📌 Detailed Explanation of the Code: Retrieval and Contextual Compression in Generative AI

This code demonstrates how **retrieval-based question-answering (QA)** works in **Generative AI** using **LangChain**, **FAISS (Facebook AI Similarity Search)**, **OpenAI embeddings**, and **contextual compression retrievers**.

1️⃣ Explanation of Each Line of Code

Let's break down the code **step by step**.

📌 Setup & Installation

```
python
CopyEdit
!pip install langchain_community
!pip install langchain_openai
!pip install faiss-cpu  # Install FAISS for vector-based retrieval
```

- **LangChain Community & OpenAI**: These are required for **retrieval and reranking**.
- **FAISS (Facebook AI Similarity Search)**: A **vector-based retrieval** library that enables **fast nearest-neighbor searches**.

📌 Loading and Splitting Documents

```
python
CopyEdit
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import CharacterTextSplitter
```

- **TextLoader**: Loads raw text from a file.
- **FAISS**: Stores and retrieves documents using **embeddings**.
- **OpenAIEmbeddings**: Converts text into **vector embeddings**.
- **CharacterTextSplitter**: Splits large text into **chunks**.

```
python
CopyEdit
documents = TextLoader("/content/state_of_the_union.txt").load()
text_splitter = CharacterTextSplitter(chunk_size=500,
chunk_overlap=100)
texts = text_splitter.split_documents(documents)
texts
```

- **Loads a document** from a text file (`state_of_the_union.txt`).
- **Splits the document** into **500-character chunks** with **100-character overlap** to preserve context.

📌 Setting Up OpenAI API Key

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

- **Fetches the OpenAI API Key** stored in Google Colab.
- **Sets it as an environment variable** to use OpenAI services.

📌 Creating a Retriever using FAISS

```
python
CopyEdit
retriever = FAISS.from_documents(texts,
OpenAIEmbeddings()).as_retriever()
```

- Converts **text chunks into embeddings** using **OpenAI's text-embedding model**.
- **FAISS stores** the embeddings for **efficient retrieval**.

```
python
CopyEdit
docs = retriever.invoke("What did the president say about Ketanji
Brown Jackson?")
```

- **Retrieves relevant documents** based on the query.

📌 Printing Retrieved Documents

```
python
CopyEdit
def pretty_print_docs(docs):
    print(
        f"\n{'-' * 100}\n".join(
            [f"Document {i+1}:\n\n" + d.page_content for i, d in
enumerate(docs)]
        )
    )
pretty_print_docs(docs)
```

- **Formats and prints** the retrieved documents.

```
python
CopyEdit
docs2 = retriever.invoke("What were the top three priorities outlined
in the most recent State of the Union address?")
pretty_print_docs(docs2)
```

- **Retrieves additional documents** based on a different query.

📌 Setting Up an LLM for Question-Answering

```
python
CopyEdit
```

```
from langchain_openai import OpenAI
llm = OpenAI(temperature=0)
```

- Uses **OpenAI's GPT model** for generating responses.
- `temperature=0` ensures **deterministic responses**.

```
python
CopyEdit
from langchain.chains import RetrievalQA
chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
query = "What were the top three priorities outlined in the most
recent State of the Union address?"
chain.invoke(query)
print(chain.invoke(query)['result'])
```

- **Creates a QA chain** where the **retrieved documents are passed to the LLM**.
- **Generates the final answer**.

2️⃣ Contextual Compression Retrieval

Now, we use **compression techniques** to **reduce irrelevant information** before passing it to the LLM.

📌 Using LLMChainExtractor for Contextual Compression

```
python
CopyEdit
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import
LLMChainExtractor
compressor = LLMChainExtractor.from_llm(llm)
compression_retriever =
ContextualCompressionRetriever(base_compressor=compressor,
base_retriever=retriever)
```

- **LLMChainExtractor** extracts **only the most relevant parts** of each document.
- **ContextualCompressionRetriever** applies this **compression before retrieval**.

```
python
CopyEdit
compressed_docs = compression_retriever.invoke("What did the president
say about Ketanji Jackson Brown?")
pretty_print_docs(compressed_docs)
```

- **Retrieves and compresses** documents using **LLM-based extraction**.

📌 Using LLMChainFilter for Document Filtering

```
python
CopyEdit
from langchain.retrievers.document_compressors import LLMChainFilter
filter = LLMChainFilter.from_llm(llm)
compression_retriever2 =
ContextualCompressionRetriever(base_compressor=filter,
base_retriever=retriever)
```

- **Filters documents** by **removing irrelevant content**.

```
python
CopyEdit
compressed_docs3 = compression_retriever2.invoke("What were the top
three priorities outlined in the most recent State of the Union
address?")
pretty_print_docs(compressed_docs3)
```

- **Retrieves only the most relevant documents**.

📌 Measuring Compression Efficiency

```
python
CopyEdit
original_contexts_len = len("\n\n".join([d.page_content for i, d in
enumerate(docs2)]))
```

```
compressed_contexts_len = len("\n\n".join([d.page_content for i, d in
enumerate(compressed_docs)]))
print("Original context length:", original_contexts_len)
print("Compressed context length:", compressed_contexts_len)
print("Compressed Ratio:",
f"{original_contexts_len/(compressed_contexts_len + 1e-5):.2f}x")
```

- **Compares the length of the original vs compressed documents**.
- **Compression Ratio** shows **how much the document size was reduced**.

📌 Using Embeddings-Based Filtering

```
python
CopyEdit
from langchain.retrievers.document_compressors import EmbeddingsFilter
embeddings = OpenAIEmbeddings()
embeddings_filter = EmbeddingsFilter(embeddings=embeddings)
compression_retriever3 =
ContextualCompressionRetriever(base_compressor=embeddings_filter,
base_retriever=retriever)
```

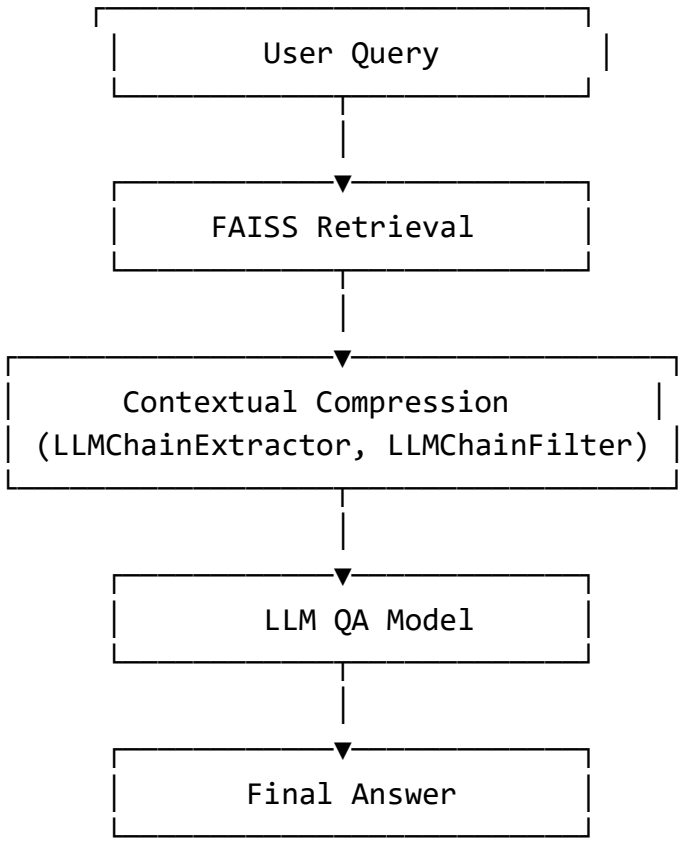
- Uses **embeddings similarity** to **filter out redundant information**.

```
python
CopyEdit
compressed_docs4 = compression_retriever3.invoke("What were the top
three priorities outlined in the most recent State of the Union
address?")
pretty_print_docs(compressed_docs4)
```

- **Retrieves only highly relevant text**.

3️⃣ Architecture Diagram

```
sql
CopyEdit
```



4️⃣ Key Takeaways

✅ **FAISS efficiently retrieves documents using embeddings.**
✅ **Contextual Compression removes irrelevant text before passing it to an LLM.**
✅ **Different compression techniques (LLM extraction, filtering, embeddings-based) can be used.**
✅ **Reduces LLM token usage, making responses more accurate and cost-effective.**

Would you like a **custom implementation** for your specific use case? 🚀

📌 Contextual Compression Retrieval in Generative AI: Detailed Notes

Contextual Compression Retrieval is a method used in **Retrieval-Augmented Generation (RAG)** to improve the efficiency of **retrieving and processing information** for **large language models (LLMs)** like GPT-4.

🔍 1️⃣ What is Contextual Compression Retrieval?

🚀 The Problem with Traditional Retrieval

In **retrieval-based AI systems**, we retrieve relevant documents based on a query. However:

- Many retrieved documents **contain redundant or irrelevant information**.
- The LLM must **process unnecessary data**, increasing **cost** and **response time**.
- Without proper filtering, **low-quality results** can affect the final answer.

☑️ Solution: Contextual Compression Retrieval

- **Contextual Compression removes irrelevant, redundant, or non-informative** sections from retrieved documents **before** passing them to an LLM.
- This **improves response quality, reduces costs, and speeds up inference**.

⚙️ 2️⃣ How Contextual Compression Retrieval Works

🔗 Step 1: Document Retrieval

- The system retrieves documents **related to the query** using a vector database (**e.g., FAISS, Pinecone**).
- Example retrieval methods:
  - **Sparse retrieval** (BM25, TF-IDF) → Matches exact words.
  - **Dense retrieval** (FAISS, OpenAI embeddings) → Matches **semantic meaning**.

🔗 Step 2: Contextual Compression

- The **retrieved documents** are **compressed** to remove unnecessary sections.
- **Techniques used:**
  - **LLM-Based Extraction** → Uses an LLM to extract only **high-value information**.
  - **Filtering** → Removes irrelevant sections **using heuristics or embeddings**.
  - **Embeddings-Based Reranking** → Keeps only the most **semantically relevant** passages.

🔗 Step 3: Query Execution & Response Generation

- The **compressed context** is **passed to an LLM**, which generates the final answer.
- Since **less text is processed**, the **LLM is faster and cheaper**.

🛠️ 3️⃣ Contextual Compression Retrieval in Code

🔹 (A) Loading & Splitting Documents

```
python
CopyEdit
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import CharacterTextSplitter

# Load the document
documents = TextLoader("/content/state_of_the_union.txt").load()

# Split document into chunks
text_splitter = CharacterTextSplitter(chunk_size=500,
chunk_overlap=100)
texts = text_splitter.split_documents(documents)
```

- The document is **loaded and split into smaller parts** to preserve context.

🔹 (B) Creating a Retriever with FAISS

```
python
CopyEdit
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

# Convert documents into embeddings and store in FAISS
retriever = FAISS.from_documents(texts,
OpenAIEmbeddings()).as_retriever()
```

- **FAISS** (Facebook AI Similarity Search) is used for **efficient vector search**.

◇ (C) Retrieving Documents for a Query

```
python
CopyEdit
query = "What did the president say about Ketanji Brown Jackson?"
docs = retriever.invoke(query)

# Print retrieved documents
def pretty_print_docs(docs):
    print("\n".join([d.page_content for d in docs]))

pretty_print_docs(docs)
```

- The **retriever searches** for the **most relevant** documents.
- However, these documents **may contain a lot of unnecessary text**.

◇ (D) Applying Contextual Compression with LLM

```
python
CopyEdit
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import
LLMChainExtractor
from langchain_openai import OpenAI

# Load the LLM
llm = OpenAI(temperature=0)

# Create a compressor using LLM extraction
compressor = LLMChainExtractor.from_llm(llm)

# Combine retriever and compressor
compression_retriever =
ContextualCompressionRetriever(base_compressor=compressor,
base_retriever=retriever)
```

- **LLMChainExtractor** automatically **extracts** the **most useful sentences** from retrieved documents.

◇ (E) Running a Compressed Retrieval Query

```
python
CopyEdit
compressed_docs = compression_retriever.invoke("What did the president
say about Ketanji Brown Jackson?")
pretty_print_docs(compressed_docs)
```

- The **compressed output** contains **only the most relevant information**.

◇ (F) Additional Filtering to Remove Noise

```
python
CopyEdit
from langchain.retrievers.document_compressors import LLMChainFilter

# Apply an LLM-based filtering method
filter = LLMChainFilter.from_llm(llm)
compression_retriever2 =
ContextualCompressionRetriever(base_compressor=filter,
base_retriever=retriever)

# Retrieve filtered documents
compressed_docs_filtered = compression_retriever2.invoke("What were
the key topics in the address?")
pretty_print_docs(compressed_docs_filtered)
```

- **LLMChainFilter** helps to **further refine results**, **removing lower-value content**.

📊 4️⃣ Measuring the Efficiency of Contextual Compression

Before & After Compression

```
python
CopyEdit
original_len = len("\n".join([d.page_content for d in docs]))
compressed_len = len("\n".join([d.page_content for d in
compressed_docs]))

print("Original Context Length:", original_len)
print("Compressed Context Length:", compressed_len)
print("Compression Ratio:", f"{original_len / (compressed_len + 1e-
5):.2f}x")
```

- **Compression Ratio** measures **how much text was removed** while keeping the core information.

📌 5️⃣ Using Embeddings-Based Filtering for Contextual Compression

◇ (A) Embeddings-Based Compression

```
python
CopyEdit
from langchain.retrievers.document_compressors import EmbeddingsFilter

# Use embeddings for compression
embeddings = OpenAIEmbeddings()
embeddings_filter = EmbeddingsFilter(embeddings=embeddings)

# Create a new compression retriever with embeddings filter
```

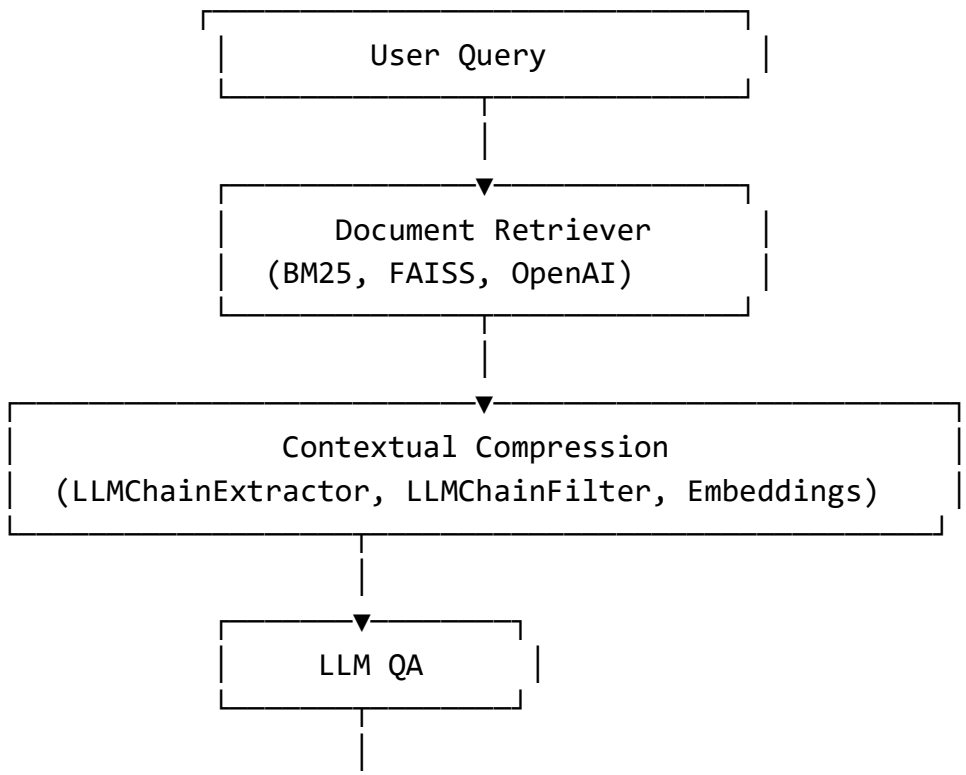
```
compression_retriever3 =
ContextualCompressionRetriever(base_compressor=embeddings_filter,
base_retriever=retriever)

# Retrieve compressed documents
compressed_docs_embeddings = compression_retriever3.invoke("What were
the top three priorities in the address?")
pretty_print_docs(compressed_docs_embeddings)
```

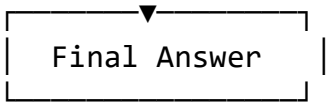
- **Embeddings-based filtering** ranks the **most semantically relevant parts**.

📌 6️⃣ Full Architecture of Contextual Compression Retrieval

🛠️ How All Components Work Together

scss
CopyEdit





- **First, retrieval fetches relevant documents**.
- **Then, compression removes unnecessary content**.
- **Finally, the LLM generates the best response**.

🔑 7️⃣ Key Takeaways

✅ **Contextual Compression Retrieval improves efficiency** by reducing irrelevant content.
✅ **Saves computational cost** by reducing the number of tokens processed.
✅ **Improves response quality** by focusing on relevant content.
✅ **Combines multiple techniques** (LLM-based, embeddings, filtering).
✅ **Essential for Generative AI models** using external knowledge.

Would you like a **real-world use case** or an **optimized implementation**? 🚀

######### Below is code from DeepSeek ##########################

1. Line-by-Line Code Explanation

Install Dependencies

```
python
Copy
!pip install langchain_community langchain_openai faiss-cpu
```

- Installs required libraries:
  - `langchain_community`: Community-contributed LangChain components.
  - `langchain_openai`: OpenAI integrations for LangChain.
  - `faiss-cpu`: Vector database for efficient similarity search.

Load and Split Documents

```
python
Copy
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import CharacterTextSplitter

documents = TextLoader("/content/state_of_the_union.txt").load()
text_splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=100)
texts = text_splitter.split_documents(documents)
```

- **TextLoader**: Loads a text file (`state_of_the_union.txt`).
- **CharacterTextSplitter**: Splits text into chunks of 500 characters with 100-character overlap to preserve context.

Set Up OpenAI API Key

```
python
Copy
from google.colab import userdata
OPENAI_API_KEY = userdata.get('OPENAI_API_KEY')
import os
os.environ["OPENAI_API_KEY"] = OPENAI_API_KEY
```

- Retrieves the OpenAI API key from Google Colab secrets and sets it as an environment variable.

Create FAISS Vector Store

```
python
Copy
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

retriever = FAISS.from_documents(texts, OpenAIEmbeddings()).as_retriever()
```

- **OpenAIEmbeddings**: Converts text chunks into vector embeddings using OpenAI's `text-embedding-ada-002` model.
- **FAISS**: Builds a vector store for fast similarity search. `as_retriever()` converts it into a retriever object.

Retrieve and Print Documents

```
python
Copy
def pretty_print_docs(docs):
    print(f"\n{'-' * 100}\n".join(
        [f"Document {i+1}:\n\n" + d.page_content for i, d in enumerate(docs)]
    ))

docs = retriever.invoke("What did the president say about Ketanji Brown Jackson")
pretty_print_docs(docs)
```

- **retriever.invoke()**: Fetches documents relevant to the query using FAISS.
- **pretty_print_docs()**: Formats and prints documents with separators.

Basic Question Answering

```
python
Copy
from langchain_openai import OpenAI
from langchain.chains import RetrievalQA

llm = OpenAI(temperature=0)
chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
query = "What were the top three priorities outlined in the most recent State of the Union address?"
print(chain.invoke(query)['result'])
```

- **RetrievalQA**: Combines retrieval and LLM to answer questions.
- `temperature=0` ensures deterministic responses.

Contextual Compression

```
python
Copy
```

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor,
LLMChainFilter, EmbeddingsFilter

# 1. LLMChainExtractor
compressor = LLMChainExtractor.from_llm(llm)
compression_retriever =
ContextualCompressionRetriever(base_compressor=compressor,
base_retriever=retriever)
compressed_docs = compression_retriever.invoke("What did the president say about
Ketanji Jackson Brown")

# 2. LLMChainFilter
filter = LLMChainFilter.from_llm(llm)
compression_retriever2 = ContextualCompressionRetriever(base_compressor=filter,
base_retriever=retriever)

# 3. EmbeddingsFilter
embeddings = OpenAIEmbeddings()
embeddings_filter = EmbeddingsFilter(embeddings=embeddings,
similarity_threshold=0.76)
compression_retriever3 =
ContextualCompressionRetriever(base_compressor=embeddings_filter,
base_retriever=retriever)

# 4. Pipeline with Splitting and Filtering
from langchain_text_splitters import CharacterTextSplitter
from langchain_community.document_transformers import EmbeddingsRedundantFilter

splitter = CharacterTextSplitter(chunk_size=300, chunk_overlap=0)
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)
pipeline_compressor = DocumentCompressorPipeline(transformers=[splitter,
redundant_filter, embeddings_filter])
compression_retriever =
ContextualCompressionRetriever(base_compressor=pipeline_compressor,
base_retriever=retriever)
```

- **LLMChainExtractor**: Uses an LLM to extract relevant snippets from documents.
- **LLMChainFilter**: Uses an LLM to filter irrelevant documents.
- **EmbeddingsFilter**: Filters documents based on embedding similarity to the query.
- **Pipeline**: Combines splitting, redundancy removal, and similarity filtering.

2. Key Concepts

Retrieval-Augmented Generation (RAG)

- **Goal**: Improve LLM answers by retrieving relevant context from external documents.
- **Steps**:
    - **Retrieve**: Fetch documents using a vector store (FAISS).
    - **Compress**: Reduce document volume while preserving relevance.
    - **Generate**: Feed compressed context to an LLM for answer synthesis.

Vector Databases (FAISS)

- **Purpose**: Efficiently index and search high-dimensional vectors (text embeddings).
- **Use Case**: Retrieve top-*k* documents based on cosine similarity to the query.

Contextual Compression

- **Problem**: Retrieved documents often contain redundant or irrelevant text.
- **Solutions**:
    - **Extractors**: Use LLMs to extract key snippets (e.g., `LLMChainExtractor`).
    - **Filters**: Remove irrelevant documents using embeddings or LLMs.
    - **Pipelines**: Combine multiple steps (split → deduplicate → filter).

Embeddings

- **Function**: Convert text into dense vectors capturing semantic meaning.
- **Example**: `text-embedding-ada-002` from OpenAI.

3. Architecture Diagram

```
Copy
[User Query]
    |
    ▼
[FAISS Retriever] → Retrieves top-k documents
    |
    ▼
[Contextual Compressor] → (Optional: Split → Deduplicate → Filter)
    |
    ▼
[LLM (OpenAI)] → Generates final answer
```

4. Missing Points

1. **Parameter Tuning**:
    a. `chunk_size=500`, `similarity_threshold=0.76`, and `chunk_overlap=100` are arbitrary. These values should be optimized based on use case.
2. **Evaluation**:
    a. No metrics (e.g., precision@k, recall) to measure retrieval or compression quality.
3. **Alternatives**:
    a. Other retrievers (e.g., BM25, Pinecone) or compressors (e.g., Cohere rerankers).
4. **Latency vs. Quality**:
    a. Compression reduces context length but may lose nuance. Tradeoffs are not quantified.
5. **Error Handling**:
    a. No fallback mechanisms for API failures or empty retrieval results.
6. **Cost Implications**:
    a. Using OpenAI embeddings/LLMs incurs API costs. No discussion of caching or cost-saving strategies.
7. **Real-World Use Cases**:
    a. Ideal for chatbots, search engines, or document QA systems.

Contextual Compression in Retrieval-Augmented Generation (RAG): Detailed Notes

1. Core Concepts

What is Contextual Compression?

Contextual compression is a technique used in retrieval-augmented generative AI to **refine and condense retrieved documents** before passing them to a large language model (LLM). It ensures only the most relevant parts of the retrieved context are used for answer generation, reducing noise and computational overhead.

Why Use It?

- **Token Limits**: LLMs have token constraints (e.g., 4k-32k tokens). Compression avoids truncation.
- **Cost Efficiency**: Fewer tokens → lower API costs (e.g., OpenAI charges per token).
- **Accuracy**: Removing irrelevant text improves answer quality.

2. Theoretical Foundations

Key Components

1. **Retriever**: Fetches candidate documents (e.g., FAISS vector store).
2. **Compressor**: Refines retrieved documents using methods like extraction, filtering, or reranking.
3. **LLM**: Generates answers using the compressed context.

Workflow

```
Copy
[User Query]
    |
    ▼
[Retriever (FAISS)] → Retrieves top-k documents
    |
    ▼
[Contextual Compressor] → Filters/Extracts key parts
    |
    ▼
[LLM (OpenAI)] → Generates final answer
```

3. Types of Contextual Compression

1. Content-Aware Compression

Uses **LLMs** to analyze and refine documents:

- **LLMChainExtractor**:
  - An LLM extracts only the snippets relevant to the query.
  - Example: For a query *"climate change policies,"* the compressor might retain sentences like *"The president proposed a 50% carbon reduction by 2030."*
- **LLMChainFilter**:
  - An LLM scores documents as "relevant" or "irrelevant," discarding the latter.

2. Content-Agnostic Compression

Uses **embeddings and similarity thresholds**:

- **EmbeddingsFilter**:
  - Computes cosine similarity between query and document embeddings.
  - Discards documents below a threshold (e.g., `similarity_threshold=0.76`).
- **EmbeddingsRedundantFilter**:

- Removes duplicate documents by comparing embeddings of chunks.

3. Pipeline Compression
Combines multiple steps:

1. **Split**: Break documents into smaller chunks (e.g., `CharacterTextSplitter`).
2. **Deduplicate**: Remove redundant chunks (e.g., `EmbeddingsRedundantFilter`).
3. **Filter**: Retain chunks above a similarity threshold (e.g., `EmbeddingsFilter`).

4. Mathematical Basis
Cosine Similarity
Used in `EmbeddingsFilter` to measure relevance:

$$Similarity(Q,D) = Q \cdot D / \|Q\| \|D\| \text{Similarity}(Q,D) = \|\mathbf{Q}\| \|\mathbf{D}\| \mathbf{Q} \cdot \mathbf{D}$$

- **Q**: Query embedding.
- **D**: Document chunk embedding.
- **Threshold**: Only chunks with similarity > threshold are retained (e.g., 0.76).

LLM-Based Extraction

- **Prompt Engineering**:
  - Example prompt for `LLMChainExtractor`:

```
Copy
"Extract the parts of this document relevant to the query: {query}.
Document: {doc}"
```

- **Token Probability**:
  - LLMs assign higher probabilities to tokens in relevant spans, enabling extraction.

5. Code Implementation Breakdown
Step 1: Document Retrieval

```
python
Copy
retriever = FAISS.from_documents(texts,
OpenAIEmbeddings()).as_retriever()
docs = retriever.invoke("Climate change policies?")
```

- **FAISS** retrieves documents using vector similarity (OpenAI embeddings).

Step 2: Compression
Example 1: LLMChainExtractor

```
python
Copy
compressor = LLMChainExtractor.from_llm(llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor, base_retriever=retriever
)
compressed_docs = compression_retriever.invoke("Climate change
policies?")
```

- **Action**: The LLM parses each document and extracts only the text answering the query.

Example 2: Pipeline Compression

```
python
Copy
splitter = CharacterTextSplitter(chunk_size=300)
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)
relevant_filter = EmbeddingsFilter(embeddings=embeddings,
similarity_threshold=0.76)
pipeline_compressor = DocumentCompressorPipeline(
    transformers=[splitter, redundant_filter, relevant_filter]
)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor, base_retriever=retriever
)
```

- **Steps**:
    - **Split**: Break documents into 300-character chunks.
    - **Deduplicate**: Remove redundant chunks.
    - **Filter**: Keep chunks with similarity > 0.76.

6. Advantages of Contextual Compression

1. **Cost Savings**:

    a. Reduces tokens sent to the LLM (e.g., 10k tokens → 2k tokens = 80% cost reduction).
2. **Improved Latency**:
    a. Smaller context → faster LLM inference.
3. **Enhanced Precision**:
    a. Removes irrelevant text that could mislead the LLM.

7. Limitations

1. **Information Loss**:
    a. Over-aggressive filtering may discard critical context.
2. **Threshold Sensitivity**:
    a. A low `similarity_threshold` includes noise; a high threshold excludes relevant docs.
3. **LLM Dependency**:
    a. Methods like `LLMChainExtractor` require additional LLM calls, increasing latency.

8. Practical Example
Query: *"What were the top three priorities in the State of the Union address?"*
Retrieved Documents: 10 chunks (~5k tokens).
After Compression:

- **LLMChainExtractor** reduces to 3 chunks (~1k tokens) containing phrases like *"1. Infrastructure investment, 2. Healthcare reform, 3. Climate action."*
  LLM Answer:
  "The three priorities were infrastructure investment, healthcare reform, and climate action."

9. Key Takeaways

1. **Balancing Act**:
    a. Trade off between compression ratio and information retention.
2. **Hybrid Approaches**:
    a. Combine LLM-based extraction with embedding filters for optimal results.
3. **Evaluation**:

a. Use metrics like **compression ratio** and **answer accuracy** to tune thresholds.

10. Architecture Diagram

```
Copy
[User Query]
    │
    ▼
[FAISS Retriever] → Retrieves Top-10 Documents
    │
    ▼
[Contextual Compressor]
    ├─ Splitter → Breaks into smaller chunks
    ├─ Redundant Filter → Removes duplicates
    ├─ Embeddings Filter → Drops low-similarity chunks
    └─ LLM Extractor → Extracts key snippets
    │
    ▼
[LLM] → Generates answer using compressed context
```

11. Missing Points in the Code

1. **Parameter Tuning**:
    a. No experimentation with `chunk_size`, `similarity_threshold`, or overlap.
2. **Error Handling**:
    a. No fallback if compression removes all documents.
3. **Evaluation Metrics**:
    a. Missing precision/recall measurements for retrieval and compression.
4. **Alternatives**:
    a. Other compressors like Cohere Rerank or BERT-based rerankers.