**EduAI- An AI Powered Chatbot for YouTube Transcripts: A RAG-Based Approach**

**Project Overview**

This project implements a Retrieval-Augmented Generation (RAG) chatbot designed to process YouTube video transcripts and allow users to query the content. By leveraging OpenAI’s GPT-4, ChromaDB, and LangChain, the chatbot efficiently retrieves the most relevant transcript sections and generates human-like responses.

The system follows these key steps:

1. Transcribing YouTube videos using Whisper.
2. Splitting transcripts into manageable chunks for efficient processing.
3. Generating embeddings to convert text into searchable vectors.
4. Storing embeddings in ChromaDB to enable vector-based retrieval.
5. Retrieving relevant transcript segments based on user queries.
6. Generating AI-powered responses by combining retrieved data with GPT-4.
7. Deploying the chatbot using Gradio for user-friendly interaction.

**Objectives**

The main objectives of this project were: Develop an AI chatbot capable of answering questions about YouTube videos using transcript data.  
Implement a RAG approach for context-aware responses.  
Enable fast & accurate transcript searches using vector embeddings in ChromaDB.

Create a user-friendly chatbot interface using Gradio.  
Ensure adaptability by supporting multiple YouTube videos.

**Implementation Steps**

Step 1: Installing Required Libraries

Before implementing the chatbot, I installed essential libraries for transcription, embeddings, retrieval, and chatbot integration.

!pip install langchain openai chromadb gradio whisper

Step 2: Transcribing YouTube Video Audio Using Whisper

Since some YouTube videos lack captions, Whisper was used to generate transcripts from the audio.

import whisper

whisper\_model = whisper.load\_model("base")

def transcribe\_audio(audio\_path):

result = whisper\_model.transcribe(audio\_path)

return result["text"]

Step 3: Splitting the Transcript into Manageable Segments

To make the transcript searchable, it was split into 100-word chunks.

import re

def split\_text\_into\_segments(text, max\_words=100):

sentences = re.split(r'\. ', text)

segments, current\_segment = [], ""

for sentence in sentences:

if len(current\_segment.split()) + len(sentence.split()) <= max\_words:

current\_segment += " " + sentence

else:

segments.append(current\_segment.strip())

current\_segment = sentence

if current\_segment:

segments.append(current\_segment.strip())

return segments

Step 4: Generating Text Embeddings for Vector Search

To allow fast and accurate searches, transcript segments were converted into vector embeddings using SentenceTransformers.

from sentence\_transformers import SentenceTransformer

embedding\_model = SentenceTransformer("all-MiniLM-L6-v2")

def generate\_embeddings(text\_segments):

return [embedding\_model.encode(segment).tolist() for segment in text\_segments]

Step 5: Storing Embeddings in ChromaDB

ChromaDB was used as the vector database for storing and retrieving transcript embeddings.

import chromadb

chroma\_client = chromadb.PersistentClient(path="/mnt/data/chroma\_db")

collection = chroma\_client.get\_or\_create\_collection(name="youtube\_transcripts")

def store\_in\_chromadb(segments):

for i, segment in enumerate(segments):

embedding = embedding\_model.encode(segment).tolist()

collection.add(

ids=[str(i)],

embeddings=[embedding],

metadatas=[{"text": segment}]

)

Step 6: Retrieving the Most Relevant Transcript Segments

The retrieval function searches for the most relevant transcript chunks based on user queries.

def retrieve\_relevant\_context(query, top\_k=3):

query\_embedding = embedding\_model.encode([query]).tolist()

results = collection.query(query\_embeddings=query\_embedding, n\_results=top\_k)

return [doc["text"] for doc in results["metadatas"][0]]

Step 7: Generating AI Responses Using Retrieval-Augmented Generation (RAG)

Once relevant transcript sections are retrieved, they are combined with a LLM prompt for response generation.

from langchain.chat\_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", temperature=0.7, openai\_api\_key="YOUR\_OPENAI\_API\_KEY")

def generate\_rag\_response(query):

retrieved\_context = retrieve\_relevant\_context(query)

context\_text = "\n".join(retrieved\_context)

prompt = f"""

You are an AI assistant using YouTube transcript data.

Answer based on the provided transcript.

Context:

{context\_text}

Question: {query}

Answer:

"""

response = llm.predict(prompt)

return response

Step 8: Deploying as an AI Chatbot Using Gradio

To make the chatbot accessible, I deployed it using Gradio.

import gradio as gr

def chatbot\_rag(query):

return generate\_rag\_response(query)

iface = gr.Interface(

fn=chatbot\_rag,

inputs="text",

outputs="text",

title="YouTube Transcript AI Chatbot",

description="Ask a question about the video, and get AI-generated responses based on the transcript."

)

iface.launch()

**Key Challenges and Solutions**

| Challenge | Solution |
| --- | --- |
| Handling large transcripts | Split text into smaller 100-word chunks |
| Slow transcript retrieval | Used vector embeddings & ChromaDB for faster searches |
| Ensuring relevant AI responses | Implemented Retrieval-Augmented Generation (RAG) |
| User-friendly interface | Integrated Gradio for easy access |

**Future Improvements**

Hybrid search: Combining keyword search + vector embeddings  
Optimized model selection: Use Llama-2 instead of GPT-4 to cut costs

Multi-video support: Allow querying across multiple YouTube videos

**Conclusion**

This project successfully implements an AI chatbot that allows users to query YouTube videos using transcript data. By leveraging RAG, OpenAI GPT-4, LangChain, and ChromaDB, the chatbot efficiently retrieves and generates responses. The integration of Gradio makes it user-friendly, while vector-based search ensures accurate and fast results.

With additional improvements like hybrid search, Llama-2 integration, and multi-video support, this project can be scaled into a full-fledged AI-powered research assistant for video content.