AskMe: AI Assistant

Overall Approach

Environment Setup

- **`load_dotenv()`**: Loads environment variables from a `.env` file, including the Groq and Google API keys necessary for embedding and model operations.

```
GROQ_API_KEY=your_groq_api_key

GOOGLE_API_KEY=your_google_api_key
```

Install Requirements

- Install the required packages using pip

```
pip install -r requirements.txt
```

Styling

- **`load_css(file_name)`**: Loads and applies custom CSS styling to the Streamlit application to enhance the UI/UX.

Model Initialization

- **`ChatGroq(groq_api_key=groq_api_key, model_name="Gemma-7b-it")`**: Initializes the ChatGroq model using the provided Groq API key and model name, setting up the language model for generating responses.

Prompt Template

- **`prompt_template_str`**: Defines a template for the chatbot's prompt to generate context-aware and relevant responses based on the provided document content.

Embedding Function with Retry Mechanism

- **`embed_with_retries(embedding_function, texts, retries=3)`**: A utility function that attempts to generate embeddings for the given texts, retrying up to a specified number of times in case of transient errors.

Document Processing

- **`vector_embedding()`**:
  - **Check Session State**: Ensures that vectors are loaded only once.
  - **Load Documents**: Uses `PyPDFDirectoryLoader` to load PDF documents from a specified directory.

- **Split Documents**: Uses `RecursiveCharacterTextSplitter` to split documents into smaller, manageable chunks.
- **Generate Embeddings**: Calls `embed_with_retries` to generate embeddings for the document chunks.
- **Store Embeddings**: Saves the embeddings in a FAISS vector store for efficient retrieval during query processing.

Conversation Management

- **`update_conversation_log(question, answer)`**: Appends each question and its corresponding answer to a log file, maintaining a history of the conversation for context.
- **`get_context_from_log()`**: Reads the conversation log file and returns the context, which includes all previous questions and answers.

Streamlit Interface

- **Embedding Button Section**:
  - **`st.button("Click to load the Documents")`**: Triggers the `vector_embedding()` function to initialize the vector store with document embeddings.
  - **`st.button("Clear Conversation")`**: Resets the chat history and clears the conversation log.
- **Chat Interface**:
  - Displays previous chat history from `st.session_state`.
  - Provides an input box (`st.text_input`) for users to type their questions.

Handling User Queries

- **Retrieve Context**: Calls `get_context_from_log()` to retrieve previous conversation context.
- **Handle Ambiguous Queries**: Detects follow-up questions referencing previous ones and prepends the previous question for clarity.
- **Create Prompt**: Uses `ChatPromptTemplate.from_template(prompt_template_str)` to create a prompt with context.
- **Retrieve Answer**: Uses LangChain to create a retrieval chain and generate responses based on the user's query and the document content.
  - **`create_stuff_documents_chain(llm, prompt_template)`**: Creates a document chain for combining document content.
  - **`create_retrieval_chain(retriever, document_chain)`**: Sets up a retrieval chain using the FAISS retriever and document chain.
- **Update Chat History and Log**:
  - Appends the new question and answer to `st.session_state.chat_history`.
  - Calls `update_conversation_log(prompt1, answer)` to update the conversation log.

Error Handling

- **Error Messages**: Uses `st.error` to display appropriate error messages in case of issues during embedding generation or retrieval, ensuring users are informed of any problems.

Running the Application

To start the Streamlit application, run the following command

```
streamlit run your_script.py
```

Frameworks/Libraries/Tools Used

Streamlit

- **Purpose**: Building the user interface for the chatbot.
- **Usage**: The entire frontend of the chatbot, including text input, displaying chat history, and styling.

LangChain

- **Purpose**: Handling the retrieval and response generation process.
- **Usage**: Creating chains for combining document chunks and generating responses based on user queries.

Google Generative AI (google-generativeai)

- **Purpose**: Generating embeddings for the document chunks.
- **Usage**: Embedding generation for document retrieval.

Python-dotenv

- **Purpose**: Managing environment variables.
- **Usage**: Loading API keys and other sensitive information from a `.env` file.

PyPDF2

- **Purpose**: Handling PDF documents.
- **Usage**: Loading and processing PDF documents for data ingestion.

ChromaDB

- **Purpose**: Managing the vector store database.
- **Usage**: Efficient storage and retrieval of document embeddings.

FAISS

- **Purpose**: Efficient similarity search and clustering of embeddings.
- **Usage**: Storing and retrieving document embeddings for quick and accurate response generation.

Problems Faced and Solutions

Problem 1: Embedding Errors

- **Issue**: Encountered errors during the embedding generation process.
- **Solution**: Implemented a retry mechanism to handle transient errors and ensure successful embedding generation.

Problem 2: Context Management

- **Issue**: Maintaining context across multiple questions and ensuring coherent responses.
- **Solution**: Utilized a log file to store the conversation history and retrieved context from it for each new query.

Problem 3: Handling Ambiguous Queries

- **Issue**: Users sometimes ask follow-up questions that reference previous ones.
- **Solution**: Implemented logic to detect such references and prepend previous questions to the current query to maintain context.

Problem 4: Slow Response Rate with Google API

- **Issue**: Experienced delays in response due to latency with the Google API.
- **Solution**: Implemented parallel processing and utilized both Google and Groq APIs concurrently with the Gemma-7b-it model to improve response times.

Future Scope

Feature Enhancements

- **Multilingual Support**: Adding support for multiple languages to cater to a wider audience.
- **Voice Interaction**: Enabling voice-based queries and responses for a more interactive experience.
- **Personalization**: Tailoring responses based on user preferences and history for a more personalized experience.

Technical Improvements

- **Scalability**: Optimizing the backend to handle a larger volume of documents and user queries.
- **Advanced Analytics**: Implementing analytics to track user interactions and improve the chatbot's performance over time.
- **Improved Error Handling**: Enhancing error handling mechanisms to make the chatbot more robust.

Integration

- **Third-Party APIs**: Integrating with other APIs to provide more comprehensive answers (e.g., real-time data, external knowledge bases).
- **Mobile App**: Developing a mobile application to make the chatbot more accessible on various devices.