**Documentation for Hackathon Chatbot Application**

Table of Contents

1. **Import Libraries**
2. **Setup and Initialization**
3. **File Uploading and Processing**
4. **Embedding and Vector Store Selection**
5. **Text Chunking and Document Handling**
6. **Retrieval QA Chain Creation**
7. **User Interaction and Input Handling**
8. **Main Function**
9. **Extracting Files from ZIP Archives**
10. **Handling PDF Text Extraction**
11. **Handling DOCX Text Extraction**
12. **Creating and Storing Documents**
13. **Conclusion and Additional Features**

**1. Import Libraries**

This section of the code is dedicated to importing necessary libraries, each serving a specific purpose in the application:

* **streamlit**: A framework for creating web applications quickly. In this project, it is used to build the user interface.
* **os**: Provides a way to interact with the operating system. Here, it is used for file path manipulations and environment variable access.
* **PyPDF2**: A library for PDF file manipulation, enabling the program to read text from PDF files.
* **pinecone**: A service for managing vector databases. This is crucial for handling the vectors generated from text data.
* **docx**: Used to read Microsoft Word .docx files, allowing the application to process text from these documents.
* **shutil**: Offers high-level file operations, such as file copying.
* **langchain.chat\_models**: From LangChain, used to integrate chat models like OpenAI's GPT-3.5-turbo.
* **langchain.vectorstores**: Parts of LangChain that deal with vector storage, specifically for Pinecone and Qdrant, which are the two vector databases used.
* **random, datetime, string**: Utilities for generating random strings, handling dates, and manipulating strings, which are useful in various parts of the application.
* **qdrant\_client**: Client library for interacting with the Qdrant vector database.
* **dotenv**: Loads environment variables from a .env file, essential for securing API keys and other sensitive data.
* **langchain.embeddings**: Part of LangChain for handling embeddings, specifically from Hugging Face models.
* **langchain.text\_splitter**: Contains utilities for splitting text into manageable chunks, critical for processing large documents.
* **streamlit\_chat**: A utility for creating chat interfaces in Streamlit apps.
* **langchain.callbacks**: Provides callbacks for LangChain functionalities.
* **langchain.docstore.document**: Deals with document storage in LangChain.
* **langchain\_community.document\_loaders**: Specialized loaders for different document types like PDF and DOCX, as part of the LangChain community contributions.
* **tempfile**: Generates temporary files and directories, useful for handling file uploads.
* **zipfile**: Handles ZIP file extraction, which is necessary for processing ZIP archives containing documents.

**2. Setup and Initialization**

In this part of the code, we set up the application environment and initialize various components:

* **Environment Setup**: The **load\_dotenv()** function is used to load environment variables from a **.env** file. This step is crucial for securely accessing API keys and other configuration settings.
* **Streamlit Configuration**: **st.set\_page\_config(page\_title="Q/A with your file")** configures the Streamlit page with a title. This function sets the browser tab's title and can also be used to set other page properties.
* **Header and Session State Initialization**: The code sets a header for the web page using **st.header("Retrieval QA Chain")**. It also initializes the session state variables: **conversation**, **chat\_history**, and **processComplete**. These variables store the current state of the conversation, the history of the chat for display, and a flag to indicate if the processing of the uploaded documents is complete, respectively.
* **Sidebar for User Input**: Using **with st.sidebar**, a sidebar is created where users can upload files, choose options for PDF and DOCX loaders, select a vector store (Pinecone or Qdrant), pick an embeddings model, and choose a text splitting method. A button **process** is also provided for starting the processing of the uploaded files.

**3. File Uploading and Processing**

This section of the code handles the uploading and processing of files by users. It includes the following key steps:

* **Uploaded Files**: Users can upload files using the **st.file\_uploader** function. This function allows users to upload PDF, DOCX, and ZIP files. Uploaded files are stored in the **uploaded\_files** variable.
* **PDF and DOCX Loaders**: The code allows users to choose between different loaders for PDF and DOCX files. However, these options are currently commented out in your code.
* **Vector Store Selection**: Users can select a vector store option from either Pinecone or Qdrant using a radio button.
* **Embeddings Model Selection**: Users can choose between two embedding models: "intfloat/e5-large-v2" and "BAAI/bge-small-en-v1.5" using a radio button.
* **Text Splitting Method Selection**: Users can select the text splitting method from either "CharacterTextSplitter" or "RecursiveCharacterTextSplitter" using a radio button.
* **Processing Button**: The **Process** button triggers the processing of uploaded files when clicked. However, before processing, the code checks if the OpenAI API key is available. If not, it displays a message asking the user to add the API key.
* **Processing Logic**: The code then processes the uploaded files based on their types. For ZIP files, it extracts the contained files and processes them individually. For PDF and DOCX files, it extracts the text content. The chosen text splitting method is applied to split the text into chunks. These chunks are collected in the **text\_chunks\_list**.
* **Vector Store Creation**: If an embeddings model is selected, a vector store is created from the text chunks using the chosen vector store option (Pinecone or Qdrant).
* **Session State Update**: The session state is updated with the conversation, and the processing completion flag is set to **True**. A message is displayed to inform the user that the vector store and QA chain have been created.

**4. Embedding and Vector Store Selection**

In this part of the code, the user is provided with options to select the embeddings model and the vector store for managing the processed text data:

* **Embeddings Model Selection**: Users can choose between two pre-trained embedding models:
  + "intfloat/e5-large-v2": This is one of the available Hugging Face models.
  + "BAAI/bge-small-en-v1.5": Another Hugging Face model specifically designed for English text.

The selected embedding model determines how text is transformed into numerical vectors, which are essential for similarity calculations.

* **Vector Store Selection**: Users can choose between two cloud-based vector databases:
  + Pinecone: A cloud-based vector database service that provides efficient storage and retrieval of vector data.
  + Qdrant: A vector database with features for similarity search and retrieval.

The choice of vector store affects how the vector representations of text chunks are stored and queried.

**5. Text Chunking and Document Handling**

This section of the code is responsible for handling the text data extracted from the uploaded files and preparing it for further processing. It includes the following steps:

* **Text Chunking Methods**: The code allows users to select between two text splitting methods: "CharacterTextSplitter" and "RecursiveCharacterTextSplitter." These methods determine how the extracted text is divided into smaller, manageable chunks.
* **Text Chunking Logic**: Depending on the selected text splitting method, the code applies the chosen method to split the text into chunks. This is crucial for processing large documents efficiently.
* **Document Creation**: The code then creates documents from the text chunks. Each document includes a portion of the text content along with metadata that identifies the source of the text. These documents are stored in the **doc\_list**.
* **Metadata**: The metadata typically includes information like the source file name and page number, making it easier to track the origin of each text chunk.
* **Document List**: The **doc\_list** contains all the documents created from the text chunks. These documents serve as the basis for generating vector representations and performing similarity searches.

**6. Retrieval QA Chain Creation**

In this part of the code, the Retrieval QA (Question-Answer) chain is created, which enables the chatbot to respond to user queries based on the processed text data. Here's how it is done:

* **Retrieval QA Chain Creation**: The RetrievalQA chain is created using the **get\_qa\_chain** function. This chain combines several components:
  + Language Model (LLM): The GPT-3.5-turbo-16k model from OpenAI, which serves as the language model for generating responses.
  + Chain Type: The chain type is set to "stuff," which indicates the type of questions or queries the chatbot can handle.
  + Retriever: The retriever is configured to use the previously created vector store as the source for retrieving relevant documents for answering questions.
  + Number of Chunks: The **num\_chunks** parameter specifies the number of text chunks to consider when answering a question.
* **Session State Update**: The resulting QA chain is stored in the session state, allowing the chatbot to maintain its state and history throughout user interactions.

**7. User Interaction and Input Handling**

This section of the code handles user interactions and input processing. It enables users to ask questions and receive responses from the chatbot. Here's how it works:

* **User Input**: Users can input questions or queries in the provided chat input field.
* **Response Generation**: When a user submits a question, the code triggers the generation of a response from the chatbot. This is done by passing the user's question to the QA chain created earlier.
* **Response Display**: The generated response is displayed to the user in the chat interface.
* **Source Document Display**: If source documents are available for the response, the code also displays information about the source document, helping the user understand the context of the response.
* **Chat History**: The chat history is maintained in the **chat\_history** variable, storing both user questions and chatbot responses. This history is displayed in the chat interface to provide context for the conversation.
* **Input Validation**: User input is processed and validated to ensure it is in the correct format and conforms to expected standards.
* **Spinner**: While the response is being generated, a spinner is displayed to indicate that the chatbot is processing the query.
* **Session State Update**: The chat history is updated in the session state, ensuring that the conversation history is preserved across user interactions.

**8. Main Function**

The **main** function is the entry point of the chatbot application. It orchestrates the overall flow of the application and manages user interactions. Here's how it operates:

* **Environment Setup**: It starts by loading environment variables using **load\_dotenv()** to access sensitive information like API keys.
* **Streamlit Configuration**: The function configures the Streamlit app with a specific page title using **st.set\_page\_config**. This title appears in the browser tab.
* **Header**: It sets a header for the web page, displaying "Retrieval QA Chain" as a title.
* **Session State Initialization**: The code initializes session state variables, including **conversation**, **chat\_history**, and **processComplete**, to manage the state of the application.
* **Sidebar and File Upload**: In the sidebar, users can upload files of supported types (PDF, DOCX, ZIP). Options for PDF and DOCX loaders, vector store selection, embeddings model selection, and text splitting method selection are provided. A "Process" button triggers the file processing.
* **File Processing**: When the "Process" button is clicked, the code handles file processing, including extracting text from uploaded files, splitting text into chunks, and creating a vector store if an embeddings model is selected.
* **User Interaction**: If the file processing is complete (indicated by **processComplete**), users can input questions in the chat interface and interact with the chatbot.
* **Response Handling**: User questions trigger responses from the chatbot, and the responses are displayed in the chat interface. The chat history is updated accordingly.
* **Spinner for Response**: While the chatbot is generating a response, a spinner is displayed to indicate processing.
* **Conclusion**: The **main** function serves as the central control point for the application, managing initialization, file processing, and user interactions.

**9. Extracting Files from ZIP Archives**

In the code, there is a function **extract\_files\_from\_zip(uploaded\_zip\_file)** specifically designed to handle ZIP archives containing multiple files. Here's how it works:

* **ZIP File Extraction**: The function takes an uploaded ZIP file as input (**uploaded\_zip\_file**).
* **Temporary Directory**: It creates a temporary directory using **tempfile.mkdtemp()** to extract the contents of the ZIP file. This temporary directory is used to store the extracted files.
* **Extraction Process**: The code then uses the **zipfile.ZipFile** class to extract the contents of the uploaded ZIP file into the temporary directory. This process iterates through the ZIP file, extracting files with extensions ".pdf" and ".docx."
* **Extracted Files List**: The function compiles a list of extracted files along with their file names and paths. This list is returned as **extracted\_files** and contains tuples of the form **(file\_name, file\_path)**.
* **Temporary Directory Cleanup**: After extraction is complete, the temporary directory is not explicitly deleted in the code, but it is left to be managed by the operating system. In some cases, depending on the system and usage, temporary directories may be automatically deleted upon program exit.

**10. Handling PDF Text Extraction**

In the code, there are two methods for extracting text from PDF files:

* **Direct PDF Text Extraction**: In the **get\_pdf\_text** function, the code directly extracts text from PDF files using the **PdfReader** class from the PyPDF2 library. It iterates through the pages of the PDF and extracts text using **page.extract\_text()**. This method is straightforward and suitable for many PDFs.
* **PDF Text Extraction with PyMuPDF (MuPDF)**: In the **get\_pdf\_text\_pymupdf** function, the code processes PDF files using the PyMuPDFLoader. PyMuPDF (MuPDF) is used to load and extract text from PDF documents. This method provides an alternative approach for extracting text from PDFs.

**11. Handling DOCX Text Extraction**

In the code, text extraction from DOCX files is handled using two methods:

* **DOCX Text Extraction with Docx2txt**: In the **get\_docx\_text\_docx2txt** function, the code utilizes the **Docx2txtLoader** to extract text from DOCX files. This method relies on the Docx2txt library to process and extract text from DOCX documents. The extracted text is then returned.
* **Unstructured DOCX Text Extraction**: In the **get\_docx\_text\_unstructured** function, the code directly loads and extracts text from DOCX files using the **UnstructuredWordDocumentLoader**. This method is used when the Docx2txt library is not employed for text extraction.

**12. Creating and Storing Documents**

In this section of the code, documents are created and stored for further processing and analysis. Here's how it works:

* **Text Chunking**: Depending on the user's choice of text splitting method (either "CharacterTextSplitter" or "RecursiveCharacterTextSplitter"), the code applies the selected method to split the extracted text into smaller, manageable chunks. Text chunking is essential for processing large documents efficiently.
* **Document Creation**: Once the text is chunked, the code creates individual documents from these chunks. Each document contains a portion of the text content along with metadata that identifies the source of the text. This metadata typically includes information such as the source file name and page number.
* **Document List**: The documents are collected in a list called **doc\_list**. This list serves as a container for the individual documents, making it easier to manage and process them collectively.
* **Document Metadata**: The metadata associated with each document allows for tracking the source of each text chunk. This is important for maintaining context and traceability during further analysis and interactions with the chatbot.

**13 Conclusion and Additional Features**

In conclusion, our code provides a comprehensive solution for text data processing and question-answering. It empowers users to extract text from various document formats, create vector representations, and interact with a chatbot for information retrieval.

Additional Features:

While the current code provides essential functionality, there is room for expansion and enhancement. Here are some potential additional features:

1. User Authentication: Implement user authentication mechanisms to secure access to the application, especially when handling sensitive data.
2. Natural Language Understanding: Enhance the chatbot's natural language understanding capabilities for more accurate responses and a broader range of user queries.
3. Scalability: Optimize the code for scalability to handle larger datasets and more extensive text processing tasks.
4. Customization: Allow users to customize the chatbot's behavior and responses to better suit their specific use cases.
5. Visualization: Integrate data visualization tools to present processed information in a more user-friendly and visually appealing manner.
6. Multi-Language Support: Extend language support to cater to users who require text processing and chatbot interactions in different languages.
7. Data Export: Enable users to export processed data and chatbot conversations for further analysis or reporting.
8. Feedback Mechanism: Implement a feedback mechanism to collect user feedback and improve the chatbot's performance over time.
9. Error Reporting: Include an error reporting system to capture and log application errors for troubleshooting and debugging.
10. Deployment Options: Offer deployment options, including cloud hosting and containerization, to make it easier for users to deploy the application.