NirDiamant /
RAG_Techniques

Code Issues Pull requests Actions Projects Security Insights

RAG_Techniques / all_rag_techniques / hierarchical_indices.ipynb

NirDiamant updated readme c924c8f · 4 months ago

375 lines (375 loc) · 14.7 KB

Hierarchical Indices in Document Retrieval

Overview

This code implements a Hierarchical Indexing system for document retrieval, utilizing two levels of encoding: document-level summaries and detailed chunks. This approach aims to improve the efficiency and relevance of information retrieval by first identifying relevant document sections through summaries, then drilling down to specific details within those sections.

Motivation

Traditional flat indexing methods can struggle with large documents or corpus, potentially missing context or returning irrelevant information. Hierarchical indexing addresses this by creating a two-tier search system, allowing for more efficient and context-aware retrieval.

1. PDF processing and text chunking
2. Asynchronous document summarization using OpenAI's GPT-4
3. Vector store creation for both summaries and detailed chunks using FAISS and OpenAI embeddings
4. Custom hierarchical retrieval function

Method Details

Document Preprocessing and Encoding

1. The PDF is loaded and split into documents (likely by page).
2. Each document is summarized asynchronously using GPT-4.
3. The original documents are also split into smaller, detailed chunks.
4. Two separate vector stores are created:
   - One for document-level summaries
   - One for detailed chunks

Asynchronous Processing and Rate Limiting

1. The code uses asynchronous programming (asyncio) to improve efficiency.
2. Implements batching and exponential backoff to handle API rate limits.

Hierarchical Retrieval

The `retrieve_hierarchical` function implements the two-tier search:

1. It first searches the summary vector store to identify relevant document sections.
2. For each relevant summary, it then searches the detailed chunk vector store, filtering by the corresponding page number.
3. This approach ensures that detailed information is retrieved only from the most relevant document sections.

Benefits of this Approach

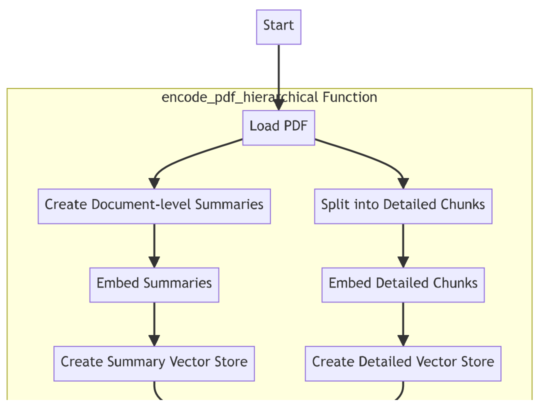
1. Improved Retrieval Efficiency: By first searching summaries, the system can quickly identify relevant document sections without processing all detailed chunks.
2. Better Context Preservation: The hierarchical approach helps maintain the broader context of retrieved information.
3. Scalability: This method is particularly beneficial for large documents or corpus, where flat searching might be inefficient or miss important context.
4. Flexibility: The system allows for adjusting the number of summaries and chunks retrieved, enabling fine-tuning for different use cases.

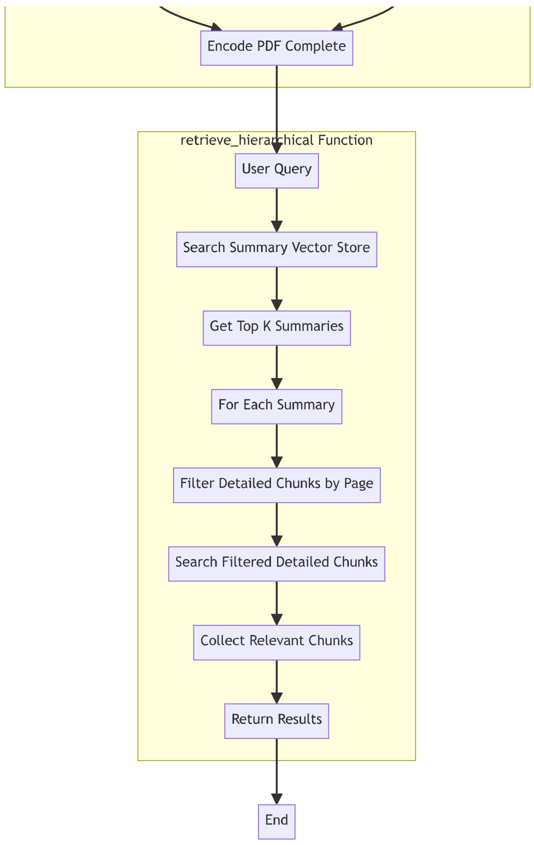
Implementation Details

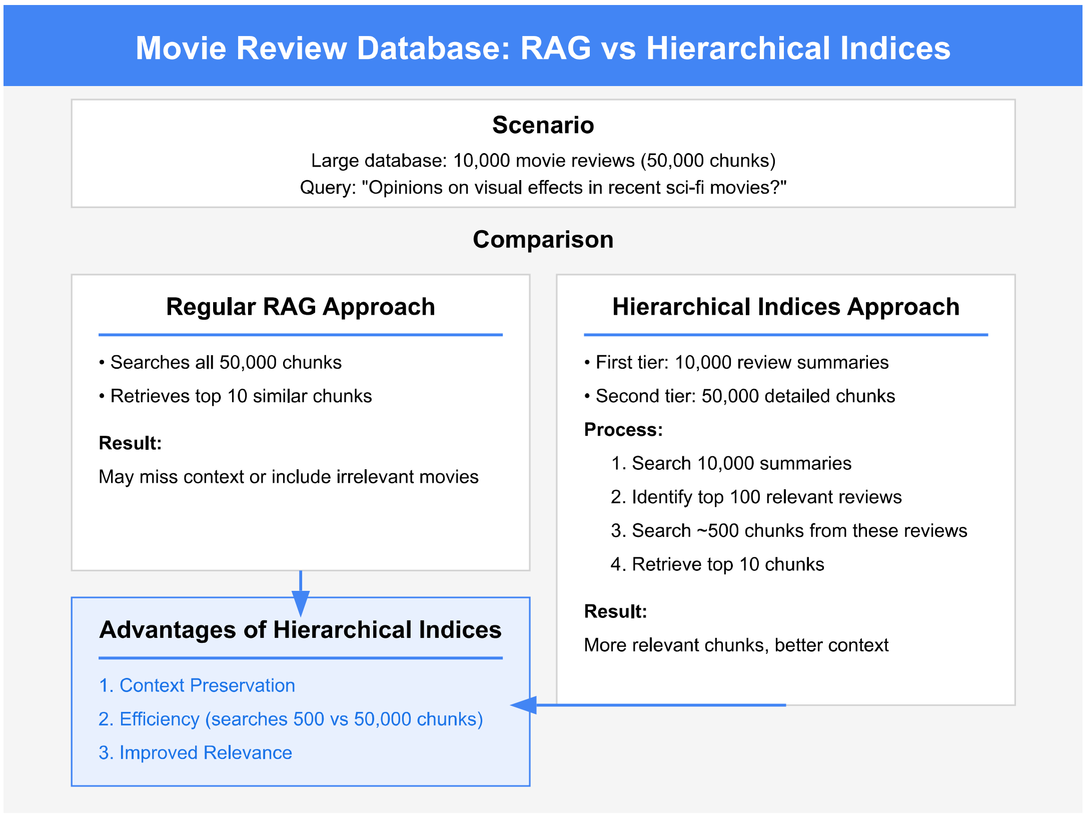
1. Asynchronous Programming: Utilizes Python's asyncio for efficient I/O operations and API calls.
2. Rate Limit Handling: Implements batching and exponential backoff to manage API rate limits effectively.
3. Persistent Storage: Saves the generated vector stores locally to avoid unnecessary recomputation.

Conclusion

Hierarchical indexing represents a sophisticated approach to document retrieval, particularly suitable for large or complex document sets. By leveraging both high-level summaries and detailed chunks, it offers a balance between broad context understanding and specific information retrieval. This method has potential applications in various fields requiring efficient and context-aware information retrieval, such as legal document analysis, academic research, or large-scale content management systems.


Start
encode_pdf_hierarchical Function
Load PDF
Create Document-level Summaries
Split into Detailed Chunks
Embed Summaries
Embed Detailed Chunks
Create Summary Vector Store
Create Detailed Vector Store


Encode PDF Complete
retrieve_hierarchical Function
User Query
Search Summary Vector Store
Get Top K Summaries
For Each Summary
Filter Detailed Chunks by Page
Search Filtered Detailed Chunks
Collect Relevant Chunks
Return Results
End


Movie Review Database: RAG vs Hierarchical Indices
Scenario
Large database: 10,000 movie reviews (50,000 chunks)
Query: "Opinions on visual effects in recent sci-fi movies?"
Comparison
Regular RAG Approach
• Searches all 50,000 chunks
• Retrieves top 10 similar chunks
Result:
May miss context or include irrelevant movies
Hierarchical Indices Approach
• First tier: 10,000 review summaries
• Second tier: 50,000 detailed chunks
Process:
1. Search 10,000 summaries
2. Identify top 100 relevant reviews
3. Search ~500 chunks from these reviews
4. Retrieve top 10 chunks
Result:
More relevant chunks, better context
Advantages of Hierarchical Indices
1. Context Preservation
2. Efficiency (searches 500 vs 50,000 chunks)
3. Improved Relevance

Import libraries

In [1]:
```
import asyncio
import os
import sys
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain.chains.summarize.chain import load_summarize_chain
from langchain.docstore.document import Document
```

```
sys.path.append(os.path.abspath(os.path.join(os.getcwd(), '..'))) # Add the p
from helper_functions import *
from evaluation.evalute_rag import *
from helper_functions import encode_pdf, encode_from_string

# Load environment variables from a .env file
load_dotenv()

# Set the OpenAI API key environment variable
os.environ["OPENAI_API_KEY"] = os.getenv('OPENAI_API_KEY')
```

```
c:\Users\N7\PycharmProjects\llm_tasks\RAG_TECHNIQUES\.venv\Lib\site-packages\de
epeval\__init__.py:45: UserWarning: You are using deepeval version 0.21.73, how
ever version 1.0.3 is available. You should consider upgrading via the "pip ins
tall --upgrade deepeval" command.
  warnings.warn(
```

Define document path

In [2]:

```
path = "../data/Understanding_Climate_Change.pdf"
```

Function to encode to both summary and chunk levels, sharing the page metadata

In [8]:

```
async def encode_pdf_hierarchical(path, chunk_size=1000, chunk_overlap=200, i
    """
    Asynchronously encodes a PDF book into a hierarchical vector store using (
    Includes rate limit handling with exponential backoff.

    Args:
        path: The path to the PDF file.
        chunk_size: The desired size of each text chunk.
        chunk_overlap: The amount of overlap between consecutive chunks.

    Returns:
        A tuple containing two FAISS vector stores:
        1. Document-level summaries
        2. Detailed chunks
    """

    # Load PDF documents
    if not is_string:
        loader = PyPDFLoader(path)
        documents = await asyncio.to_thread(loader.load)
    else:
        text_splitter = RecursiveCharacterTextSplitter(
            # Set a really small chunk size, just to show.
            chunk_size=chunk_size,
            chunk_overlap=chunk_overlap,
            length_function=len,
            is_separator_regex=False,
        )
        documents = text_splitter.create_documents([path])


    # Create document-level summaries
    summary_llm = ChatOpenAI(temperature=0, model_name="gpt-4o-mini", max_tok
    summary_chain = load_summarize_chain(summary_llm, chain_type="map_reduce"
async def summarize_doc(doc):
    """
    Summarizes a single document with rate limit handling.

    Args:
        doc: The document to be summarized.

    Returns:
        A summarized Document object.
    """
    # Retry the summarization with exponential backoff
    summary_output = await retry_with_exponential_backoff(summary_chain.a
    summary = summary_output['output_text']
    return Document(
        page_content=summary,
        metadata={"source": path, "page": doc.metadata["page"], "summary"
    )

# Process documents in smaller batches to avoid rate limits
batch_size = 5  # Adjust this based on your rate limits
summaries = []
for i in range(0, len(documents), batch_size):
    batch = documents[i:i+batch_size]
    batch_summaries = await asyncio.gather(*[summarize_doc(doc) for doc i
    summaries.extend(batch_summaries)
    await asyncio.sleep(1)  # Short pause between batches

# Split documents into detailed chunks
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size, chunk_overlap=chunk_overlap, length_function=l
)
detailed_chunks = await asyncio.to_thread(text_splitter.split_documents,

# Update metadata for detailed chunks
for i, chunk in enumerate(detailed_chunks):
    chunk.metadata.update({
        "chunk_id": i,
        "summary": False,
        "page": int(chunk.metadata.get("page", 0))
    })

# Create embeddings
embeddings = OpenAIEmbeddings()

# Create vector stores asynchronously with rate limit handling
async def create_vectorstore(docs):
    """
    Creates a vector store from a list of documents with rate limit handl

    Args:
        docs: The list of documents to be embedded.

    Returns:
        A FAISS vector store containing the embedded documents.
    """
    return await retry_with_exponential_backoff(
        asyncio.to_thread(FAISS.from_documents, docs, embeddings)
    )

# Generate vector stores for summaries and detailed chunks concurrently
summary_vectorstore, detailed_vectorstore = await asyncio.gather(
    create_vectorstore(summaries),
    create_vectorstore(detailed_chunks)
    )

    return summary_vectorstore, detailed_vectorstore
```

Encode the PDF book to both document-level summaries and detailed chunks if the vector stores do not exist

In [9]:

```
if os.path.exists("../vector_stores/summary_store") and os.path.exists("../ve
   embeddings = OpenAIEmbeddings()
   summary_store = FAISS.load_local("../vector_stores/summary_store", embeddi
   detailed_store = FAISS.load_local("../vector_stores/detailed_store", embed

else:
    summary_store, detailed_store = await encode_pdf_hierarchical(path)
    summary_store.save_local("../vector_stores/summary_store")
    detailed_store.save_local("../vector_stores/detailed_store")
```

Retrieve information according to summary level, and then retrieve information from the chunk level vector store and filter according to the summary level pages

In [10]:

```
def retrieve_hierarchical(query, summary_vectorstore, detailed_vectorstore, k
    """
    Performs a hierarchical retrieval using the query.

    Args:
        query: The search query.
        summary_vectorstore: The vector store containing document summaries.
        detailed_vectorstore: The vector store containing detailed chunks.
        k_summaries: The number of top summaries to retrieve.
        k_chunks: The number of detailed chunks to retrieve per summary.

    Returns:
        A list of relevant detailed chunks.
    """

    # Retrieve top summaries
    top_summaries = summary_vectorstore.similarity_search(query, k=k_summarie

    relevant_chunks = []
    for summary in top_summaries:
        # For each summary, retrieve relevant detailed chunks
        page_number = summary.metadata["page"]
        page_filter = lambda metadata: metadata["page"] == page_number
        page_chunks = detailed_vectorstore.similarity_search(
            query,
            k=k_chunks,
            filter=page_filter
        )
        relevant_chunks.extend(page_chunks)

    return relevant_chunks
```

Demonstrate on a use case

In [ ]:

```python
query = "What is the greenhouse effect?"
results = retrieve_hierarchical(query, summary_store, detailed_store)

# Print results
for chunk in results:
    print(f"Page: {chunk.metadata['page']}")
    print(f"Content: {chunk.page_content}...")  # Print first 100 characters
    print("---")
```