Revolutionizing YouTube Video Summaries and Q&A with LangChain, Llama 3.2, and Gradio: A Game-Changer for Students, Creators, and Professionals

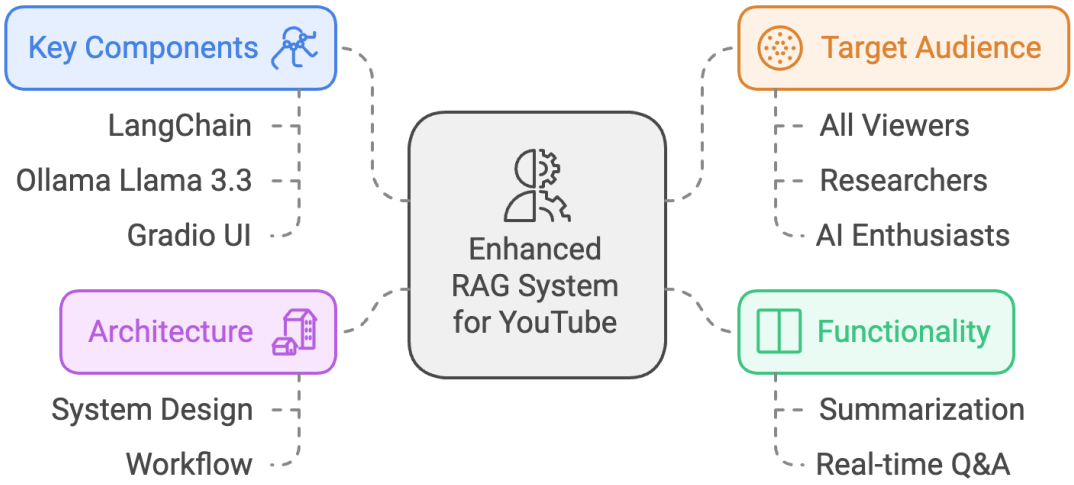


Image created using napkin.ai

Introduction

How many times have you sat through a really long YouTube video just to find the answer to one specific question? Or watched an entire video for just a couple of key words from the author? Whether you're a student trying to extract valuable knowledge from a lecture, a content creator analyzing trends, or a professional seeking quick answers, the process can be tedious and time-consuming.

Enter the enhanced Retrieval-Augmented Generation (RAG) system, powered by cutting-edge AI tools like LangChain, Llama 3.2, and Gradio. This system transforms how you interact with YouTube videos by offering two key functionalities: **summarization** and **interactive Q&A**. Not only can it distill lengthy videos into concise summaries, but it also allows you to ask specific questions and get precise, contextually relevant answers in real time.

In this blog, we'll explore how this innovative AI solution caters to the growing demand for smarter tools, from helping students tackle dense educational content to enabling creators and professionals to work more efficiently. Along the way, you'll discover how LangChain structures workflows, how Llama 3.2 powers conversational capabilities, and how Gradio delivers a seamless user interface for anyone to use. Whether you're an AI enthusiast, a tech-savvy student, or someone curious about automating tedious tasks, this post will guide you step by step through building a tool that redefines the way we engage with video content.

Github Repo: The complete code for this project is available on GitHub. Follow along with this blog to understand the implementation step by step.

Prerequisites to Run the Project

Before we start with the setup and code, let's go through the prerequisites to make sure you have everything you need to run this project smoothly.

1. **Python 3.x**: Ensure that Python 3 or above is installed on your system.

2. **Install Required Libraries**

The project requires several Python libraries that can be installed via pip. You can install them by running the following command:

pip install -r requirements.txt

Some of the key libraries include:

- **gradio**: Used for creating interactive user interfaces to allow users to input data (such as a YouTube URL) and interact with the system (e.g., asking questions or receiving summaries).
- **langchain**: A framework for building applications that integrate with large language models (LLMs), enabling composability and easy creation of complex workflows.
- **langchain_huggingface**: Provides integration between LangChain and Hugging Face models, enabling the use of Hugging Face embeddings within LangChain workflows.
- **langchain_community**: A collection of community-contributed tools and extensions for LangChain, including document loaders and utilities for enhancing LLM-based applications.
- **langchain_ollama**: Integrates Ollama's LLaMA models (such as LLaMA 3.2) into LangChain, enabling conversational AI and natural language processing within the system.
- **faiss-cpu**: A library for efficient similarity search and clustering of dense vectors, providing fast retrieval of relevant video segments or answers from large datasets of embeddings.

3. Ollama's LLaMA 3.2 Model

Since the system leverages **Ollama's LLaMA 3.2 model**, you will need to have **Ollama** installed and running on your local machine. You can download and set up Ollama from their official site. After installation, ensure that the LLaMA 3.2 model is available.

Step-by-Step Code Walkthrough

This code implements an interactive YouTube video Q&A system using a combination of tools: Gradio for the user interface, LangChain for managing the retrieval and processing of information, FAISS for efficient vector storage, and Ollama LLaMA for conversational capabilities. Here's how it works, step by step:

1. Importing Libraries

```
import gradio as gr
from langchain_huggingface import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.document_loaders import YoutubeLoader
from langchain_ollama import ChatOllama
```

- **Gradio** is used to build the interactive UI.
- **LangChain** and **HuggingFaceEmbeddings** provide tools for processing, embedding, and storing video data.
- **FAISS** is used for efficient similarity-based document retrieval.

- **RecursiveCharacterTextSplitter** helps in splitting the text into manageable chunks.
- **ChatOllama** interfaces with the LLaMA 3.2 model for conversational capabilities.
- **YoutubeLoader**: Extracting text from YouTube video captions.

2. Processing YouTube URL

```
def process_youtube_url(youtube_url):
  loader = YoutubeLoader.from_youtube_url(youtube_url, add_video_info=False)
  documents = loader.load()

  splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=256,
    separators=["\n\n", "\n", " "]
  )

  split_docs = splitter.split_documents(documents)

  embedding_model = 'sentence-transformers/all-MiniLM-L6-v2'
  embeddings = HuggingFaceEmbeddings(model_name=embedding_model)

  db = FAISS.from_documents(split_docs, embeddings)
  return db
```

process_youtube_url(youtube_url) function takes a YouTube URL as input and extracts the video transcript using the YoutubeLoader.from_youtube_url method. It then processes the transcript by splitting it into smaller, manageable chunks using the RecursiveCharacterTextSplitter, which divides the text into segments of 512 characters. The resulting chunks are embedded using the HuggingFaceEmbeddings model (specifically, 'all-MiniLM-L6-v2'), which transforms the text into numerical representations suitable for machine learning tasks. These embeddings are then stored in a FAISS vector database, enabling efficient querying and retrieval of information related to the video.

Note: In order to modify the code to use captions in a different language, refer to this documentation. By default, the code reads English captions.

3. Submitting YouTube URL

```
def submit_url(youtube_url):
  global global_vector_db
  try:
    global_vector_db = process_youtube_url(youtube_url)
    default_question = "Summarize this video."
    chat_history = []

    summary, _ = answer_question(default_question, chat_history)

    status_message = "Video indexed successfully ✅! You can now ask questions about the video in the chatbot."
  except Exception as e:
```

```
        status_message = f"❌ Error processing the video: {str(e)}"
        summary = ""

    return status_message, summary
```

The submit_url function processes the provided YouTube URL by calling the process_youtube_url function to create a vector store from the video's transcript. After processing, it uses a default question ("Summarize this video") to initiate a retrieval-based question-answering process. If successful, it returns a status message indicating the video has been indexed successfully, along with a summary. In case of an error, the function catches exceptions and returns an error message, ensuring a smooth user experience.

4. Answering Questions Based on the Video

```
def answer_question(question, chat_history):
    global global_vector_db
    if global_vector_db is None:
        return "Please process a YouTube video URL first.", chat_history

    try:
        local_llm = 'llama3.2'
        llama3 = ChatOllama(model=local_llm, temperature=0)

        retriever = global_vector_db.as_retriever(search_kwargs={"k": 5})

        system_prompt = (
            "You are a video assistant tasked with answering questions based on the provided YouTube video context. "
            "Use the given context given by the video author to provide accurate, concise answers in three sentences. "
            "If the context does not contain the answer, say you are not sure "
            "Context: {context}"
        )

        prompt = ChatPromptTemplate.from_messages(
            [
                ("system", system_prompt),
                ("human", "{input}"),
            ]
        )

        question_answer_chain = create_stuff_documents_chain(llama3, prompt)
        chain = create_retrieval_chain(retriever, question_answer_chain)

        response = chain.invoke({"input": question})

        if "answer" not in response:
            raise ValueError("Response does not contain an 'answer' key.")
```

```
        chat_history.append((question, response['answer']))

        return response['answer'], chat_history
    except Exception as e:
        error_message = f"Error: {str(e)}"
        chat_history.append((question, error_message))
        return error_message, chat_history
```

The answer_question function answers user questions based on the video's indexed content. It first checks if the vector database is available; if not, it prompts the user to process a video. Then, it uses the ChatOllama model and a retrieval chain to search for relevant information in the vector database and generate a concise answer. The question and corresponding answer are appended to the chat_history, allowing for a continuous, interactive conversation. If an error occurs, the function handles it by appending the error message to the chat history.

5. Asking Questions via Gradio

```
def ask_question(question, chat_history):
    response, updated_chat_history = answer_question(question, chat_history)
    return updated_chat_history, updated_chat_history
```

The ask_question function interacts with the user to answer their query. It calls the answer_question function, passing the question and the current chat_history to retrieve an answer. The updated chat history, which includes the user's question and the assistant's response, is returned, allowing the conversation to continue smoothly. This function ensures a seamless Q&A interaction with the chatbot.

6. Creating Gradio Interface

```
def create_gradio_interface():
    with gr.Blocks() as demo:
        gr.Markdown("<h1 style='text-align: center; color: #4A90E2;'>YouTube Video Q&A</h1>")
        gr.Markdown("<p style='text-align: center;'>Enter a YouTube video URL to extract information and ask questions about it.</p>")

        with gr.Row():
            with gr.Column(scale=1):
                youtube_url = gr.Textbox(label="YouTube Video URL", placeholder="Enter the YouTube video URL here...", lines=1)
                submit_btn = gr.Button("Submit URL", variant="primary")
                status_info = gr.Textbox(label="Status Info", placeholder="Indexing status will appear here...", interactive=False, lines=2)
                summary_box = gr.Textbox(label="Video Summary", placeholder="Summary will appear here...", interactive=False, lines=6)

                submit_btn.click(fn=submit_url, inputs=youtube_url, outputs=[status_info, summary_box])

            with gr.Column(scale=1):
                chat_history = gr.Chatbot()
                question = gr.Textbox(label="Your Question", placeholder="Ask a question
```

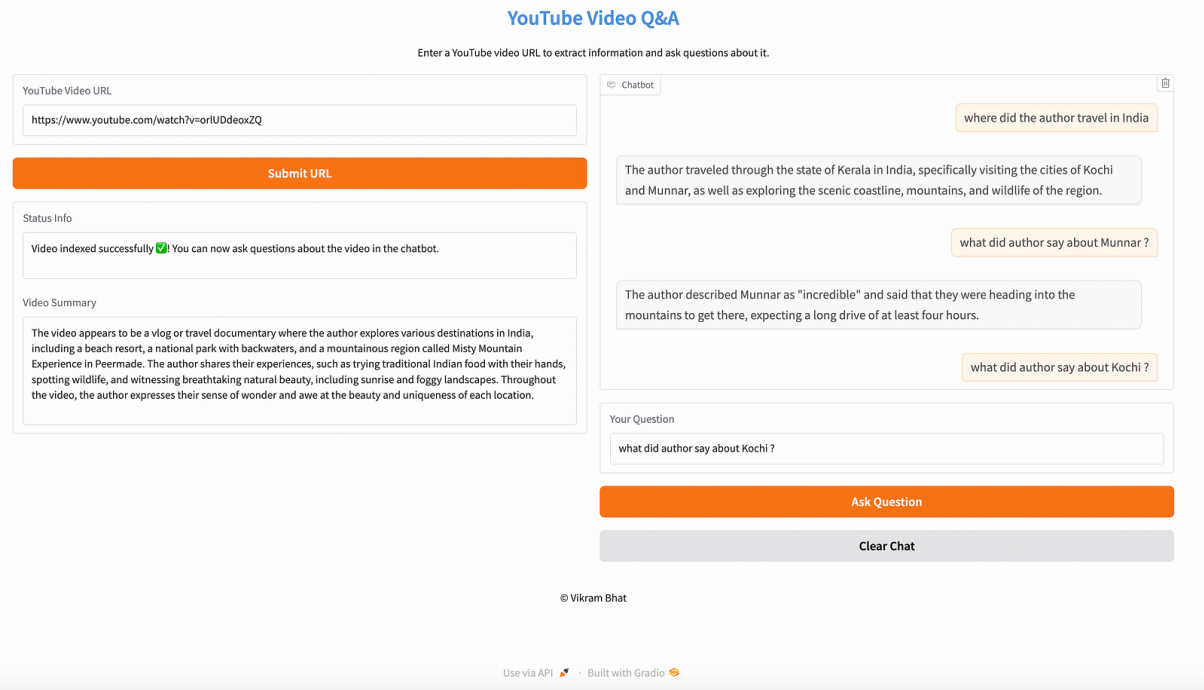
```
about the video...", lines=1)
            ask_btn = gr.Button("Ask Question", variant="primary")
            clear_btn = gr.Button("Clear Chat", variant="secondary")
            state = gr.State([])

            ask_btn.click(fn=ask_question, inputs=[question, state], outputs=[chat_history, state])
            clear_btn.click(fn=lambda: ([], []), inputs=[], outputs=[chat_history, state])

        gr.Markdown("<footer style='text-align: center; margin-top: 20px;'>© Vikram Bhat</footer>")

    return demo
```

create_gradio_interface() function constructs the Gradio interface, providing a user-friendly layout for interaction with the system. It includes input fields for users to enter a YouTube URL and ask questions about the video. The interface features a submission button, status information box, and a summary box to display the video summary. Users can also ask questions, view responses, and clear the chat history as needed. Additionally, the interface includes a footer for supplementary information. This function defines the overall structure and interaction flow of the application, making it accessible and intuitive for users.

7. Launching the Gradio Interface

```
interface = create_gradio_interface()
interface.launch(share=True)
```

interface.launch starts the Gradio interface, allowing users to interact with the system by entering a YouTube URL, asking questions, and receiving answers.



Gradio Interface for YouTube Video Q&A

Conclusion

This blog demonstrated how to build an enhanced Retrieval-Augmented Generation (RAG) system that can summarize and interact with YouTube videos using advanced tools like LangChain, Ollama Llama 3.2, and Gradio UI. We explored how the system processes YouTube video captions (transcripts) to create a vector database, enabling real-time Q&A and summarization. By leveraging LangChain's modular framework, the Ollama Llama 3.2 model's conversational abilities, and Gradio's intuitive interface, we've created a seamless solution for users to quickly extract key information from lengthy videos. This system represents a significant leap forward in video content interaction, allowing users to gain insights and answers with minimal effort. Whether you are a developer looking to integrate similar functionalities or simply someone who values efficient video summarization, this approach offers an innovative way to engage with video content.

Explore More:

Building a Smart Trip Planner: How I Combined LLMs and Gradio for Interactive Itineraries

*AI-RouteRover*ai.gopubby.com

Multi-Agent System for Research Summarization and Reporting with Crew AI

*Leveraging Crew AI and Gradio for Efficient Multi-Agent Research Summarization and Automated Reporting*ai.gopubby.com

You can also connect with me on LinkedIn for more updates and professional insights.