RAG Pipeline for Coding Pilot: Google Colab Implementation

By:Khushi Malik

---

my_colab_notebook:[colab.research.google.com/drive/10iI32qEqV1jIMZBXZw_PnzVnITgn7IM3#scrollTo=8koZennMSgfU]
coding_copilot_research https://docs.google.com/document/d/1KGo6q_H9NBX_aP2eNpYPWZpQYCZqVoeLg5MwL4UEvh0/edit?addon_store&tab=t.0
rag_research:https://docs.google.com/document/d/1V7L6gJCzp8Nu_x0Xt_i1gJXQKf0riz86fLsYwdwOYO8/edit?usp=sharing

---

This document provides a comprehensive guide to setting up and using the Retrieval-Augmented Generation (RAG) pipeline designed for coding assistance, specifically tailored for the Flask codebase. This implementation leverages BAAI/bge-code-v1 for code embeddings, bigcode/starcoder2-3b for tokenization, and incorporates power prompting and semantic search techniques within a Google Colab environment.

---

1. Overview of the RAG Pipeline

The RAG pipeline enhances the capabilities of Large Language Models (LLMs) by providing them with relevant, up-to-date, and domain-specific information. In this context, the LLM is augmented with knowledge from the Flask codebase, enabling it to answer questions about the code, explain functionalities, and potentially assist in code generation with higher accuracy and reduced hallucinations.
The pipeline consists of the following key components:
•**Data Ingestion and Preprocessing:** The Flask codebase is cloned, and its Python files are read and split into manageable chunks. While a simple RecursiveCharacterTextSplitter is used in this example, advanced implementations might employ code-aware chunking strategies (e.g., Abstract Syntax Tree (AST) based chunking) for more semantically meaningful code segments.
•**Embedding and Vector Storage:** Each code chunk is converted into a numerical vector (embedding) using the BAAI/bge-code-v1 embedding model. These embeddings, along along with their corresponding text chunks, are stored in a vector database (ChromaDB in this Colab example). This allows for efficient similarity searches.

•**Retrieval (Semantic Search):** When a user poses a question, the question itself is embedded using the same model. A semantic search is then performed against the vector database to retrieve code chunks that are most semantically similar to the user's query. This ensures that the most relevant parts of the codebase are identified.
•**Generation (Power Prompting):** The retrieved code snippets are combined with the user's original question to form an augmented prompt. This
augmented prompt is then fed to a Large Language Model (LLM). The concept of "power prompting" is applied here, where the prompt is carefully crafted to guide the LLM to generate accurate, context-aware, and helpful responses, potentially including examples or specific instructions based on the retrieved code.

2. Google Colab Implementation Detail

2.1. Setup and Installation

```
!pip install transformers datasets faiss-cpu accelerate
!pip install sentence-transformers transformers langchain chromadb langchain-community
```

This section ensures that all necessary Python libraries are installed within the Colab environment. sentence-transformers is used for the BGE embedding model, transformers for the StarCoder tokenizer, langchain provides utilities for text splitting and interacting with vector stores, chromadb is the chosen in-memory vector database for this example, and langchain-community provides additional integrations for LangChain.

2.2. Clone the Flask Codebase

```
!git clone https://github.com/pallets/flask /content/sample_codebase
```

This command clones the official Flask GitHub repository into the /content/sample_codebase directory within the Colab environment. This serves as our domain-specific knowledge base for the RAG pipeline.

2.3. Load Models

```
from transformers import PreTrainedModel
from sentence_transformers import SentenceTransformer
import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:true"  # Reduce fragmentation

import torch
```

```
from sentence_transformers import SentenceTransformer

embed_model_id = SentenceTransformer(
    "BAAI/bge-code-v1",
    device="cuda" if torch.cuda.is_available() else "cpu"
)

torch.cuda.empty_cache()

from transformers import AutoModelForCausalLM
from transformers import AutoTokenizer, AutoModel

starcoder_tokenizer = AutoTokenizer.from_pretrained("bigcode/starcoder2-3b")
starcoder_model = AutoModelForCausalLM.from_pretrained("bigcode/starcoder2-3b", device_map="auto")
```

Here, the BAAI/bge-code-v1 model is loaded using SentenceTransformer for generating code embeddings. The bigcode/starcoder2-3b tokenizer is loaded using AutoTokenizer from the transformers library. While the tokenizer is loaded, its primary use in this RAG setup would be for potential future integration with a generative LLM that might require specific tokenization for its input.

2.4. Data Ingestion and Preprocessing

Python
```
#  Data Ingestion and Preprocessing
import os
from langchain.text_splitter import RecursiveCharacterTextSplitter

code_files = []
for root, _, files in os.walk('/content/sample_codebase'):
    for file in files:
        if file.endswith('.py'):
            code_files.append(os.path.join(root, file))

print(f"Found {len(code_files)} Python files.")

all_code_content = ""
for file_path in code_files:
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        all_code_content += f.read() + "\n\n"

# Using a generic text splitter for now, a code-aware splitter would be more advanced
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len,
)

code_chunks = text_splitter.split_text(all_code_content)
print(f"Split code into {len(code_chunks)} chunks.")
```

This section first identifies all Python files within the cloned Flask codebase. It then reads the content of these files and concatenates them into a single string. A **RecursiveCharacterTextSplitter** from langchain is used to break down this large text into smaller, overlapping chunks. The **chunk_size and chunk_overlap parameters** are crucial for ensuring that each chunk contains sufficient context while remaining within the token limits of the embedding model. For code, more sophisticated chunking strategies (e.g., based on Abstract Syntax Trees or function/class definitions) could be employed for better semantic coherence, but RecursiveCharacterTextSplitter provides a good starting point.

2.5. Embedding and Vector Storage

```
from langchain.vectorstores import Chroma
from langchain_community.embeddings import HuggingFaceBgeEmbeddings

embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-code-v1")
vectorstore = Chroma.from_texts(texts=code_chunks, embedding=embeddings)
```

This creates an **embedding function** based on the `"BAAI/bge-code-v1"` model.This specific model (`bge-code-v1`) is optimized for encoding source code into dense vector representations that capture semantic meaning.Any text or code passed through this embedding will be transformed into a high-dimensional vector suitable for similarity search.

2.6. Retrieval Function

```
def retrieve_code_snippets(query: str, k: int = 5):
    docs = vectorstore.similarity_search(query, k=k)
    return [doc.page_content for doc in docs]

print("Retrieval function defined.")
```

This function takes a user query and an optional k parameter (number of top similar documents to retrieve). It uses the vectorstore.similarity_search method to find the most relevant code chunks based on the semantic similarity between the query embedding and the stored code chunk embeddings. The page_content of the retrieved documents (which are the code chunks themselves) is then returned.

2.7. Generation (Power Prompting and LLM Interaction

~~~
    context = "\n\n".join(retrieved_snippets)

    # Power Prompting Example:
    # Instruct the LLM to act as a coding assistant, provide context, and ask for specific output.
    prompt = f"""You are an expert Python coding assistant.
Based on the following code snippets from the Flask codebase, answer the user's question.
Focus on providing accurate and concise information, and if applicable, provide code examples from the
snippets.

Code Context:
```python
{context}
```

User Question: {query}

Expert Assistant's Answer (including relevant code examples if appropriate):
"""

    # In a real application, you would send this prompt to an LLM.
    # For now, we'll just return the constructed prompt.
    return prompt

print("Generation function defined.")
~~~

User Question: {query}
Expert Assistant's Answer (including relevant code examples if appropriate): """
Plain Text
return prompt

```
print("Generation function defined.")
```
Plain Text

This `generate_response` function demonstrates the "power prompting" technique. It constructs a detailed prompt for an LLM by combining the user's `query` with the `retrieved_snippets`. The prompt explicitly instructs the LLM to act as an "expert Python coding assistant" and to use the provided "Code Context" to answer the question, emphasizing accuracy, conciseness, and the inclusion of code examples. This structured prompting helps in guiding the LLM to produce high-quality, relevant outputs. In a full implementation, the `prompt` would be sent to an actual LLM API (e.g., OpenAI's GPT models, Google's Gemini, or a self-hosted model).

This section provides a concrete example of how to use the implemented functions. It defines a user_query, retrieves relevant code snippets, and then generates the final prompt that would be sent to an LLM. The commented-out lines indicate where an actual LLM API integration would occur.

3. Future Enhancements

•**Advanced Code Chunking:** Implement more sophisticated code-aware chunking strategies (e.g., using tree-sitter or AST parsing) to ensure that code chunks are semantically meaningful and do not break functions or classes in the middle.

•**LLM Integration:** Integrate with a specific LLM API (e.g., OpenAI, Google Gemini, HuggingFace Inference API) to get actual generated responses.

•**Evaluation Metrics:** Implement metrics to evaluate the quality of retrieved snippets and generated responses.

•**User Interface:** Develop a simple web interface (e.g., using Gradio or Streamlit) to interact with the RAG pipeline more easily.

•**Persistent Vector Store:** For larger codebases or long-term use, consider persisting the vector store to disk (e.g., using ChromaDB's persistent client or other vector databases like LanceDB, Pinecone, Weaviate).

•**Re-ranking:** Explore re-ranking techniques for retrieved documents to improve the relevance of the context provided to the LLM.

References

[1] Medium Article on RAG:
https://medium.com/@drjulija/what-is-retrieval-augmented-generation-rag-938e4f6e03d1
[2] LanceDB Blog on RAG for Codebases: https://blog.lancedb.com/rag-codebase-1/