Gen AI Engineer / Machine Learning Engineer Assignment

Overview:

In this project, the goal is to build a **QA bot** that can retrieve relevant information from a dataset or document using a combination of **embeddings** stored in a vector database (such as **Pinecone**) and a **generative model** (such as Cohere or GPT-3) to generate answers based on the retrieved information. This approach is known as **Retrieval-Augmented Generation (RAG)**, and it's particularly useful for building conversational agents that can give contextually relevant, accurate answers.

Step-by-Step Implementation:

1. Data Preprocessing:

Before we get into the actual embedding creation, it's important to properly prepare the dataset. If the data is in PDF format or consists of unstructured text, we'll need to extract the text first. Tools like **PyPDF2** or **Textract** are great for this task, allowing us to parse out the raw text from the documents.

Once we have the text, we'll **chunk** it into smaller, manageable pieces. This is essential because larger documents can't be fed into the model directly, and breaking them into smaller segments (e.g., 200-300 words) ensures that the retrieval process will be more precise when users ask questions.

2. Embedding Creation:

Now that we have our text chunks, the next step is to create **embeddings**. This involves converting each text chunk into a numeric representation that the model can process. For this, we can use pre-trained models like **BERT** or **sentence-transformers**, both of which are accessible via libraries like Hugging Face.

This step is crucial for the retrieval process, as it allows the system to efficiently compare the user's query to the document's contents in a meaningful way.

Here's how the code looks:

```
from sentence_transformers import SentenceTransformer


model = SentenceTransformer('paraphrase-MiniLM-L6-v2')  # This is a lightweight model for creating embeddings

chunks = ["Text chunk 1...", "Text chunk 2...", "Text chunk 3..."]  # Split your document into chunks

embeddings = model.encode(chunks)
```

3. Storing Embeddings in Pinecone:

Once the embeddings are generated, we need to store them in a vector database like **Pinecone**. Pinecone is a highly efficient solution for managing and retrieving document embeddings. After setting up an account and initializing a vector index, you can upload your embeddings and link them to document IDs, making them searchable.

Here's a quick look at the setup:

```
import pinecone


pinecone.init(api_key='YOUR_API_KEY', environment='us-west1-gcp')

index = pinecone.Index("document-qa-bot")


# Store the document embeddings in the index

for idx, embed in enumerate(embeddings):

    index.upsert([(f"id_{idx}", embed)])
```

4. Query Processing:

When a user asks a question, the system needs to convert the query into an embedding (just like the document chunks) and then compare it to the stored embeddings in Pinecone. The top matches will be retrieved, giving us the most relevant pieces of text to base our answer on.

```
query = "What is the return policy?"

query_embedding = model.encode([query])

result = index.query(queries=query_embedding, top_k=5)  # Retrieve the top 5 most relevant chunks
```

5. Answer Generation Using a Generative Model:

Once we have the relevant document chunks, we use a **generative model** like Cohere or GPT-3 to create a coherent response. The retrieved chunks and the user's query are combined to form a prompt that guides the model in generating the most relevant answer.

Here's how that might look with Cohere:

```
import cohere


co = cohere.Client('YOUR_API_KEY')


response = co.generate(
```

```
    model='large',
    prompt=f"Context: {result['matches'][0]['text']} \n Question: {query}\n Answer:",
    max_tokens=100
)

print(response.generations[0].text)
```

6. Testing the Model:

To ensure everything is working correctly, it's important to test the model with several example queries. You'll want to check that the system is retrieving the correct document segments and generating coherent, contextually accurate answers.

For instance, if a user asks, "What are the product features?", the model should retrieve the relevant document section and generate an answer like, "The product features include X, Y, and Z."

Interactive QA Bot Interface (Part 2):

To make this system more user-friendly, the next step is to create an interactive interface where users can upload documents and ask questions in real-time. **Streamlit** or **Gradio** are excellent options for this.

Here's a high-level breakdown:

1. Frontend Interface:

Using **Streamlit**, we can build a clean, simple interface with a file uploader for PDF documents and a text input for queries. Once a user uploads a document, the backend processes it, creates embeddings, and stores them for retrieval.

Here's an example of the Streamlit code:

```
import streamlit as st
from PyPDF2 import PdfReader

st.title("QA Bot - Document Based Question Answering")

uploaded_file = st.file_uploader("Upload a PDF document", type="pdf")

if uploaded_file is not None:
```

```
    reader = PdfReader(uploaded_file)
    document_text = ""
    for page in reader.pages:
        document_text += page.extract_text()

    st.write("Document uploaded successfully.")

    user_query = st.text_input("Enter your query:")

    if user_query:
        answer = generate_answer(document_text, user_query)  # Call backend function
        st.write(f"Answer: {answer}")
```

2. Backend Integration:

The backend will use the same process described in Part 1 to handle PDF text extraction, embedding creation, and answer generation.

3. Handling Multiple Queries:

To make sure the system is efficient, especially when handling multiple queries, caching the document embeddings after upload is critical. This prevents the need to re-process the document for every query.

Here's an example of how to cache the embeddings in Streamlit:

```
@st.cache
def create_embeddings_for_document(document_text):
    chunks = chunk_document(document_text)
    document_embeddings = create_embeddings(chunks)
    return document_embeddings
```

Final Steps:

1. **Testing the Bot**: Make sure to run several tests with different documents and queries to ensure that the retrieval and generation processes are working smoothly.
2. **Deployment**: Once everything is working, you can containerize the app using Docker and deploy it on a cloud service like **Heroku**, **AWS**, or **Streamlit Cloud**.