RAG System with ChatGPT Documentation

Introduction

This **Python** code implements a Retrieval-Augmented Generation (RAG) system using OpenAI's language models (GPT-4o-mini in this case). The RAG setup allows a language model to retrieve relevant documents from a vector database before generating responses, which improves the accuracy and relevance of answers to user queries.

This code is a reimplementation of the solution discussed in this tutorial: RAG With Llama 3.1 8B, Ollama, and Langchain: Tutorial. If you are interested in building an RAG system with **Llama** (an open-sourced LLM that can run locally on your device), please check that tutorial instead.

Prerequisites

- **OpenAI API Key**: This script uses the OpenAI API for processing the data and generating responses. Set up your API key as an environment variable: `OPENAI_API_KEY`.

Dependencies

To use this code, you will need to install the following libraries:
- `langchain`
- `openai`
- `pickle`
- `os`
- `time`

You can install these in the terminal by executing the command:

pip install langchain langchain-chroma langchain-community langchain-core langchain-openai langchain-ollama openai

Code Walkthrough

Import Libraries

```python
import os
import pickle  # Import pickle for serialization
import time  # Import time for measuring execution time

from langchain.prompts import PromptTemplate
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_chroma import Chroma  # Updated import
from langchain_community.document_loaders import PyPDFLoader, WebBaseLoader
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama import ChatOllama
from langchain_openai import OpenAIEmbeddings  # Updated import
from openai import OpenAI
```

This section imports all the required libraries:
- `langchain` module handle prompt templates, text splitting, embeddings, and document loaders.
- `openai` library provides an interface to interact with the OpenAI API.
- `pickle` is used to serialize (save) data to disk.

Note that this code assumed the original data format was PDF. Therefore, if other types of file need to be processed, you will need to import the corresponding file loader from langchain_community.document_loaders.

Setting Environment Variables

```python
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
)
```

- **OpenAI client**: The `OpenAI` client initializes with your API key for interacting with OpenAI's API.

Define Paths for Serialized Data

```python
# Define paths for serialized data
SPLIT_DOCS_PATH = 'split_documents_pdf.pkl'  # Path to save split documents
VECTOR_STORE_DIR = 'chroma_vectorstore_PDF'  # Directory to save Chroma data
```

These paths specify where serialized data, including split documents and vector store data, will be stored.

Step 1: Loading and Splitting Documents

Loading from PDF Files

```python
file_paths = [
    "documents/physics.pdf"
]

def load_and_split_documents_PDF(file_paths, split_docs_path):
    if os.path.exists(split_docs_path):
        print("Loading existing split documents...")
        with open(split_docs_path, 'rb') as f:
            doc_splits = pickle.load(f)
    else:
        print("Loading and splitting documents...")
        # Initialize loaders for each file and load documents
        loaders = [PyPDFLoader(file_path) for file_path in file_paths]
        docs = [loader.load() for loader in loaders]
        docs_list = [item for sublist in docs for item in sublist]  # Flatten the list

        text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
            chunk_size=250, chunk_overlap=30
        )
        doc_splits = text_splitter.split_documents(docs_list)

        # Save split documents to disk
        with open(split_docs_path, 'wb') as f:
            pickle.dump(doc_splits, f)
        print(f"Split documents saved to {split_docs_path}.")
    return doc_splits

# Load or split and save documents
doc_splits = load_and_split_documents_PDF(file_paths, SPLIT_DOCS_PATH)
```

This function loads PDF documents, splits them into chunks, and saves them for reuse.

Step 2: Create or Load Vector Store with Embeddings

```python
def create_or_load_vectorstore(vector_store_dir, doc_splits):
    if os.path.exists(vector_store_dir):
        print("Loading existing Chroma vector store...")
        embedding = OpenAIEmbeddings(openai_api_key=os.getenv("OPENAI_API_KEY"))
        vectorstore = Chroma(
            persist_directory=vector_store_dir,
            embedding_function=embedding
        )
    else:
        print("Creating new Chroma vector store...")
        embedding = OpenAIEmbeddings(openai_api_key=os.getenv("OPENAI_API_KEY"))
        vectorstore = Chroma.from_documents(
            documents=doc_splits,
            embedding=embedding,
            persist_directory=vector_store_dir
        )
         # Save the vector store to disk
        print(f"Chroma vector store saved to {vector_store_dir}.")
    return vectorstore

vectorstore = create_or_load_vectorstore(VECTOR_STORE_DIR, doc_splits)
retriever = vectorstore.as_retriever(search_kwargs={"k": 4})  # Retrieve top 4 relevant documents
```

This function either loads an existing vector store or creates a new one from document chunks. Embeddings are generated using OpenAIEmbeddings, and the store is saved for reuse.

Step 3: Define Prompt Templates

```python
# Step 3: Define Prompt Template for the Language Model
prompt_template = PromptTemplate(
    template="""You are an assistant for question-answering tasks.
Use the following documents to answer the question. Use your own knowledge when documents do not have the answer.

NEVER share the source of your response.
Do not include document in your response.

Question: {question}
Documents: {documents}
Answer:
""",
    input_variables=["question", "documents"]
)

prompt_template_og = PromptTemplate(
    template="""You are an assistant for question-answering tasks.

Question: {question}
Answer:
""",
    input_variables=["question"]
)
```

Two prompt templates are defined: one for the RAG setup and one without retrieval for comparison purposes.

Step 4: Define Chat Function with OpenAI's API

```python
# Step 4: Define a function to interact with OpenAI's ChatGPT API
def ask_chatgpt(question, documents, model="gpt-4o-mini"):
    # Format the prompt with question and documents
    prompt = prompt_template.format(question=question, documents=documents)

    response = client.chat.completions.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=0
    )
    return response.choices[0].message.content
```

This function formats the question and document context for RAG-based queries and uses OpenAI's API to get a response.

Step 5: RAG Application Class and OG Application Class

```python
# Step 5: Define Application Classes
class RAGApplication:
    def __init__(self, retriever):
        self.retriever = retriever

    def run(self, question):
        # Retrieve relevant documents using the 'invoke' method
        documents = self.retriever.invoke(question)
        # Extract content from retrieved documents
        doc_texts = "\n".join([doc.page_content for doc in documents])
        # Get the answer from the language model
        answer = ask_chatgpt(question, doc_texts)
        return answer

# Original LLM pipeline (without RAG) for comparison
class BasicLLMApplication:
    def __init__(self, model="gpt-4o-mini"):
        self.model = model

    def run(self, question):
        prompt = prompt_template_og.format(question=question)
        response = client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0
        )
        return response.choices[0].message.content
```

The RAG class initializes with a retriever, retrieves documents based on the question, and formats them for the language model's answer generation.

The basic LLM class simply formats the prompt and ask the language model to generate answer based on the model's own knowledge

Step 6: Main Execution Loop

```python
# Step 6: Test the Application with Interactive Loop
if __name__ == "__main__":
    try:
        # Initialize the RAG application
        rag_application = RAGApplication(retriever)

        # Initialize the original LLM without RAG
        basic_llm_application = BasicLLMApplication()

        # Welcome message and instructions
        print("Welcome to the RAG Question-Answering System!")
        print("Type your questions below. To exit, type 'exit' or 'quit'.\n")

        while True:
            # Prompt user for input
            question = input("Your question: ")

            # Handle exit commands
            if question.lower() in ['exit', 'quit']:
                print("Exiting the application. Goodbye!")
                break
            elif question.strip() == "":
                print("Please enter a valid question.\n")
                continue

            # Measure Basic LLM response time
            start_basic = time.perf_counter()
            basic_answer = basic_llm_application.run(question)
            end_basic = time.perf_counter()
            basic_time = end_basic - start_basic
            # Measure RAG response time
            start_rag = time.perf_counter()
            rag_answer = rag_application.run(question)
            end_rag = time.perf_counter()
            rag_time = end_rag - start_rag



            # Display the answers with elapsed times
            print("\nAnswer with RAG:")
            print(rag_answer)
            print(f"Time taken: {rag_time:.2f} seconds\n")

            print("Original Model Answer:")
            print(basic_answer)
            print(f"Time taken: {basic_time:.2f} seconds\n")

    except KeyboardInterrupt:
        # Handle Ctrl+C gracefully
        print("\nInterrupted by user. Exiting the application. Goodbye!")
    except Exception as e:
        print(f"An error occurred during execution: {e}")
```

This interactive loop takes user questions, retrieves responses from both RAG and basic LLM setups, and displays the answers and time taken.

Conclusion

This script provides a framework for creating an RAG system with ChatGPT, enabling better quality answers by integrating document retrieval into language model queries. The most important part of the code is Step2, where we used OpenAI's embedding API to create a vector store (embeddings) from our data chunks. Embeddings are the key for LLM to generate high quality answers in a short amount of time.