This code is implementing **a document retrieval and question-answering pipeline using LangChain and ChromaDB**. Below is a structured breakdown:

1) Code Explanation (Step-by-Step)

Step 1: Install Required Libraries

```
python
CopyEdit
!pip install -qU langchain chromadb huggingface_hub sentence-transformers pypdf openai tiktoken
!pip install -U langchain-community
```

- Installs necessary libraries for **document loading, embedding models, vector storage, retrieval, and LLM-based question answering**.

Step 2: Load PDF Data

```
python
CopyEdit
from langchain.document_loaders import PyPDFLoader
from google.colab import drive
drive.mount('/content/drive')

loader_harrypotter  = PyPDFLoader("/content/harry_potter_book.pdf")
documnet_harrypotter = loader_harrypotter.load()
print(len(documnet_harrypotter))

loader_got = PyPDFLoader("/content/got_book.pdf")
documnet_got = loader_got.load()
print(len(documnet_got))
```

- Mounts Google Drive to access files.
- Uses PyPDFLoader to extract text from **Harry Potter** and **Game of Thrones (GoT)** books.

- The `load()` function reads the PDF content into **LangChain Document objects**.

Step 3: Chunking Text for Vector Storage

```
python
CopyEdit
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=100)

text_harrypotter = text_splitter.split_documents(documnet_harrypotter)
text_got = text_splitter.split_documents(documnet_got)
```

- Splits text into **overlapping chunks (500 characters, with 100 overlapping)** to maintain context for embeddings.

Step 4: Load Embedding Models

```
python
CopyEdit
from langchain.embeddings import HuggingFaceEmbeddings,
OpenAIEmbeddings, HuggingFaceBgeEmbeddings

hf_embeddings = HuggingFaceEmbeddings(model_name="sentence-
transformers/all-MiniLM-L6-v2")
hf_bge_embeddings = HuggingFaceBgeEmbeddings(model_name="BAAI/bge-
large-en")
```

- Embeddings are numerical representations of text for similarity search.
- Uses **Sentence Transformers** and **BGE Large** models.

```
python
CopyEdit
from google.colab import userdata
OPENAI_API_KEY=userdata.get('OPENAI_API_KEY')
os.environ["OPENAI_API_KEY"]=OPENAI_API_KEY
openai_embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
```

- Uses **OpenAI embeddings** for improved vector search.

Step 5: Store Data in ChromaDB

```
python
CopyEdit
from langchain.vectorstores import Chroma
import chromadb
import os

DB_DIR = "/content/db"
client_settings = chromadb.config.Settings(
    is_persistent=True,
    persist_directory=DB_DIR,
    anonymized_telemetry=False,
)

harrypotter_vectorstore = Chroma.from_documents(
    text_harrypotter, hf_bge_embeddings,
client_settings=client_settings,
    collection_name="harrypotter", collection_metadata={"hnsw":
"cosine"},
    persist_directory="/store/harrypotter"
)

got_vectorstore = Chroma.from_documents(
    text_got, hf_bge_embeddings, client_settings=client_settings,
    collection_name="got", collection_metadata={"hnsw": "cosine"},
    persist_directory="/store/got"
)
```

- **Stores** vector embeddings of text chunks in **ChromaDB**.
- Uses **Hierarchical Navigable Small World (HNSW) with cosine similarity** for fast retrieval.

Step 6: Create Retrievers

```
python
CopyEdit
retriever_harrypotter =
harrypotter_vectorstore.as_retriever(search_type="mmr",
search_kwargs={"k": 5, "include_metadata": True})
retriever_got = got_vectorstore.as_retriever(search_type="mmr",
search_kwargs={"k": 5, "include_metadata": True})
```

- Retrieves **top 5** most relevant document chunks using **Maximal Marginal Relevance (MMR)**.

Step 7: Merge Retrievers

```
python
CopyEdit
from langchain.retrievers.merger_retriever import MergerRetriever

lotr = MergerRetriever(retrievers=[retriever_harrypotter,
retriever_got])
```

- **Merges multiple retrievers** into one. This is also called **"Lord of Retrievers" (LOTR)**.

```
python
CopyEdit
for chunks in lotr.get_relevant_documents("Who was the jon snow?"):
    print(chunks.page_content)
```

- Fetches relevant documents across **both books**.

Step 8: Context Reordering and Filtering

```
python
CopyEdit
from langchain.document_transformers import
EmbeddingsClusteringFilter, EmbeddingsRedundantFilter
```

```
from langchain.retrievers.document_compressors import
DocumentCompressorPipeline
from langchain.retrievers import ContextualCompressionRetriever
from langchain.document_transformers import LongContextReorder

filter = EmbeddingsRedundantFilter(embeddings=hf_bge_embeddings)
reordering = LongContextReorder()
pipeline = DocumentCompressorPipeline(transformers=[filter,
reordering])
compression_retriever_reordered = ContextualCompressionRetriever(
    base_compressor=pipeline, base_retriever=lotr, search_kwargs={"k":
3, "include_metadata": True}
)
```

- **Filters redundant data** and **reorders context** for clarity.

Step 9: Load LLM Model

```
python
CopyEdit
!pip install llama-cpp-python
from langchain.llms import LlamaCpp

llms = LlamaCpp(
    model_path="/content/drive/MyDrive/zephyr-7b-beta.Q4_K_M.gguf",
    max_tokens=1500, temperature=0.75, top_p=1, gpu_layers=0,
    verbose=True, n_threads=int(os.cpu_count()/2), n_ctx=4096
)
```

- Loads **Zephyr 7B model** for text generation.
- Uses **LLamaCpp** for inference.

Step 10: Question Answering Pipeline

```
python
CopyEdit
```

```
from langchain.chains import RetrievalQA

qa = RetrievalQA.from_chain_type(
      llm=llms, chain_type="stuff",
      retriever=compression_retriever_reordered,
      return_source_documents=True
)
```

- Creates a **retrieval-based question-answering pipeline**.

```
python
CopyEdit
query ="who is jon snow?"
results = qa(query)
print(results['result'])
```

- Queries the retriever and generates a response using LLM.

2) Purpose of the Code

1. **Extracts text from PDF books** (Harry Potter & GoT).
2. **Embeds text into vectors** using multiple models.
3. **Stores vectors in ChromaDB** for retrieval.
4. **Creates a retriever** for fast search.
5. **Merges retrievers** for querying across multiple books.
6. **Enhances retrieval** by filtering redundant info & reordering context.
7. **Uses an LLM model** to answer questions from retrieved data.
8. **Generates final output** in a question-answering format.

3) Architecture of the Code

Below is the **high-level architecture**:

```
less
CopyEdit
           [PDF Files: Harry Potter, GoT]
                        |
```

```
                v
        [Text Extraction]
                |
                v
        [Text Chunking]
                |
                v
       --------------------------------
       |                              |
 [Embedding Model]           [ChromaDB Vector Store]
       |                              |
       --------------------------------
                |
        [Retriever: MMR, Filter, Merge]
                |
                v
         [Retriever Optimization: Filtering & Reordering]
                |
                v
          [LLM (Zephyr-7B)] - LlamaCpp
                |
                v
       [Question-Answering Pipeline]
                |
                v
           [Final Answer]
```

This architecture **efficiently retrieves and processes information from multiple sources**, making it suitable for **multi-book Q&A systems**.

Final Thoughts

- This **pipeline efficiently retrieves context** from multiple sources.
- **Merging retrievers** allows cross-book querying.
- **LLM integration** generates **context-aware answers**.
- **Context reordering & filtering** improve **response quality**.

Would you like further **optimizations or visual representations**? 🚀

You said:

above is the merger retriever & LongContextReorder? if yes please expalin the concept

ChatGPT said:

Yes, the provided code implements **MergerRetriever** and **LongContextReorder** concepts in a retrieval-augmented generation (RAG) pipeline. Below is a detailed explanation of these concepts:

1) What is MergerRetriever?

Concept:

A **MergerRetriever** is a mechanism to **combine multiple retrievers into a single retriever**. This is particularly useful when data is stored in **different vector databases** or **retrieval models**.

How it works:

- You might have different document collections stored in **separate retrievers**.
- Instead of querying each retriever separately, a **MergerRetriever** merges results from multiple retrievers.
- It allows for **multi-source retrieval**, which is useful when dealing with **multiple books, databases, or document collections**.

Implementation in Code:

```
python
CopyEdit
from langchain.retrievers.merger_retriever import MergerRetriever

lotr = MergerRetriever(retrievers=[retriever_harrypotter,
```

```
retriever_got])
```

- Here, `retriever_harrypotter` and `retriever_got` are **separate retrievers** that fetch relevant documents from **Harry Potter** and **Game of Thrones** books, respectively.
- The `MergerRetriever` (`lotr`) **combines both** so that a single query can retrieve information from **both books**.

Example Query using MergerRetriever:

```
python
CopyEdit
for chunks in lotr.get_relevant_documents("Who was Jon Snow?"):
    print(chunks.page_content)
```

- This **fetches results from both Harry Potter and GoT books**, ensuring **cross-source search**.

Why is MergerRetriever Useful?

✅ Queries multiple document sources efficiently.
✅ Ensures diverse retrieval without manually searching each source.
✅ Reduces dependency on a single vector database.

2) What is LongContextReorder?

Concept:

When retrieving long documents, **the most relevant information may be scattered throughout different parts of the text**. The **LongContextReorder** transformer reorders text chunks to improve the relevance of answers.

Problem it Solves:

- Sometimes, **important context appears late in the retrieved text**.

- If the **most relevant information is buried deep in a document**, a model may **miss** it.
- **LongContextReorder ensures that the most important chunks are at the top.**

Implementation in Code:

```
python
CopyEdit
from langchain.document_transformers import LongContextReorder

reordering = LongContextReorder()
```

- Here, `LongContextReorder` is initialized to **rearrange document chunks** based on **importance**.

Using LongContextReorder in a Pipeline:

```
python
CopyEdit
from langchain.document_transformers import EmbeddingsRedundantFilter
from langchain.retrievers.document_compressors import
DocumentCompressorPipeline
from langchain.retrievers import ContextualCompressionRetriever

filter = EmbeddingsRedundantFilter(embeddings=hf_bge_embeddings)
pipeline = DocumentCompressorPipeline(transformers=[filter,
reordering])

compression_retriever_reordered = ContextualCompressionRetriever(
    base_compressor=pipeline, base_retriever=lotr, search_kwargs={"k":
3, "include_metadata": True}
)
```

- `EmbeddingsRedundantFilter`: **Removes duplicate or less important results**.
- `LongContextReorder`: **Reorders text chunks to prioritize key context**.
- `DocumentCompressorPipeline`: **Combines filtering & reordering into a single retrieval step**.
- `ContextualCompressionRetriever`: **Applies the compression pipeline to the merged retriever**.

Example Query using LongContextReorder:

```
python
CopyEdit
results = qa.invoke("How does Jon Snow's relationship with the Stark
family influence his identity?")
print(results['result'])
```

- This query **retrieves information** from GoT books.
- **Reordering ensures** that the most relevant parts of Jon Snow's story appear **first** in the retrieval.
- The LLM processes **ordered content** to generate a better answer.

Why is LongContextReorder Useful?

✅ Ensures that the **most relevant** text appears **first** in retrieval.
✅ **Improves answer quality** by ensuring context is clear.
✅ **Reduces confusion** when processing long documents.

3) Combined Power: MergerRetriever + LongContextReorder

The **combination of MergerRetriever and LongContextReorder** creates an **optimized retrieval pipeline**:

1. **MergerRetriever**: Merges multiple retrievers → Fetches diverse, cross-source data.
2. **LongContextReorder**: Reorders retrieved text → Ensures the most important information is at the top.

This combination allows **high-quality retrieval and structured responses** from multiple sources.

Conclusion

Feature	MergerRetriever	LongContextReorder
Function	Merges multiple retrievers	Reorders long text chunks
Use Case	Searching across multiple sources	Optimizing text structure for clarity
Benefit	Unified retrieval from different datasets	Ensures important info appears first
Example Query	"Who is Jon Snow?"	"How does Jon Snow's identity evolve?"

Both **concepts together** create a **highly efficient Retrieval-Augmented Generation (RAG) pipeline**.

Would you like an architectural diagram for this? 🚀

######### Below is from Deep Seek #########

1. Explanation of the Code

This code is a **Retrieval-Augmented Generation (RAG) pipeline** designed to answer questions using content from two books (*Harry Potter* and *Game of Thrones*). Here's a breakdown:
Key Steps:

1. **Install Libraries**: Installs dependencies for text processing, embeddings, vector databases, and LLMs.
2. **Load Data**: Uses `PyPDFLoader` to load PDFs of the books from Google Drive.

3. **Chunk Documents**: Splits text into smaller chunks (`chunk_size=500`, `overlap=100`) using `RecursiveCharacterTextSplitter`.
4. **Embeddings**: Uses `HuggingFaceBgeEmbeddings` to convert text chunks into vectors.
5. **Vector Storage**: Stores embeddings in **ChromaDB** (a vector database) with persistent storage for reuse.
6. **Retrievers**: Creates retrievers for each book to fetch relevant chunks using **MMR (Maximal Marginal Relevance)**.
7. **Merge Retrievers**: Combines retrievers using `MergerRetriever` to query both books simultaneously.
8. **Context Refinement**: Applies `EmbeddingsRedundantFilter` (removes duplicates) and `LongContextReorder` (optimizes chunk order) to improve results.
9. **LLM Integration**: Loads the **Zephyr-7B** model (via `LlamaCpp`) and connects it to the refined retriever using `RetrievalQA`.

2. Purpose of the Code

The goal is to **answer questions by combining information from multiple documents** (Harry Potter and Game of Thrones books). It addresses challenges like:

- **Cross-document retrieval**: Fetching relevant chunks from multiple sources.
- **Context management**: Reordering chunks to avoid "lost in the middle" issues.
- **Noise reduction**: Filtering redundant or irrelevant text.

3. Architecture & Detailed Explanation

Architecture Diagram:

Copy
```
[PDFs] → [Text Splitting] → [Embeddings] → [ChromaDB]
                                               │
                                               ├─ Retriever (Harry Potter)
                                               ├─ Retriever (Game of Thrones)
                                               │
                                               └─ Merge → Filter/Reorder → LLM →
Answer
```

Components:

1. **Document Loading**:

   a. Uses `PyPDFLoader` to load PDFs into `Document` objects (each page as a document).
2. **Text Splitting**:
   a. Splits text into overlapping chunks (`RecursiveCharacterTextSplitter`) to preserve context.
3. **Embeddings**:
   a. Converts text to vectors using `BAAI/bge-large-en`, a high-quality embedding model.
4. **Vector Database**:
   a. **ChromaDB** stores embeddings persistently, enabling fast similarity searches.
   b. Separate collections for Harry Potter (`harrypotter`) and Game of Thrones (`got`).
5. **Retrievers**:
   a. **MMR Retrievers**: Fetch top-5 chunks per book, balancing relevance and diversity.
   b. **MergerRetriever**: Combines results from both books into a single list.
6. **Context Refinement**:
   a. **EmbeddingsRedundantFilter**: Removes duplicate chunks using embeddings.
   b. **LongContextReorder**: Reorders chunks to prioritize important content (avoids middle bias).
7. **LLM**:
   a. Uses **Zephyr-7B** (a quantized model) for answer generation via `RetrievalQA`.
   b. `chain_type="stuff"` concatenates chunks into the LLM's context window.

Example Workflow:

1. **Query**: "Who is Jon Snow?"
2. **Retrieval**: Fetches chunks about Jon Snow from both books.
3. **Refinement**: Filters duplicates and reorders chunks.
4. **Generation**: LLM synthesizes the answer using the refined context.

Key Tools & Techniques:

- **LangChain**: Orchestrates the RAG pipeline (retrievers, chains, transformers).
- **ChromaDB**: Efficient vector storage and retrieval.

- **BGE Embeddings**: High-quality text representation.
- **Zephyr-7B**: Lightweight, open-source LLM for answer generation.

This pipeline is ideal for **multi-document QA systems** where context from diverse sources must be combined and optimized.

above is the merger retriever & LongContextReorder? if yes please expalin the concept

1. MergerRetriever (LOTR: "Lord of the Retrievers")
What is it?

- A **LangChain component** that combines results from multiple retrievers (e.g., from different document collections) into a single ranked list.
- **Purpose**: To query multiple sources (e.g., Harry Potter and Game of Thrones books) simultaneously and aggregate their results.

How it works in the code:

1. **Separate Retrievers**:
    a. `retriever_harrypotter` and `retriever_got` are created for each book's ChromaDB collection.
    b. Each retriever uses **MMR (Maximal Marginal Relevance)** to fetch `k=5` chunks, balancing relevance and diversity.
2. **Merge Results**:

python
Copy
```
from langchain.retrievers.merger_retriever import MergerRetriever
lotr = MergerRetriever(retrievers=[retriever_harrypotter,
retriever_got])
```

    a. The `MergerRetriever` (`lotr`) combines results from both retrievers into a single list of chunks.

Why use it?

- **Cross-Document Search**: Answers questions requiring knowledge from multiple sources (e.g., comparing characters from different books).
- **Scalability**: Easily add more retrievers for additional document collections.

2. LongContextReorder
What is it?

- A **document transformer** that reorders retrieved chunks to address the **"lost in the middle" problem**, where LLMs struggle to retain information from the middle of long contexts.
- **Purpose**: Optimize the order of chunks to prioritize the most relevant information for the LLM.

How it works in the code:

1. **Pipeline Setup**:
    a. **EmbeddingsRedundantFilter**: Removes duplicate chunks using embeddings similarity.
    b. **LongContextReorder**: Reorders the remaining chunks to improve LLM comprehension.

python
Copy
```
pipeline = DocumentCompressorPipeline(transformers=[filter,
reordering])
```

2. **Reordering Strategy**:
    a. Chunks are rearranged to place the **most relevant** chunks at the **beginning** and **end** of the context.
    b. Example: If 6 chunks are retrieved, the order becomes `[1, 3, 5, 6, 4, 2]`.

Why use it?

- **Improved LLM Performance**: LLMs (like Zephyr-7B) process the start and end of contexts more effectively. Reordering ensures critical information isn't buried in the middle.
- **Noise Reduction**: Filters redundant content to keep the context concise.

Workflow Example:
For the query *"Who is Jon Snow?"*:

1. **Retrieval**:
    a. `MergerRetriever` fetches chunks from both Harry Potter and Game of Thrones collections.
2. **Filtering**:
    a. `EmbeddingsRedundantFilter` removes duplicate chunks.
3. **Reordering**:
    a. `LongContextReorder` rearranges chunks to highlight the most relevant ones.
4. **Answer Generation**:
    a. The reordered chunks are fed to the LLM (Zephyr-7B) to generate a coherent answer.

Key Benefits:

1. **MergerRetriever**:
    a. Enables **multi-source QA** without manual aggregation.
    b. Supports hybrid search (e.g., combining vector and keyword-based retrievers).
2. **LongContextReorder**:
    a. Addresses LLM weaknesses in processing long contexts.
    b. Works with any LLM to improve answer quality.

When to Use These Tools:

- **Multi-Document QA**: When answers depend on multiple documents (e.g., research papers, product manuals).
- **Long Contexts**: When retrieved chunks exceed the LLM's "attention window" (e.g., GPT-4 has 128k tokens, but smaller models like Zephyr-7B have 4k).

The code combines both techniques to create a robust, scalable RAG pipeline for complex queries! 🚀