Test cases generator

Deep learning team

February 2024

1 Introduction

In this study, we explore the enhancement of code coverage through the integration of Large Language Models (LLMs) into the test case generation process.

2 Data Preprocessing

For our problem, we employed the CodeContests database, recognized for its competitive programming dataset tailored for machine learning purposes. We adapted this dataset to suit our specific objectives by modifying the database schema. Our modifications focused on extracting code written in the C++ programming language along with their associated test cases. Additionally, we utilized GCOV to compute the coverage of these code snippets. Consequently, our database now contains a curated collection of codes, their respective test cases, and the corresponding code coverage metrics.

3 Large Language Models (LLMs)

In our approach, we experimented with several LLMs using Reg, three LLM: LangChain, CodeLlama, and BART, to generate test cases that aim to achieve comprehensive coverage. The integration of these models into our system architecture plays a pivotal role in automating and enhancing the test case generation process.

4 RAG Architecture

The Retrieval-Augmented Generation (RAG) architecture is a novel approach that combines retrieval-based and generative models to generate natural language responses. A typical RAG application has two main components: Indexing: a pipeline for ingesting data from a source and indexing it. This usually happens offline. Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model. The most common full sequence from raw data to

answer looks like: Indexing Load: First we need to load our data. Split: Text splitters break large Documents into smaller chunks. This is useful both for indexing data and for passing it into a model, since large chunks are harder to search over and won't fit in a model's finite context window. Store: We need somewhere to store and index our splits so that they can later be searched over. This is often done using a VectorStore and Embeddings model. The second step is Retrieval and generation : Retrieve: Given a user input, relevant splits are retrieved from storage using a Retriever. Generate: A Chat Model / LLM produces an answer using a prompt that includes the question and the retrieved data

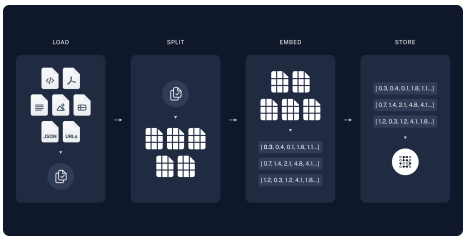

Figure 1: RAG indexing

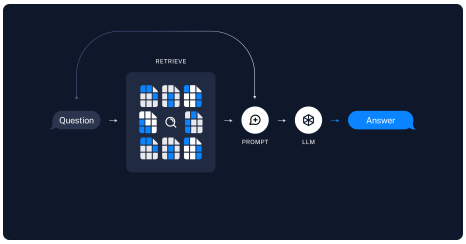

Figure 2: Retrieval and generation

5 System Architecture

In the architectural framework proposed, the first step involves inserting the code intended for test case generation into a Large Language Model (LLM). We use Retrieval using RAG pipeline. The database houses code examples along with their respective test cases and coverage metrics. The most relevant example is then selected using cosine similarity and presented to the user as a prompt. The output of this prompt consists of test cases aimed at ensuring sufficient coverage. The utility unit computes the coverage of the code based on these test cases. If the coverage falls short of 100%, feedback is relayed, comprising the coverage percentage and details of unexecuted lines, to the LLM. If the coverage

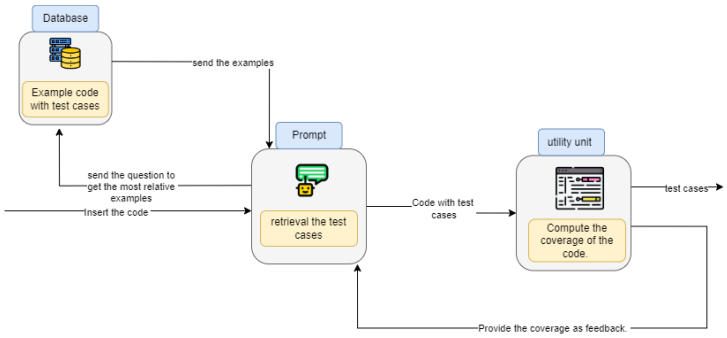

Figure 3: System Architecture

fails to improve after a certain number of attempts or reaches full coverage, the utility unit ceases feedback provision and returns the outcome.

6 Future work

Update the database to be more optimize for the problem and publish the database card

7 Conclusion

Our proposed system leverages the capabilities of LLMs to significantly enhance the efficiency and effectiveness of test case generation, aiming for optimal code coverage. The integration of these models into a structured architecture demonstrates a promising approach to automating and improving software testing processes.