Practical Task

RAG-Based Knowledge Q&A Chatbot using Ollama & LangChain

- ## Project Overview:

This project is a RAG (Retrieval-Augmented Generation) based chatbot built using Python. It allows users to ask questions related to a set of documents, such as PDFs about machine learning, and provides accurate answers by combining retrieval from a knowledge base with generation using an LLM (Ollama Llama model). The chatbot first ingests PDFs, converts them into vector embeddings stored in a Chroma vector database, and then retrieves the most relevant information based on user queries. Using LangChain and HuggingFace embeddings, it integrates the retrieved information into the LLM prompt, enabling context-aware answers. The system also maintains chat history to make the conversation more coherent and supports caching for faster repeated queries. Overall, it demonstrates Python programming expertise, data pipeline management, LLM integration, and a structured approach to building a knowledge-based AI chatbot.

- ## Requirement Documentation:

Functional Requirements:

1. The system should allow users to ask natural language questions related to a set of documents (PDFs).
2. The chatbot should retrieve relevant information from the knowledge base using vector similarity search.
3. The chatbot should generate accurate and context-aware answers using the Ollama LLM.
4. The system should maintain chat history to provide coherent multi-turn conversations.
5. The chatbot should handle multiple queries in a session.

Non-Functional Requirements:

1. The system should be fast and memory-efficient, supporting smaller LLM models if system RAM is limited.

2. The solution should be modular and reusable, with separate modules for ingestion, vector DB, and chatbot logic.
3. The project should be reproducible, with the ability to regenerate the vector database anytime from source PDFs.
4. The system should have unit testing to validate ingestion, retrieval, and LLM responses.
5. The code should follow Python best practices, including proper folder structure, documentation, and use of .gitignore to exclude unnecessary files.

Inputs:

- User queries (text input)
- PDFs in the knowledge base folder

Outputs:

- Context-aware answers to user queries
- Optional: cached responses for faster repeated queries

• HLD:

The RAG chatbot is designed as a modular system with clearly defined components to handle knowledge retrieval and answer generation. At a high level, the system consists of the following components:

1. **User Interface (UI / CLI):**
    - Users interact with the chatbot through a command-line interface or a simple front-end.
    - Accepts user questions and displays responses.
2. **Ingestion Module:**
    - Processes PDF documents in the knowledge base.
    - Converts text into vector embeddings using HuggingFace sentence transformers.
    - Stores the embeddings in a Chroma vector database for efficient similarity search.
3. **Vector Database (Chroma):**
    - Stores embeddings of all documents for fast retrieval.

   - Allows the chatbot to fetch the most relevant chunks of text based on the user query.

4. **RAG (Retrieval-Augmented Generation) Engine:**
   - Combines retrieval from the vector database with generation from the LLM.
   - Uses Ollama Llama3.2:1b model to generate answers using retrieved context and chat history.
   - Supports caching to reuse previous responses and improve performance.

5. **Chat History & Memory:**
   - Maintains the last few interactions to provide context for multi-turn conversations.

6. **Output:**
   - The final response is sent back to the user through the interface.

• LLD:

The project is organized into multiple Python files for modularity:

- ingest.py: Handles PDF ingestion, splits documents into chunks, generates embeddings using HuggingFace sentence-transformers, and stores them in a Chroma vector database.
- rag.py: Main chatbot logic. Accepts user queries, retrieves the top relevant chunks from the vector database, formats a prompt with context and chat history, and invokes the Ollama LLM to generate answers.
- cache.py (optional): Stores previous query results in memory to improve response time for repeated queries.
- test_rag.py: Contains unit tests using pytest to validate ingestion, retrieval, and LLM responses.
- data/: Stores PDFs (knowledge base).
- db/: Local vector database (Chroma) generated during ingestion; excluded from GitHub.

• Dataflow Diagram & Database Design:

• Data Flow:

The RAG chatbot follows a linear yet modular data flow:

1. User Input: The user sends a query via the chatbot interface (CLI or UI).
2. Retrieval Step: The system performs a similarity search on the vector database (ChromaDB) using the user query embeddings generated by HuggingFace embeddings.
3. Context Preparation: The top relevant chunks retrieved from the vector DB are combined with chat history to form a context-rich prompt.
4. LLM Processing: The Ollama LLM receives the prompt and generates a context-aware response.
5. Response Output: The chatbot displays the answer to the user and optionally caches the result for future queries.
6. Update: Chat history and cache are updated for the next query.

• Database Design:

The RAG chatbot uses a **vector database** to efficiently store and retrieve embeddings generated from PDF documents. Each PDF is split into smaller chunks, and **HuggingFace embeddings** are created for these chunks and stored in **ChromaDB**, which is a local vector database stored in the db/ folder. The database maintains both the embedding vectors and metadata, such as the source PDF name, page number, and chunk identifier, allowing the system to retrieve the most relevant information during a user query. Since the database is **automatically generated during ingestion**, it is not included in GitHub, but it can be recreated anytime by running the ingest.py script. The database design focuses on **fast similarity search**, **context-aware retrieval**, and **efficient integration** with the LLM to provide accurate answers. Additionally, in-memory caching is optionally used to speed up repeated queries, enhancing performance without modifying the database.

• Architecture diagram:

The architecture of the RAG chatbot is modular and scalable, designed to integrate document retrieval with LLM-based generation. The system has three main components: Ingestion, Vector Database, and Chatbot Interface.

1. Ingestion Module (ingest.py): Reads PDFs from the knowledge_base folder, splits them into chunks, generates embeddings using HuggingFace models, and stores them in Chroma vector database.

2. Vector Database (db/ folder): Stores embeddings along with metadata like PDF name and page number. Supports fast similarity search to retrieve the most relevant chunks for a query.
3. Chatbot Interface (rag.py / CLI): Accepts user queries, retrieves relevant chunks from the vector database, combines them with chat history and optional cache, and passes them as a prompt to the Ollama LLM. The LLM generates context-aware answers, which are returned to the user and stored in the chat history for future context.

Optional Components:

- Cache Module: Stores recent query results in memory to speed up repeated requests.
- Unit Tests: Ensure ingestion, retrieval, and LLM integration work as expected.

• Rag:

The core of the chatbot is the RAG pipeline, which combines retrieval of relevant information with generation using an LLM. When a user asks a question, the system first performs a similarity search on the vector database to retrieve the top relevant document chunks. These chunks are then combined with the chat history to form a context-rich prompt. The Ollama LLM uses this information to generate an answer that is accurate, context-aware, and grounded in the knowledge base. This approach ensures that the chatbot can answer questions beyond what the LLM knows by default, leveraging the ingested PDFs as external knowledge.

• Knowledge Base:

The knowledge base consists of PDF documents stored in the knowledge_base folder. Each document is split into smaller chunks to improve retrieval accuracy. During ingestion, each chunk is converted into a vector embedding using HuggingFace sentence-transformers. These embeddings are stored in the Chroma vector database along with metadata such as source PDF name and page number. This setup allows the chatbot to quickly find the most relevant information for any user query, providing document-grounded answers.

• Caching:

To improve performance, the system uses in-memory caching for recently asked queries. When a user asks a question, the chatbot first checks if a similar query has been processed recently. If so, it can return the cached answer instantly, avoiding repeated computations and vector searches. This caching mechanism reduces response time, optimizes resource usage, and makes the chatbot more efficient for multiple queries.

• Database & Vector Database:

The project does not use a traditional relational database; instead, it relies on a **vector database** (ChromaDB) to store embeddings of document chunks. Each chunk of a PDF is converted into a **vector representation** using HuggingFace embeddings and stored with metadata such as the source PDF, page number, and chunk ID. This allows **fast similarity-based retrieval**, which is essential for the RAG workflow. The vector database is automatically generated during the ingestion step (ingest.py) and can be regenerated anytime, ensuring reproducibility.

• LLM Integration, Chain of Thoughts:

The Ollama LLM (llama3.2:1b) is integrated into the pipeline to generate context-aware answers. The LLM prompt includes:

1. Chat history – so it can maintain coherent multi-turn conversations.
2. Retrieved document chunks – from the vector database, which provides factual grounding.
3. User query – to generate the final answer.

This forms a chain of thought, where retrieved knowledge and conversation context guide the LLM to produce accurate and relevant responses. This approach demonstrates advanced Python skills, modular code design, and understanding of LLM prompt engineering.

• Unit Testing:

Unit testing is implemented using pytest to ensure each component works correctly:

- test_rag.py tests PDF ingestion, embedding generation, and vector retrieval.
- Retrieval correctness is validated by checking that relevant chunks are returned for sample queries.
- LLM integration is tested (mocked if necessary) to ensure the prompt is correctly formatted and responses are received.

• Data Pipeline:

The data pipeline in the project is end-to-end and modular:

1. Ingestion – PDFs → chunking → embedding generation.
2. Storage – embeddings and metadata stored in Chroma vector database.

3. Query Processing – user query → similarity search → context + chat history → LLM prompt.
4. Response Generation – LLM generates answer → update chat history and cache → return to user.