DAY – 12

On Day 12, we were introduced to a cutting-edge concept in the field of Generative AI — RAG (Retrieval-Augmented Generation). This architecture bridges the gap between static knowledge stored in large language models (LLMs) and dynamic, real-time document-based information. It allows AI systems to generate responses that are grounded in actual data, such as uploaded PDFs, websites, or private datasets.

1. WHAT IS RAG?

RAG (Retrieval-Augmented Generation) is a framework that combines:

- **Information Retrieval (IR)** — for fetching relevant external knowledge
- **Text Generation** — for generating human-like responses using LLMs

Rather than relying solely on what the model was trained on, RAG retrieves content from uploaded documents, integrates it into the prompt, and produces answers that are more accurate, current, and context-aware.

2. FOUR KEY STEPS OF RAG

We explored the four fundamental components of the RAG pipeline:

a. Indexing

- The document(s) are split into smaller, manageable chunks.
- These chunks are vectorized (converted to embeddings) and stored in a vector database (e.g., FAISS or Chroma DB).

b. Retrieval

- When a user asks a question, the system retrieves the most semantically relevant chunks based on the query.

c. Augmentation

- The retrieved chunks are appended to the user's prompt as context, ensuring that the LLM has access to relevant information during generation.

d. Generation

- The prompt + retrieved context are passed to the LLM, which then generates a grounded response.

3. CONCEPT OF OVERLAPPING CHUNKS

We learned about the importance of overlapping chunks while splitting long documents:

- Helps preserve semantic continuity between neighbouring chunks.
- Avoids cutting off sentences or losing key context at chunk boundaries.
- Improves the retrieval accuracy, especially for multi-sentence answers.

This technique is crucial for ensuring the retrieved content remains coherent and useful when presented to the LLM.

4. HANDS-ON IMPLEMENTATION OF RAG USING PYTHON

We built two versions of a functional RAG pipeline in Python:

A. Single PDF RAG System

- **PDF Upload:** Used file upload widgets or tkinter/CLI to accept a single PDF file.
- **Text Extraction:** Used libraries like PyMuPDF (fitz) or pdfplumber to extract raw text from the document.
- **Chunking:** Split the extracted text into overlapping chunks using a window size and stride approach.
- **Vectorization:** Transformed each chunk into embeddings using models from Hugging Face or SentenceTransformers.
- **Indexing:** Stored all embeddings into a vector store like FAISS or Chroma.
- **User Query:** The user asked questions from the terminal or UI.
- **Context Retrieval:** Retrieved top-k most relevant chunks based on semantic similarity.
- **LLM Integration:** Sent the combined context + query to an LLM (e.g., Gemini/GPT-4 via API) to generate the final answer.
- **Context Display:** Showed both the retrieved context and the answer for transparency.

B. Multi-PDF RAG System

We then extended the above pipeline to support multiple PDF uploads:

- All uploaded PDFs were read in a loop.
- Text from each file was extracted, chunked, and embedded.

- All embeddings were added to a shared vector store, allowing unified retrieval across all documents.
- This enabled users to ask questions that could span multiple files or topics, simulating a personalized knowledge base.

CONCLUSION

Day 12 was a highly technical and practical session where we not only learned the theoretical foundations of RAG (Retrieval-Augmented Generation) but also built two working prototypes using Python — one for single-file processing and the other for multi-document intelligence.

This architecture is a major advancement in making LLMs more context-aware, reliable, and grounded in user-specific content. It's especially useful in enterprise, education, and research-based applications where document-based reasoning is critical.