PDF-Based Question Answering Chatbot

Project Overview

This project is a PDF-based Question Answering (QA) chatbot that allows users to ask questions based on the content of a provided PDF document. If the chatbot cannot find a relevant answer, it gracefully handles the situation with a fallback response.

Approach

The development of the PDF-based Question Answering Chatbot was guided by the following systematic approach:

1. Problem Understanding

- Clearly defined the goal: A chatbot capable of answering questions based on a provided PDF document.
- Identified two key challenges:
    - Extracting and organizing content from the PDF.
    - Handling queries with appropriate fallback responses.
    - Building backend and frontend for making a complete system

---

2. Tool and Library Selection

- PDF Parsing: Selected `PDFMiner` for extracting text from PDF documents.
- Embedding and Search:
    - HuggingFace Bge Embeddings: For generating semantic embeddings of the document content.
    - ChromaDB: For efficient storage and similarity-based retrieval of document chunks.
- Chat Interface: Chose a simple JavaScript-based frontend for user interaction.

- Backend Framework: Used `FastAPI` to implement the chatbot's API.
- AI Prompting: Utilized `LangChain` for managing AI prompt templates and generating responses.

---

3. Document Preparation

- PDF Loading: Loaded the content of the PDF using `PDFMinerLoader`.
- Chunking: Split the document into semantically similar chunks to ensure better query matching by using `semantic-text-splitter`, using a token limit for efficient processing. Chose this over traditional splitting as this generally gives better retrieved documents
- Embedding: Converted chunks into semantic vectors using `HuggingFace Bge Embeddings` for indexing in ChromaDB.

---

4. Semantic Search Implementation

- Configured ChromaDB to retrieve the most relevant document chunks for a given query.
- Set a **similarity score threshold** to filter out irrelevant matches, ensuring only meaningful context is considered. This is done in order to ensure that application does not return any irrelevant responses and gives a fallback answer if the model cannot find the answer in the document
- Fallback returned is

  “Sorry, I didn’t understand your question. Do you want to connect with a live agent?”

---

5. Chatbot Response Generation

- Designed a prompt template in `LangChain` that:
  - Provides the retrieved context for the AI to answer the query.
  - Implements fallback logic if the retrieved context is irrelevant.
  - The promp is designed in such a way that the mode only returns answer if it is confident about it , otherwise a fallback response is given
- Integrated the prompt template with AI model to dynamically generate responses.

- Groq is used to get response from the AI model through its API
- *Mixtral-8x7b-32768* model is used because of its great accuracy in question answering tasks

---

6. Frontend Development

- Built a simple web interface using JavaScript, allowing users to:
    - Input their questions.
    - View the chatbot's responses.
    - Handle loading and error states.

---

7. Backend-Frontend Integration

- Connected the frontend to the backend via API endpoints hosted on `FastAPI`.
- Configured the frontend to make POST requests to the backend with the conversation data and display the responses.

---

8. Fallback Logic

- Implemented fallback logic to handle cases where no relevant context is retrieved from the document:
    - Checked the similarity score of retrieved chunks.
    - Responded with:
      "**Sorry, I didn't understand your question. Do you want to connect with a live agent?"**
    - Ensured this logic was consistent in both the backend API and the frontend UI.
    - Also added a safety check in *prompt* so that the model returns fallback if the context is not present in the pdf

---

9. Testing and Debugging

- Conducted extensive testing for:
    - PDF parsing and chunking accuracy.
    - Query matching and retrieval quality.

    - AI-generated responses.
    - Seamless frontend-backend communication.
- Resolved edge cases such as invalid queries, empty PDF content, and connection errors.
- The *score threshold* is set to **0.5** through testing with different human made queries and found this threshold to perform decent

---

10. Deployment

- Developed a Python script to automate the starting of both backend and frontend servers.
- Ensured the application could be run locally with minimal setup.

Technologies Used

Frontend

- HTML, CSS, JavaScript: For the web interface.
- HTTP Server: Serves the static frontend files.

Backend

- FastAPI: For building the chatbot API.
- LangChain: For handling document splitting, vector embeddings, and semantic search.
- ChromaDB: For storing and querying the document embeddings.
- HuggingFace Embeddings: For semantic vector generation.
- PDFMiner: For parsing and extracting text from PDF documents.

Other Tools

- Uvicorn: For running the FastAPI application.
- Python HTTP Server: For hosting the frontend.

Installation and Setup

Prerequisites

- Python 3.7 or higher
- pip (Python package manager)

Project structure

project_directory/

|---- backend/
|------__init__.py
|------ app.py
|------input_apport.pdf

|---- frontend/
|------index.html
|------app.js
|------style.css
|------assets/

├── startup.py
|------.env
|-------.gitignore
|-------requirements.txt

Follow these steps to set up and run the PDF-based Question Answering Chatbot:

---

1. Open the Project in Visual Studio Code

1. Open Visual Studio Code on your computer.
2. Use the menu option `File > Open Folder` (or `Open` on macOS).
3. Navigate to the project directory and select it.

---

2. Create and Activate a Virtual Environment

For macOS/Linux:

1. Open the terminal in Visual Studio Code or your system terminal.
2. Run the following commands to create and activate the virtual environment:

```
python3 -m venv venv
source venv/bin/activate
```

For Windows:

1. Open the terminal in Visual Studio Code or the Command Prompt.
2. Run the following commands to create and activate the virtual environment:

```
python -m venv venv
venv\Scripts\activate
```

---

3. Install Project Dependencies

1. Ensure the virtual environment is activated.
2. Install the dependencies specified in `requirements.txt` by running:

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

3. Verify that all dependencies have been installed without errors.
4. Run the application using the following command

```
python start_app.py
```

5. Access the Application:
    - Backend API Documentation: http://127.0.0.1:8000/docs
    - Frontend Interface: http://127.0.0.1:5500/

Usage Instructions

1. Upload or specify the PDF document (`input_apport.pdf`) before starting the server.
2. Open the frontend interface in your browser.
3. Type a question related to the content of the PDF.
4. Receive a response:
    - If the answer is found, it is displayed.
    - If no answer is found, you will see the fallback message:
      "Sorry, I didn't understand your question. Do you want to connect with a live agent?"

Troubleshooting

If the application is not starting using startup.py , manually start backend and frontend using the below steps:

Start the Backend

1. While the virtual environment is activated, navigate to the project directory if not already there.
2. Run the following command to start the backend server:

```
uvicorn backend.app:app --reload --host 127.0.0.1 --port 8000
```

The backend server should now be running on http://127.0.0.1:8000

. Start the Frontend

1. Open another terminal instance (or tab) while keeping the backend terminal running.
2. Navigate to the `frontend` directory within the project

```
cd frontend
```

3. Start a simple HTTP server to serve the frontend files:

```
python -m http.server 5500
```

4. The frontend should now be running on http://127.0.0.1:5500

Acknowledgments

- LangChain for document processing and querying.
- HuggingFace for embedding models.
- FastAPI for building the API.
