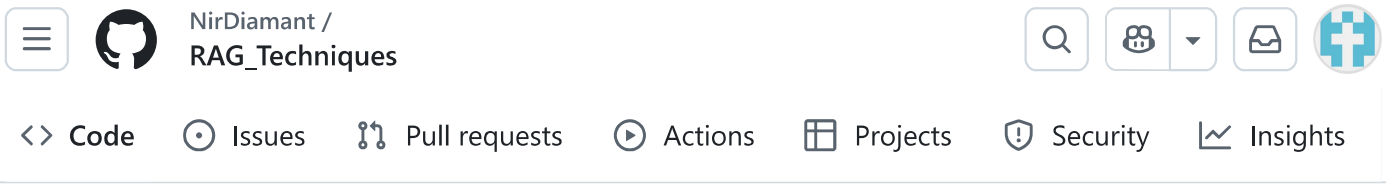


RAG_Techniques / all_rag_techniques / raptor.ipynb

eliavs Code updates: bc8e374 · 4 months ago

603 lines (603 loc) · 22.2 KB

Preview Code Blame Raw

RAPTOR: Recursive Abstractive Processing and Thematic Organization for Retrieval

Overview

RAPTOR is an advanced information retrieval and question-answering system that combines hierarchical document summarization, embedding-based retrieval, and contextual answer generation. It aims to efficiently handle large document collections by creating a multi-level tree of summaries, allowing for both broad and detailed information retrieval.

Motivation

Traditional retrieval systems often struggle with large document sets, either missing important details or getting overwhelmed by irrelevant information. RAPTOR addresses this by creating a hierarchical structure of the document collection, allowing it to navigate between high-level concepts and specific details as needed.

Key Components

1. **Tree Building**: Creates a hierarchical structure of document summaries.
2. **Embedding and Clustering**: Organizes documents and summaries based on semantic similarity.
3. **Vectorstore**: Efficiently stores and retrieves document and summary embeddings.
4. **Contextual Retriever**: Selects the most relevant information for a given query.
5. **Answer Generation**: Produces coherent responses based on retrieved information.

Method Details

Tree Building

1. Start with original documents at level 0.
2. For each level:
   - Embed the texts using a language model.
   - Cluster the embeddings (e.g., using Gaussian Mixture Models).
   - Generate summaries for each cluster.
   - Use these summaries as the texts for the next level.
3. Continue until reaching a single summary or a maximum level.

Embedding and Retrieval

1. Embed all documents and summaries from all levels of the tree

1. Embed all documents and summaries from all levels of the tree.
2. Store these embeddings in a vectorstore (e.g., FAISS) for efficient similarity search.
3. For a given query:
   - Embed the query.
   - Retrieve the most similar documents/summaries from the vectorstore.

Contextual Compression

1. Take the retrieved documents/summaries.
2. Use a language model to extract only the most relevant parts for the given query.

Answer Generation

1. Combine the relevant parts into a context.
2. Use a language model to generate an answer based on this context and the original query.

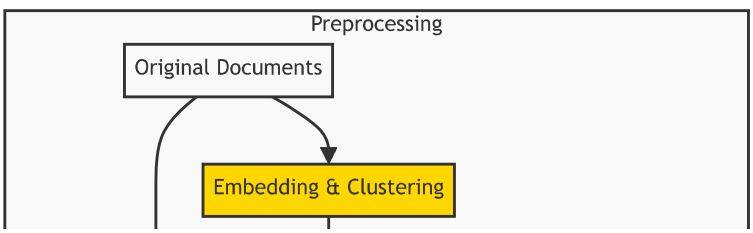
Benefits of this Approach

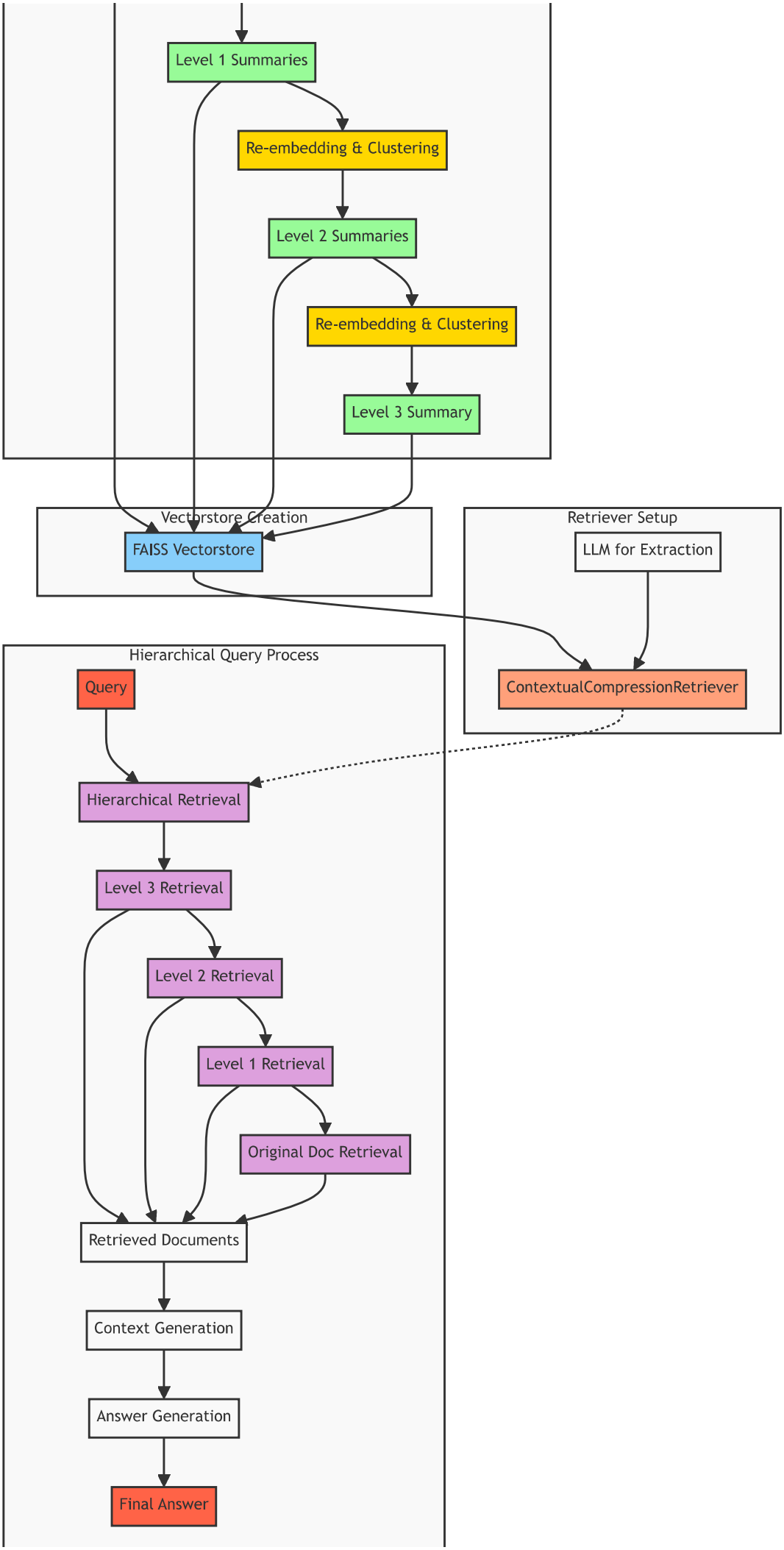
1. **Scalability**: Can handle large document collections by working with summaries at different levels.
2. **Flexibility**: Capable of providing both high-level overviews and specific details.
3. **Context-Awareness**: Retrieves information from the most appropriate level of abstraction.
4. **Efficiency**: Uses embeddings and vectorstore for fast retrieval.
5. **Traceability**: Maintains links between summaries and original documents, allowing for source verification.

Conclusion

RAPTOR represents a significant advancement in information retrieval and question-answering systems. By combining hierarchical summarization with embedding-based retrieval and contextual answer generation, it offers a powerful and flexible approach to handling large document collections. The system's ability to navigate different levels of abstraction allows it to provide relevant and contextually appropriate answers to a wide range of queries.

While RAPTOR shows great promise, future work could focus on optimizing the tree-building process, improving summary quality, and enhancing the retrieval mechanism to better handle complex, multi-faceted queries. Additionally, integrating this approach with other AI technologies could lead to even more sophisticated information processing systems.




Level 1 Summaries
Re-embedding & Clustering
Level 2 Summaries
Re-embedding & Clustering
Level 3 Summary
Vectorstore Creation
FAISS Vectorstore
Retriever Setup
LLM for Extraction
ContextualCompressionRetriever
Hierarchical Query Process
Query
Hierarchical Retrieval
Level 3 Retrieval
Level 2 Retrieval
Level 1 Retrieval
Original Doc Retrieval
Retrieved Documents
Context Generation
Answer Generation
Final Answer

Imports and Setup

In [75]:

```
import numpy as np
import pandas as pd
from typing import List, Dict, Any
from sklearn.mixture import GaussianMixture
from langchain.chains.llm import LLMChain
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain.schema import AIMessage
from langchain.docstore.document import Document

import matplotlib.pyplot as plt
import logging
import os
import sys
from dotenv import load_dotenv

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

# 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')
```

Define logging, llm and embeddings

In [30]:

```
# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -

embeddings = OpenAIEmbeddings()
llm = ChatOpenAI(model_name="gpt-4o-mini")
```

Helper Functions

In [85]:

```
def extract_text(item):
    """Extract text content from either a string or an AIMessage object."""
    if isinstance(item, AIMessage):
        return item.content
    return item

def embed_texts(texts: List[str]) -> List[List[float]]:
    """Embed texts using OpenAIEmbeddings."""
    logging.info(f"Embedding {len(texts)} texts")
    return embeddings.embed_documents([extract_text(text) for text in texts])

def perform_clustering(embeddings: np.ndarray, n_clusters: int = 10) -> np.nd
```

```
    """Perform clustering on embeddings using Gaussian Mixture Model."""
    logging.info(f"Performing clustering with {n_clusters} clusters")
    gm = GaussianMixture(n_components=n_clusters, random_state=42)
    return gm.fit_predict(embeddings)

def summarize_texts(texts: List[str]) -> str:
    """Summarize a list of texts using OpenAI."""
    logging.info(f"Summarizing {len(texts)} texts")
    prompt = ChatPromptTemplate.from_template(
        "Summarize the following text concisely:\n\n{text}"
    )
    chain = prompt | llm
    input_data = {"text": texts}
    return chain.invoke(input_data)

def visualize_clusters(embeddings: np.ndarray, labels: np.ndarray, level: int
    """Visualize clusters using PCA."""
    from sklearn.decomposition import PCA
    pca = PCA(n_components=2)
    reduced_embeddings = pca.fit_transform(embeddings)

    plt.figure(figsize=(10, 8))
    scatter = plt.scatter(reduced_embeddings[:, 0], reduced_embeddings[:, 1],
    plt.colorbar(scatter)
    plt.title(f'Cluster Visualization - Level {level}')
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.show()
```

RAPTOR Core Function

In [86]:

```
def build_raptor_tree(texts: List[str], max_levels: int = 3) -> Dict[int, pd.
    """Build the RAPTOR tree structure with level metadata and parent-child r
    results = {}
    current_texts = [extract_text(text) for text in texts]
    current_metadata = [{"level": 0, "origin": "original", "parent_id": None}

    for level in range(1, max_levels + 1):
        logging.info(f"Processing level {level}")

        embeddings = embed_texts(current_texts)
        n_clusters = min(10, len(current_texts) // 2)
        cluster_labels = perform_clustering(np.array(embeddings), n_clusters)

        df = pd.DataFrame({
            'text': current_texts,
            'embedding': embeddings,
            'cluster': cluster_labels,
            'metadata': current_metadata
        })

        results[level-1] = df

        summaries = []
        new_metadata = []
        for cluster in df['cluster'].unique():
            cluster_docs = df[df['cluster'] == cluster]
            cluster_texts = cluster_docs['text'].tolist()
            cluster_metadata = cluster_docs['metadata'].tolist()
            summary = summarize_texts(cluster_texts)
            summaries.append(summary)
```

```
            new_metadata.append({
                "level": level,
                "origin": f"summary_of_cluster_{cluster}_level_{level-1}",
                "child_ids": [meta.get('id') for meta in cluster_metadata],
                "id": f"summary_{level}_{cluster}"
            })

        current_texts = summaries
        current_metadata = new_metadata

        if len(current_texts) <= 1:
            results[level] = pd.DataFrame({
                'text': current_texts,
                'embedding': embed_texts(current_texts),
                'cluster': [0],
                'metadata': current_metadata
            })
            logging.info(f"Stopping at level {level} as we have only one summ
            break

    return results
```

Vectorstore Function

In [87]:

```
def build_vectorstore(tree_results: Dict[int, pd.DataFrame]) -> FAISS:
    """Build a FAISS vectorstore from all texts in the RAPTOR tree."""
    all_texts = []
    all_embeddings = []
    all_metadatas = []

    for level, df in tree_results.items():
        all_texts.extend([str(text) for text in df['text'].tolist()])
        all_embeddings.extend([embedding.tolist() if isinstance(embedding, np
        all_metadatas.extend(df['metadata'].tolist())

    logging.info(f"Building vectorstore with {len(all_texts)} texts")

    # Create Document objects manually to ensure correct types
    documents = [Document(page_content=str(text), metadata=metadata)
                 for text, metadata in zip(all_texts, all_metadatas)]

    return FAISS.from_documents(documents, embeddings)
```

Define tree traversal retrieval

In [88]:

```
def tree_traversal_retrieval(query: str, vectorstore: FAISS, k: int = 3) -> L
    """Perform tree traversal retrieval."""
    query_embedding = embeddings.embed_query(query)

    def retrieve_level(level: int, parent_ids: List[str] = None) -> List[Docu
        if parent_ids:
            docs = vectorstore.similarity_search_by_vector_with_relevance_sco
                query_embedding,
                k=k,
                filter=lambda meta: meta['level'] == level and meta['id'] in
            )
        else:
```

```
            docs = vectorstore.similarity_search_by_vector_with_relevance_sco
                query_embedding,
                k=k,
                filter=lambda meta: meta['level'] == level
            )

        if not docs or level == 0:
            return docs

        child_ids = [doc.metadata.get('child_ids', []) for doc, _ in docs]
        child_ids = [item for sublist in child_ids for item in sublist]  # Fl

        child_docs = retrieve_level(level - 1, child_ids)
        return docs + child_docs

    max_level = max(doc.metadata['level'] for doc in vectorstore.docstore.val
    return retrieve_level(max_level)
```

Create Retriever

In [80]:

```
def create_retriever(vectorstore: FAISS) -> ContextualCompressionRetriever:
    """Create a retriever with contextual compression."""
    logging.info("Creating contextual compression retriever")
    base_retriever = vectorstore.as_retriever()

    prompt = ChatPromptTemplate.from_template(
        "Given the following context and question, extract only the relevant
        "Context: {context}\n"
        "Question: {question}\n\n"
        "Relevant Information:"
    )

    extractor = LLMChainExtractor.from_llm(llm, prompt=prompt)

    return ContextualCompressionRetriever(
        base_compressor=extractor,
        base_retriever=base_retriever
    )
```

Define hierarchical retrieval

In [100…

```
def hierarchical_retrieval(query: str, retriever: ContextualCompressionRetrie
    """Perform hierarchical retrieval starting from the highest level, handli
    all_retrieved_docs = []

    for level in range(max_level, -1, -1):
        # Retrieve documents from the current level
        level_docs = retriever.get_relevant_documents(
            query,
            filter=lambda meta: meta['level'] == level
        )
        all_retrieved_docs.extend(level_docs)

        # If we've found documents, retrieve their children from the next lev
        if level_docs and level > 0:
            child_ids = [doc.metadata.get('child_ids', []) for doc in level_d
            child_ids = [item for sublist in child_ids for item in sublist if
```

```
            if child_ids:  # Only modify query if there are valid child IDs
                child_query = f" AND id:({' OR '.join(str(id) for id in child
                query += child_query

    return all_retrieved_docs
```

RAPTOR Query Process (Online Process)

In [101…

```
def raptor_query(query: str, retriever: ContextualCompressionRetriever, max_l
    """Process a query using the RAPTOR system with hierarchical retrieval."""
    logging.info(f"Processing query: {query}")

    relevant_docs = hierarchical_retrieval(query, retriever, max_level)

    doc_details = []
    for i, doc in enumerate(relevant_docs, 1):
        doc_details.append({
            "index": i,
            "content": doc.page_content,
            "metadata": doc.metadata,
            "level": doc.metadata.get('level', 'Unknown'),
            "similarity_score": doc.metadata.get('score', 'N/A')
        })

    context = "\n\n".join([doc.page_content for doc in relevant_docs])

    prompt = ChatPromptTemplate.from_template(
        "Given the following context, please answer the question:\n\n"
        "Context: {context}\n\n"
        "Question: {question}\n\n"
        "Answer:"
    )
    chain = LLMChain(llm=llm, prompt=prompt)
    answer = chain.run(context=context, question=query)

    logging.info("Query processing completed")

    result = {
        "query": query,
        "retrieved_documents": doc_details,
        "num_docs_retrieved": len(relevant_docs),
        "context_used": context,
        "answer": answer,
        "model_used": llm.model_name,
    }

    return result


def print_query_details(result: Dict[str, Any]):
    """Print detailed information about the query process, including tree lev
    print(f"Query: {result['query']}")
    print(f"\nNumber of documents retrieved: {result['num_docs_retrieved']}")
    print(f"\nRetrieved Documents:")
    for doc in result['retrieved_documents']:
        print(f"  Document {doc['index']}:")
        print(f"    Content: {doc['content'][:100]}...")  # Show first 100 ch
        print(f"    Similarity Score: {doc['similarity_score']}")
        print(f"    Tree Level: {doc['metadata'].get('level', 'Unknown')}")
        print(f"    Origin: {doc['metadata'].get('origin', 'Unknown')}")
        if 'child_docs' in doc['metadata']:
```

```
            print(f"    Number of Child Documents: {len(doc['metadata']['chil
        print()

    print(f"\nContext used for answer generation:")
    print(result['context_used'])

    print(f"\nGenerated Answer:")
    print(result['answer'])

    print(f"\nModel Used: {result['model_used']}")
```

Example Usage and Visualization

Define data folder

In [36]:
```
path = "../data/Understanding_Climate_Change.pdf"
```

Process texts

In [37]:
```
loader = PyPDFLoader(path)
documents = loader.load()
texts = [doc.page_content for doc in documents]
```

Create RAPTOR components instances

In [ ]:
```
# Build the RAPTOR tree
tree_results = build_raptor_tree(texts)
```

In [ ]:
```
# Build vectorstore
vectorstore = build_vectorstore(tree_results)
```

In [ ]:
```
# Create retriever
retriever = create_retriever(vectorstore)
```

Run a query and observe where it got the data from + results

In [ ]:
```
# Run the pipeline
max_level = 3  # Adjust based on your tree depth
query = "What is the greenhouse effect?"
result = raptor_query(query, retriever, max_level)
print_query_details(result)
```