**Naive RAG Implementation**

**1. Introduction to Retrieval-Augmented Generation (RAG)**

Retrieval-Augmented Generation (RAG) is a technique that enhances the capabilities of Large Language Models (LLMs) by integrating them with an external knowledge base. This approach allows LLMs to generate responses that are not only fluent but also factually accurate and up-to-date, overcoming the limitations of models trained on static datasets. In essence, RAG enables an LLM to "look up" information before generating an answer, much like a human would consult a reference book.

**2. The Provided RAG Code: A Step-by-Step Breakdown**

The Python code provided implements a basic RAG pipeline. Let's break down each part to understand its role in the overall process.

**2.1. Setup and Dependencies**

**import os**

**from dotenv import** load\_dotenv

**import chromadb**

**import google.generativeai as genai**

**from chromadb.utils.embedding\_functions import**

SentenceTransformerEmbeddingFunction

**from pypdf import** PdfReader

**from helper\_utils import** word\_wrap

**from langchain.text\_splitter import** (

RecursiveCharacterTextSplitter,

SentenceTransformersTokenTextSplitter,

)

load\_dotenv()

gemini\_api = os.getenv("GEMINI\_API\_KEY")

This section handles the initial setup. It imports necessary libraries such as os for environment variables, dotenv for loading API keys, chromadb for the vector database, google.generativeai for interacting with the Gemini LLM, pypdf for reading PDF files, and langchain.text\_splitter for breaking down text into manageable chunks. The helper\_utils.word\_wrap is also imported, presumably for formatting the output.

**2.2. Data Loading and Preprocessing**

reader = PdfReader("./1408487-EN.pdf")

pdf\_texts = [p.extract\_text().strip() **for** p **in** reader.pages]

pdf\_texts = [text **for** text **in** pdf\_texts **if** text]

character\_splitter = RecursiveCharacterTextSplitter(

separators=["**\n\n**", "**\n**", ". ", ", ", " ", ""], chunk\_size=1000, chunk\_overlap=20

)

character\_split\_texts = character\_splitter.split\_text("**\n\n**".join(pdf\_texts))

token\_splitter = SentenceTransformersTokenTextSplitter(

chunk\_overlap=0, tokens\_per\_chunk=256

)

token\_split\_texts = []

**for** text **in** character\_split\_texts:

token\_split\_texts += token\_splitter.split\_text(text)

Here, the code first loads text from a PDF file named 1408487-EN.pdf . It then processes this text using two types of text splitters:

**RecursiveCharacterTextSplitter** : This splitter attempts to split text by various separators (like double newlines, single newlines, spaces, etc.) to keep chunks semantically coherent, with a specified chunk\_size and chunk\_overlap .

**SentenceTransformersTokenTextSplitter** : This splitter further refines the chunks based on token limits, which is crucial for embedding models that have specific input token constraints. This ensures that each piece of text is small enough to be processed by the embedding model.

This process transforms a large document into smaller, manageable, and semantically meaningful text chunks.

**2.3. Embedding and Vector Database Storage**

embedding\_function = SentenceTransformerEmbeddingFunction()

chroma\_client = chromadb.Client()

chroma\_collection = chroma\_client.create\_collection(

"microsoft-collection", embedding\_function=embedding\_function

)

ids = [str(i) **for** i **in** range(len(token\_split\_texts))]

chroma\_collection.add(ids=ids, documents=token\_split\_texts)

This is where the

retrieval part of RAG comes into play. An embedding\_function (likely a Sentence Transformer model) is initialized to convert text chunks into numerical vector embeddings. These embeddings are then stored in a chromadb collection, which acts as our vector database. Each text chunk is assigned a unique ID. This step is crucial because it allows for efficient semantic search later on; when a query comes in, it will be embedded, and the vector database will quickly find the most similar text chunks.

**2.4. LLM Configuration**

genai.configure(api\_key=gemini\_api)

This line configures the google.generativeai library with your Gemini API key, enabling the code to interact with Google's Gemini Large Language Models for text generation.

**2.5. Naive RAG Function**

**def** naive\_rag(query, n\_results=5):

results = chroma\_collection.query(query\_texts=[query], n\_results=n\_results) retrieved\_documents = results["documents"][0]

context = "**\n\n**".join(retrieved\_documents)

prompt = f"""Answer the following question based on the provided context:

Context:

**{**context**}**

Question: **{**query**}**

Answer:"""

model = genai.GenerativeModel("gemini-1.5-flash")

response = model.generate\_content(prompt)

**return** response.text

This naive\_rag function encapsulates the core RAG logic:

1. **Retrieval**: It takes a query and uses chroma\_collection.query to search the vector database for the n\_results most relevant documents (text chunks). These are the retrieved\_documents .

2. **Augmentation**: The retrieved\_documents are then joined together to form a context string. This context, along with the original query , is then used to construct a prompt for the LLM. This is the

key step where external knowledge is injected into the LLM's input. 3. **Generation**: A gemini-1.5-flash model is initialized, and model.generate\_content(prompt) is called. The LLM uses the provided context to generate an answer to the query .

This function demonstrates the fundamental retrieve-then-generate pattern of RAG.

**2.6. Execution**

query = "is there any information related to 2023?"

answer = naive\_rag(query)

print(word\_wrap(answer))

This final part of the code defines a specific query and then calls the naive\_rag function with this query. The answer returned by the RAG system is then printed, formatted using the word\_wrap utility.

**3. Demo: RAG in Action**

To illustrate how RAG works with this code, let's trace the execution with the example query: "is there any information related to 2023?"

1. **User Query**: The user asks, "is there any information related to 2023?"

2. **Retrieval**: The naive\_rag function takes this query. The

chroma\_collection.query method searches the vector database (which contains chunks from the 1408487-EN.pdf document). It retrieves the top n\_results (defaulting to 5) text chunks that are semantically most similar to the query. These chunks are the external knowledge relevant to the query.

3. **Context Formulation**: The retrieved text chunks are concatenated to form a context string. This context now contains specific information from the PDF that is likely to address the query about "2023".

4. **Prompt Engineering**: A detailed prompt is constructed for the Gemini LLM. This prompt explicitly instructs the LLM to answer the question based on the provided context. This is crucial for RAG, as it constrains the LLM to use the retrieved information and reduces the chance of hallucinations or relying on its general training data.

``` Answer the following question based on the provided context: Context: [Retrieved relevant text chunks about 2023 from the PDF] Question: is there any information related to 2023?

Answer: ```

5. **Generation**: The Gemini LLM receives this augmented prompt. Instead of trying to answer from its vast, but potentially outdated, internal knowledge, it focuses on the provided context . It synthesizes the information from the retrieved chunks to formulate a precise and accurate answer regarding "2023" from the PDF.

6. **Output**: The LLM's generated answer is then returned and printed to the console. This answer will be directly grounded in the content of the 1408487-EN.pdf file, demonstrating how RAG effectively leverages external data to provide targeted responses.