**Project Documentation:   
AI ChatBot Application**

This project implements a question-answering system using LangChain and Streamlit, utilizing state-of-the-art text embedding and language models. It performs **Retrieval-Augmented Generation (RAG)**, combining a vector database for efficient similarity search with a Large Language Model (LLM) to generate accurate and context-aware answers. The project comprises three key components:

1. generate\_database.py: Handles text extraction, chunking, and embedding into a Chroma vector store.
2. query\_data.py: Retrieves relevant context from the vector store and interacts with the LLM to generate responses.
3. app.py: Provides a user-friendly Streamlit interface for query input and result display.

**Prerequisites**

* Python 3.8+
* Required Python libraries:
  + streamlit
  + PyPDF2
  + langchain
  + chromadb
  + langchain\_ollama
  + transformers

Install dependencies using:

pip install -r requirements.txt

**Module 1: generate\_database.py**

**Purpose**

Processes a large PDF document (e.g., a book), extracts text, splits it into manageable chunks, and stores them in a Chroma vector database using embeddings.

**Code Explanation**

**Key Functions:**

1. **extract\_text\_from\_pdf(pdf\_path: str) -> str**
   * Extracts text from the first half of a PDF file.
   * **Parameters**:
     + pdf\_path: Path to the PDF file.
   * **Returns**: Extracted text as a single string.
2. **load\_documents() -> list[Document]**
   * Calls extract\_text\_from\_pdf and wraps the text into a LangChain Document object.
   * **Returns**: A list of Document objects.
3. **split\_text(documents: list[Document]) -> list[Document]**
   * Splits the input text into smaller chunks using RecursiveCharacterTextSplitter.
   * **Parameters**:
     + documents: List of documents containing large blocks of text.
   * **Returns**: List of smaller Document chunks.
4. **save\_to\_chroma(chunks: list[Document]) -> None**
   * Embeds the chunks using OllamaEmbeddings and stores them in a Chroma vector database.
5. **generate\_data\_store()**
   * Coordinates the entire workflow: loading documents, splitting text, and saving to Chroma.

**Module 2: query\_data.py**

**Purpose**

Handles retrieval and interaction with the vector database to fetch relevant context for a given query and generate a response using an LLM.

**Code Explanation**

**Key Functions:**

1. **create\_instructor\_embeddings() -> OllamaEmbeddings**
   * Initializes the embedding model (nomic-embed-text) used for similarity search.
2. **query\_database(query\_text: str) -> tuple[str, list[str]]**
   * Main function for querying the database and generating answers.
   * **Steps**:
     1. Loads the Chroma vector database.
     2. Performs a similarity search to retrieve relevant text chunks.
     3. Constructs a prompt using the retrieved context and user query.
     4. Runs the prompt through the LLM to generate a response.
   * **Parameters**:
     1. query\_text: The user's input query.
   * **Returns**: A tuple containing the response and source metadata.

**Module 3: app.py**

**Purpose**

Provides a Streamlit-based web interface for the question-answering system.

**Code Explanation**

1. **UI Components**:
   * A text input box for user queries.
   * A "Search" button to trigger the query.
2. **Backend Interaction**:
   * Calls query\_database from query\_data.py to process the query and display results.
   * Displays both the generated response and the associated sources.

**Workflow**

1. **Data Preparation**:
   * Use generate\_database.py to process the dataset and create the Chroma vector database.
2. **Query Handling**:
   * Use query\_data.py to fetch relevant context and interact with the LLM.
3. **User Interaction**:
   * Run app.py to start the Streamlit web app for end-user interaction.

**Example Usage**

1. Run the data preparation script:

python generate\_database.py

1. Start the Streamlit app:

streamlit run app.py

1. Enter a query into the text input box, e.g., "What is Harry Potter's first encounter with magic?"

**Conclusion**

This project demonstrates a modular and extensible architecture for building a retrieval-augmented question-answering system. It effectively integrates modern machine learning techniques with an interactive user interface, making it a robust solution for querying large textual datasets.

**Citations**

* ChatGPT for debugging.
* Claude.ai for debugging.
* GitHub for reference.