Autonomous Code Summarization with Large Language Models

Objective

The primary objective is to develop an autonomous AI tool leveraging Large Language Models (LLMs) for the automatic generation of concise summaries for each function within a given application's codebase. The ultimate goal is to significantly improve code comprehension without necessitating additional input from the user.

Research Overview

Current State of Code Summarization

1. **Instruction Prompting**

- **Approach:** Utilizes natural language prompts for LLMs to generate code summaries.
- **Pros:** Enables zero-shot or few-shot learning.
- **Cons:** Relies heavily on prompt quality and specificity.

2. **Task-oriented Fine-tuning**

- **Approach:** LLMs are fine-tuned on datasets of code snippets and summaries.
- **Pros:** Improves performance and consistency.
- **Cons:** Requires substantial training data and computational resources.

3. **Prompt Learning**

- **Approach:** Combines instruction prompting and fine-tuning using a prompt agent.
- **Pros:** Offers increased flexibility and potential resource efficiency.
- **Cons:** Increased complexity and reduced interpretability.

Key Findings and Challenges

1. **Code LLMs vs. Generic LLMs:**
    - **Finding:** LLMs pre-trained on large code corpora outperform generic LLMs.
    - **Implication:** Emphasizes the importance of domain-specific pre-training for optimal performance.

2. **Zero-shot Methods:**
    - **Finding:** Zero-shot methods excel when training and test sets have different distributions.
    - **Implication:** Highlights LLMs' generalization ability and the need for caution in few-shot methods.
3. **Challenges in Code Summarization:**
    - **Finding:** LLMs may struggle with complex or lengthy code.
    - **Implication:** Human evaluation remains vital for quality assurance.

Advanced Approaches

1. Hybrid Prompting

- **Proposed Approach:** Start with human-written instruction prompts and refine them using continuous prompts from a prompt agent.
- **Rationale:** Balances performance and resource efficiency, leveraging human expertise.

2. Code Structure-Aware Fine-tuning

- **Proposed Approach:** Incorporate code structure using a code parser for abstract syntax trees (ASTs) and a graph neural network (GNN) for encoding.
- **Rationale:** Enhances the LLM's ability to capture hierarchical and syntactic information, resulting in more coherent summaries.

3. Multi-Task Learning

- **Proposed Approach:** Leverage the synergy between code generation, code explanation, and code summarization in a multi-task LLM.
- **Rationale:** Facilitates knowledge transfer and diversified summary generation.

Choice of LLMs

Criteria for Selection

1. Architecture

- Choose an architecture (e.g., Transformer, GPT, BERT) capable of handling large vocabularies, long code sequences, and natural language summaries.

2. Pre-training Technique

- Select a pre-training technique (e.g., MLM, CLM, contrastive learning) effectively leveraging large and diverse code corpora.

3. Adaptability to Autonomy

- Prioritize LLMs (e.g., CodeBERT, GraphCodeBERT, Codex) demonstrating adaptability to autonomously generate summaries for any code snippet.

Implementation Strategy

1. Data Collection and Preparation

- **Strategy:** Collect diverse datasets, including existing code summarization datasets and the application's codebase.
- **Preprocessing:** Thoroughly preprocess data through tokenization, normalization, and alignment of code and natural language.

2. Model Selection and Training

- **Choice:** Select a suitable LLM (CodeBERT, GraphCodeBERT, Codex, or custom-built).
- **Fine-tuning:** Train the LLM on a dedicated code summarization dataset using appropriate loss functions (e.g., cross-entropy, ROUGE).
- **Evaluation:** Assess LLM performance on test sets using metrics like BLEU, ROUGE, METEOR, or human evaluation.

3. Model Deployment and Testing

- **Interface Design:** Develop a user-friendly interface for the autonomous AI code summarization tool.
- **Deployment:** Deploy and rigorously test the LLM to ensure it autonomously generates high-quality summaries for diverse code snippets.

Application of Synthesize, Execute, Debug (SED) Approach

In the context of program synthesis with LLMs, the "near miss syndrome" is addressed through the Synthesize, Execute, Debug (SED) approach. This involves generating a draft solution, followed by a program repair phase. We explore optimal prompts and balance repair-focused, replace-focused, and hybrid debug strategies.

GSCS: Generating Summaries for Java Methods

A novel approach, Graph-based Semantic Code Summarizer (GSCS), enhances source code summarization for Java methods. Utilizing semantic and structural information, GSCS employs Graph Attention Networks on tokenized abstract syntax trees. Evaluation on Java datasets confirms GSCS outperforms state-of-the-art baselines.

LLMs in Problem Solving: Addressing Limitations

While LLMs like GPT-4 provide intuitive natural language interfaces for problem-solving, they face limitations in fetching recent data or domain-specific knowledge. Overcoming these limitations involves incorporating external data and tools alongside LLMs. The article provides insights into the architecture behind autonomous agents.

Evolution of LLM-based Agents

GPT-4 Plugins and Auto-GPT represent LLM-based agents, offering solutions to user-computer interactions. This article provides an engineer's perspective on the architecture behind autonomous agents, ranging from basic forms to those mirroring human-like complexities.

Source Code Summarization: Enhancing Tool Capabilities

Source Code Summarization emerges as a vital technology for automatically generating concise code descriptions. We present an eye-tracking study involving professional Java programmers to identify key elements. Applying these findings, we develop a novel summarization tool, enhancing the selection process and overall quality of code summaries.

Autofolding: Enhancing Code Comprehension

We introduce the autofolding problem, aiming to automatically create a code summary by folding less informative code regions. Our solution formulates the problem as a sequence of AST folding decisions, utilizing a scoped topic model for code tokens. Evaluation on open-source projects demonstrates a 28% error reduction, making autofolding a valuable tool for program comprehension.