[Autonomous Code Summarization Using LLMs]

[Literature Review]

In recent times, large, pre-trained, language models had shown high potential in several tasks. Such as Question Answering, Sentiment Analysis, Abstractive Summary etc. Some of the important models include [ELMo (Peterset al., 2018)], [GPT (Radford et al., 2018)], [BERT (Devlin et al., 2018)], [XLNet (Yang et al., 2019)], and [RoBERTa (Liu et al., 2019)].

They all follow the base architecture proposed by Vaswani et. al. in their Seminal Paper: [Attention is All You Need (Vaswani et al., 2017)]. All these LLMs are then used to solve the challenging problem of **Code Summarization**.

Here, are the important research studies and their findings:

- Code Summarization (Open-source): An end-to-end pipeline using Hugging Face Transformers Library, Microsoft's Open-Source Large Scale code model Code **BERT**, and Codist tree-hugger. Based on the naturalness hypothesis of source code, proposed by Allamanis et. al. It is thus preferable to try to treat large code corpora in a similar fashion and exploit their Statistical Properties.

- CodeTrans (submitted on 6 april,2021): An **encoder-decoder transformer model** for tasks in the software engineering domain, that explores the effectiveness of encoder-decoder transformer models for six software engineering tasks, including thirteen sub-tasks. Moreover, they have investigated the effect of different training strategies, including single-task learning, transfer learning, multi-task learning, and multi-task learning with fine-tuning.

  This is the specific link to the single task of Source Code Summarization.

- Few-Shot Training LLMs for Code Summarization (October,2022): In this paper, we investigate the use **few-shot training** with the exceptionally large GPT (Generative Pre-trained Transformer) Codex model and find evidence suggesting that one can significantly surpass state-of-the-art models for code-summarization, leveraging project-specific training.

  Important findings: - **Codex** outperforms all finetuned foundation models CodeT5, CodeBERT, GraphCodeBERT, Polyglot CodeBERT, and PolyGlotGraphCodeBERT in

all six programming languages, even though the fine-tuned models are trained with thousands of data.

- Deep is Better? An Empirical Comparison of Information Retrieval and Deep Learning Approaches to Code Summarization (Novemeber,2023): This paper presents the first large-scale empirical study of 18 IR, DL, and hybrid code summarization approaches on five benchmark datasets.

  DL models perform **better** in predicting tokens from method signatures and capturing structural similarities in code, simple IR approaches tend to perform better in the presence of code with high similarity or long reference summaries, and that existing hybrid approaches do not perform as well as individual approaches in their respective areas of strength.

  Important findings: - The study recommends adopting the **BM25-spl information retrieval (IR) technique** for code summarization due to its consistent and superior performance across different datasets.

Key Findings -

- ✓ **Existing Approaches:** Prior research has explored various methodologies for autonomous code summarization using Large Language Models (LLMs). Techniques range from traditional rule-based methods to more recent data-driven approaches.
- ✓ **Data-Driven Approaches:** Recent advancements focus on leveraging pre-trained LLMs, such as GPT-3, BERT, and RoBERTa, for code summarization. These models capture semantic understanding and context, enhancing their ability to generate meaningful summaries.
- ✓ **Evaluation Metrics:** Metrics like **BLEU, ROUGE, and METEOR** are commonly used to evaluate the performance of code summarization models. However, there is ongoing debate on the adequacy of these metrics in capturing the essence of code functionality.

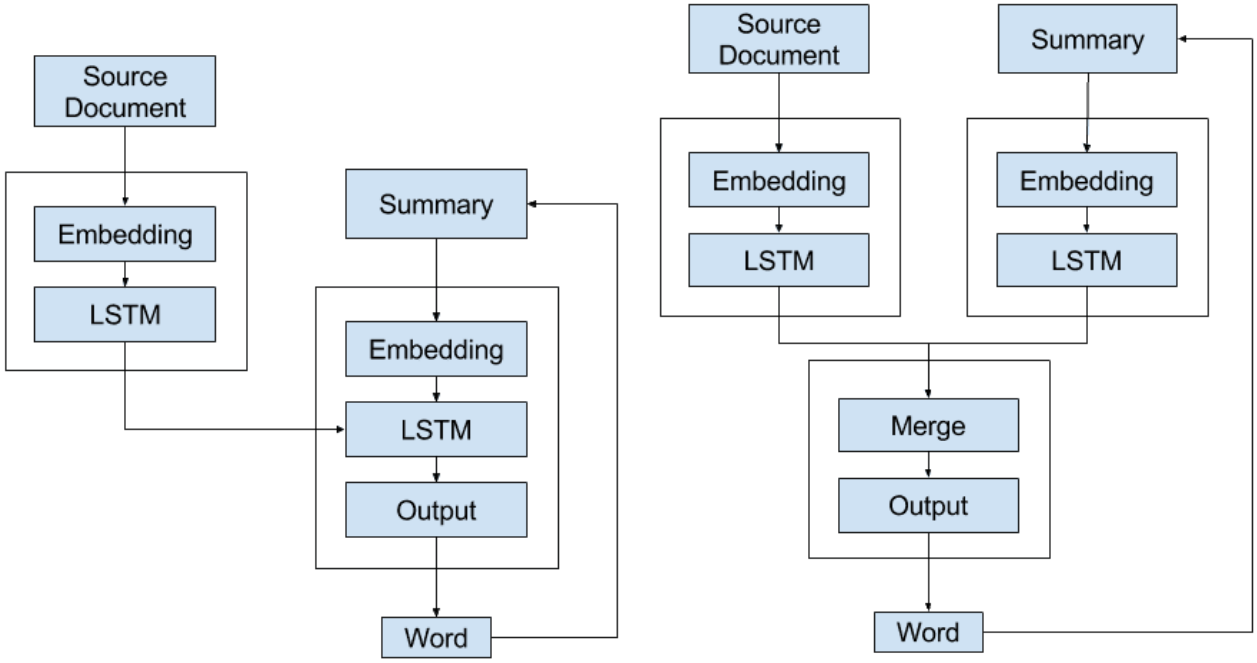
Challenges -

- ✓ **Code Ambiguity:** Code often contains implicit and context-dependent information, leading to ambiguity. LLMs may struggle to precisely capture such nuances.
- ✓ **Handling Code Variability**: Diverse coding styles and variations in code structures pose challenges. An effective model should be adaptable to these diversities.
- ✓ **Lack of Standardization:** The absence of a standardized dataset and evaluation metric complicates the comparison and benchmarking of different models.

[Approaches for Autonomous Code Summarization]

1. Encoder-Decoder Architectures:

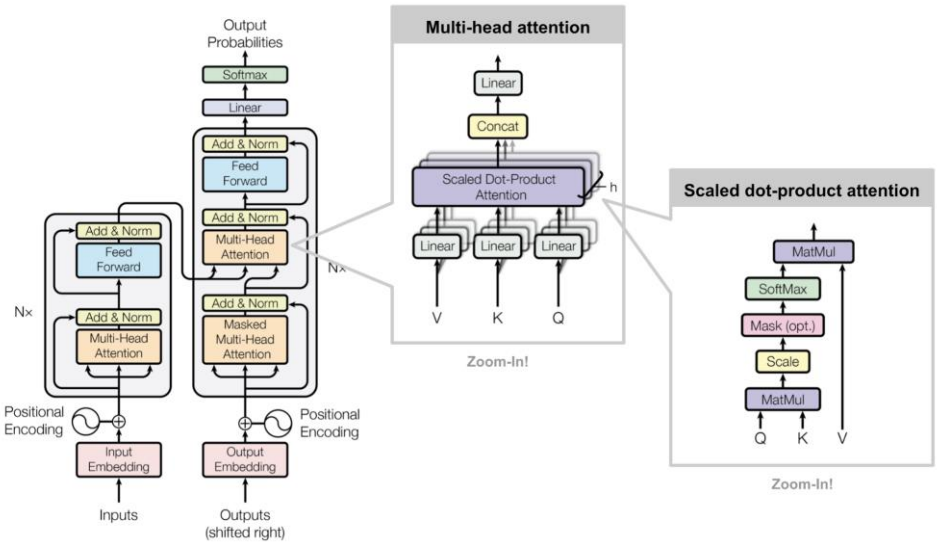
Utilize encoder-decoder architectures, where the encoder processes the code, and the decoder generates the summary. Attention mechanisms enhance the model's ability to capture dependencies.
Example - The architecture used in **CodeTrans** follows an encoder-decoder architecture.
Below are 2 types of Recursive Code Summarization Models in the form of flow diagrams.



2. Transfer Learning with LLMs:

Leverage pre-trained LLMs to benefit from their understanding of natural language. Fine-tune these models on code-specific datasets for enhanced performance in code summarization tasks. This is the method of our interest, as it is simpler to make and highly effective as compared to any other approach.

Example – **Code Summarization** (1st Literature Review) applies transfer learning. The full model architecture of the transformer. (Image source: Fig 1 & 2 in Vaswani, et al., 2017.)



3. Hybrid Models:

Hybrid models in the context of code summarization involve combining rule-based approaches with Large Language Models (LLMs) to capitalize on the strengths of both methods. The rationale behind this approach is to leverage the precision and explicit guidance provided by rule-based systems to handle code-specific nuances, while benefiting from the contextual understanding and flexibility offered by LLMs.
Example - Deep is Better? An Empirical Comparison of Information Retrieval and Deep Learning Approaches to Code Summarization provides a good insight and comparison of hybrid models performances. **Rencos and Re2Com** gave the best results in hybrid models.

Table 3. Evaluation metric scores for selected approaches on five cleansed datasets

	Approaches	TLC			CSN			HDC			FCM			PCSD		
		B	R	M	B	R	M	B	R	M	B	R	M	B	R	M
IR	BM25-spl	24.68	36.87	18.66	6.28	17.31	8.15	11.56	19.2	9.72	18.53	32.07	16.13	13.77	26.99	13.54
IR	BM25-ast	24.56	36.2	18.21	6.43	16.88	7.65	11.11	18.57	9.35	18.49	30.65	15.12	13.51	26.25	13.1
IR	BM25-alpha	23.91	36.25	18.44	6.07	16.84	8.01	10.48	18.66	9.52	18.07	31.43	15.93	13.63	26.72	13.43
IR	NNGen	23.35	33.99	16.88	5.77	15.66	7.04	11.22	17.91	8.83	17.61	29.74	14.08	11.93	23.44	11.3
DL	RNN-cos	23.68	35.48	17.71	5.54	15.67	7.0	8.58	18.95	8.39	9.63	25.81	11.38	12.13	25.17	12.05
DL	ast-attendgru	17.78	36.78	15.8	3.86	25.85	10.15	5.6	22.45	7.7	16.42	39.01	18.42	8.54	27.66	11.68
DL	code2seq	17.58	36.53	15.69	3.7	24.28	9.55	5.22	20.44	7.17	16.31	38.94	18.19	8.67	27.84	11.66
DL	Graph2Seq	17.55	36.32	15.6	3.1	23.82	9.11	6.11	21.69	7.94	15.85	38.41	17.82	8.78	27.64	11.71
DL	NCS	20.81	38.41	17.86	3.45	26.32	9.94	9.61	23.44	9.9	18.41	41.08	19.55	12.81	30.77	14.11
DL	AST-Trans	22.15	37.11	18.29	5.78	28.83	12.06	8.14	25.08	9.89	16.38	39.35	18.76	8.96	25.06	11.62
DL	SiT	20.79	39.34	18.25	5.68	27.2	11.17	9.43	25.33	10.25	15.17	38.12	17.17	12.32	30.35	13.82
DL	CodeBERT	20.04	40.86	19.03	5.99	30.91	12.65	11.34	27.93	12.0	18.43	40.96	19.54	13.16	28.55	15.46
DL	PLBART	23.58	**42.99**	**20.54**	8.37	33.82	14.52	11.48	28.06	12.24	19.76	41.78	20.28	**15.45**	**36.23**	**17.99**
DL	CodeT5	21.94	42.65	20.15	**8.98**	**34.71**	**14.89**	**12.53**	**29.64**	**13.31**	20.89	**42.71**	**21.07**	15.02	31.58	17.6
DL	StarCoder	5.3	26.88	11.19	6.34	28.22	12.41	5.38	24.46	9.27	11.22	32.43	15.7	8.91	31.87	14.98
Hybrid	Rencos	**25.35**	40.18	19.54	7.28	24.01	10.54	9.68	23.08	9.42	15.26	35.35	16.18	13.79	29.91	14.33
Hybrid	Re^{2}Com	20.73	37.4	17.2	6.64	26.57	11.23	11.78	22.01	9.72	**23.41**	39.97	20.03	10.04	26.02	11.94
Hybrid	EditSum	9.74	27.96	10.93	3.77	17.72	6.94	7.97	23.3	9.18	17.69	32.24	15.11	4.37	19.68	7.65

*B, R, M represent BLEU, ROUGE-L, and METEOR, respectively.

[Choice of LLMs]

1. **GPT-3 (Generative Pre-trained Transformer 3):** To be more specific **CODEX.** OpenAI Codex is a descendant of GPT-3; its training data contains both natural language and billions of lines of source code from publicly available sources, including code in public GitHub repositories. OpenAI Codex is most capable in **Python**, but it is also proficient in over a dozen languages including JavaScript, Go, Perl, PHP, Ruby, Swift and TypeScript, and even Shell. It has a memory of 14KB for Python code, compared to GPT-3 which has only 4KB—so it can consider over 3x as much contextual information while performing any task.

   Also, CODEX outperformed every other LLM in few-shot training method.

2. **BERT (Bidirectional Encoder Representations from Transformers) and RoBERTa (Robustly optimized BERT approach):** BERT and RoBERTa are bidirectional models that capture contextual information effectively by considering both left and right contexts. This **bidirectional nature** allows them to understand the relationships between various parts of the code and understand the dependencies and relationships within the code.

3. **T5 (Text-To-Text Transfer Transformer):** T5 introduces a unified framework where all **NLP** tasks, including code summarization, are framed as text-to-text tasks. This simplifies the training process and enhances model performance.

4. **PLBART:** PLBart is a multilingual encoder-decoder (sequence-to-sequence) model primarily intended for code-to-text, text-to-code, code-to-code tasks. This is a BART-like model which can be used to perform code-summarization, code-generation, and code-translation tasks. The pre-trained model plbart-base has been trained using multilingual denoising task on Java, Python and English. PLBART learns program syntax, style (e.g., identifier naming convention), logical flow (e.g., if block inside an else block is equivalent to else if block) that are crucial to program semantics and thus excels even with limited annotations.

[Implementation Strategy]

1. Function Identification: (Utilizing Static Code Analysis and NLP Techniques)

- Static Code Analysis:

  Approach: Employ static code analysis tools to analyze the codebase without executing it. Extract structural information, dependencies, and identify functions based on syntactic patterns.

  Tools: Use tools like **AST (Abstract Syntax Tree)** parsers to represent the hierarchical structure of the code, aiding in function identification.

  Benefits: Static analysis provides a non-intrusive way to understand the code's structure and can be used to identify functions, classes, and methods.

- Natural Language Processing (NLP):

Integration: Apply NLP techniques to analyze comments and documentation within the code. Identify natural language descriptions of functions, which can further assist in function recognition.

Named Entity Recognition (NER): Employ NER to identify function names and related entities within comments.

Benefits: NLP helps bridge the gap between code and natural language, contributing to a more comprehensive understanding of the codebase.

2. Summarization Process: (Implementation pipeline)

- Processing Identified Functions:

  Data Pipeline: Develop a pipeline that takes identified functions as input for summarization.

  LLM Integration: Implement mechanisms to send these functions through the selected LLMs (such as Codex, BERT, or others) for generating summaries.

  Tokenization: Tokenize the code and associated comments to feed into the LLM, ensuring the model understands both the code structure and natural language context. This is one of the most important parts of implementing our AI model.

- **Fine-Tuning LLMs:**

  Dataset Preparation: Curate code-specific datasets containing pairs of functions and their corresponding human-generated summaries.

  Fine-Tuning Process: Fine-tune the selected LLMs on this dataset to enhance their understanding of programming language semantics specific to code summarization tasks.

  Transfer Learning: Leverage pre-trained models and adapt them to the nuances of code summarization.

3. User Intervention: (Balancing Automation and User Feedback)

- Automation:

  Minimizing User Input: Design the tool to operate autonomously, minimizing the need for user intervention in the summarization process.

  Real-time Processing: Implement the system to process functions and generate summaries in real-time as code is analyzed.

- Optional Feedback Mechanism:

  User Feedback Loop: Allow for an optional feedback loop where users can provide input on the quality of generated summaries.

  Model Iteration: Use user feedback to iteratively improve the model, addressing challenges or nuances specific to the codebase.

- Continuous Improvement:

  Model Versioning: Implement a system for versioning models, allowing for seamless updates based on user feedback and ongoing improvements.

  Monitoring and Evaluation: Regularly monitor the tool's performance, assess user feedback, and use this information for continuous enhancement.

** Although the task is to minimize human interaction and analyze the code through the model only, sometimes USER Feedback could be useful option to keep.

[Handling Code Structure and Context]

This section is mostly based on the research paper - Summarizing Source Code from Structure and Context.

1. **<u>Abstract Syntax Tree (AST) and Root-to-leaf paths:</u>** The abstract syntax tree (AST) serves as a fundamental representation of source code, commonly used in compilers for token-to-tree conversion. This choice is driven by two key factors: first, the AST omits inessential components like punctuation, emphasizing the inherent code structure; second, it enables the model to grasp semantic context and consider long-range dependencies induced by the usage of elements across distant locations in the code. The AST provides a condensed and language-agnostic view of the code, highlighting both its structural nature and semantic implications. It consists of nodes, representing constructs in the source code, categorized as terminals (e.g., identifiers) and non-terminals (e.g., functions, loops), each denoted by type and value.

   The process involves traversing the Abstract Syntax Tree (AST) of a code snippet to simultaneously capture semantic and syntactic information. Root-to-leaf paths are generated by considering all paths between the root and terminals in the AST, representing sequences that fully expose the code's structural and semantic aspects. Each path contributes local semantic information (e.g., arguments) and preserves global structural interactions (e.g., functions). Leveraging root-to-leaf paths ensure accessibility to the AST for code summarization, providing both local and global context. The generated paths can be unambiguously reconstructed into the original AST, facilitating code representation learning and summarization.

   Example – Fig1. Source code in python and Fig 2. AST and Root-to-leaf paths for the code.

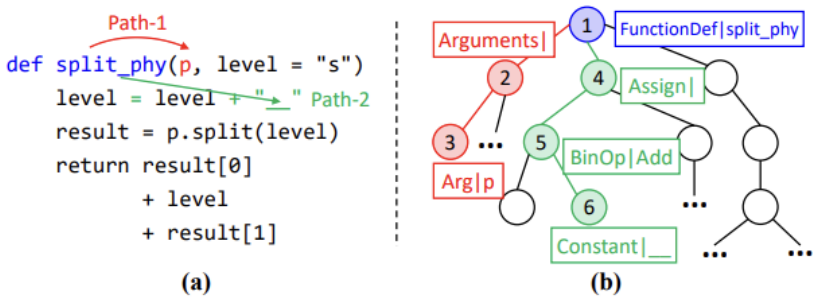


Fig. 1. An example of source code in Python and its AST, where each root-to-leaf path indicates a local semantic operation in the source code context.

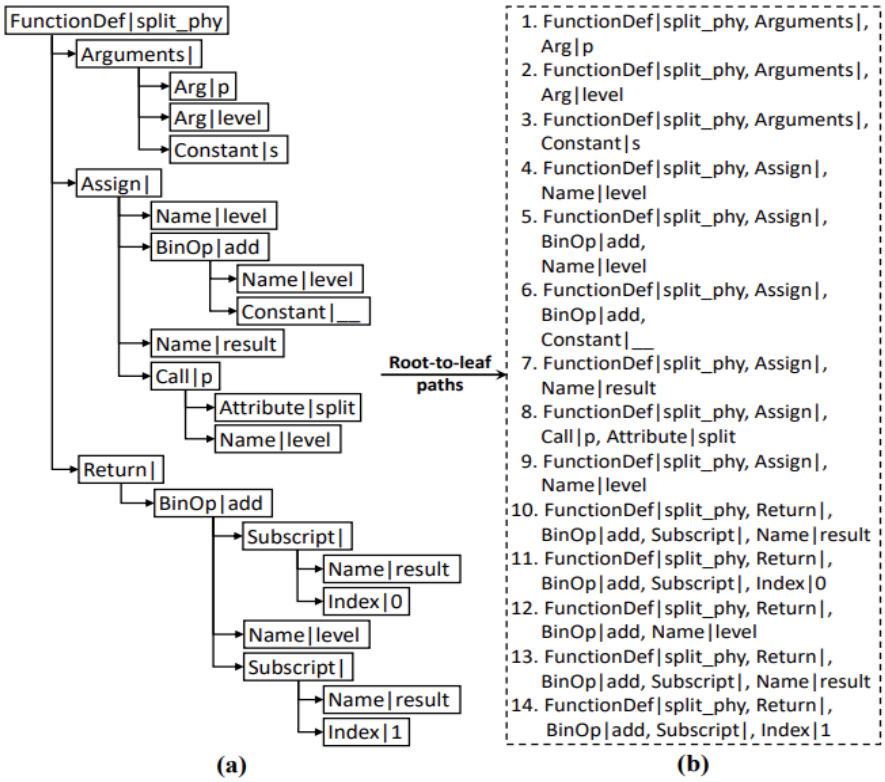


Fig. 2. AST for the code snippet in Python of Fig. 1(a), and its generated root-to-leaf paths.

2. **Tree-based Transformer Encoder [TreeXFMR]:** The TreeXFMR model adopts an encoder-decoder architecture for code summarization, focusing on mapping root-to-leaf paths of input code snippets to code representations. The model utilizes token and path representations, including root value embeddings and leaf value embeddings, to capture both structural and contextual information. Positional

encoding is employed for intra-path and inter-path information, ensuring the preservation of order within paths and among paths. A hierarchical attention mechanism is applied to highlight important content in both tokens and paths, contributing to effective code representation learning. The final output represents the code summary.

The decoder in the TreeXFMR model generates natural language words one at a time by paying attention to the code representation Z. It predicts each word based on Z and the words generated so far. The training involves minimizing cross-entropy loss, and during inference, a beam search decoder is used. Additionally, the model undergoes pre-training, where it learns to predict masked variables in the source code's root-to-leaf paths, enhancing its overall language generation abilities.

The conclusion is TreeXFMR is the best model considering its ability to understand the code structure and context.

Table 1. TreeXFMR test-loss and BLEU-4 comparison with other popular methods

COMPARISONS WITH BASELINES AND ABLATION STUDY

Method	Test Loss	BLEU-4 (%)
CODE-NN	2.93	17.37
Tree-LSTM	2.95	17.42
Code2Seq	2.88	18.86
CodeBERT	2.55	19.08
CodeBERT w/o pre-train	3.35	13.11
DeepCom	3.90	11.35
TreeXFMR w/o pre-train	3.24	13.45
TreeXFMR w/o node type path	2.73	18.29
TreeXFMR w/o node value	4.89	7.94
TreeXFMR	**2.23**	**20.16**

Fig 3. TreeXFMR model

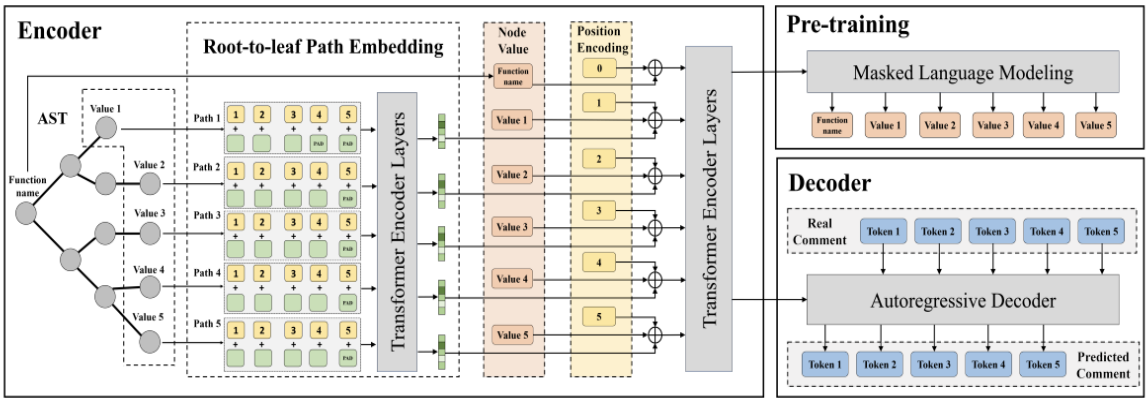


Fig. 3. The overview of our proposed code summarization model TreeXFMR.

[Challenges and Solutions]

- **Code Variability:**
  - *Challenge:* Codebases exhibit diverse styles, making it challenging for the model to adapt effectively.
  - *Solution:* Regularly update the model with diverse datasets to ensure adaptability to various coding styles.
- **Lack of Context Understanding:**
  - *Challenge:* The model may struggle to understand intricate programming language semantics.
  - *Solution:* Enhance model pre-training with code-specific tasks to improve its comprehension of programming language semantics.
- **Ambiguity in Code:**
  - *Challenge:* Code snippets may contain ambiguous constructs.
  - *Solution:* Incorporate feedback mechanisms, allowing users to provide clarification on ambiguous code snippets. Or just try to find new methods to eliminate human interaction.
- **Standardization:**
  - *Challenge:* Lack of standardized datasets and evaluation metrics for code summarization.
  - *Solution:* Advocate for the development of standardized datasets and evaluation metrics specific to code summarization tasks.