Unleashing the Power of Retrieval-Augmented Generation (RAG)

AI/ML Team, Kanaka Software

September 12, 2024

kanaka Presents

RAG - Retrieval Augmented Generation

Unleashing the Power of Retrieval-Augmented Generation (RAG)

Talk to your data with RAG

Innovating Technology, Empowering Developers

At Kanaka Software, our AI/ML Team combines deep expertise with a relentless drive for innovation to deliver exceptional results. This guide reflects our dedication to empowering businesses with cutting-edge AI and machine learning solutions, helping them achieve their goals and stay ahead in a competitive landscape. Trust Kanaka Software as your strategic partner in navigating the future of technology, ensuring your success every step of the way.

Ready to transform your understanding of AI?

Let's embark on this exciting journey together!

A "Great Place to Work" certified organization

www.kanakasoftware.com | info@kanakasoftware.com

September 12, 2024

RAG - Retrieval Augmented Generation

Unleashing the Power of Retrieval-Augmented Generation (RAG)

by

AI/ML Team, Kanaka Software

Nitish Katkade

Sanket Nagare

Mahadev Godbole

Rajesh Pandhare

September 12, 2024

Contents

Unleashing the Power of Retrieval-Augmented Generation (RAG)

What is RAG? And Why do we need it?

Retrieval Augmented Generation is known as RAG. RAG is a technique used to increase the LLM knowledge base. LLM's are trained on lots of data after the training is completed they are used in applications. Now the issue arises when need to interact with new data. i.e. Data that is generated after the training. We can't just train it again because training a Large Language Model Model is not a piece of cake. It takes a huge amount of computational resources, electricity, and money. So how do we increase the knowledge base of our LLM's? that's where RAG comes into the picture.

We understand exactly what RAG is and why we require it. We can now proceed with our task.

The relevant data is obtained from the data and sent to the model once the query is executed at runtime.

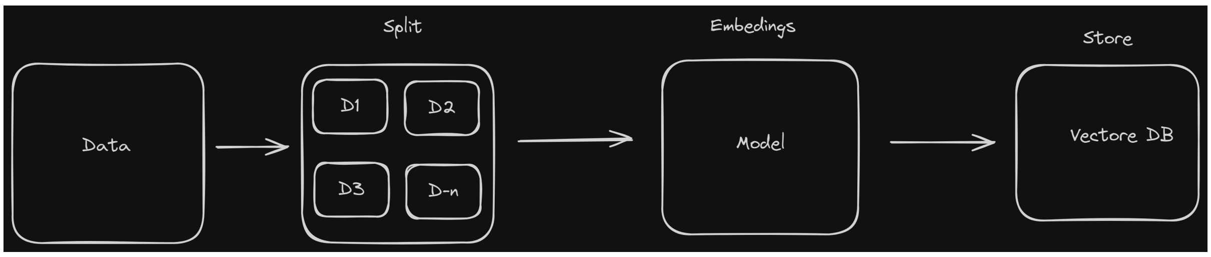

Figure 1: What is Rag

Let's explore how to prepare data for our model.

1. Load: Our data must first be loaded.
2. Split: Divide lengthy materials into manageable portions.
3. Store: To make finding easier later, we store our data, usually in vector format.
4. Retrieve: Based on the user's input, relevant chunks are obtained.
5. Generate: The model will be provided with the user's inquiry and the gathered data to provide an answer.

In this guide, we'll build an app that answers questions about a website's content. For this tutorial, we'll use article from realpython website "**Python Polars: A Lightning-Fast DataFrame Library**" as our source material. This will allow us to demonstrate how to ask questions about the post's content using RAG.

RAG Architecture

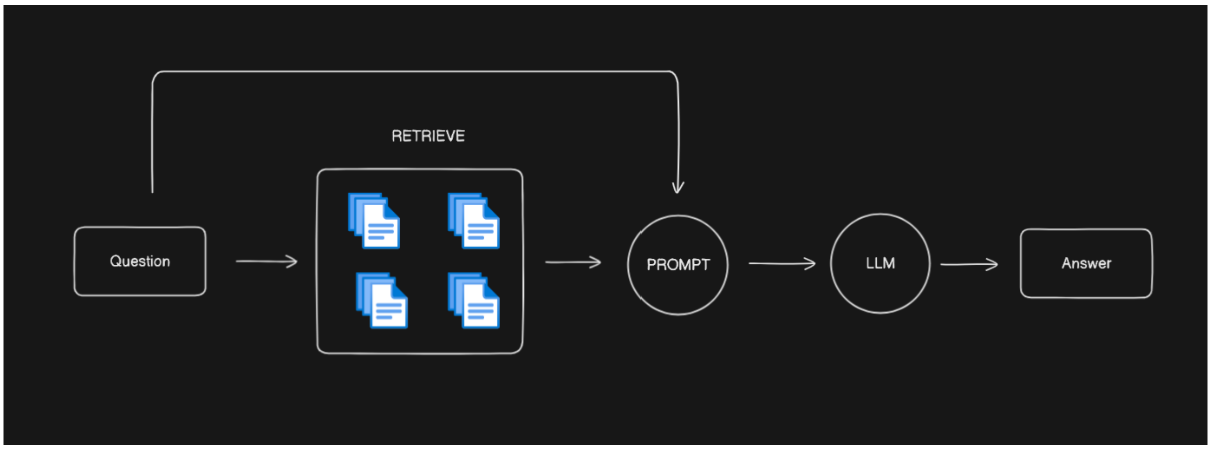

Figure 2: Rag Architecture

Installation & env setup

For this tutorial, we'll be using Langchain. **LangChain** is a framework for developing applications powered by large language models (LLMs).

Create a Virtual Environment

Create a project directory and Navigate to it:

- **For Windows:**

```
mkdir rag
cd rag
python -m venv rag_app
```

- **For Mac/Linux:**

```
mkdir rag
cd rag
python3 -m venv rag_app
```

Activate the Virtual Environment

- **For Windows:**

```
.\rag_app\Scripts\activate
```

- **For Mac/Linux:**

```
source rag_app/bin/activate
```

Create a requirements.txt:

```
langchain
langchain_community
langchain_chroma
langchain-google-genai
beautifulsoup4
langchain-openai
```

Install Required Packages

```
pip install -r requirements.txt
```

To access Google AI models you'll need to create a Google Account, get a Google AI API key

Head to https://ai.google.dev/gemini-api/docs/api-key to generate a Google AI API key. Once you've done this set the GOOGLE_API_KEY environment variable:

For this tutorial, we'll be using Gemini-1.5 Pro model

Create `.env` file in your project directory. Make sure the env names are as per the given format.

```
GOOGLE_API_KEY = "YOUR KEY"
OPENAI_API_KEY = "YOUR KEY"
```

Our dir. structure

```
PROJECT DIR
|
|--- .env
|--- main.py
```

```python
import dotenv
from langchain_google_genai import ChatGoogleGenerativeAI
# from langchain_openai import ChatOpenAI # Uncomment this if you have openai client
dotenv.load_dotenv()
# Instantiate the Google Gemini model (gemini-1.5-pro)
llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")
# llm = ChatOpenAI(model="gpt-4o") # Uncomment this if you have openai client
```

Load

To begin, we'll need to load the blog post content, which can be efficiently handled using Document Loaders. For this task, we'll utilize `WebBaseLoader`, which leverages `urllib` to fetch HTML from web URLs and `BeautifulSoup` to parse it into text. In this case only HTML tags with class "post-content", "post-title", or "post-header" are relevant, so we'll remove all others.

Langchain document loaders support various types of file formats like PDF, CSV, EXCEL, MS Word,

DB, JSON ,etc.

```python
import bs4
from langchain_community.document_loaders import WebBaseLoader

# Load content from a specific web page using WebBaseLoader
loader = WebBaseLoader(
    web_paths=("https://realpython.com/polars-python/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("with-headerlinks", "article-body")
        )
    ),
)

# Load documents from the web
docs = loader.load()
```

Split

LLM has a smaller context window. Even if they have a large context window they will struggle to find the right piece of text. To deal with this we'll split the document. This should help us retrieve only the most relevant bits of the blog post at run time.

In this case, we'll split our documents into chunks of 1000 characters with 200 characters of overlap between chunks. The overlap helps mitigate the possibility of separating a statement from an important context related to it. We use the `RecursiveCharacterTextSplitter`, which will recursively split the document using common separators like new lines until each chunk is the appropriate size. This is the recommended text splitter for generic text use cases.

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Set up a RecursiveCharacterTextSplitter to chunk the text
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)

# Split documents into chunks
splits = text_splitter.split_documents(docs)
```

Store

We can embed and store all of our document splits in a single command using the Chroma vector store and **GoogleGenerativeAIEmbeddings** model.

Embeddings create a vector representation of a piece of text. This is useful because it means we can think about text in the vector space, and do things like semantic search where we look for pieces of text that are most similar in the vector space. We are using google's `embedding-001` model.

```python
from langchain_chroma import Chroma
from langchain_google_genai import GoogleGenerativeAIEmbeddings
# from langchain_openai import OpenAIEmbeddings
# Uncomment this if you have openai client

# Create a vector store with Chroma, using GoogleGenerativeAIEmbeddings
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    # embedding=OpenAIEmbeddings(model="text-embedding-3-large")
    # Uncomment this if you have openai client
)
```

Retrieve

Retrievers are responsible for taking a query and returning relevant documents.

```python
retriever = vectorstore.as_retriever()
```

Generate

Let's put it all together into a chain that takes a question, retrieves relevant documents, constructs a prompt, passes that to a model, and parses the output.

```python
from langchain_core.prompts import ChatPromptTemplate , MessagesPlaceholder
from langchain.chains import create_history_aware_retriever
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
```

```
# The system prompt instructs the model to rephrase
# a user's question based on chat history.

contextualize_q_system_prompt = (
    "Given a chat history and the latest user question "
    "which might reference context in the chat history, "
    "formulate a standalone question which can be understood "
    "without the chat history. Do NOT answer the question, "
    "just reformulate it if needed and otherwise return it as is."
)

# This prompt is part of a chat template that includes
the system instruction and placeholders for chat history and user input.
contextualize_q_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", contextualize_q_system_prompt),
        # System message provides the instruction
        MessagesPlaceholder("chat_history"),
        # Placeholder for the chat history
        ("human", "{input}"),
        # Placeholder for the latest user input
    ]
)

# This function creates a retriever that uses the chat history and the rephrased question.
history_aware_retriever = create_history_aware_retriever(
    llm, retriever, contextualize_q_prompt
)

# System prompt for question-answering based on retrieved context.
#It limits the response length and encourages conciseness.

system_prompt = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer "
    "the question. If you don't know the answer, say that you "
    "don't know. Use three sentences maximum and keep the "
    "answer concise."
    "\n\n"
    "{context}"
)

# This is another chat template for the Q&A chain.
#It takes the system instruction and the chat history, and waits for user input.
```

```
qa_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),  # System message with the answer instruction
        MessagesPlaceholder("chat_history"),  # Placeholder for the chat history
        ("human", "{input}"),  # Placeholder for the latest question from the user
    ]
)

# This chain uses the retriever to fetch relevant information
# and the LLM to answer the question.

question_answer_chain = create_stuff_documents_chain(llm, qa_prompt)

# Combine the history-aware retriever and the Q&A
# chain into a retrieval-augmented generation (RAG) chain.

rag_chain = create_retrieval_chain(history_aware_retriever, question_answer_chain)

# A store to save chat session histories.
store = {}

# Function to get or create a chat message history object for a given session ID.
def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = ChatMessageHistory()  # Create new history if it doesn't exist
    return store[session_id]  # Return the existing history

# Runnable that uses the RAG chain with the stored chat history.
# The chat history is retrieved based on the session ID to maintain
# context across the conversation.

conversational_rag_chain = RunnableWithMessageHistory(
    rag_chain,
    get_session_history,  # Retrieves chat history based on the session
    input_messages_key="input",  # The user input message key
    history_messages_key="chat_history",  # The key for chat history
    output_messages_key="answer",  # The key for storing the output answer
)
```

```python
# Invoking the conversational RAG chain by passing
# the user question and session ID for tracking.
res_answer = conversational_rag_chain.invoke(
    {"input": "What is Lazy Api?"},  # User input question
    config={
        "configurable": {"session_id": "1234"}
        # Session ID to maintain the history context
    },
)["answer"]  # Fetch the generated answer

print(res_answer)  # Fetch the generated answer
```

Output :

" The Lazy API in Polars defers data manipulation until the result is explicitly requested. It represents operations as an execution plan, optimizing performance for large datasets. You can create a LazyFrame using `pl.LazyFrame()` or convert an existing DataFrame with `.lazy()` "

Conclusion

In conclusion, by enabling large language models (LLMs) to integrate and adapt to new data without undergoing retraining, Retrieval-Augmented Generation (RAG) provides an innovative approach to enhance AI applications. RAG bridges in the gap by enabling models to dynamically retrieve and use important external data during runtime, whereas LLMs are often restricted to the data they initially trained on.

RAG provides developers with the tools they want to create applications that are more efficient and responsive by integrating procedures which include data loading, splitting, vector format storing, retrieval. Using the help of this framework, models might react to questions quickly as well as the most recent and relevant data. If you are willing to utilize the OpenAI client, we have added the comments in the code. Uncomment to use it.

```python
import bs4
from langchain_google_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_chroma import Chroma
from langchain_openai import ChatOpenAI
from langchain_openai import OpenAIEmbeddings
from langchain_core.prompts import ChatPromptTemplate , MessagesPlaceholder
from langchain.chains import create_history_aware_retriever
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
import dotenv

dotenv.load_dotenv()

# Instantiate the Google Gemini model (gemini-1.5-pro)
llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")
# llm = ChatOpenAI(model="gpt-4o") # Uncomment this if you have openai client

# Load content from a specific web page using WebBaseLoader
loader = WebBaseLoader(
    web_paths=("https://realpython.com/polars-python/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("with-headerlinks", "article-body")
        )
    ),
)
```

```
# Load documents from the web
docs = loader.load()

# Set up a RecursiveCharacterTextSplitter to chunk the text
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)

# Split documents into chunks
splits = text_splitter.split_documents(docs)

# Create a vector store with Chroma, using GoogleGenerativeAIEmbeddings
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    # embedding=OpenAIEmbeddings(model="text-embedding-3-large")
    # Uncomment this if you have openai client
)

# Set up a retriever from the vector store
retriever = vectorstore.as_retriever()

# The system prompt instructs the model to
# rephrase a user's question based on chat history.

contextualize_q_system_prompt = (
    "Given a chat history and the latest user question "
    "which might reference context in the chat history, "
    "formulate a standalone question which can be understood "
    "without the chat history. Do NOT answer the question, "
    "just reformulate it if needed and otherwise return it as is."
)

# This prompt is part of a chat template that includes the
# system instruction and placeholders for chat history and user input.
contextualize_q_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", contextualize_q_system_prompt),
        # System message provides the instruction
        MessagesPlaceholder("chat_history"),
        # Placeholder for the chat history
        ("human", "{input}"),
        # Placeholder for the latest user input
    ]
)
```

```
# This function creates a retriever that uses the
# chat history and the rephrased question.
history_aware_retriever = create_history_aware_retriever(
    llm, retriever, contextualize_q_prompt
)

# System prompt for question-answering based on retrieved context.
# It limits the response length and encourages conciseness.

system_prompt = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer "
    "the question. If you don't know the answer, say that you "
    "don't know. Use three sentences maximum and keep the "
    "answer concise."
    "\n\n"
    "{context}"
)

# This is another chat template for the Q&A chain.
#  It takes the system instruction and the chat history, and waits for user input.

qa_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),
        # System message with the answer instruction
        MessagesPlaceholder("chat_history"),
        # Placeholer for the chat history
        ("human", "{input}"),
        # Placeholder for the latest question from the user
    ]
)

# This chain uses the retriever to fetch
# relevant information and the LLM to answer the question.
question_answer_chain = create_stuff_documents_chain(llm, qa_prompt)

# Combine the history-aware retriever
# and the Q&A chain into a retrieval-augmented generation (RAG) chain.
rag_chain = create_retrieval_chain(history_aware_retriever, question_answer_chain)

# A store to save chat session histories.
store = {}
```

```python
# Function to get or create a chat message
#  history object for a given session ID.
def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = ChatMessageHistory()  # Create new history if it doesn't exist
    return store[session_id]  # Return the existing history

# Runnable that uses the RAG chain with the stored chat history.
# The chat history is retrieved based on the
# session ID to maintain context across the conversation.
conversational_rag_chain = RunnableWithMessageHistory(
    rag_chain,
    get_session_history,  # Retrieves chat history based on the session
    input_messages_key="input",  # The user input message key
    history_messages_key="chat_history",  # The key for chat history
    output_messages_key="answer",  # The key for storing the output answer
)

# Invoking the conversational RAG chain
# by passing the user question and session ID for tracking.
res_answer = conversational_rag_chain.invoke(
    {"input": "What is Lazy Api?"},
     # User input question
    config={
        "configurable": {"session_id": "1234"}
        # Session ID to maintain the history context
    },
)["answer"]
# Fetch the generated answer

print(res_answer)
# Fetch the generated answer
```