Task 1

Develop an LLM-based Retrieval-Augmented Generation (RAG) system designed to handle 10 or more documents. The system will use an open-source vector database for document vectorization and incorporate a user-friendly interface built with Gradio or Streamlit. The interface will feature two primary screens:

- **Document Upload Screen**: Allows users to upload documents.
- **Chatbot Screen**: Provides a chat interface where users can select an LLM model and engage in conversation.

Key Features to Implement:

- **PDF Reading and Vectorization**: Extract text, images, tables, formulas, and charts and process them from PDF documents for vector storage.
- **Semantic Similarity Search**: Enable efficient retrieval of relevant content based on user queries.
- **Support for Open-Source LLM Models**: Integrate five different models to provide a range of options for user interaction.
- **Prompt Engineering**: Optimize prompts for effective responses and better user experience.
- **UI Development**: Design a seamless and intuitive interface for both document upload and chatbot interactions.

Task 1.1

- PDF Text Extraction using ocr (3 DEC)
- Extraction done (if image in a graph format then how to extract graph without label?)

Task 1.2

- Document Preprocessing (long documents into smaller chunks)
- Lemmatization(nlp based), Handling Case Sensitivity

—----------testing—----------

Task 1.3

- Learn Vectorization of Documents
- Count vectorization, TF-IDF, Word Embeddings, Doc2Vec,
- One-Hot Encoding, Position Encoding (in NLP and Transformers), Hashing (Feature Hashing), Latent Semantic Analysis (LSA), Latent Dirichlet Allocation (LDA), BERT and Other Transformer-based Models, BERT

Task 1.3.1

- Use model **text-embedding-ada-002**

Task 1.4

- Learn RAG

Task 1.4.1

–

—----------testing—----------

Task 1.5

- Prompt Engineering

Llm hsging fscr model llm model impediments

Rag concept

Sample Electric Bill

The information needed to generate electric bills for the tenants of a master metered mobile home park can be found on the park's monthly electric bill. To determine the effective rate per kWh (see Commodity Charge below), please refer to the Public Utilities Commission of Nevada's sample form, "Computing the Effective Rate per kWh for a Master Meter Park – Electric Bill," available online at puc.nv.gov/Utilities/MHP/Sample_Forms/. Complete Section 1 for each occupied space. If each space is equipped with individual lot meters, complete Section 2 and stop. If each space is not equipped with individual lot meters, skip Section 2 and complete Section 3.

SECTION 1

Tenant Name Individual Tenant **Space No:** 1

Billing Period Jul 1 – Jul 31

SECTION 2

Meter Reading

This Month = 1200 **(1)**

Last Month = 850 **(2)**

Usage (Difference kWh) = 350 **(3)**

Commodity Charge

Usage in kWh 350 **(4)** x Effective Rate $ .12665 **(5)** = $ 44.33 **(6)**

Parks Service Charge (Prorated) = $ 2.67 **(7)**

Tenant Service Charge (Optional) = $ 0 **(8)**

Amount Due = $ 47.00 **(9)**

Step 1: Set Up the Environment

- **Install Dependencies:**
  - Install Python, if not already done.
  - Install required libraries for document processing (e.g., `PyPDF2`, `pdfplumber` for text extraction, `langchain`, `faiss` or `milvus` for vector storage, `gradio` or `streamlit` for UI, `transformers` for LLMs).

bash
Copy code

```
pip install langchain faiss-cpu gradio transformers streamlit pypdf2 pdfplumber
```

-
- **Set Up the Vector Database:**
  - Choose an open-source vector database such as FAISS, Milvus, or Pinecone.
  - Install and configure the vector database to store document vectors for semantic search.

Step 2: PDF Document Upload and Extraction

- **PDF Upload Functionality:**
  - Create the interface for uploading documents using Gradio or Streamlit.
  - Define a function to handle PDF uploads.
- **PDF Text Extraction:**
  - Use libraries like `pdfplumber` or `PyPDF2` to extract raw text from the uploaded PDFs.
  - For extracting tables, images, and charts, leverage `pdfplumber` (for tables) or use OCR tools like `pytesseract` for images.

    - Store the extracted content (text, tables, charts) into the system.
- **Document Preprocessing:**
    - Clean the text (remove unnecessary characters, newlines, etc.).
    - Process and split long documents into smaller chunks (e.g., by paragraphs or sentences) for better vectorization and search.

Step 3: Vectorization of Documents

- **Text Embedding:**
    - Use an embedding model (like OpenAI's `text-embedding-ada-002`, Sentence-BERT, or a similar model) to convert text from the documents into vectors.
    - Process the chunks of text from the documents and generate vectors for each chunk.
- **Store Vectors in the Database:**
    - After generating embeddings for each chunk, store them in the vector database (FAISS or Milvus).
    - Ensure each vector entry has a reference to the corresponding document section, so it can be retrieved easily during queries.

Step 4: Build Retrieval System

- **Semantic Search:**
    - Implement a retrieval system that performs semantic similarity searches in the vector database.
    - Given a user query, convert the query into a vector and search for the most relevant document chunks based on cosine similarity or other distance metrics.
    - Retrieve top-N relevant chunks (text, tables, images) that are semantically similar to the query.

Step 5: Integrate Open-Source LLM Models

- **LLM Selection:**
  - Choose 5 different open-source LLM models (e.g., `GPT-2`, `GPT-Neo`, `LLaMA`, `Bloom`, or `T5`).
  - Use the `transformers` library to integrate these models and allow users to select a model for interaction.
- **Prompt Engineering:**
  - Design and optimize the prompts for generating responses based on the retrieved information.
  - Use context from the retrieved documents to form prompts, ensuring that the LLM can answer user queries in a coherent and contextually accurate manner.
- **LLM Query Handling:**
  - For each query, retrieve relevant document chunks and append them as context to the input prompt.
  - Feed the prompt into the selected LLM model and generate a response.
  - Optimize the generation process for clarity, conciseness, and relevance.

Step 6: User Interface Development

- **Document Upload Screen:**
  - Use Gradio or Streamlit to create a user-friendly interface where users can upload PDFs.
  - Display an upload button that triggers the document extraction and processing.
  - Show progress indicators or feedback on the extraction status.
- **Chatbot Interface:**
  - Create a chat interface where users can enter queries.

    - Provide a dropdown to allow users to select an LLM model (e.g., GPT-2, GPT-Neo, etc.).
    - Display the generated responses in a chat-like format, allowing users to interact with the system conversationally.
- **Feedback Mechanism:**
    - Implement a feedback mechanism to refine and improve the LLM's responses over time (e.g., thumbs up/down).

Step 7: Testing and Optimization

- **Test Document Upload:**
    - Test the document upload functionality with different PDF formats to ensure text extraction works accurately (check tables, images, and text).
- **Evaluate Vectorization:**
    - Test the accuracy and efficiency of semantic similarity search by querying different topics and evaluating the returned results.
- **Optimize LLM Interactions:**
    - Test the chatbot interface to ensure that different models respond correctly to user queries.
    - Fine-tune the prompts and LLM configurations for optimal performance.

Step 8: Deployment

- **Host the Application:**
    - Use services like Hugging Face Spaces, Heroku, or AWS to deploy your application.
    - Ensure the vector database is hosted in a scalable manner for handling multiple queries concurrently.
    - Deploy the interface and make the system accessible to users via a web browser.
- **User Access and Documentation:**

    - Provide user documentation for using the platform, explaining the upload process, selecting models, and interacting with the chatbot.

Step 9: Maintenance and Updates

- **Monitor Performance:**
    - Continuously monitor the system's performance, including document upload time, search query speed, and LLM response quality.
- **Iterate and Improve:**
    - Collect user feedback to improve the system.
    - Add new features like supporting other document formats (e.g., Word, PowerPoint) or new models.

By following these steps, you'll be able to build and deploy a Retrieval-Augmented Generation system that can handle a range of document types, provide semantic search capabilities, and allow users to interact with multiple open-source LLMs.

The **Vectorization** step is crucial in your Retrieval-Augmented Generation (RAG) system as it converts text data (from documents) into numerical representations (vectors) that a machine can process

for similarity-based retrieval and further processing by a language model. Here's a detailed explanation of the vectorization process:

What is Vectorization?

Vectorization refers to the process of converting textual data (like the content of your documents) into fixed-length vectors (numerical representations) that capture the meaning of the text. These vectors are used for tasks like semantic search, where you compare the similarity of different pieces of text, and for feeding into a language model like GPT, BERT, or others.

Steps Involved in Vectorization:

1. **Text Preprocessing:** Before vectorizing, you need to clean and preprocess the text to ensure it's in the best format for vectorization. This includes:

    - **Lowercasing:** Convert all text to lowercase to avoid case-sensitive differences.
    - **Removing Special Characters:** Remove unwanted characters like punctuation, extra spaces, or HTML tags.
    - **Tokenization:** Split the text into smaller pieces (tokens), like words or subwords. This is usually done with a tokenizer from an NLP library.
    - **Stopword Removal (optional):** Remove common words (e.g., "the", "and", "is") that don’t add much meaning in the context.
    - **Stemming or Lemmatization (optional):** Reduce words to their base or root form (e.g., “running” becomes “run”).
2. **Choosing a Vectorization Method:** The next step is selecting a method to transform the preprocessed text into vectors. There are several techniques available, ranging from traditional methods to state-of-the-art transformer-based models.

Here are some popular methods:

1. TF-IDF (Term Frequency-Inverse Document Frequency):

- **What it is:** It calculates the importance of each word in a document relative to its frequency across a corpus.
- **How it works:**
  - **Term Frequency (TF):** Measures how frequently a term appears in a document.
  - **Inverse Document Frequency (IDF):** Measures how important a word is across all documents. Words that appear in many documents are considered less important.
- **Use Case:** TF-IDF is simple and effective for smaller datasets or when you just need to capture basic word frequency information. However, it doesn't capture semantic meaning beyond word frequencies.

3. **2. Word2Vec:**

- **What it is:** Word2Vec is a model that creates word embeddings (vectors) based on the context of words in large text corpora. It learns vector representations by considering nearby words in the text (context).
- **How it works:** Words with similar meanings will have similar vectors (e.g., "dog" and "puppy" might have very similar vectors).
- **Use Case:** Useful for general semantic representation of words, but it does not consider the entire sentence or document context.

4. **3. Doc2Vec (Paragraph Vectors):**

- **What it is:** An extension of Word2Vec that learns fixed-length vectors for entire sentences or documents rather than just individual words.
- **How it works:** In addition to word embeddings, Doc2Vec learns a vector for each document or sentence, capturing the overall meaning and context.
- **Use Case:** Good for documents or sentences where the context across words matters (i.e., understanding entire paragraphs or sections of a document).

5. **4. BERT (Bidirectional Encoder Representations from Transformers):**

    - **What it is:** A transformer-based model that provides contextual embeddings for words based on both the preceding and succeeding context. This is different from Word2Vec, which only looks at local context.
    - **How it works:** BERT generates embeddings that consider the full context of the sentence, allowing it to capture more complex meanings.
    - **Use Case:** Great for context-rich and complex text. Often used for tasks such as question answering or semantic search, where the meaning of words is deeply tied to the surrounding text.
    - **Hugging Face's Transformers library** allows you to use pretrained BERT or similar models, such as RoBERTa or DistilBERT, to obtain high-quality embeddings.

6. **5. Sentence Transformers (e.g., Sentence-BERT):**

    - **What it is:** A variant of BERT optimized for generating sentence-level embeddings. It uses the BERT model but is fine-tuned specifically to produce fixed-size vectors for sentences, rather than individual words.
    - **How it works:** This approach produces embeddings for sentences or paragraphs rather than individual words.

These sentence-level embeddings are well-suited for semantic search.
    - **Use Case:** Ideal for document-level vectorization, particularly when you need to search for relevant documents or paragraphs based on meaning rather than keyword matching.
    - **Example:** You can use models like `sentence-transformers/all-MiniLM-L6-v2` available in the Hugging Face Model Hub.
7. **6. OpenAI's Embeddings (e.g., Ada, Curie):**
    - **What it is:** OpenAI provides powerful embeddings through its GPT-3 models. You can use these embeddings for semantic search and similar tasks.
    - **How it works:** OpenAI's models provide high-quality vector representations of text that capture the meaning of the text.
    - **Use Case:** Very effective if you have access to OpenAI's API and prefer not to deal with setting up your own model or fine-tuning. OpenAI embeddings are commonly used for semantic search and generating responses in conversational systems.

Step 3: Generating Document Vectors

Once you have selected the method to vectorize your documents, you will perform the following steps:

Load Pretrained Model: If you're using a transformer-based model like BERT, you can use libraries like `Hugging Face Transformers` to easily load pretrained models. For example, if you're using Sentence-BERT:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
```

1.

Vectorize the Document: You can now feed your document (or a chunk of it, like a paragraph or section) into the model to generate the corresponding vector.

```
document_text = "Your document text goes here"

document_vector = model.encode(document_text)
```

For non-transformer models like Word2Vec or TF-IDF, you can use libraries like `Gensim` or `Scikit-learn` respectively:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()

document_vector = tfidf_vectorizer.fit_transform([document_text])
```

2.

Store the Vectors in a Database: After vectorizing the documents, you store the generated vectors in a vector database (like `FAISS`, `Weaviate`, or `Chroma`). This database will allow you to efficiently retrieve the most similar vectors to any query vector later on.

Example using `FAISS`:

```
import faiss

index = faiss.IndexFlatL2(d)  # d is the dimension of the vector
```

index.add(document_vector) # Add the document vectors to the index

3.

Step 4: Using the Vectors for Retrieval

Now that the documents are vectorized and stored, you can retrieve relevant documents based on user queries. For this:

1. Vectorize the user query.
2. Perform a similarity search by comparing the query vector with the document vectors in the vector database.
3. Retrieve the top N most similar documents based on their cosine similarity or dot product.

Example query vectorization and search:

query_text = "What is the importance of vectorization?"

query_vector = model.encode(query_text)

D, I = index.search(query_vector, k) # Search for the top k similar documents

---

Conclusion

The vectorization step is a key part of your RAG system that transforms your textual data into a machine-readable format, enabling efficient semantic search and interactions with language models. By selecting an appropriate vectorization method based on the

complexity and requirements of your application, you ensure that your system can handle and retrieve the most relevant information from the documents in response to user queries.