Objective:
Your task is to design and implement an autonomous AI tool that utilises the capabilities of Large Language Models (LLMs) to automatically generate concise summaries for each function within a given application's codebase. The primary goal is to enhance code comprehension without requiring additional input from the user.

Sources used in Research :

Link : A Code Summarization Approach for Object Oriented Programs

Link : Using Artificial Intelligence in Source Code Summarization: A Review

Link : On the Use of Automated Text Summarization Techniques for Summarising Source Code

Made by : SHRAVAN PARIKH *for* DEVZERY INTERNSHIP ASSIGNMENT TASK

Autonomous code documentation using AI.

Introduction to Autonomous Code Summarization

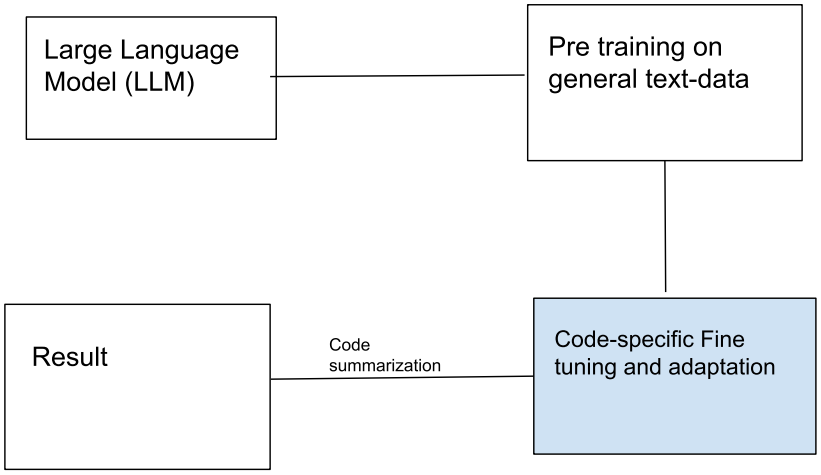
Definition and Significance

Autonomous Code Summarization involves the use of advanced techniques, particularly Large Language Models (LLMs), to automatically generate concise and contextually relevant summaries for functions within a codebase. The significance of this approach lies in improving code comprehension, aiding developers in understanding complex code structures, and facilitating faster debugging and maintenance.

It will improve software maintenance and other developers can easily work on prior developers work, which currently is a huge problem new developers sometimes take weeks just to understand the codebase and functions of different parameters involved which is a loss of important man power.

LLMs in Code Summarization

Large Language Models, such as GPT-3, BERT, Llama and Transformer-based architectures, have demonstrated remarkable capabilities in understanding and generating human-like text. When applied to code summarization, these models leverage pre-training on vast amounts of text data to capture semantic relationships, context, and syntactic structures in programming languages.



Utilisation of LLMs (GPT-3, BERT, Llama)

- LLMs' are great tools to summarise and we can utilise them by firstly adapting our code in a certain way that gives better understanding to the LLM aware of context. There are various techniques to make LLm aware of the context, one simple technique is using comments in the code.

Context-Aware Summarization:

- One key finding is the importance of context-aware summarization. Contextual understanding enables LLMs to consider the surrounding code when generating summaries. This approach ensures that the generated summaries are not only syntactically correct but also semantically meaningful, providing developers with a better grasp of the code's functionality.

Transfer Learning and Fine-Tuning:

- Research indicates that transfer learning, especially fine-tuning pre-trained LLMs on code-related tasks, is a crucial strategy for achieving high-performance in code summarization. By transferring knowledge gained from general language understanding to the specific domain of code, LLMs can adapt and produce accurate and relevant summaries.

Challenges in Autonomy

Diverse Code Structures:
Codebases can vary significantly in terms of structure, coding styles, and language constructs, making it challenging for an autonomous system to adapt universally.

Variable Codebase Context:
Understanding the specific context and purpose of code within different projects or domains poses a challenge. Contextual differences may lead to inaccuracies in generated summaries.

\

Handling Uncommon or Specialised Libraries:
Code often involves the use of specialised libraries or frameworks, and an autonomous system may struggle to comprehend and summarise functions that rely heavily on these external components.

Maintaining Coherence Across Multiple Functions:
Ensuring coherence in summaries when dealing with interconnected functions is challenging. The tool must understand and maintain logical connections between different parts of the codebase.

Challenges in Auto Code Documentation:

Dynamic Code Changes:
Frequent updates or changes in the codebase can lead to outdated documentation. Maintaining synchronisation between the code and its documentation

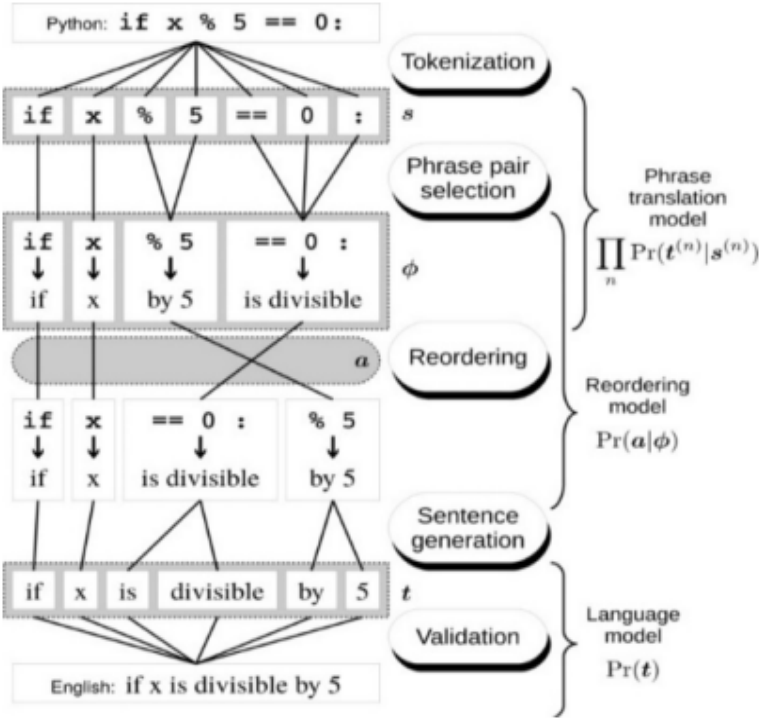
Lack of Detailed Inline Comments:
Some code may lack detailed comments or documentation, relying on the tool to infer and generate accurate summaries.
.
Ensuring Accuracy in Descriptive Summaries:
Generating descriptive and accurate documentation requires understanding the intricacies of the code's functionality, and inaccuracies may arise if the tool misinterprets complex logic or algorithms.

Handling Large Codebases:
For large codebases, generating comprehensive and concise documentation while avoiding information overload is a delicate balance. Ensuring that documentation remains useful and digestible is a challenge.

Code Summarization Techniques

Statistical Machine Translation-based Code Summarization:

Adopts Statistical Machine Translation (SMT) using Phrase-Based Machine Translation (PBMT).
Involves the identification of phrase pairs, reordering, and fluency assessment using a language model.
Challenges addressed by Tree to String Machine Translation (T2SMT) for hierarchical correspondence in the source code.



Python to English pseudo-code generation using PBMT technique |

Deep Reinforcement Learning-based Code Summarization

Utilises deep reinforcement learning, specifically the Actor-Critic network.
Employs a hybrid code representation using Long Short-Term Network (LSTM) for sequential tokens and Abstract Syntax Tree (AST) for structural information.
Resolves exposure bias through actor-critic network predictions.
Evaluated based on Bilingual Evaluation Understudy (BLEU) score.

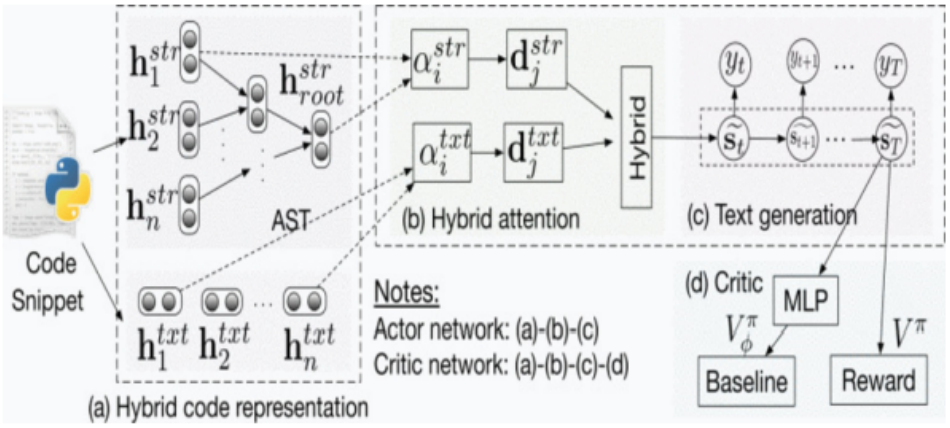


Figure 1. Overview of proposed code summarization framework [13].

Code Summarization based on Natural Language Generation :

Utilises Natural Language Generation (NLG) to translate Java code into English descriptions.
Applies PageRank algorithm to identify important methods and Software Word Usage Model (SWUM) to extract keywords.
Employs a custom NLG technique to generate English descriptions based on contextual information.

Table 1. Types of messages

Message Type	Explanation
Quick Summary Message	This is short sentence which gives description of the function.
Return Message	Return type of the method is given by this type of message.
Importance Message	It shows the importance of the method based on PageRank.
Output Used Message	This is to describe maximum 2 methods which calls this method.
Call Message	This describes maximum 2 methods which is called by this method.

Source Code Markup Language (SrcML) based Code Summarization :

Converts source code to XML using SrcML and generates a document file.
Focuses on core components using XML tags (e.g., <class>, <function>) for description.
Considers attributes, conditions, calls, and functions for feature extraction.
Generates comments based on the source code and related information.

Regarding "Maintaining Coherence in Autonomously Generated Summaries," this concept is not explicitly discussed in the provided contexts. If you have specific questions or need clarification on coherence maintenance, feel free to provide more details.

Fig. 3. Part of the Xml file after conversion using srcml.

Choice of LLMs:
The choice of Language Models (LLMs) for autonomous code summarization depends on various factors Here's a comparison and contrast of different LLMs based on their architectures, pre-training techniques, and adaptability to autonomy constraints:

BERT (Bidirectional Encoder Representations from Transformers):

Architecture: BERT employs a transformer architecture, allowing bidirectional understanding of context by considering both left and right contexts.
Pre-training: BERT is pre-trained on a large corpus using masked language modeling, where it learns to predict missing words in sentences.
Adaptability: BERT can be fine-tuned for specific tasks, making it adaptable to code summarization. Its bidirectional nature helps capture contextual information effectively.

GPT (Generative Pre-trained Transformer):

Architecture: GPT uses a transformer architecture but is autoregressive, predicting the next word in a sequence based on the context of the preceding words.
Pre-training: GPT is pre-trained by predicting the next word in a sentence, allowing it to generate coherent and contextually relevant text.
Adaptability: GPT models are versatile and can be fine-tuned for various tasks, including code summarization. They excel in generating human-like text.

XLNet:

Architecture: XLNet combines bidirectional and autoregressive approaches, addressing some limitations of BERT and GPT.
Pre-training: It employs permutation language modelling, considering all permutations of a sequence to predict missing tokens.
Adaptability: XLNet is adaptable and suitable for tasks like code summarization, benefiting from bidirectional context understanding and autoregressive generation.

RoBERTa (Robustly optimised BERT approach):

Architecture: Based on BERT, RoBERTa optimises training techniques and removes certain limitations, leading to improved performance.
Pre-training: RoBERTa utilises masked language modelling, similar to BERT, but with additional optimizations.
Adaptability: RoBERTa can be fine-tuned for specific tasks, and its robust training approach makes it suitable for code summarization.

Conclusion:

- For code summarization, the bidirectional nature of BERT and the autoregressive generation of GPT provide valuable contextual understanding.

- XLNet's combination of bidirectional and autoregressive approaches might offer a balance between the strengths of BERT and GPT.

- RoBERTa, with its robust optimization, can be considered for improved performance.

Implementation Strategy for Autonomous AI Code Summarization:

Data Collection and Preprocessing:

Collect Codebase: Gather a diverse set of code snippets and projects to create a comprehensive dataset.

Preprocessing: Tokenize code into meaningful elements, handle special characters, and create an organised dataset.

Architecture Selection:

Choose LLM: Based on the requirements, select a suitable Language Model for code summarization (e.g., BERT, GPT, or XLNet).
Fine-tuning the Language Model:

Dataset Preparation: Create a labelled dataset with code snippets paired with their corresponding summaries.

Fine-tuning: Train the selected LLM on the dataset, optimising it for code summarization.

Function Identification:

AST Parsing: Implement Abstract Syntax Tree (AST) parsing to analyse the code structure.
Identify Functions: Utilise AST to autonomously identify functions, methods, and their relationships within the code.

Context-aware Code Representation:

Hybrid Representation: Enhance code representation by combining sequential tokens (e.g., using LSTM) and structural information (e.g., using AST).
Embedding Generation: Create embeddings that capture both syntactic and semantic aspects of the code.

Summarization Generation:

Autonomous Inference: Implement a mechanism for the tool to autonomously analyse code snippets, identify functions, and generate summaries without user intervention.
Utilise Language Model: Leverage the fine-tuned LLM to generate coherent and contextually relevant summaries for identified functions.
Evaluation and Refinement:

Quality Metrics: Implement evaluation metrics (e.g., BLEU score) to assess the generated summaries' quality.

Feedback Loop: Establish a feedback loop for continuous learning and refinement, allowing the tool to adapt and improve over time.
User Interface (Optional):

Integration: If necessary, integrate the summarization tool with a user interface for user-friendly interaction.
Feedback Mechanism: Implement a mechanism for users to provide feedback on generated summaries for further improvement.
Testing and Validation:

Unit Testing: Conduct thorough unit testing to ensure the correctness of each component.
Validation: Validate the tool on diverse codebases to assess its generalizability and performance.
Deployment:

Integration with Development Environments: Integrate the tool with popular development environments to facilitate easy adoption.
Continuous Monitoring: Implement monitoring mechanisms for continuous performance evaluation and updates.

One High level strategy from my reading is :

Designing a Code Writing Template:

- Create a standardised code writing template with a simple structure.
- Ensure the template is designed to facilitate future documentation needs.
- Align the code writing style with the characteristics that suit the fine-tuned Language Model (LLM).

Fine-Tuning the Language Model:

- Train the LLM with data specifically tailored to the designed code writing template.
- Emphasise a balanced approach during fine-tuning to optimise the model for both training and generating data.

Two-Phase Code Base Processing:

Phase 1: Comment Generation

Train the LLM to focus on generating comments for each section of the code.
Leverage the fine-tuned LLM to ensure generated comments align with the chosen code writing style.

Phase 2: Summarizer/Document Generation

Implement a sequential process: first, generate comments, and then create summaries.

Utilise the LLM's proficiency in comment generation as a foundation for effective summarization.
Consider employing a separate LLM specialised in summarization to process and enhance the generated comments.

Effectiveness:

- Leverage the fine-tuned LLM's capability for comment generation to feed into the summarization phase.
- Utilize another LLM dedicated to summarization to enhance and condense the generated comments into comprehensive summaries.
- This strategy aims to capitalise on the strengths of the LLM by tailoring its training to a specific code writing style and leveraging its sequential abilities for effective comment and summary generation.

Handling Code Structure and Context:

Context Preservation:

Mechanisms for Retaining Surrounding Code Context:

- Utilise the hybrid code representation approach discussed in the source code summarization methodologies.

- Leverage the encoder-decoder framework with LSTM for sequential tokens and AST for structural information.

- This method captures both sequential and structural contents, preserving the broader context of the code.

Relationship Understanding:

Semantic Analysis for Identifying Dependencies:

- Draw insights from the deep reinforcement learning-based code summarization.

- Utilise AST to effectively capture the semantics of the code snippet.

- Leverage semantic analysis techniques to identify dependencies between variables and functions.

Algorithms for Recognizing Connections Between Functions:

- Build upon the natural language generation (NLG) technique discussed in the methodologies.
- Implement PageRank algorithm to identify the importance of methods and understand their relationships.

- Use SWUM to extract keywords and recognize connections between important methods.

Coherence Maintenance:

Integration of Coherence Checks:

- Extend insights from the NLG technique for coherence maintenance.
- Implement coherence checks to ensure consistent language and terminology in function summaries.
- Utilise NER to identify and maintain coherence in the naming of entities across functions.

Techniques for Handling Variations in Code Structures:

- Consider the Tree Convolutional Neural Network (Tree CNN) approach for source code summarization.

- Develop adaptive models capable of handling variations in coding styles using hierarchical representations.

- Train the LLM to recognize and adapt to different code structures as discussed in the methodologies.

Conclusion:

- Exploring source code summarization methodologies reveals diverse approaches, addressing challenges and enhancing code understanding.
- The need for summarization becomes evident in software maintenance, code categorization, and search efficiency.
- The proposed autonomous AI tool, guided by a fine-tuned LLM, promises revolutionary code understanding and documentation. Strategies for handling code structure and context leverage advancements to ensure coherence and relationship understanding.
- Together, these components contribute to an effective tool for code summarization.