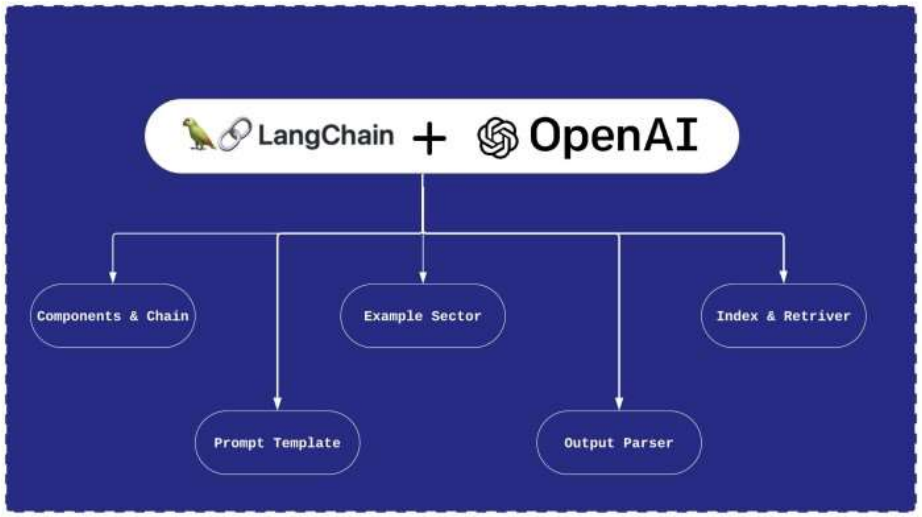


Transforming Question Answering with OpenAI and LangChain: Harnessing the Potential of Retrieval

Augmented Generation (RAG)

Vraj Routu
Cloud Solutions Architect at Siemens Healthineers | Gen AI

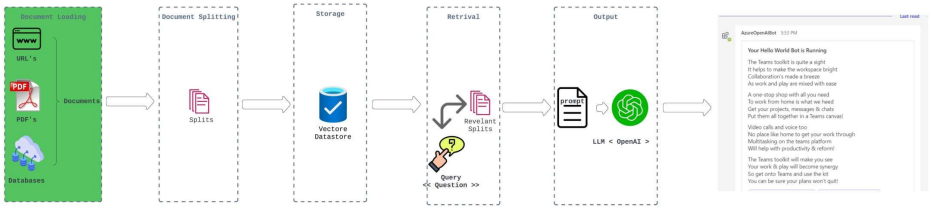
5 juillet 2023

Retrieval augmented generation (RAG) is a powerful approach that combines the capabilities of language models with the ability to retrieve relevant information from external documents. In RAG, a language model retrieves contextual documents from an external dataset to enhance its understanding and generate more accurate and informative responses.

This retrieval-based approach is particularly useful when we want to ask questions or generate content based on specific documents such as PDFs, videos, web pages, or even Notion databases. By incorporating external documents into the language model's execution, RAG

expands its knowledge and provides more contextually rich output

Document Loading



Before we delve into the details of retrieval augmented generation, let's explore how documents can be loaded into the system. In the examples below, we demonstrate loading different types of documents.

PDFs

To load a PDF document, we can use the PyPDFLoader from the langchain library. Let's consider an example where we load a PDF transcript.

```
from langchain.document_loaders import PyPDFLoader

loader = PyPDFLoader("docs/langchain/MachineLearning.pdf'
pages = loader.load()
```

Each page of the PDF is represented as a **Document** object, containing the text content of the page and metadata. We can access individual pages and their metadata using the following code:

```
page = pages[0]
print(page.page_content[0:500])
print(page.metadata)
```

YouTube Videos

In addition to PDFs, we can also load content from YouTube videos. The langchain library provides a GenericLoader that supports various document loaders and parsers. Let's load the content from a YouTube video using the YouTubeAudioLoader and OpenAIWhisperParser.

```
from langchain.document_loaders.generic import GenericLoa
from langchain.document_loaders.blob_loaders.youtube_audi
from langchain.document_loaders.parsers import OpenAIWhis

url = "https://www.youtube.com/watch?v=jGwO_UgTS7I"
save_dir = "docs/youtube/"
loader = GenericLoader(
    YoutubeAudioLoader([url], save_dir),
    OpenAIWhisperParser()
)
docs = loader.load()

print(docs[0].page_content[0:500])
```

URLs

We can also load content directly from web pages using the WebBaseLoader. Here's an example of loading content from a GitHub page:

```
from langchain.document_loaders import WebBaseLoader
```

```
loader = WebBaseLoader("https://github.com/basecamp/handb
docs = loader.load()

print(docs[0].page_content[:500])
```

Notion Databases

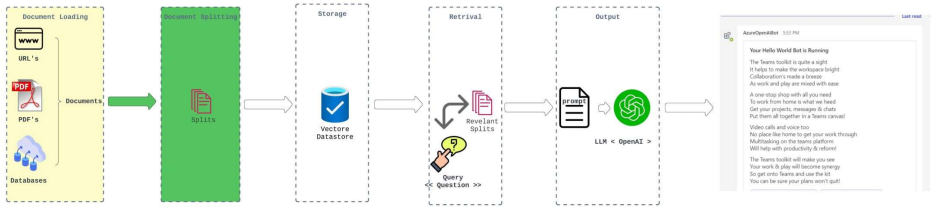
Furthermore, we can load content from Notion databases by utilizing the NotionDirectoryLoader. Follow the steps mentioned in the link provided to export a Notion site as Markdown/CSV and save it as a folder containing the markdown files.

```
from langchain.document_loaders import NotionDirectoryLoa

loader = NotionDirectoryLoader("docs/Notion_DB")
docs = loader.load()

print(docs[0].page_content[0:200])
print(docs[0].metadata)
```

Document Splitting



After loading the documents, we may need to split them into smaller chunks to effectively process the content. The langchain library offers various text splitters for this purpose.

Recursive Character Text Splitter

The RecursiveCharacterTextSplitter is recommended for generic text splitting. It splits the text based on specified chunk size and overlap, while preserving the document's context.

```
from langchain.text_splitter import RecursiveCharacterTe

chunk_size = 26
chunk_overlap = 4
```

```
r_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap
)

text1 = 'abcdefghijklmnopqrstuvwxyz'
text2 = 'abcdefghijklmnopqrstuvwxyzabcdefg'
text3 = "a b c d e f g h i j k l m n o p q r s t u v w x

print(r_splitter.split_text(text1))
print(r_splitter.split_text(text2))
print(r_splitter.split_text(text3))
```

Token Text Splitter

If we want to split the text based on token count, we can use the TokenTextSplitter. This approach is useful when considering language model context windows that are designated in tokens.

```
from langchain.text_splitter import TokenTextSplitter

text_splitter = TokenTextSplitter(chunk_size=10, chunk_o
text = "foo bar bazzyfoo"
```

```
print(text_splitter.split_text(text))
```

Markdown Header Text Splitter

When dealing with structured documents like Markdown, we can use the MarkdownHeaderTextSplitter to split the text while preserving the header metadata. This is especially useful when headers contain valuable information that should be included in the generated content.

```
from langchain.document_loaders import NotionDirectoryLoa
from langchain.text_splitter import MarkdownHeaderTextSpl


markdown_document = """# Title\n\n \
## Chapter 1\n\n \
Hi this is Mona\n\n Hi this is Loki\n\n \
### Section \n\n \
Hi this is Hamilton \n\n
## Chapter 2\n\n \
Hi this is Vraj"""

headers_to_split_on = [
    ("#", "Header 1"),
```

```
    ("##", "Header 2"),
    ("###", "Header 3"),
]

markdown_splitter = MarkdownHeaderTextSplitter(
    headers_to_split_on=headers_to_split_on
)
md_header_splits = markdown_splitter.split_text(markdown_

print(md_header_splits[0])
print(md_header_splits[1])
```

Embeddings

Next, we focus on embeddings, which play a vital role in representing the textual content in a numerical format. We utilize the OpenAIEmbeddings module to generate embeddings for sentences. These embeddings capture the semantic information of the sentences and enable us to perform similarity calculations between them:

```
from langchain.embeddings.openai import OpenAIEmbeddings

embedding = OpenAIEmbeddings()
```

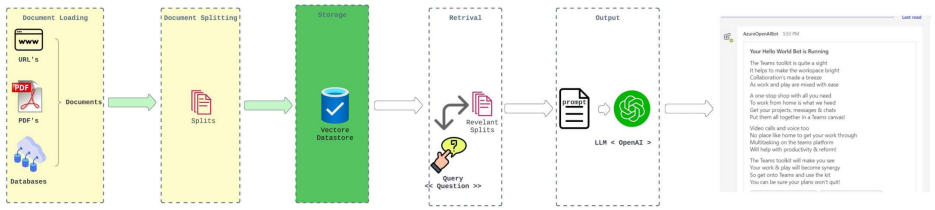
```
sentence1 = "I like dogs"
sentence2 = "I like canines"
sentence3 = "The weather is ugly outside"

embedding1 = embedding.embed_query(sentence1)
embedding2 = embedding.embed_query(sentence2)
embedding3 = embedding.embed_query(sentence3)

import numpy as np

np.dot(embedding1, embedding2)
np.dot(embedding1, embedding3)
np.dot(embedding2, embedding3)
```

Vector Datastore



VectorDB and LLM Configuration

To enable effective question answering, we need to utilize a vector database (VectorDB) for efficient retrieval of relevant information. In our example, we'll load the VectorDB and configure the language model (LLM) accordingly.

```
import os
import openai
import sys
sys.path.append('../..')

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

openai.api_key = os.environ['OPENAI_API_KEY']

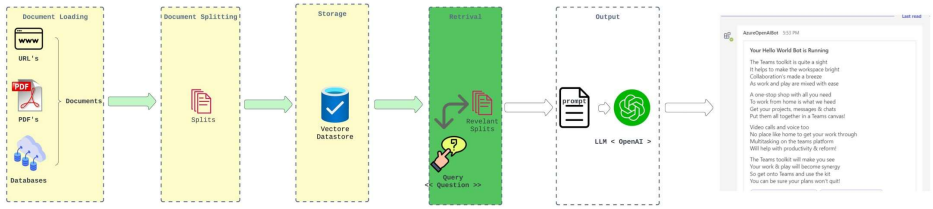
# Check the current date to determine the LLM version
import datetime
current_date = datetime.datetime.now().date()
if current_date < datetime.date(2023, 9, 2):
    llm_name = "gpt-3.5-turbo-0301"
else:
    llm_name = "gpt-3.5-turbo"
print(llm_name)

from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
persist_directory = 'docs/chroma/'
embedding = OpenAIEmbeddings()
vectordb = Chroma(persist_directory=persist_directory, em
```

```
print(vectordb._collection.count())
```

Question & Answer

RetrievalQA Chain



The RetrievalQA chain is a crucial component for question answering in RAG. It combines the language model with the VectorDB's retrieval capabilities. Let's set up the RetrievalQA chain and perform a sample question answering task.

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name=llm_name, temperature=0)

from langchain.chains import RetrievalQA
```

```
qa_chain = RetrievalQA.from_chain_type(
    llm,
    retriever=vectordb.as_retriever()
)

question = "What are the major topics for this class?"
result = qa_chain({"query": question})

result["result"]
```

Adding Prompt for Question Answering

To provide additional context and structure to the question answering process, we can use a prompt template. In this example, we'll build a prompt template that includes context and the question, guiding the model to provide a concise answer.

```
from langchain.prompts import PromptTemplate

# Build prompt
template = """Use the following pieces of context to answ
{context}
Question: {question}
```

```
Helpful Answer:"""
QA_CHAIN_PROMPT = PromptTemplate.from_template(template)

# Run the RetrievalQA chain with prompt
qa_chain = RetrievalQA.from_chain_type(
    llm,
    retriever=vectordb.as_retriever(),
    return_source_documents=True,
    chain_type_kwargs={"prompt": QA_CHAIN_PROMPT}
)

question = "Is probability aclass topic?"

result = qa_chain({"query": question})
result["result"]
result["source_documents"][0]

## RetrievalQA Chain Types
The RetrievalQA chain supports different chain types, suc

```python
qa_chain_mr = RetrievalQA.from_chain_type(
    llm,
    retriever=vectordb.as_retriever(),
    chain_type="map_reduce"
)

result = qa_chain_mr({"query": question})
result["result"]

qa_chain_mr = RetrievalQA.from_chain_type(
    llm,
```
```

```
    retriever=vectordb.as_retriever(),
    chain_type="refine"
)

result = qa_chain_mr({"query": question})

result["result"]
```

Limitations of RetrievalQA

While RetrievalQA is a powerful approach, it has certain limitations. One such limitation is the inability to preserve conversational history. Each question is treated independently, and the model does not have access to past questions or answers. Let's examine this limitation further.

```
qa_chain = RetrievalQA.from_chain_type(
    llm,
    retriever=vectordb.as_retriever()
)

question = "Is probability a class topic?"
result = qa_chain({"query": question})
result["result"]
```

```
question = "Why are those prerequisites needed?"
result = qa_chain({"query": question})
result["result"]
```

Conclusion

Retrieval augmented generation (RAG) is a powerful technique that incorporates external documents to enhance the capabilities of language models. By loading and splitting documents, RAG provides a broader context for more accurate and informative responses. Additionally, RAG can be utilized for question answering tasks, revolutionizing language understanding and generation. The LangChain library offers developers the necessary tools to implement RAG and build sophisticated question answering systems. While there are limitations, such as the inability to preserve conversational history, ongoing advancements in RAG continue to push the boundaries of question answering. Embrace the possibilities of RAG and its transformative impact on language understanding and generation.

```
Repository : https://github.com/kishorerv93/langchain-qna
```

Signaler ceci

Publié par

Vraj Routu 3 articles + Suivre

Cloud Solutions Architect at Siemens Healthineers | Gen AI

Publié • 1 mois

📣 Excited to share my latest article on LangChain and OpenAI! 🚀 In this article, I dive deep into the world of retrieval augmented generation (RAG) and how it can revolutionize chatting with your data. By leveraging the powerful combination of Azure OpenAI and LangChain, we explore how RAG enhances contextual understanding by incorporating external documents. #RAG #QuestionAnswering #AzureOpenAI #LangChain #NLP #MachineLearning Repo: https://lnkd.in/eAu_UZjt

J'aime Commenter Partager 29 1 commentaire

Réactions

+17

1 commentaire

Les plus pertinents

Ajouter un commentaire...

Hadi Arbabi • + que 3e
Director Of Engineering

1 mois •••

Thanks for sharing your experiences on these libraries and approaches on RAG and AI.

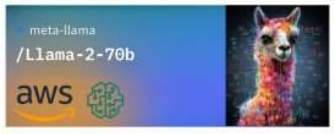
Voir la traduction

J'aime · 2 | Répondre

Vraj Routu
Cloud Solutions Architect at Siemens Healthineers | Gen AI

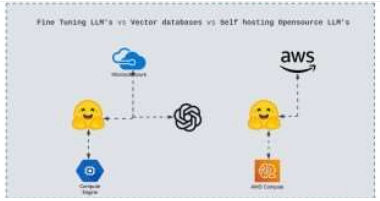
+ Suivre

Plus de Vraj Routu



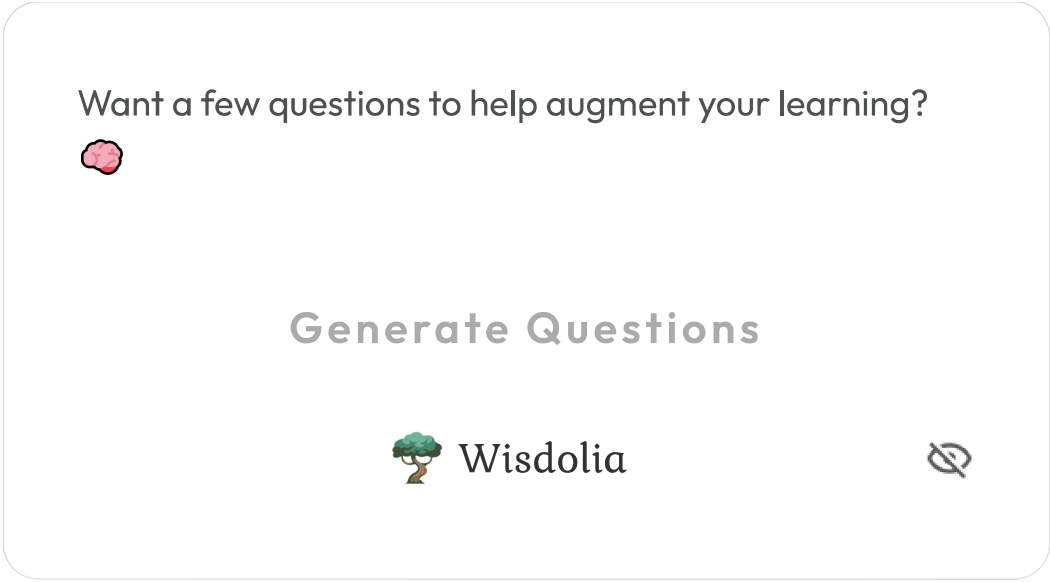
Enhancing Language Models with QLoRA: Efficient Fine-Tuning of LLaMA 2 on...

Vraj Routu sur LinkedIn



Overcoming OpenAI Model Expiration: Efficient Strategies for Machine Learning...

Vraj Routu sur LinkedIn


Want a few questions to help augment your learning?
Generate Questions
Wisdolia