**Manual for DME Chatbot Code and Deployment**

**1. Introduction**

**Overview of the Chatbot**

This chatbot is designed to answer questions from Texas Medicaid Handbook for providers of Durable Medical Equipment (DME) and supplies. It utilizes advanced natural language processing (NLP) techniques to extract relevant information from a PDF version of the manual and provide users with accurate, contextually appropriate answers. The chatbot is particularly useful for providers who need quick access to information without having to manually search through extensive documentation.

**Key Features**

* **PDF Document Parsing**: The chatbot processes and chunks the PDF manual to retrieve specific sections or information.
* **Natural Language Understanding**: Powered by a large language model (LLM) to understand and respond to user queries in natural language.
* **Contextual Conversations**: Maintains context across multiple user queries to provide coherent and relevant responses.
* **User-Friendly Interface**: Implemented using Streamlit to provide an easy-to-use graphical user interface.

**2. Environment Setup**

**Prerequisites**

Before setting up the chatbot, ensure that you have the following tools and libraries installed:

* **Python 3.8+**: The programming language used for the chatbot.
* **pip**: Python package installer to install the required libraries.
* **Tesseract OCR**: For Optical Character Recognition, if dealing with scanned PDFs.
* **Poppler**: For rendering of PDF documents.
* **NLTK**: A suite of libraries and programs for symbolic and statistical natural language processing.
* **Libraries**: All the required libraries mentioned in the requirements.txt file

**Installation Instructions**

1. **Clone the Repository**: The code is hosted on Texas HHS-OIG GitHub. You can clone the repository to your local machine:
2. **Install Dependencies**: Navigate to the project directory and install the required Python libraries using pip.
3. **Set Up Environment Variables**: If needed, set up any environment variables required by the chatbot, such as API keys for the LLM.
4. **Tesseract OCR Installation**:
   * **Linux**: *sudo apt-get install tesseract-ocr*
   * **Windows**: Download the installer from [Tesseract OCR](https://github.com/tesseract-ocr/tesseract) and follow the instructions.
5. **Poppler Installation**:
   * **Linux**: *sudo apt-get install poppler-utils*
   * **Windows**: Download the package from [Poppler](https://github.com/oschwartz10612/poppler-windows) and follow the instructions

**3. Core Components**

**PDF Processing**

The chatbot leverages libraries like *pdfplumber* and *partition\_pdf* to extract and process text from PDF documents. These tools allow for the extraction of both text and tables from the PDF, enabling the chatbot to provide accurate responses based on the manual's content.

* ***partition\_pdf***: Utilized for splitting the PDF into manageable chunks. This helps in focusing on specific sections of the document relevant to the user's query. This package is very useful in splitting the document by title and maintaining the semantic context of the document. However, it is not very efficient in extracting the tables and maintaining their structure.
* ***pdfplumber***: This is another library for PDF processing, often used for extracting more complex structures like tables or images. It does not provide the utility of splitting the document by titles but it is very reliable in extracting tables and maintaining their structure.

**Language Model (LLM) Integration**

The chatbot uses an Llama3.1 to understand and respond to user queries. This model has been fine-tuned on large datasets to handle complex natural language queries, making it well-suited for the task of interpreting and responding to questions about the manual.

* **Why Llama3.1?** Llama3.1 is an open-source LLM provided by Meta that boasts performance comparable to the best LLMs available. Open-source models, unlike GPT models hosted by openai, allow free of cost response generation and hosting of the chatbot locally. This is more economical and addresses the privacy concerns of those who do not want to share their data to a third party. It also integrates seamlessly with the LangChain framework, which the chatbot uses extensively. Another advantage of using Llama3.1 is that it has a large token limit which allows for inputting longer context chunks, prompts and queries. The downside to having a larger token limit is that the model requires more computing power to process and generate responses.
* **Why nomic-embed-text?** Nomic-embed-text is an embedding model available within the Ollama module. It is a widely used embedding model that integrates well with the Llama models.
* **Why Ollama module:** Ollama is a module available in Langchain setup that allows for importing the Llama LLM models and nomic embedding models. To run Ollama, the user needs to download Ollama interface in their local machine and run Ollama and download the models with the command in the shell ‘Ollama pull Llama3.1’. Once the model is downloaded it can be used through the Ollama module. This allows for easy import of the model and running them locally without downloading them every time the application is reloaded.

**Summaries**

Creating summaries of the text is an optional choice. Since document has both texts and table elements it makes more sense to create summaries of these chunks instead of letting the retriever run through original chunks which would be computationally more expensive and might run into errors when going through tables. The advantage of summaries is that the retriever must go through only text to retrieve relevant chunks which is easier. However, the disadvantage of summaries is that if the summaries are not very accurate depiction of the text or table chunks then they might miss important information for retriever to look at. The retriever only goes through the embeddings of the summaries and finds most relevant ones and then pulls the original chunks corresponding to those summaries. If the summaries don’t refer to some important piece of information in the original chunks, then they would never be retrieved affecting the quality of the output.

**Embeddings Vector Store**

To efficiently search through the large amount of text in the PDF manual, the chatbot uses embeddings and a multi-vector retriever:

* **Embeddings**: Text from the manual is converted into numerical vectors, which capture the semantic meaning of the text.
* **Vectorstore:** A vector store that stores the embeddings of the summaries created for each table or text chunk.
* **Docstore:** A memory store that keeps the original text and table chunks corresponding to the summaries.
* **Chroma**: An open-source vector store that indexes these embeddings, allowing for quick and efficient retrieval of relevant text segments when a user asks a question.

**Conversational Memory**

To maintain context over a conversation, the chatbot uses *ConversationBufferMemory*. This allows the chatbot to remember previous interactions, making it more effective in handling follow-up questions or multi-turn conversations.

* **ConversationBufferMemory**: Stores past interactions and is used to append context to the ongoing conversation.

**4. Code Walkthrough**

***get\_pdf\_chunks()***

This function is responsible for processing the PDF document and dividing it into smaller, more manageable chunks using two different packages (PDFPlumber and Partition\_PDF). The chunking is based on specific strategies such as splitting by title or character count, which helps in isolating relevant sections of the document.

* **Parameters for *partition\_pdf***:
  + *path*: The file path of the PDF document.
  + *Strategy*: The chunking strategy (e.g., by title, by character count).
* **Output**: Returns a list of chunks, each representing a portion of the document that can be individually processed by the chatbot.

***get\_table\_text\_elements***

This function extracts the text and metadata from the table and text elements. Partition\_pdf module returns the output in text and table objects, so they need to be processed to extract the text and metadata from them.

***LangChain Setup***

The chatbot uses the LangChain framework to connect various components like the language model, vector store, and memory. This setup allows the chatbot to efficiently process user queries and provide accurate responses.

* **Prompts**: Custom templates are used to structure the input to the language model, ensuring that the responses are relevant and concise.
* **Chains**: The Conversational Retrieval Chain combines the LLM, memory, and vector store, creating a robust pipeline for handling queries.

***get\_summaries***

This function is responsible for creating the summaries of table and text chunks extracted. It used the Llama3.1 model to create summaries.

***create\_vectorstore***

This function is responsible for creating the vectorstore and docstore for the retriever to search for relevant summaries and then retrieve the corresponding documents. For the summaries and their corresponding chunks, there is a unique ID attached to both so that the retriever can identify the correct documents. The metadata is attached to the original documents in docstore as they are retrieved to generate responses and we will use the metdata of the retrieved documents to give references in the responses.

***Streamlit Interface***

The user interface is built using Streamlit, a framework that allows for the quick creation of web applications. This section of the code defines the layout, input fields, and how user interactions are handled.

* **Templates**: HTML templates (*css, bot\_template, user\_template*) are used for styling the chatbot interface, making it visually appealing and easy to navigate.

***Saving and Loading Data***

Data persistence is handled using Python's pickle and Json module. Since the processing of PDF and creation of summaries can be a computationally intensive process, it is more efficient to store the chunks and summaries and retrieve them directly. This allows for quick loading and reuse of data without needing to reprocess the entire document every time the chatbot starts.

**5. Customization and Extensibility**

**Modifying the PDF Processing Strategy**

Users can modify the PDF processing strategy by adjusting the parameters in the get\_pdf\_chunks() function. For example, changing the chunking strategy from 'by title' to 'by section' might yield different chunks that could be more suitable for certain types of queries.

**Changing the Language Model**

If you wish to use a different LLM, you can replace the Llama3.1 model in the LangChain setup with another model. There are certain models available in the Ollama module that you can call directly. If other models that are not available through Ollama module are required then they can be used using the HuggingFace, GPT4All or Langchain modules.

**Prompt Engineering**

Prompt engineering plays a significant role in determining the quality of the output. The prompt can include specific instructions like limiting the length of the answer or acting as an expert of a field which would affect the output of the model. There is a lot of scope in testing different prompts and assessing the quality of the output. That can be an area of research and improvement for this app for the future.

**UI Customization**

The Streamlit interface can be customized by modifying the HTML and CSS templates. This allows for changes in layout, color scheme, or adding new interactive elements.

**6. Deployment**

**Running the Chatbot Locally**

1. Ensure all dependencies are installed and environment variables are set up.
2. Run the chatbot using Streamlit:

streamlit run DME\_chatbot\_app\_final.py

1. Access the chatbot locally via http://localhost:8501.

**Deploying to the Web**

1. **Cloud Platforms**: You can deploy the chatbot on platforms like Heroku, AWS, or Google Cloud.
2. **Docker**: Create a Dockerfile and build a Docker image to containerize the application for easier deployment.

**7. Troubleshooting**

**Common Issues**

* **Dependencies**: Ensure all required libraries are installed. Missing packages can cause the chatbot to fail during runtime.
* **PDF Parsing Errors**: If the PDF is not processed correctly, try adjusting the chunking strategy or use a different PDF processing library.
* **LLM Response Accuracy**: If the LLM does not provide accurate responses, consider fine-tuning the model or adjusting the prompt templates.

**Error Handling**

The code includes basic error handling, but you may need to add custom handlers depending on the environment or specific issues that arise during use.

**8. Conclusion**

**Summary**

This guide provided a detailed walkthrough of the Handbook for Texas Medicaid Providers of DME Chatbot, covering everything from environment setup to customization and deployment. By understanding the core components and their interactions, users can effectively use, modify, and extend the chatbot to suit their needs.