AI WhatsApp Assistant

Overview

This code implements an AI-powered support assistant using Langchain, OpenAI, and Streamlit. The assistant utilizes a RAG (Retrieval-Augmented Generation) architecture to answer questions based on a combination of retrieved information from a document store and its own knowledge if the query is out of provided knowledge .

RAG Architecture

The RAG (Retrieval-Augmented Generation) architecture is a key component of this AI Assistant. It combines the strengths of retrieval-based and generation-based approaches to provide more accurate and contextually relevant responses.

Key Components of RAG:

1. **Retriever**:
   - Implemented using Chroma vector store.
   - Indexes and stores document chunks for efficient retrieval.
   - Uses OpenAI's `text-embedding-3-large` model for document embeddings.
2. **Generator**:
   - Utilizes OpenAI's `gpt-4o-mini` model.
   - Generates responses based on retrieved context and the question.
3. **Knowledge Integration**:
   - Combines retrieved information with the model's inherent knowledge.

RAG Workflow:

1. User query is received.
2. The query is used to retrieve relevant documents from the vector store.
3. Retrieved documents are combined with the query and conversation history.
4. This combined input is sent to the language model for generation.
5. The model generates a response that leverages both retrieved information.

Key Components

1. Document Loading and Processing

- Uses `TextLoader` to load documents from a file.
- Implements `RecursiveCharacterTextSplitter` for text splitting.
- Creates a Chroma vector store for efficient document retrieval.

2. Embeddings

- Utilizes OpenAI's `text-embedding-3-large` model for document embeddings.

3. Retriever

- Implements a custom `load_and_process_documents` function to load, split, and store documents in the vector store.
- Uses Chroma's retriever for finding relevant documents based on user queries.

4. Language Models

- Primary LLM: OpenAI's `gpt-4o-mini` for general text generation.
- Router LLM: Another instance of `gpt-4o-mini` for query routing.

5. Prompts and Templates

- Defines a custom template for the main conversation chain.
- Implements a routing prompt to decide between using the vector store or the model's knowledge.

6. Conversation Memory

- Uses `ConversationSummaryMemory` to maintain context across interactions.

7. Query Routing

- Implements a `RouteQuery` class to determine whether to use the vector store or the model's knowledge.
- Uses a structured output from the LLM to make routing decisions.

8. State Management

- Defines a `GraphState` class to manage the state of the conversation, including questions, generated responses, retrieved documents, and memory.

9. Workflow Graph

- Utilizes `StateGraph` to define the workflow of the assistant.
- Includes nodes for retrieval, generation, and conditional routing.

10. Streamlit UI

- Implements a chat interface using Streamlit.
- Manages conversation history and displays messages.
- Streams the AI's response for a more interactive experience.

Langraph Technique:

Further we have utilised an agentic approach to handle the response of a model using langchian's latest technique called langraph. Below its states are defined.

`retrieve(state)`

- Retrieves relevant documents based on the user's question.

`generate(state)`

- Generates a response using the RAG (Retrieval-Augmented Generation) chain.
- Incorporates retrieved documents, the user's question, and conversation history.

`route_question(state)`

- Determines whether to use the vector store or the model's own knowledge to answer the question.

Workflow

1. The user inputs a question.
2. The question is routed to either the vector store or directly to generation.
3. If routed to the vector store, relevant documents are retrieved.
4. The generation step creates a response using the retrieved documents (if any), the question, and conversation history.
5. The response is displayed to the user in a streaming fashion.
6. The conversation memory is updated.

Environment and Dependencies

- Uses environment variables for API keys and configurations.
- Key dependencies: langchain, langraph, langsmith, OpenAI, Streamlit, Chroma, dotenv.

Code:

main.py

```
from langchain_community.document_loaders import
TextLoader
from prompt import template
from langchain.chains import ConversationChain
from langchain.memory import ConversationSummaryMemory
from langchain_community.tools.tavily_search import
TavilySearchResults
from pprint import pprint
from langchain.text_splitter import
RecursiveCharacterTextSplitter
from langchain_community.document_loaders import
WebBaseLoader
from langchain_community.vectorstores import Chroma
from typing import Literal
from langchain_core.prompts import ChatPromptTemplate,
PromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_openai import ChatOpenAI,
OpenAIEmbeddings
from langchain import hub
from langchain_core.output_parsers import
StrOutputParser
from typing import List
from typing_extensions import TypedDict
from langchain.schema import Document
from langgraph.graph import END, StateGraph, START
import streamlit as st
from dotenv import load_dotenv
import os
# import sys
# __import__('pysqlite3')
```

```
# sys.modules['sqlite3'] = sys.modules.pop('pysqlite3')


# Load environment variables
load_dotenv("var.env")
os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_ENDPOINT"] =
"https://api.smith.langchain.com"
os.getenv("LANGCHAIN_API_KEY")

# Set embeddings
embd = OpenAIEmbeddings(model="text-embedding-3-large")


# Load and split documents
@st.cache_resource
def load_and_process_documents():
    # Use os.path.join for cross-platform compatibility
    file_path = os.path.join(os.path.dirname(__file__),
'Data', 'liran.txt')

    # Try different encodings
    encodings = ['utf-8']
    for encoding in encodings:
        try:
            loader = TextLoader(file_path,
encoding=encoding)
            docs = loader.load()
            break
        except UnicodeDecodeError:
            continue
    else:
        raise RuntimeError(
```

```
            f"Unable to load {file_path} with any of
the attempted encodings")

    text_splitter =
RecursiveCharacterTextSplitter.from_tiktoken_encoder(
        chunk_size=10000, chunk_overlap=2000
    )
    doc_splits = text_splitter.split_documents(docs)

    vectorstore = Chroma.from_documents(
        documents=doc_splits,
        collection_name="rag-chroma",
        embedding=embd,
    )
    return vectorstore.as_retriever()


retriever = load_and_process_documents()
# Router


class RouteQuery(BaseModel):
    datasource: Literal["vectorstore"] = Field(
        ...,
        description="Given a user question choose to
route it to a vectorstore or use your own knowledge.",
    )


# LLM with function call
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
structured_llm_router =
with_structured_output(RouteQuery)
```

```python
# Routing prompt
system = """You are an expert at routing a user
question to a vectorstore.
The vectorstore contains documents related to Users
WhatsApp Conversation.
Use the vectorstore for questions on related to that
WhatsApp Conversation. Otherwise, use your own
knowledge."""
route_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        ("human", "{question}"),
    ]
)

question_router = route_prompt | structured_llm_router

# Generate
prompt = PromptTemplate(
    input_variables=["context", "question", "history"],
    template=template
)

# LLM
llm = ChatOpenAI(model_name="gpt-4o-mini",
temperature=0.7)


def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in
docs)


# Chain
```

```python
rag_chain = prompt | llm | StrOutputParser()

# Graph state


class GraphState(TypedDict):
    question: str
    generation: str
    documents: List[str]
    memory: ConversationSummaryMemory


def initialize_memory():
    return
ConversationSummaryMemory(llm=ChatOpenAI(model_name="gp
t-4o-mini", temperature=0))


def retrieve(state):
    print("---RETRIEVE---")
    question = ["question"]
    documents = retriever.invoke(question)
    return {"documents": documents, "question":
question, "memory": ["memory"]}


def generate(state):
    print("---GENERATE---")
    question = state["question"]
    documents = state["documents"]
    memory = state["memory"]

    history =
memory.load_memory_variables({})["history"]
```

```
    generation = rag_chain.invoke({
        "context": format_docs(documents),
        "question": question,
        "history": history
    })

    memory.save_context({"input": question}, {"output":
generation})

    return {"documents": documents, "question":
question, "generation": generation, "memory": memory}


def route_question(state):
    print("---ROUTE QUESTION---")
    question = state["question"]
    source = question_router.invoke({"question":
question})
    return "vectorstore" if source.datasource ==
"vectorstore" else "generate"


workflow = StateGraph(GraphState)
workflow.add_node("retrieve", retrieve)
workflow.add_node("generate", generate)
workflow.add_conditional_edges(
    START,
    route_question,
    {
        "vectorstore": "retrieve",
    },
)
workflow.add_edge("retrieve", "generate")
```

```python
workflow.add_edge("generate", END)

# Compile
app = workflow.compile()

# Streamlit UI
st.title("AI Support Assistant")

if "messages" not in st.session_state:
    st.session_state.messages = []

if "memory" not in st.session_state:
    st.session_state.memory = initialize_memory()

for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

if prompt := st.chat_input("What is your question?"):
    st.session_state.messages.append({"role": "user",
"content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        message_placeholder = st.empty()
        full_response = ""

        inputs = {
            "question": prompt,
            "memory": st.session_state.memory
        }

        for output in app.stream(inputs):
```

```
                for key, value in output.items():
                    if key == "generate":
                        full_response = value["generation"]

message_placeholder.markdown(full_response + "▌")

        message_placeholder.markdown(full_response)
        st.session_state.memory = value["memory"]

    st.session_state.messages.append(
        {"role": "assistant", "content":
full_response})


# import sqlite3
# print(sqlite3.sqlite_version)
```

prompt.py:

```
template = """

You are an expert AI Chat Assistant, Your job is to analyse whole provided data and response user according to its query.

Relevant Information:

{context}

History:
{history}
Conversation:
Human: {question}
AI:Let's think it step by step
"""

```