Sample Set

Sample set is a Rag application project

Anshu Kumar Singh

Mob: 9896843511

anshukumar9728@gmail.com

Overview

In recent advancements in Natural Language Processing (NLP), Retrieval-Augmented Generation (RAG) models have shown promise in enhancing the quality of generated responses by integrating retrieval mechanisms with generative capabilities. This document explores two innovative techniques for optimizing RAG models, aiming to improve relevance, accuracy, and overall performance.

In that Assignment I have used the Agentic RAG for optimizing retrieval and improve the relevance accuracy and overall performance

Objective and Scope

The primary objective of this project is to develop a robust Retrieval-Augmented Generation (RAG) system that efficiently integrates a vector database for enhanced information retrieval and response generation. The scope encompasses the installation of necessary dependencies, data loading, and the implementation of various tools to streamline the process, ultimately aiming to improve the accuracy and relevance of generated responses in natural language processing tasks using langchain and langgraph.

Installing Dependency

Libraries Used

- **Langchain and Langgraph**: These libraries facilitate the integration of large language models (LLMs) with vector databases, enabling seamless information retrieval and processing.
- **Groq**: This library serves as the chat model, providing advanced capabilities for generating conversational responses.
- **Pinecone**: Utilized as the vector database, Pinecone vector database efficiently stores and retrieves vector representations of data, enhancing the overall performance of the retrieval-augmented generation system.

Data Loading and Text embeddings

In this implementation, I utilized LangChain to load PDF data and convert it into a **LangChain Document** object. This document is then processed and stored in the

Pinecone vector database for efficient retrieval during question-answering tasks.

Steps Overview:

1. **PDF Data Loading**:
    - The PDF data is first read and loaded into the LangChain document format. This format is compatible with various downstream tasks like chunking, embedding generation, and storage.
2. **Chunking the Document**:
    - Large documents exceed the context window limitations of LLM models. The context window refers to the maximum amount of data that an LLM can process in a single request.
    - To ensure the model can handle the data, I implemented a chunking process. This breaks the document into smaller, manageable parts before passing it to the model for embeddings and response generation.
3. **Embeddings using Hugging Face**:
    - For embedding generation, I utilized the **Hugging Face Hub** model: `Snowflake/snowflake-arctic-embed-s-small`. This embedding model was chosen due to its **high accuracy** and **small size**, which makes it both efficient and effective for generating meaningful embeddings for document chunks.
4. **Storing in Pinecone**:
    - Once the document chunks are embedded, they are stored in **Pinecone**, a cloud-based vector database.
    - Pinecone stores these embeddings, allowing efficient similarity searches during query processing. When a user poses a question, the system retrieves relevant chunks based on vector similarity, ensuring quick and relevant responses.

Key Tools:

- **LangChain**: Handles document loading and chunking to prepare data for vector storage and retrieval.
- **Pinecone**: A vector database used to store embeddings and enable fast retrieval of relevant documents.
- **Hugging Face Model**: The `Snowflake/snowflake-arctic-embed-s-small` model is used for generating accurate embeddings for the document chunks.

By combining these technologies, the system efficiently processes large documents,

breaking them into smaller, query-able chunks and storing them for fast, accurate retrieval during the chatbot's operations.

```
def get_embedding_model(model_name="Snowflake/snowflake-arctic-embed-s"):
    encode_kwargs = {'normalize_embeddings': True} # set True to compute cosine similarity
    base_embedding_model = HuggingFaceEmbeddings(
        model_name=model_name,
        # model_kwargs={'device': device},
        encode_kwargs=encode_kwargs
    )
    return base_embedding_model
def create_documents_chunks(documents):
    text_splitter = RecursiveCharacterTextSplitter(
        # Set a really small chunk size, just to show.
        chunk_size=1000,
        chunk_overlap=200,
        length_function=len,
        is_separator_regex=False,
    )
    docs = text_splitter.split_documents(documents)
    return docs
```

```
class VectorStore():

    def __init__(self, index_name, embedding_model) -> None:

        self.index_name = index_name

        self.embedding_model = embedding_model

        self.pc = Pinecone(api_key=os.getenv('PINECONE_API_KEY'))

        self.index = None



    def pinecone_index(self):

        # if not self.pc.has_index(self.index_name):

        if not self.pc.list_indexes().get('indexes'):

            self.pc.create_index(

                name=self.index_name,

                # dimension=len(self.embedding_model.embed_query("Hi")).

                # dimension=384,

                dimension=len(self.embedding_model.embed_query('hi')),

                metric="cosine",

                spec=ServerlessSpec(

                    cloud='aws',

                    region='us-east-1'

                )

            )

            while not
```

```
self.pc.describe_index(self.index_name).status['ready']:

                time.sleep(1)

        index = self.pc.list_indexes().get('indexes')

        # print("Index name", index[0].get('name'))

        self.index = index[0].get('name')

        if self.index:

            print(f"Pinecone Index {self.index_name} Created
Successfully")

        else:

            print(f"Pinecone Index {self.index_name} Creation Failed"

    def store_vector_embedding_to_pinecone(self, documents):



        # index = self.pinecone_index()

        self.pinecone_index()

        print(" Data Storinig...   Pinecone Index Name",self.index)

        if self.index:



            documents_Chunk =create_documents_chunks(documents=documents)

            vector_db = PineconeVectorStore.from_documents(

                documents_Chunk,

                index_name = self.index_name,

                embedding=self.embedding_model
```

```
            )

            print('Data stored succusfully')

            return vector_db

        print("Cannot store vector embeddings.")

        return None
```

Model Utilization

In this section, I utilized the **Groq** module for model implementation due to its capability to offer **low inference times**, which significantly reduces response latency. The Groq platform provides optimized hardware acceleration, making it ideal for applications that require fast, efficient responses.

To integrate Groq into the project, the following steps were taken:

1. **Groq API Key Setup**:
    - I initialized the Groq module by providing my **Groq API key**. This enables seamless access to the platform's accelerated model serving capabilities.
2. **Model Selection**:
    - For this project, I selected the **Meta LLaMA model** variant, known for its effectiveness in handling Retrieval-Augmented Generation (RAG) applications. The Meta LLaMA model is highly capable of generating high-quality, contextually relevant responses.
3. **Performance Benefits**:
    - The Groq infrastructure enhances the model's performance by reducing inference times, ensuring that responses are delivered quickly and efficiently. This makes the chatbot both **responsive** and **scalable** for real-time user interactions, without compromising on the relevance or accuracy of generated content.

By leveraging Groq's hardware-accelerated environment and the Meta LLaMA model, the project ensures **rapid and efficient response generation** while maintaining the high-quality performance needed for RAG-based applications

Implementing Self RAG (Retrieval-Augmented Generation)

In this implementation, I have utilized **Self-RAG** to enhance accuracy. Self-RAG is an advanced strategy for **Retrieval-Augmented Generation (RAG)** that incorporates self-reflection and self-grading on both retrieved documents and generated responses. This iterative process ensures that the responses are both relevant and grounded in the provided data.

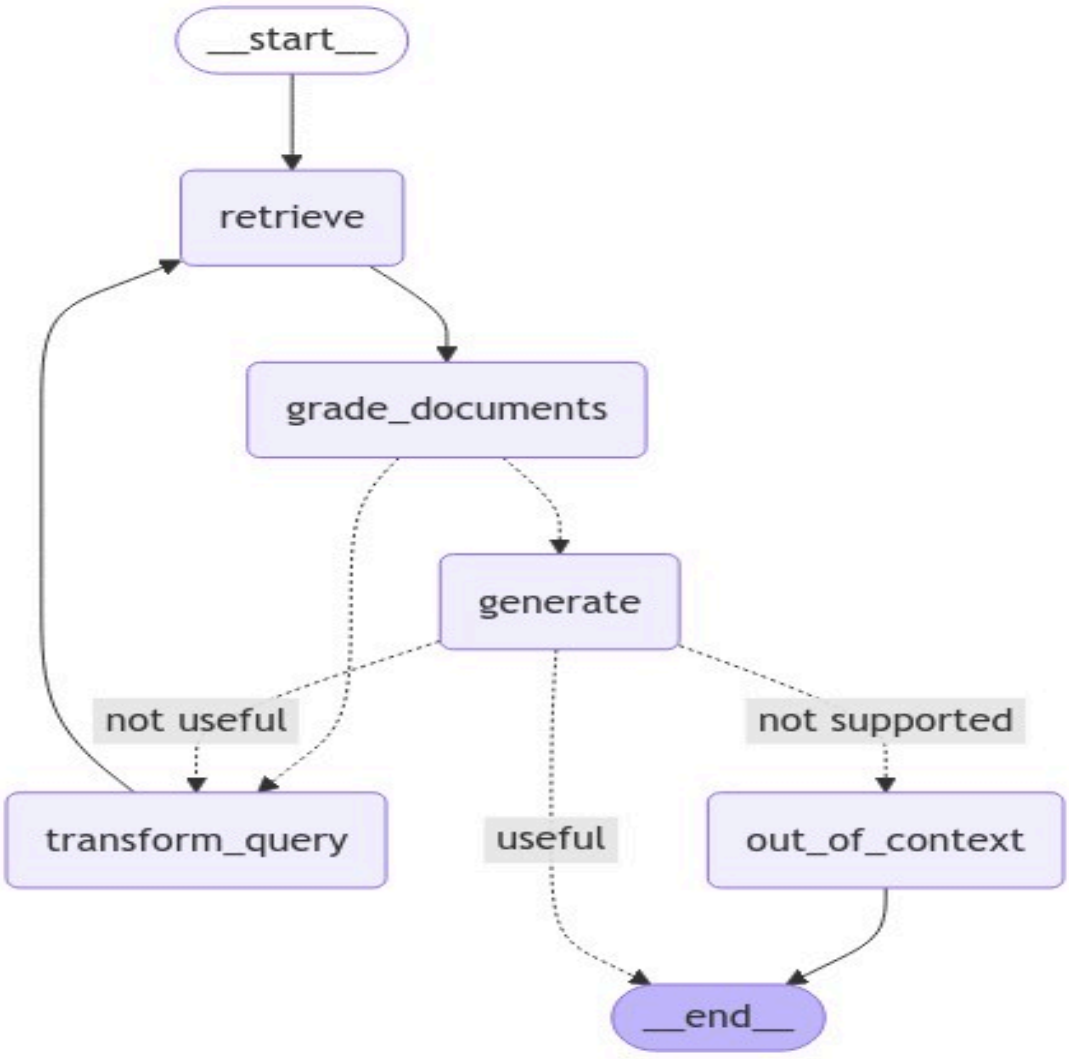
Workflow Overview:

1. **Query Input**:
    - The process begins with the user submitting a query. This query serves as the input for the document retrieval step.
2. **Document Retrieval**:
    - The system retrieves relevant documents from a vector database (e.g., Pinecone) based on the user query.
    - I used **LangChain's document objects** to load the PDF data, which was chunked into smaller segments to handle the context window limitations of the model effectively. These chunks are then stored in the vector database using embeddings from the **Snowflake/snowflake-arctic-embed-s** model (selected for its high accuracy).
3. **Self-Reflection and Document Relevance Grading**:
    - After retrieval, the system employs a **self-grading mechanism** to evaluate the relevance of the retrieved documents against the query.
    - The model assesses whether the documents contain information that directly addresses the query, and irrelevant documents are filtered out before proceeding.
4. **Answer Generation**:
    - With the filtered, relevant documents, the model generates an answer.
    - The Self-RAG process then evaluates whether the generated response is well-grounded in the retrieved documents, using self-assessment techniques to check for any hallucinations or unsupported content.
5. **Self-Grading of Generated Answers**:

- The generated response is also graded for accuracy, ensuring that it resolves the user's query effectively.
- If the response is deemed inaccurate or unsupported by the retrieved content, the process is repeated with a refined query.

6. **Final Answer**:
   - After passing the self-grading steps, the final answer is presented to the user, ensuring a higher degree of accuracy and relevance.

This iterative self-reflection process within Self-RAG significantly enhances the quality of the generated responses, minimizing hallucinations and ensuring that the answers are well-supported by the retrieved content.



User Interface

For the user interface, I used **Streamlit**, a powerful and easy-to-use framework for building interactive web applications in Python. Streamlit allows for seamless integration of user input, enabling users to interact with the chatbot, upload documents, and query the system. The interface is designed to be user-friendly and intuitive, allowing for smooth communication with the backend retrieval and generation system.

Deployment

To ensure easy deployment and containerization, I utilized **Docker**. Docker simplifies the process of packaging the application along with all its dependencies into a lightweight, portable container. This makes the deployment process efficient and scalable, allowing the application to run consistently across different environments. By containerizing the application, we ensure that the system is easy to maintain, update, and deploy on various platforms, making it highly flexible for future development.