Retrievers in RAG Systems

Introduction (2 minutes)

Good morning/afternoon everyone! Today we're going to explore retrievers, which are a fundamental component of Retrieval Augmented Generation (RAG) systems. As we know, large language models like GPT or Claude are powerful, but they have limitations - they can't access specific documents you have or information beyond their training data. That's where RAG comes in.

The RAG Pipeline (3 minutes)

Let's think of RAG as giving an AI a research assistant. The retriever is like that assistant who searches through your documents and provides relevant information to the AI. Our notebook demonstrates the four key steps in this process:

1. We load documents and split them into manageable chunks
2. We convert these text chunks into numerical vectors using embeddings
3. We store these vectors in a database for quick searching
4. When given a query, we find the most similar chunks in our database

Document Processing (4 minutes)

In our notebook, we start by loading a simple text file. In real applications, this could be PDFs, web pages, or even private company documents. We then split these documents into smaller chunks.

Why do we chunk? Two reasons:

- Large language models have context length limitations
- Smaller chunks allow for more precise retrieval

Our code uses what's called a RecursiveCharacterTextSplitter, which is smart enough to break text at natural boundaries like paragraphs or sentences. We set a chunk size of 500 characters with a 100-character overlap between chunks to maintain context.

Embeddings (3 minutes)

Once we have our chunks, we need to convert them into a format that computers can understand semantically. This is where embeddings come in.

Embeddings are like magic translators that convert text into numerical vectors. In these vectors, similar concepts end up close together in the vector space. For example, "dog" and "puppy" would be closer together than "dog" and "refrigerator."

We're using OpenAI's embedding model here, but there are many options available, including open-source alternatives.

Vector Databases (4 minutes)

Now that we have our text converted to vectors, we need a place to store them that allows for efficient searching. That's where FAISS comes in - it's a vector database developed by Facebook AI.

FAISS is optimized for similarity search, which means finding vectors that are "close" to our query vector. This is fundamentally different from traditional keyword search. We're looking for semantic similarity, not just matching words.

One great feature we demonstrate is saving our vector database to disk. This is important because computing embeddings can be expensive and time-consuming. By saving our database, we can reuse it across multiple sessions.

Retrieval in Action (5 minutes)

Let's look at the retrieval process. When a user asks a question like "Can you give some decorative styles in ancient Greek life?", we:

1. Convert this query to an embedding vector
2. Search our vector database for the most similar chunks
3. Retrieve the top 3 results (we set k=3 in our code)

In the example, our retriever successfully found chunks discussing Greek decorative styles, including information about Geometric Style, Black-Figure Technique, and architectural orders like Doric and Ionic.

Building a Complete RAG System (4 minutes)

The final part of our notebook shows how to build a complete RAG pipeline with helper functions for processing documents and creating vector stores. We demonstrate this with a different document - an AI report - and a new query about AI-driven tutoring solutions.

This modular approach makes our code reusable and easier to maintain. It's a pattern you'll see in many production RAG systems.

Key Takeaways (2 minutes)

As you build your own RAG applications, remember these key points:

1. Chunking strategy matters - experiment with different chunk sizes and overlaps
2. Choose embedding models appropriate for your use case and budget
3. Consider the retrieval method - we used simple similarity search, but there are more advanced techniques
4. Always evaluate your retriever's performance with real queries

Conclusion (1 minute)

Retrievers are what give large language models access to specific knowledge. They're the bridge between general AI capabilities and domain-specific information. As you continue your AI journey, understanding and optimizing retrievers will be crucial for building effective information retrieval systems.

Any questions before we move on?

Multi-Query Retrieval

Introduction (1-2 minutes)

Good morning/afternoon everyone! Today we're going to explore a fascinating technique in AI information retrieval called Multi-Query Retrieval. This approach solves one of the most common problems when working with vector databases - how to find the information you need when you don't know exactly how it's phrased in your database.

The Problem (2-3 minutes)

Imagine this scenario: You've built a knowledge base with thousands of documents. A user asks "When was MKUltra declassified?" But what if the document actually phrases it as "declassification occurred in 2001" or "information was made public in 2001"?

In traditional vector searches, this mismatch between query phrasing and document phrasing can cause relevant information to be missed. It's like trying to find someone in a crowd by describing them, but using different words than everyone else uses.

This is a significant limitation because:

- Users don't know what phrasing is in your database
- Natural questions rarely match technical documentation exactly
- Multiple ways to express the same information lead to retrieval gaps

The Solution: Multi-Query Retrieval (3-4 minutes)

[SHOW SLIDE WITH DIAGRAM]

This is where Multi-Query Retrieval comes in. The core idea is brilliantly simple:

1. Take the user's original question
2. Use an LLM to generate multiple alternative phrasings
3. Search for each phrasing
4. Combine the unique results

It's like sending out multiple scouts to look for information, each asking in a slightly different way.

The Notebook Demonstration (5-6 minutes)

Let's walk through the notebook to see this in action:

First, we gathered Wikipedia data about MKUltra - a declassified CIA program. We then:

- Split it into smaller chunks (about 500 characters each)
- Created embeddings using OpenAI's embedding model
- Stored these in a FAISS vector database

Now for the magic part - we asked a simple question: "When was this declassified?"

The system then:

- Generated three variations of this question
- Searched for each variation
- Found the relevant document stating "Some surviving information about MKUltra was declassified in 2001."

All this happened automatically, without us having to guess the right phrasing!

Why This Matters (3-4 minutes)

This technique is a game-changer for several reasons:

1. **User Experience**: It allows users to ask questions naturally, without having to guess the "right" way to ask

2. **Improved Results**: By casting a wider net with multiple phrasings, we're more likely to find relevant information
3. **Practical Implementation**: It's relatively easy to implement with modern LLM APIs and vector databases

This represents a shift from "search exactly what I ask" to "understand what I mean and find it" - a much more human-like approach to information retrieval.

Implementation Details (2-3 minutes)

For those interested in the technical aspects:

- We used LangChain's MultiQueryRetriever
- The query generation is handled by ChatGPT (temperature=0 for consistency)
- FAISS provides efficient similarity search
- The whole pipeline requires minimal code to implement

Conclusion and Applications (2 minutes)

Multi-Query Retrieval bridges the gap between how humans naturally ask questions and how machines traditionally search for information. It's particularly valuable for:

- Question-answering systems
- Knowledge bases and documentation search
- Customer support tools
- Research assistants

As you build your own AI applications, consider how this technique might improve your users' ability to find information without having to know exactly how to ask for it.

Any questions?