📚 RAG Pipeline with LangChain, Google Gemini, and Agents

Hands-on

This notebook demonstrates how to build a Retrieval-Augmented Generation (RAG) system. RAG allows an AI model to answer questions based on specific documents (like a PDF) rather than just its training data.

We will use:

- **Google Gemini:** As the LLM (Language Model) and for Embeddings.
- **ChromaDB:** As the Vector Database to store document chunks.
- **LangGraph/LangChain:** To create an agent that can decide when to search the documents.

---

1. 🛠️ Setup and Installation

First, we need to install the necessary Python libraries.

- langchain & langgraph: Frameworks for building LLM applications.
- langchain-google-genai: Connects to Google's Gemini models.
- langchain-chroma: Connects to the Chroma vector database.
- pypdf: Used to read PDF files.

```
# Install necessary libraries in one go to keep the environment clean
!pip install -qU langchain langchain-community langchain-google-genai langc
hain-chroma pypdf langgraph
```

---

2. 🔑 Environment Configuration

We need to set up API keys.

1. **LangSmith:** Used for tracing/debugging (optional but recommended to see how the agent thinks).
2. **Google API Key:** Required to use Gemini models.

Note: This code uses userdata.get() which works in Google Colab. If running locally, you might use .env files instead.

```
import os
import getpass
from google.colab import userdata

# 1. Setup LangSmith for tracing (Optional: helps visualize the agent's steps)
try:
    Langsmith_api = userdata.get('Langsmith_api')
    os.environ["LANGSMITH_TRACING"] = "true"
    os.environ["LANGSMITH_API_KEY"] = Langsmith_api
except Exception as e:
    print("LangSmith API key not found. Tracing skipped.")

# 2. Setup Google Gemini API
try:
    os.environ["GOOGLE_API_KEY"] = userdata.get('GOOGLE_API_KEY')
except Exception as e:
    # Fallback if not using Colab secrets
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API
Key: ")
```

---

3. 🧠 Initialize Models

We need two types of models:

1. **Chat Model (LLM):** The "brain" that generates answers (Gemini Flash).
2. **Embedding Model:** Converts text into numbers (vectors) so we can perform mathematical similarity searches.

```
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain.chat_models import init_chat_model

# Initialize the Chat Model
# Note: Ensure the model name is valid. Standard naming is usually "gemini-1.
5-flash"
model = init_chat_model("google_genai:gemini-2.5-flash-lite")

# Initialize the Embedding Model
embeddings = GoogleGenerativeAIEmbeddings(model="models/gemini-embe
dding-001")
```

4. 🗄️ Vector Store Setup

We use **Chroma** as our vector database. This is where our document data will be stored as numerical vectors.

- persist_directory: Saves the database to the disk so we don't have to rebuild it every time (optional).

```
from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=embeddings,
    persist_directory="./chroma_langchain_db",  # Where to save data locally, r
emove if not necessary
)
```

5. 📄 Data Loading

Here we load the content from your PDF file (data.pdf).

We use PyPDFLoader which extracts text page by page.

```
from langchain_community.document_loaders import PyPDFLoader

# Path to your file
file_path = "data.pdf"

# Load the document
loader = PyPDFLoader(file_path)
docs = loader.load()

# Basic verification
if docs:
    print(f"Loaded {len(docs)} pages.")
    print(f"Preview of page 1:\n{docs[0].page_content[:500]}...")
else:
    print("No documents loaded. Please check the file path.")
```

---

6. ✂️ Text Splitting

LLMs have a limit on how much text they can read at once. Furthermore, smaller chunks make search more accurate.

We use RecursiveCharacterTextSplitter to break the PDF into chunks of 1000 characters with a 200-character overlap (to maintain context between chunks).

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # chunk size (characters)
    chunk_overlap=200,  # chunk overlap (characters)
    add_start_index=True,  # track index in original document
)
all_splits = text_splitter.split_documents(docs)

print(f"Split blog post into {len(all_splits)} sub-documents.")
```

---

7. 📥 Indexing Documents (with Rate Limiting)

We now convert text chunks into vectors and store them in Chroma.

Why the batching? Google's Embedding API has rate limits (quotas). If we send too much text at once, it will fail. We process in batches of 90 and sleep for 60 seconds between batches to stay safe.

```
import time

batch_size = 90
document_ids = []

for i in range(0, len(all_splits), batch_size):
    batch = all_splits[i : i + batch_size]
    print(f"Adding batch {i//batch_size + 1}/{len(all_splits)//batch_size + 1} with
{len(batch)} chunks...")
    batch_ids = vector_store.add_documents(documents=batch)
    document_ids.extend(batch_ids)
    if i + batch_size < len(all_splits):
        print(f"Sleeping for 60 seconds to respect rate limits...")
        time.sleep(60)

print(f"Successfully added {len(document_ids)} documents to the vector stor
e.")
print(document_ids[:3])
```

---

8. 🛠️ Define the Retrieval Tool

An **Agent** needs tools to interact with the outside world. We define a tool called retrieve_context.

- When the Agent calls this function, it searches the Chroma database for relevant chunks.
- It returns the content to the Agent so it can formulate an answer.

```
from langchain.tools import tool

@tool(response_format="content_and_artifact")
def retrieve_context(query: str):
    """Retrieve information to help answer a query."""
    retrieved_docs = vector_store.similarity_search(query, k=2)
    serialized = "\n\n".join(
        (f"Source: {doc.metadata}\nContent: {doc.page_content}")
        for doc in retrieved_docs
    )
    return serialized, retrieved_docs
```

9. 🤖 Create and Run the Agent

We use **LangGraph** (the modern way to build agents in LangChain) to create a "ReAct" agent.

- **ReAct:** Reason + Act. The model considers the user question, decides if it needs the retrieve_context tool, uses it, reads the results, and then answers.

```
from langchain.agents import create_agent


tools = [retrieve_context]
# If desired, specify custom instructions
prompt = (
    "You have access to a tool that retrieves context from a blog post. "
    "Use the tool to help answer user queries. Always provide a detailed answe
r based on the retrieved context."
)
agent = create_agent(model, tools, system_prompt=prompt)
```

```
query = (
    "In case of a claim involving multiple illnesses or injuries within the same po
licy year, how does the policy handle sum insured restoration and claim settle
```

```
ment priority under the Re-fill Benefit?"

)

for event in agent.stream(
    {"messages": [{"role": "user", "content": query}]},
    stream_mode="values",
):
    event["messages"][-1].pretty_print()
```

https://www.menti.com/alcrho7zomhv