RAG System: Model Architecture, Retrieval Approach, and Generative Responses

This document explains the architecture, retrieval approach, augmentation, and generative response creation of the **Retrieval Augmented Generation (RAG)** system. The system combines three key components: **retrieval** of relevant information from a PDF document, **Augmenting** it with the user query and **generation** of answers using a language model. It leverages document retrieval, augmentation techniques, and generative capabilities to provide accurate, contextually relevant responses.

1. **Retrieval**: It extracts relevant content from a large body of text (in this case, a PDF document) using vector embeddings.
2. **Augmentation**: Augment the query from the user along with the relevant content.
3. **Generation**: It uses a generative language model to create natural language responses based on the retrieved context.

Model Architecture

The architecture of the RAG system can be broken down into three main stages: document embedding, retrieval, and generative response creation. Each stage is designed to ensure that the answer is both contextually accurate and linguistically fluent.

1. Document Embedding

After loading the PDF document, the text is processed into smaller chunks for efficient retrieval. These chunks are then converted into vector embeddings using a pre-trained model (thenlper/gte-small). Note that max_seq_length': 512 this is the reason I choose chunk size as 500.

- **Embedding Model**: The embedding model transforms the textual content of each chunk into a high-dimensional vector representation. This vector encodes the semantic information of the text, enabling the system to perform similarity searches later.
- **Chunking**: The document is split into 500-character chunks with a 50-character overlap. Chunking ensures that the system retains sufficient context while keeping the input manageable.

2. Approach to Retrieval

Retrieval is a crucial step where the system identifies relevant pieces of text from the document that match the user's query.

- **Chroma Vector Database**: Once the document is split and embedded, the embeddings are stored in a vector database using Chroma. This database allows for fast and efficient similarity-based searches.
- **Similarity Search**: When a user asks a question, the query is converted into an embedding using the same model. The system then performs a **similarity search** within the vector database, finding the top 3 most relevant chunks based on the cosine distance between the query embedding and the stored embeddings.

This approach ensures that the system retrieves the most relevant context to answer the user's question accurately.

3. Generative Response Creation

Once the relevant document chunks are retrieved, the generative model uses this context to produce an answer to the query.

- **OpenAI Chat Model**: The language model, initialized with the OpenAI API, receives the retrieved context along with the user's question. It is designed to use this information to generate fluent, contextually relevant responses.
- **Chat Prompt Template**: The retrieved chunks are formatted into a pre-defined prompt structure. The template presents System Message, the retrieved context followed by the user's question, guiding the model to focus on answering based on the provided information.
- **Output Parsing**: The response from the chat model is processed through an output parser (StrOutputParser), ensuring that it returns a clean and human-readable string.

4. RAG Chain Execution

The entire process is wrapped into a Retrieval Augmented Generation (RAG) chain. This chain executes the following steps sequentially:

1. Takes the user's question.
2. Retrieves relevant document chunks using the vector embeddings.
3. Formats the context and question in a prompt template.
4. Sends the final prompt to the language model for generation.
5. Converts the model's output into a string and returns it to the user as the answer.

Retrieval Approach

The retrieval mechanism is built to be fast and scalable, ensuring that the most relevant sections of the document are surfaced for answering any given query. Here's how it works:

1. **Embedding Creation**: Each chunk of the document is converted into a vector embedding using a pre-trained model (thenlper/gte-small).
2. **Vector Storage**: These embeddings are stored in a vector database (Chroma) that allows for similarity-based searches.
3. **Query Embedding**: The user's question is also embedded into a vector format, which is then compared against the stored embeddings to find relevant chunks.
4. **Similarity Search**: The system calculates the cosine distance between the query embedding and the stored embeddings, retrieving the top 3 most similar chunks. This method ensures that the retrieved content is highly relevant to the user's query.

Generative Response Creation

After retrieving the most relevant document chunks, the system proceeds to generate a response using a language model.

1. **Chat Prompt Template**: The system utilizes a prompt template that guides the model to respond concisely and only using the information provided in the retrieved context.

2. **OpenAI Chat Model:** The OpenAI GPT Chat model takes the user's question along with the retrieved document chunks and generates a natural language response. The model is context-aware and uses the retrieved text to ensure that the answer is factually correct and contextually appropriate.

3. **Structured Response:** To ensure that the output remains concise and relevant, the system converts the generative model's response into a structured format using the StrOutputParser.