Interactive Q/A Bot with Document Upload

Pradeep Dubey

September 22, 2024

1 Introduction

The Interactive QA Bot is a web application designed to facilitate document-based question answering. Users can upload PDF files, ask questions about the content, and receive contextually relevant answers. The system integrates several advanced technologies: Pinecone for efficient document retrieval, Cohere for natural language generation, and Streamlit for the user interface.

2 Uploading Files

2.1 Application Launch

Open your web browser and navigate to the URL where the Streamlit application is hosted. You will see the application's title "Interactive QA Bot with Document Upload" at the top.

2.2 PDF Upload Interface

Find the file uploader widget in the center of the page labeled "Upload a PDF document". This widget allows you to choose a file from your local file system.

2.3 Selecting the File

Click on the uploader widget. A file dialog will appear. Navigate to the location of your PDF document, select it, and click "Open". Only PDF files are supported.

2.4 Processing the PDF

After selecting the file, the application starts processing the PDF. You will see a spinner icon with the message "Processing PDF...". This indicates that the file is being read and processed.

2.5 Text Extraction

The application uses `PyPDF2` to extract text from each page of the PDF. The extracted text is then prepared for embedding.

2.6 Embedding and Indexing

The extracted text is split into manageable chunks. Each chunk is embedded into a vector representation using the `sentence-transformers/all-MiniLM-L6-v2` model. The embeddings are uploaded to Pinecone for efficient retrieval.

2.7 Completion Notification

Once processing and indexing are complete, a success message "Document processed and embeddings saved!" will be displayed, indicating that the document is ready for querying.

3 Asking Questions

3.1 Query Input

Locate the text input field labeled "Ask a question about the document:". This field allows you to type in your question related to the uploaded document.

3.2 Query Submission

Type your question into the text input field. Ensure that your question is clear and specific to get the most accurate results.

3.3 Information Retrieval

After submitting your query, the application begins retrieving relevant information from the Pinecone index. A spinner icon with the message "Retrieving information..." will appear while the application performs the retrieval process.

3.4 Document Retrieval Process

The query is embedded into a vector using the same embedding model used for document text. The embedded query is used to query Pinecone for the most relevant document chunks based on similarity.

3.5 Displaying Retrieved Documents

The relevant document chunks are displayed under the section "Retrieved Documents:". Each chunk is shown with its content, which helps in understanding the context used for generating the response.

4 Viewing the Bot's Response

4.1 Generating the Answer

Once the relevant documents are retrieved, the application uses Cohere to generate a response. A spinner icon with the message "Generating answer..." will be displayed during the response generation process.

4.2 Contextual Answer Generation

The application aggregates the text from the retrieved documents to form a context. Cohere's model, `command-xlarge-nightly`, is used to generate a coherent and contextually accurate answer based on the provided context and user query.

4.3 Displaying the Answer

The generated answer is shown under the section "Generated Answer:". This provides the final response to the user's query, derived from the context of the document.

5 Challenges Faced and Solutions

5.1 Out-of-Range Error in Pinecone

Problem: During the integration of Pinecone for document retrieval, I encountered an "out-of-range" error when attempting to query or update the Pinecone database. This issue typically arises when invalid IDs or excessive data is stored, leading to retrieval or insertion failures.

Solution: I resolved the issue by clearing stored records in the Pinecone database. This involved reviewing the state of the database and ensuring that it was properly managed and reset when needed. After clearing the unnecessary records, the issue was resolved, allowing smooth interaction between the QA bot and Pinecone.

5.2 Token Limit Exceeded Error in Cohere API

Problem: The Cohere API returned a token limit exceeded error because the free version has a restriction of 100 tokens per minute. This caused interruptions in generating answers from the QA bot, especially when dealing with large documents or complex queries.

Solution: To bypass this tokenization limitation, I incorporated a more efficient approach using the `sentence-transformers/all-MiniLM-L6-v2` model from Hugging Face. By switching to this tokenizer and model, I was able to process larger chunks of data without hitting the Cohere API's token limits. The specific setup was as follows:

```
tokenizer = AutoTokenizer.from_pretrained('sentence-transformers/all-MiniLM-L6-v2')
model = AutoModel.from_pretrained('sentence-transformers/all-MiniLM-L6-v2')
```

This alternative not only optimized token usage but also improved the overall embedding quality of the documents.

5.3 Challenges in Embedding Text

Problem: Embedding large documents into vectors for Pinecone posed a significant challenge. Handling large amounts of text and ensuring that embeddings were accurate and efficiently stored was critical for effective document retrieval.

Solution: I found an efficient method to embed the text through a custom function that splits the document into manageable chunks and then upserts the resulting embeddings into the Pinecone index. This approach allows for better management of memory and processing time, especially for large documents.

```
def save_document_embeddings(text):
    docs = text.split("\n")
    for i, doc in enumerate(docs):
        vector = embed_text(doc)
        index.upsert(vectors=[(str(i), vector)])
    return docs
```

This function systematically processes each chunk of text, ensuring that embeddings are correctly stored in the database for fast and accurate retrieval during queries.

6 Example Interactions

6.1 Example 1

Document: A PDF titled "Introduction to Artificial Intelligence" is uploaded.
User Question: "What is Artificial Intelligence?"
Output: [Generated answer here]

6.2 Example 2

Document: A PDF titled "Machine Learning Algorithms" is uploaded.
User Question: "What is supervised learning?"
Output: [Generated answer here]

7 Deployment and Performance Considerations

7.1 Guidelines for Deployment

7.1.1 Containerization with Docker

Dockerfile Creation: Create a Dockerfile to containerize the application. The Dockerfile should include steps to install dependencies, set up the environment, and run the Streamlit server.

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8501
CMD ["streamlit", "run", "Front_End.py"]
```

Build and Run:

- Build the Docker image: `docker build -t qa-bot-app .`
- Run the container: `docker run -p 8501:8501 qa-bot-app`

7.1.2 Handling Large Documents

- **Chunking:** Implement chunking strategies to split large documents into smaller segments for processing and embedding.
- **Memory Management:** Monitor and manage memory usage to handle large document sizes without performance degradation.

7.1.3 Query Performance

- **Index Optimization:** Optimize Pinecone index configurations to handle high query loads efficiently.
- **Scalability:** Ensure the application is scalable to accommodate multiple concurrent users and large volumes of queries.