangChain to first generate function headers for each code snippet and defined GPT as another chat-based tool to generate a project description or file header from multiple function headers. Here, depending on the setup, we can have different multi-agent structures where each component can be switched out by a custom model trained for the purpose. One such example can be seen in 3, out of which we only implemented the function header generator with the file header generator in our experiment.

8 Future Work

The future work should focus on three key areas: modeling approaches, evaluation techniques, and multi-agent implementation.

Modeling approaches

DietCodeBERT. DietCodeBERT (Zhang et al., 2022) is a lightweight, simplified version of CodeBERT for code summary and code search. It uses the same data from CodeSearchNet and uses 3 different strategies to simplify the input code for CodeBERT. DietCodeBERT uses word dropout, frequency filtering, and an attention-based strategy that selects statements and tokens that receive the

most attention weights during pre-training. Given the large amounts of data and time taken for training, this is an option worth exploring.

LAMNER. LAMNER (Sharma et al., 2022), on the other hand, takes a different approach by enhancing the representation of code tokens through character-level language modeling and named entity recognition (NER). LAMNER aims to address the challenge of encoding code semantics more effectively by focusing on the entire form of a code token and its structural properties. The use of NER helps identify and categorize various code constructs, providing richer contextual information that improves the quality of generated comments. This approach allows LAMNER to capture syntactic and semantic nuances in code, resulting in more precise and informative documentation. Through the fusion of these techniques, LAMNER shows potential for outperforming baseline models in generating high-quality code comments.

CodeRL. CodeRL (Le et al., 2022) is a novel framework designed for program synthesis that leverages the power of pre-trained language models combined with deep reinforcement learning. Unlike conventional approaches that rely solely on supervised fine-tuning, CodeRL introduces a critic network during the generation process, which evaluates the functional correctness of generated code based on feedback from unit tests. This feedback allows the model to refine and regenerate code until it meets the desired specifications. By incorporating this iterative feedback loop, CodeRL aims to overcome the limitations of standard models, which often struggle with complex and unseen coding tasks. The model's ability to self-correct during inference holds the potential to improve the quality and robustness of code documentation generation.

Evaluation techniques

While BLEU scores are the most widely used scores in MT system evaluation, various recent studies (see discussion section) have indicated that BLEU scores may not be the best metric to evaluate models on code generation and summarization (Wei et al., 2019; Tran et al., 2019; Mathur et al., 2020b; Roy et al., 2021; Evtikhiev et al., 2023). This would necessitate the incorporation of other evaluation metrics such as ROGUE-L, METEOR and chrF for improved performance of our MT system.

Multi-agent system implementation

Following experimentation and empirical evaluation of different models presented in this study, they can be integrated into a multi-agent system for code documentation generation. This system could be organized as a modular workflow, with each model handling the specific tasks it performs best, such as generating codebase summaries, creating function headers, providing variable descriptions, and so on.

9 Conclusion

The task of machine translation for code summarization presents unique challenges due to the inherent differences between programming languages and natural languages, as well as the subjectivity and variability in docstring conventions. This study highlights the complexities of mapping code to human-readable descriptions, emphasizing the importance of addressing ambiguities, inconsistencies, and computational demands in this domain.

Our implementation code space can be found here

Our experiments demonstrate that while BLEU is a widely used metric, it may not effectively capture the nuances of code summarization and generation tasks. The lack of statistical significance in performance differences between the base and fine-tuned models underscores the limitations of BLEU in distinguishing between models. Alternative metrics such as ROUGE-L, METEOR, and chrF offer promising avenues for more sensitive and reliable evaluation, as they better align with the characteristics of code-related tasks and human judgment.

Additionally, the integration of models into a multi-agent system introduces opportunities for modular workflows tailored to specific subtasks, such as generating function headers and file-level summaries. By leveraging tools like LangChain and GPT, the system can scale to accommodate more sophisticated pipelines, where individual components can be replaced or enhanced with custom-trained models. This modular approach fosters adaptability and extensibility, paving the way for more robust and efficient solutions in code documentation generation.

In conclusion, while significant progress has been made, further research is required to refine evaluation metrics, address computational challenges, and explore modular, multi-agent implementations for advancing machine translation in code summarization.

References

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Nathanael Chambers and Dan Jurafsky. 2011. Template-based information extraction without the templates. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pages 976–986.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

David Goodger and Guido van Rossum. 2001. PEP 257 – docstring conventions. Python Enhancement Proposal. https://peps.python.org/pep-0257/.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Elias Iosif and Alexandros Potamianos. 2009. Unsupervised semantic similarity computation between terms using web documents. *IEEE Transactions on knowledge and data engineering*, 22(11):1637–1647.

Thorsten Joachims et al. 1997. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. In *ICML*, volume 97, pages 143–151. Citeseer.

Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using gpt-3. *Preprint*, arXiv:2209.02235.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic. Association for Computational Linguistics.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Preprint*, arXiv:2207.01780.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384.

Nitika Mathur, Timothy Baldwin, and Trevor Cohn. 2020a. Tangled up in bleu: Reevaluating the evaluation of automatic machine translation evaluation metrics. *arXiv preprint arXiv:2006.06264*.

Nitika Mathur, Timothy Baldwin, and Trevor Cohn. 2020b. Tangled up in BLEU: Reevaluating the evaluation of automatic machine translation evaluation metrics. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4984–4997, Online. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Maja Popović. 2015. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395.

Xiaojun Quan, Gang Liu, Zhi Lu, Xingliang Ni, and Liu Wenyin. 2010. Short text similarity based on probabilistic topics. *Knowledge and information systems*, 25:473–491.

Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple bm25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 42–49.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1105–1116.

Rishab Sharma, Fuxiang Chen, and Fatemeh Fard. 2022. Lamner: Code comment generation using character language model and named entity recognition. *Preprint*, arXiv:2204.09654.

Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *International journal of machine learning and cybernetics*, 1:43–52.

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1073–1084, New York, NY, USA. Association for Computing Machinery.

Tingwei Zhu, Zhong Li, Minxue Pan, Chaoxuan Shi, Tian Zhang, Yu Pei, and Xuandong Li. 2024. Deep is better? an empirical comparison of information retrieval and deep learning approaches to code summarization. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–37.