AI-Powered YouTube Summarizer, QA Tool with RAG, LangChain, FAISS

Estimated time needed: 1 hour

In this project, you'll build a question-answering (QA) tool capable of extracting and summarizing information from YouTube videos. Leveraging LangChain and a large language model (LLM), the tool will answer specific questions based on a video's transcript. You'll work with components like video transcript loaders, text processors, embedding models, vector databases, and retrievers, while using Streamlit for a user-friendly interface.

With the explosion of online video content, manually searching through lengthy footage is inefficient. This project automates that process, transforming dense transcripts into concise summaries and enabling precise video segment identification using Facebook AI Similarity Search (FAISS). By the end of the project, you'll have developed a powerful system that streamlines how we interact with multimedia data, making video content more accessible and insightful.

Setup

Setting up a virtual environment

First, let's create a virtual environment. A virtual environment allows you to manage dependencies for different projects separately, avoiding conflicts between package versions.

To open a terminal, go to the top menu and click **Terminal > New Terminal**.

In the terminal of your Cloud IDE, ensure that you are in the path `/home/project`, then run the following commands to create a Python virtual environment.

```
pip install virtualenv
virtualenv my_env # create a virtual environment named my_env
source my_env/bin/activate # activate my_env
```

Installing prerequisite libraries

To ensure seamless execution of your scripts, and considering that certain functions within these scripts rely on external libraries, it's essential to install some prerequisite libraries before you begin. For this project, the key libraries you'll need are:

- **`youtube-transcript-api`** for extracting transcripts from YouTube videos.
- **`faiss-cpu`** for efficient similarity search.
- **`langchain`** and **`langchain-community`** for text processing and language models.
- **`ibm-watsonx-ai`** and **`langchain-ibm`** for integrating IBM Watson services.
- **`streamlit`** for building the web application interface.

Here's how to install these packages (from your terminal):

```
# installing necessary packages in my_env
pip install youtube-transcript-api==1.2.1
pip install faiss-cpu==1.8.0
pip install langchain==0.2.6 | tail -n 1
pip install langchain-community==0.2.6 | tail -n 1
pip install ibm-watsonx-ai==1.0.10 | tail -n 1
pip install langchain_ibm==0.1.8 | tail -n 1
pip install gradio==4.44.1 | tail -n 1
# Uninstall any existing huggingface_hub first
python3.11 -m pip uninstall -y huggingface_hub
# Install a version compatible with gradio 4.44.1
python3.11 -m pip install huggingface_hub==0.16.4
```

The environment is now ready to create the application.

Construct the YouTube bot

It's time to construct the YouTube bot!

Let's start off by creating a new Python file that will store your bot. Click on the button below to create a new Python file, and call it `ytbot.py`. If, for whatever reason, the button does not work, you can create the new file by clicking **File > New Text File**. Be sure to save the file as `ytbot.py`.

Open **ytbot.py** in IDE

In the following sections, you will populate `ytbot.py` with your bot.

Import necessary libraries

Inside `ytbot.py`, import the following libraries from `streamlit`, `youtube_transcript_api`, `ibm_watsonx_ai`, `langchain_ibm`, `langchain`, and `langchain_community`. The imported classes are required for initializing models with the correct credentials, splitting text, initializing a vector store, loading YouTube transcripts, generating a question-answer retriever, and using Gradio.

```
# Import necessary libraries for the YouTube bot
import gradio as gr
import re  #For extracting video id
from youtube_transcript_api import YouTubeTranscriptApi  # For extracting transcripts from YouTube videos
from langchain.text_splitter import RecursiveCharacterTextSplitter  # For splitting text into manageable segments
from ibm_watsonx_ai.foundation_models.utils.enums import ModelTypes  # For specifying model types
from ibm_watsonx_ai import APIClient, Credentials  # For API client and credentials management
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams  # For managing model parameters
from ibm_watsonx_ai.foundation_models.utils.enums import DecodingMethods  # For defining decoding methods
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings  # For interacting with IBM's LLM and embeddings
from ibm_watsonx_ai.foundation_models.utils import get_embedding_model_specs  # For retrieving model specifications
from ibm_watsonx_ai.foundation_models.utils.enums import EmbeddingTypes  # For specifying types of embeddings
from langchain_community.vectorstores import FAISS  # For efficient vector storage and similarity search
from langchain.chains import LLMChain  # For creating chains of operations with LLMs
from langchain.prompts import PromptTemplate  # For defining prompt templates
```

Extracting YouTube transcripts

Video ID extraction

YouTube video URLs typically follow this format: `https://www.youtube.com/watch?v=VIDEO_ID`

The `VIDEO_ID` is a unique 11-character string that identifies the video. To extract this ID, we'll use a regular expression that captures this 11-character string from the URL.

Define a function to extract the video ID from the provided YouTube URL.

```
def get_video_id(url):
    # Regex pattern to match YouTube video URLs
    pattern = r'https:\/\/www\.youtube\.com\/watch\?v=([a-zA-Z0-9_-]{11})'
    match = re.search(pattern, url)
    return match.group(1) if match else None
```

Usage details

The `get_video_id()` function is designed to extract the unique `VIDEO_ID` from a YouTube URL.

Function call

```
url = "https://www.youtube.com/watch?v=dQw4w9WgXcQ"
video_id = get_video_id(url)
print(video_id)  # Output: dQw4w9WgXcQ
```

Expected output

If the URL matches the YouTube format: The function returns the extracted 11-character VIDEO_ID (for example, `dQw4w9WgXcQ`).
If the URL does not match: The function returns `None`.

Fetching transcripts from YouTube

The `YouTubeTranscriptApi` allows us to retrieve transcripts (subtitles) for a given video. This function first extracts the video ID from the YouTube URL, then fetches the transcripts available for that video. Transcripts can be either automatically generated or manually provided by the video uploader.

Here's the function to fetch the transcript:

```
def get_transcript(url):
    # Extracts the video ID from the URL
    video_id = get_video_id(url)

    # Create a YouTubeTranscriptApi() object
    ytt_api = YouTubeTranscriptApi()

    # Fetch the list of available transcripts for the given YouTube video
    transcripts = ytt_api.list(video_id)

    transcript = ""
    for t in transcripts:
        # Check if the transcript's language is English
        if t.language_code == 'en':
            if t.is_generated:
                # If no transcript has been set yet, use the auto-generated one
                if len(transcript) == 0:
                    transcript = t.fetch()
            else:
                # If a manually created transcript is found, use it (overrides auto-generated)
                transcript = t.fetch()
                break  # Prioritize the manually created transcript, exit the loop

    return transcript if transcript else None
```

Example usage

Following is an example of how to call `get_transcript()` using a sample YouTube video URL:

```
# Sample YouTube URL
url = "https://www.youtube.com/watch?v=dQw4w9WgXcQ"
# Fetching the transcript
transcript = get_transcript(url)
# Output the fetched transcript
print(transcript)
```

Example output format


```
[
    {
        "text": "We're no strangers to love.",
        "start": 0.0,
        "duration": 3.5
    },
    {
        "text": "You know the rules and so do I.",
        "start": 3.5,
        "duration": 4.0
    },
    {
        "text": "A full commitment's what I'm thinking of.",
        "start": 7.5,
        "duration": 4.0
    }
]
```


If no transcript is available or if the video ID is invalid, the function will return an empty result or an appropriate error, depending on the availability of the transcript for the provided video ID.

Processing the transcript

When we fetch the transcript, it often comes in a structured format. Each entry in the transcript is typically represented as a dictionary with the following structure:

```
{
    "text": "Transcript text here",
    "start": 0.0,
    "duration": 3.0
}
```

- `text`: The spoken text from the video.
- `start`: The time (in seconds) when the text starts in the video.
- `duration`: The duration (in seconds) for which the text is displayed.

This function transforms the fetched transcript into a more readable format by extracting the text and its corresponding start time.

```
def process(transcript):
    # Initialize an empty string to hold the formatted transcript
    txt = ""

    # Loop through each entry in the transcript
    for i in transcript:
        try:
            # Append the text and its start time to the output string
            txt += f"Text: {i.text} Start: {i.start}\n"
        except KeyError:
            # If there is an issue accessing 'text' or 'start', skip this entry
            pass

    # Return the processed transcript as a single string
    return txt
```

Example usage

Following is an example of how to call `process()` using a fetched transcript:

```
# Sample transcript list
transcript = [
    {
        "text": "We're no strangers to love.",
        "start": 0.0,
        "duration": 3.5
    },
    {
        "text": "You know the rules and so do I.",
        "start": 3.5,
        "duration": 4.0
    },
    {
        "text": "A full commitment's what I'm thinking of.",
        "start": 7.5,
        "duration": 4.0
    }
]
# Processing the transcript
formatted_transcript = process(transcript)
# Output the processed transcript
print(formatted_transcript)
```

Expected output

The `process()` function returns a formatted string that contains the text and start time for each entry in the transcript. The output format will look like this:

```
Text: We're no strangers to love. Start: 0.0
Text: You know the rules and so do I. Start: 3.5
Text: A full commitment's what I'm thinking of. Start: 7.5
```

Chunking the transcript

The `RecursiveCharacterTextSplitter` from LangChain helps split long transcripts into smaller, more manageable chunks for easier processing. This function takes a processed transcript and breaks it down into specified chunk sizes, with some overlap between chunks to ensure context is preserved across segments. This is useful when handling large texts that need to be processed by models or other tools.

Here's the function to chunk the transcript:

```
def chunk_transcript(processed_transcript, chunk_size=200, chunk_overlap=20):
    # Initialize the RecursiveCharacterTextSplitter with specified chunk size and overlap
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap
    )
    # Split the transcript into chunks
    chunks = text_splitter.split_text(processed_transcript)
    return chunks
```

Explanation

- **Chunking the transcript**: This function splits a large transcript into smaller chunks using a specified `chunk_size` (default is 200 characters) and an optional `chunk_overlap` (default is 20 characters) to maintain context across the chunks.
- **RecursiveCharacterTextSplitter**: Ensures the text is split intelligently, without breaking up sentences or paragraphs unnaturally, making it ideal for processing large documents or transcripts.

Example usage

Below is an example of how to call `chunk_transcript()` using a processed transcript:

```
# Sample processed transcript string
processed_transcript = """Text: We're no strangers to love. Start: 0.0
Text: You know the rules and so do I. Start: 3.5
Text: A full commitment's what I'm thinking of. Start: 7.5"""
# Chunking the transcript
chunks = chunk_transcript(processed_transcript)
# Output the chunks
print(chunks)
```

Expected output

The `chunk_transcript()` function returns a list of strings, each representing a chunk of the original processed transcript. For example:

```
[
    "Text: We're no strangers to love. Start: 0.0\nText: You know the rules and so do I. Start: 3.5",
    "Text: You know the rules and so do I. Start: 3.5\nText: A full commitment's what I'm thinking of. Start: 7.5"
]
```

In this output, each chunk contains overlapping segments of the transcript to maintain context, which is useful when processing each chunk with models or tools that require shorter inputs.

Setting up a watsonx model

Credentials setup

Set up the necessary credentials to access IBM Watson services. This function initializes the required credentials, client, and project details for interacting with the watsonx model.

```
def setup_credentials():
    # Define the model ID for the WatsonX model being used
    model_id = "meta-llama/llama-3-3-70b-instruct"

    # Set up the credentials by specifying the URL for IBM Watson services
    credentials = Credentials(url="https://us-south.ml.cloud.ibm.com")

    # Create an API client using the credentials
    client = APIClient(credentials)

    # Define the project ID associated with the WatsonX platform
    project_id = "skills-network"

    # Return the model ID, credentials, client, and project ID for later use
    return model_id, credentials, client, project_id
```

Defining parameters

Configure the parameters for the watsonx model. This function sets up various generation parameters, such as the decoding method and token limits, to customize the behavior of the model during text generation.

```
def define_parameters():
    # Return a dictionary containing the parameters for the WatsonX model
    return {
        # Set the decoding method to GREEDY for generating text
        GenParams.DECODING_METHOD: DecodingMethods.GREEDY,

        # Specify the maximum number of new tokens to generate
        GenParams.MAX_NEW_TOKENS: 900,
    }
```

Initializing the watsonx LLM

Instantiate the watsonx LLM for summarization and Q&A tasks. This function initializes the watsonx language model by providing the necessary model ID, credentials, project ID, and parameters for its configuration.

```
def initialize_watsonx_llm(model_id, credentials, project_id, parameters):
    # Create and return an instance of the WatsonxLLM with the specified configuration
    return WatsonxLLM(
        model_id=model_id,          # Set the model ID for the LLM
        url=credentials.get("url"),      # Retrieve the service URL from credentials
        project_id=project_id,            # Set the project ID for accessing resources
        params=parameters                  # Pass the parameters for model behavior
    )
```

Embedding and similarity search

This section covers the process of embedding transcript chunks and implementing similarity search using FAISS.

Embedding the transcript chunks

We use the `IBM SLATE-30M (ENG)` model to generate embeddings for the transcript chunks. This function initializes the embedding model, which converts the textual data into numerical vectors that can be utilized for various natural language processing tasks, such as similarity calculations, clustering, and machine learning model training.

Embeddings are dense vector representations of text, where similar pieces of text are mapped to nearby points in the vector space. This allows models to understand the semantic relationships between words and phrases, making embeddings crucial for tasks like information retrieval, clustering, and classification.

The `SLATE-30M` model is specifically designed for English language embeddings, providing high-quality representations that capture the semantic meaning of the input text.

```
def setup_embedding_model(credentials, project_id):
    # Create and return an instance of WatsonxEmbeddings with the specified configuration
    return WatsonxEmbeddings(
        model_id='ibm/slate-30m-english-rtrvr-v2',  # Set the model ID for the SLATE-30M embedding model
        url=credentials["url"],                          # Retrieve the service URL from the provided credentials
        project_id=project_id                             # Set the project ID for accessing resources in the Watson environment
    )
```

Implementing FAISS for similarity search

FAISS (Facebook AI Similarity Search) is a library designed for efficient similarity search and clustering of dense vectors, enabling rapid retrieval of nearest neighbors in high-dimensional spaces. This function creates a FAISS index from a list of text chunks using the specified embedding model. By converting text chunks into embeddings and indexing them with FAISS, we can quickly find the most similar chunks based on cosine similarity or other distance metrics. This is particularly useful in applications such as information retrieval, recommendation systems, and natural language understanding.

The function takes two parameters:

- `chunks`: A list containing the text chunks that have been processed and are ready for indexing. These chunks typically represent segments of text that need to be compared or searched.
- `embedding_model`: The model used to generate embeddings for the text chunks. This model transforms the chunks into numerical vectors that represent their semantic meaning.

```
def create_faiss_index(chunks, embedding_model):
    """
    Create a FAISS index from text chunks using the specified embedding model.

    :param chunks: List of text chunks
    :param embedding_model: The embedding model to use
    :return: FAISS index
    """
    # Use the FAISS library to create an index from the provided text chunks
    return FAISS.from_texts(chunks, embedding_model)
```

Performing similarity search

In this section, we will search for specific queries within the embedded transcript using the FAISS index. The function takes a query and finds the top `k` most similar text chunks from the indexed embeddings. This is useful for retrieving relevant information based on user queries or for identifying related content within a larger dataset.

The function takes the following parameters:

- `faiss_index`: The FAISS index created from the embedded transcript chunks. This index allows for efficient similarity searches based on vector representations.
- `query`: The text input for which we want to find similar chunks. This could be a user question or a topic of interest.
- `k`: An optional parameter that specifies the number of similar results to return (default is 3).

```
def perform_similarity_search(faiss_index, query, k=3):
    """
    Search for specific queries within the embedded transcript using the FAISS index.

    :param faiss_index: The FAISS index containing embedded text chunks
    :param query: The text input for the similarity search
    :param k: The number of similar results to return (default is 3)
    :return: List of similar results
    """
    # Perform the similarity search using the FAISS index
    results = faiss_index.similarity_search(query, k=k)
    return results
```

Summarizing the transcript

Define the prompt template

In this section, we define a prompt template for the language model to summarize a YouTube video transcript. The prompt serves as a structured instruction that guides the model in generating a coherent summary. The template includes placeholders for dynamic content, specifically the transcript of the video.

The function returns a `PromptTemplate` object configured to accept the transcript as an input variable. This ensures that when the prompt is utilized, it can seamlessly incorporate the specific transcript text that needs summarization. The summary generated by the model should focus on the key points, omitting any timestamps present in the original transcript.

```
def create_summary_prompt():
    """
    Create a PromptTemplate for summarizing a YouTube video transcript.

    :return: PromptTemplate object
    """
    # Define the template for the summary prompt
    template = """
    <|begin_of_text|><|start_header_id|>system<|end_header_id|>
    You are an AI assistant tasked with summarizing YouTube video transcripts. Provide concise, informative summaries that capture t
    Instructions:
    1. Summarize the transcript in a single concise paragraph.
    2. Ignore any timestamps in your summary.
    3. Focus on the spoken content (Text) of the video.
    Note: In the transcript, "Text" refers to the spoken words in the video, and "start" indicates the timestamp when that part begi
    Please summarize the following YouTube video transcript:
    {transcript}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
    """

    # Create the PromptTemplate object with the defined template
    prompt = PromptTemplate(
        input_variables=["transcript"],
        template=template
    )

    return prompt
```

Instantiate the LLMChain for summarization

In this section, we create an `LLMChain` for generating summaries from the provided transcript using the defined prompt template. An `LLMChain` is a construct that combines a language model with a specific prompt, facilitating the process of generating text outputs based on the input data.

The function accepts the following parameters:

- `llm`: An instance of the language model that will be used for summarization. This model processes the input and generates the summary.
- `prompt`: A `PromptTemplate` instance that contains the structured prompt for the model. It guides the LLM in generating a concise summary of the transcript.
- `verbose`: A boolean parameter that determines whether to enable verbose output during the summary generation process (default is `True`).

This function returns an `LLMChain` instance, which can then be used to generate summaries efficiently.

```
def create_summary_chain(llm, prompt, verbose=True):
    """
    Create an LLMChain for generating summaries.

    :param llm: Language model instance
    :param prompt: PromptTemplate instance
    :param verbose: Boolean to enable verbose output (default: True)
    :return: LLMChain instance
    """
    return LLMChain(llm=llm, prompt=prompt, verbose=verbose)
```

Retrieving relevant context and generating answers

Define the retrieval function

This function retrieves relevant context from the `FAISS` index based on the user's query. It leverages the similarity search capabilities of FAISS to find the most pertinent documents or chunks that relate to the input query.

Parameters:

- `query(str)`: The user's query string that specifies what information is being sought.
- `faiss_index(FAISS)`: The FAISS index containing the embedded documents, which allows for efficient similarity searches.
- `k(int, optional, default=3)`: The number of most relevant documents to retrieve.

Returns:

- `list`: A list of the `k` most relevant documents (or document chunks) that match the query.

Function workflow:

1. Takes the user's query as input.
2. Uses the FAISS index to perform a similarity search based on the vector representations (embeddings) of the query and the documents in the index.
3. Returns the `k` most similar documents or chunks to the query.

```
def retrieve(query, faiss_index, k=7):
    """
    Retrieve relevant context from the FAISS index based on the user's query.
    Parameters:
        query (str): The user's query string.
        faiss_index (FAISS): The FAISS index containing the embedded documents.
        k (int, optional): The number of most relevant documents to retrieve (default is 3).
    Returns:
        list: A list of the k most relevant documents (or document chunks).
    """
    relevant_context = faiss_index.similarity_search(query, k=k)
    return relevant_context
```

Creating the Q&A prompt template

This function structures the prompt for answering questions based on the video content. It is designed to guide the AI in providing accurate and detailed responses to user queries.

Returns:

- `PromptTemplate`: A configured `PromptTemplate` object for Q&A tasks.

Function workflow:

1. Defines a template string that instructs the AI on its role and the task it needs to perform.
2. Creates a `PromptTemplate` object with the template and the required input variables.
3. Returns the configured `PromptTemplate`.

```
from langchain import PromptTemplate
def create_qa_prompt_template():
    """
    Create a PromptTemplate for question answering based on video content.
    Returns:
        PromptTemplate: A PromptTemplate object configured for Q&A tasks.
    """

    # Define the template string
    qa_template = """
    You are an expert assistant providing detailed answers based on the following video content.
    Relevant Video Context: {context}
    Based on the above context, please answer the following question:
    Question: {question}
    """
    # Create the PromptTemplate object
    prompt_template = PromptTemplate(
        input_variables=["context", "question"],
        template=qa_template
    )
    return prompt_template
```

Example usage

Following is an example of how to call `create_qa_prompt_template()` to create a prompt template for Q&A tasks:

```
    # Creating the Q&A prompt template
```

```
qa_prompt_template = create_qa_prompt_template()
# Example of how to use the prompt template with context and a question
context = "This video explains the fundamentals of quantum physics."
question = "What are the key principles discussed in the video?"
# Generating the prompt
generated_prompt = qa_prompt_template.format(context=context, question=question)
# Output the generated prompt
print(generated_prompt)
```

Expected output

The `create_qa_prompt_template()` function returns a `PromptTemplate` object, which can be used to format the context and question. The output from the `format()` method will look like this:

```
You are an expert assistant providing detailed answers based on the following video content.
Relevant Video Context: This video explains the fundamentals of quantum physics.
Based on the above context, please answer the following question:
Question: What are the key principles discussed in the video?
```

Setting up the Q&A LLMChain

This section demonstrates how to instantiate an `LLMChain` for generating answers to questions based on a given prompt template and language model.

Returns:

- `LLMChain`: An instantiated `LLMChain` ready for question answering.

Function workflow:

1. Takes a language model and a prompt template as inputs.
2. Creates an `LLMChain` that combines the model and the prompt.
3. Sets the verbosity of the chain.
4. Returns the configured `LLMChain` object.

```
def create_qa_chain(llm, prompt_template, verbose=True):
    """
    Create an LLMChain for question answering.
    Args:
        llm: Language model instance
            The language model to use in the chain (e.g., WatsonxGranite).
        prompt_template: PromptTemplate
            The prompt template to use for structuring inputs to the language model.
        verbose: bool, optional (default=True)
            Whether to enable verbose output for the chain.
    Returns:
        LLMChain: An instantiated LLMChain ready for question answering.
    """

    return LLMChain(llm=llm, prompt=prompt_template, verbose=verbose)
```

Generating an answer

This section demonstrates how to retrieve relevant context from a FAISS index and generate an answer based on user input.

Returns:

- `str`: The generated answer to the user's question.

Function workflow:

1. Retrieves relevant context using the FAISS index based on the user's question.
2. Uses the question-answering chain (`LLMChain`) to generate an answer based on the retrieved context and the user's question.

3. Returns the generated answer.

```
def generate_answer(question, faiss_index, qa_chain, k=7):
    """
    Retrieve relevant context and generate an answer based on user input.
    Args:
        question: str
            The user's question.
        faiss_index: FAISS
            The FAISS index containing the embedded documents.
        qa_chain: LLMChain
            The question-answering chain (LLMChain) to use for generating answers.
        k: int, optional (default=3)
            The number of relevant documents to retrieve.
    Returns:
        str: The generated answer to the user's question.
    """
    # Retrieve relevant context
    relevant_context = retrieve(question, faiss_index, k=k)
    # Generate answer using the QA chain
    answer = qa_chain.predict(context=relevant_context, question=question)
    return answer
```

Summarizing a video

This function generates a summary of a video using the preprocessed transcript. It uses IBM Watson's services to create an effective summary, ensuring that if the transcript hasn't been fetched yet, it fetches it first.

Returns:

- `str`: The generated summary of the video or a message indicating that no transcript is available.

Function workflow:

1. Checks if the transcript needs to be fetched based on the `processed_transcript` global variable.
2. If the transcript is not fetched, it retrieves the transcript from the given YouTube URL.
3. Sets up IBM Watson credentials.
4. Initializes the watsonx LLM for summarization.
5. Creates a summary prompt and chain.
6. Generates and returns the summary of the video.

```
# Initialize an empty string to store the processed transcript after fetching and preprocessing
processed_transcript = ""
def summarize_video(video_url):
    """
    Title: Summarize Video
    Description:
    This function generates a summary of the video using the preprocessed transcript.
    If the transcript hasn't been fetched yet, it fetches it first.
    Args:
        video_url (str): The URL of the YouTube video from which the transcript is to be fetched.
    Returns:
        str: The generated summary of the video or a message indicating that no transcript is available.
    """
    global fetched_transcript, processed_transcript


    if video_url:
        # Fetch and preprocess transcript
        fetched_transcript = get_transcript(video_url)
        processed_transcript = process(fetched_transcript)
    else:
        return "Please provide a valid YouTube URL."
    if processed_transcript:
        # Step 1: Set up IBM Watson credentials
        model_id, credentials, client, project_id = setup_credentials()
        # Step 2: Initialize WatsonX LLM for summarization
        llm = initialize_watsonx_llm(model_id, credentials, project_id, define_parameters())
        # Step 3: Create the summary prompt and chain
        summary_prompt = create_summary_prompt()
        summary_chain = create_summary_chain(llm, summary_prompt)
        # Step 4: Generate the video summary
        summary = summary_chain.run({"transcript": processed_transcript})
        return summary
    else:
        return "No transcript available. Please fetch the transcript first."
```

Answering a user's question

This function retrieves relevant context from the FAISS index based on the user's query and generates an answer using the preprocessed transcript. It first checks if the transcript has been fetched; if not, it fetches and processes the transcript from the provided YouTube video URL.

If the transcript is available and a user question is provided, the function proceeds to chunk the transcript for better context retrieval. It then sets up IBM Watson credentials and initializes the watsonx LLM specifically for Q&A tasks.

Next, it creates a FAISS index using the chunked transcript and sets up the Q&A prompt template and chain. Finally, it generates an answer to the user's question using the FAISS index and returns the answer. If the transcript hasn't been fetched or if the user fails to provide a valid question, the function returns a relevant message indicating the issue.

```
def answer_question(video_url, user_question):
    """
    Title: Answer User's Question
    Description:
    This function retrieves relevant context from the FAISS index based on the user’s query
    and generates an answer using the preprocessed transcript.
    If the transcript hasn't been fetched yet, it fetches it first.
    Args:
        video_url (str): The URL of the YouTube video from which the transcript is to be fetched.
        user_question (str): The question posed by the user regarding the video.
    Returns:
        str: The answer to the user's question or a message indicating that the transcript
             has not been fetched.
    """
    global fetched_transcript, processed_transcript
    # Check if the transcript needs to be fetched
    if not processed_transcript:
        if video_url:
            # Fetch and preprocess transcript
            fetched_transcript = get_transcript(video_url)
            processed_transcript = process(fetched_transcript)
        else:
            return "Please provide a valid YouTube URL."
    if processed_transcript and user_question:
        # Step 1: Chunk the transcript (only for Q&A)
        chunks = chunk_transcript(processed_transcript)
        # Step 2: Set up IBM Watson credentials
        model_id, credentials, client, project_id = setup_credentials()
        # Step 3: Initialize WatsonX LLM for Q&A
        llm = initialize_watsonx_llm(model_id, credentials, project_id, define_parameters())
        # Step 4: Create FAISS index for transcript chunks (only needed for Q&A)
        embedding_model = setup_embedding_model(credentials, project_id)
        faiss_index = create_faiss_index(chunks, embedding_model)
        # Step 5: Set up the Q&A prompt and chain
        qa_prompt = create_qa_prompt_template()
        qa_chain = create_qa_chain(llm, qa_prompt)
        # Step 6: Generate the answer using FAISS index
        answer = generate_answer(user_question, faiss_index, qa_chain)
        return answer
    else:
        return "Please provide a valid question and ensure the transcript has been fetched."
```

Setting up a Gradio interface

This section describes the setup of a Gradio interface for interacting with a YouTube video, allowing users to fetch its transcript, summarize it, or ask questions based on the content of the video. The interface is built using Gradio's Blocks API, which facilitates the creation of interactive web applications with minimal code.

```
with gr.Blocks() as interface:
    # Input field for YouTube URL
    video_url = gr.Textbox(label="YouTube Video URL", placeholder="Enter the YouTube Video URL")

    # Outputs for summary and answer
    summary_output = gr.Textbox(label="Video Summary", lines=5)
    question_input = gr.Textbox(label="Ask a Question About the Video", placeholder="Ask your question")
    answer_output = gr.Textbox(label="Answer to Your Question", lines=5)
    # Buttons for selecting functionalities after fetching transcript
    summarize_btn = gr.Button("Summarize Video")
    question_btn = gr.Button("Ask a Question")
    # Display status message for transcript fetch
    transcript_status = gr.Textbox(label="Transcript Status", interactive=False)
    # Set up button actions
    summarize_btn.click(summarize_video, inputs=video_url, outputs=summary_output)
    question_btn.click(answer_question, inputs=[video_url, question_input], outputs=answer_output)
# Launch the app with specified server name and port
interface.launch(server_name="0.0.0.0", server_port=7860)
```

Description

- **Input field**: A textbox (`video_url`) is created for users to enter the URL of the YouTube video they want to analyze.
- **Output fields**: Two additional textboxes, `summary_output` and `answer_output`, are used to display the generated summary and answers to user questions, respectively. A `question_input` textbox allows users to type their queries regarding the video content.
- **Buttons**: Two buttons, `summarize_btn` and `question_btn`, are included to allow users to trigger the summarization of the video or to ask a specific question about it.
- **Transcript status**: A textbox (`transcript_status`) displays feedback to the user regarding the status of the transcript fetching process, indicating whether it was successful or if there were issues (for example, invalid URL).
- **Button actions**: The `summarize_btn` is linked to the `summarize_video` function, which takes the YouTube URL as input and returns the summary to `summary_output`. The `question_btn` is linked to the `answer_question` function, which takes both the YouTube URL and user question as inputs and returns the answer to `answer_output`.
- **Launch configuration**: Finally, the `interface.launch(server_name="0.0.0.0", server_port=7860)` line starts the Gradio application, enabling users to access it using a web browser on their local server.

Complete code reference

In this section, you'll find the full, consolidated code for the application, which includes all code snippets provided in previous steps. Use this as a reference to ensure that your implementation is consistent with the complete code structure required for the application to function as intended.

```
# Import necessary libraries for the YouTube bot
import gradio as gr
import re  #For extracting video id
from youtube_transcript_api import YouTubeTranscriptApi  # For extracting transcripts from YouTube videos
from langchain.text_splitter import RecursiveCharacterTextSplitter  # For splitting text into manageable segments
from ibm_watsonx_ai.foundation_models.utils.enums import ModelTypes  # For specifying model types
from ibm_watsonx_ai import APIClient, Credentials  # For API client and credentials management
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams  # For managing model parameters
from ibm_watsonx_ai.foundation_models.utils.enums import DecodingMethods  # For defining decoding methods
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings  # For interacting with IBM's LLM and embeddings
from ibm_watsonx_ai.foundation_models.utils import get_embedding_model_specs  # For retrieving model specifications
from ibm_watsonx_ai.foundation_models.utils.enums import EmbeddingTypes  # For specifying types of embeddings
from langchain_community.vectorstores import FAISS  # For efficient vector storage and similarity search
from langchain.chains import LLMChain  # For creating chains of operations with LLMs
from langchain.prompts import PromptTemplate  # For defining prompt templates
def get_video_id(url):
    # Regex pattern to match YouTube video URLs
    pattern = r'https:\/\/www\.youtube\.com\/watch\?v=([a-zA-Z0-9_-]{11})'
    match = re.search(pattern, url)
    return match.group(1) if match else None
def get_transcript(url):
    # Extracts the video ID from the URL
    video_id = get_video_id(url)
    # Create a YouTubeTranscriptApi() object
    ytt_api = YouTubeTranscriptApi()

    # Fetch the list of available transcripts for the given YouTube video
    transcripts = ytt_api.list(video_id)

    transcript = ""
    for t in transcripts:
        # Check if the transcript's language is English
        if t.language_code == 'en':
            if t.is_generated:
                # If no transcript has been set yet, use the auto-generated one
                if len(transcript) == 0:
                    transcript = t.fetch()
            else:
                # If a manually created transcript is found, use it (overrides auto-generated)
                transcript = t.fetch()
                break  # Prioritize the manually created transcript, exit the loop

    return transcript if transcript else None
def process(transcript):
    # Initialize an empty string to hold the formatted transcript
    txt = ""

    # Loop through each entry in the transcript
    for i in transcript:
        try:
            # Append the text and its start time to the output string
            #txt += f"Text: {i['text']} Start: {i['start']}\n"
            txt += f"Text: {i.text} Start: {i.start}\n"
        except KeyError:
            # If there is an issue accessing 'text' or 'start', skip this entry
            pass

    # Return the processed transcript as a single string
    return txt
def chunk_transcript(processed_transcript, chunk_size=200, chunk_overlap=20):
    # Initialize the RecursiveCharacterTextSplitter with specified chunk size and overlap
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap
    )
    # Split the transcript into chunks
    chunks = text_splitter.split_text(processed_transcript)
    return chunks
def setup_credentials():
    # Define the model ID for the WatsonX model being used
```

```
    model_id = "meta-llama/llama-3-2-3b-instruct"

    # Set up the credentials by specifying the URL for IBM Watson services
    credentials = Credentials(url="https://us-south.ml.cloud.ibm.com")

    # Create an API client using the credentials
    client = APIClient(credentials)

    # Define the project ID associated with the WatsonX platform
    project_id = "skills-network"

    # Return the model ID, credentials, client, and project ID for later use
    return model_id, credentials, client, project_id
def define_parameters():
    # Return a dictionary containing the parameters for the WatsonX model
    return {
        # Set the decoding method to GREEDY for generating text
        GenParams.DECODING_METHOD: DecodingMethods.GREEDY,

        # Specify the maximum number of new tokens to generate
        GenParams.MAX_NEW_TOKENS: 900,
    }
def initialize_watsonx_llm(model_id, credentials, project_id, parameters):
    # Create and return an instance of the WatsonxLLM with the specified configuration
    return WatsonxLLM(
        model_id=model_id,          # Set the model ID for the LLM
        url=credentials.get("url"),      # Retrieve the service URL from credentials
        project_id=project_id,            # Set the project ID for accessing resources
        params=parameters                 # Pass the parameters for model behavior
    )
def setup_embedding_model(credentials, project_id):
    # Create and return an instance of WatsonxEmbeddings with the specified configuration
    return WatsonxEmbeddings(
        model_id=EmbeddingTypes.IBM_SLATE_30M_ENG.value,  # Set the model ID for the SLATE-30M embedding model
        url=credentials["url"],                            # Retrieve the service URL from the provided credentials
        project_id=project_id                              # Set the project ID for accessing resources in the Watson environment
    )
def create_faiss_index(chunks, embedding_model):
    """
    Create a FAISS index from text chunks using the specified embedding model.

    :param chunks: List of text chunks
    :param embedding_model: The embedding model to use
    :return: FAISS index
    """
    # Use the FAISS library to create an index from the provided text chunks
    return FAISS.from_texts(chunks, embedding_model)
def perform_similarity_search(faiss_index, query, k=3):
    """
    Search for specific queries within the embedded transcript using the FAISS index.

    :param faiss_index: The FAISS index containing embedded text chunks
    :param query: The text input for the similarity search
    :param k: The number of similar results to return (default is 3)
    :return: List of similar results
    """
    # Perform the similarity search using the FAISS index
    results = faiss_index.similarity_search(query, k=k)
    return results
def create_summary_prompt():
    """
    Create a PromptTemplate for summarizing a YouTube video transcript.

    :return: PromptTemplate object
    """
    # Define the template for the summary prompt
    template = """
    <|begin_of_text|><|start_header_id|>system<|end_header_id|>
    You are an AI assistant tasked with summarizing YouTube video transcripts. Provide concise, informative summaries that capture t
    Instructions:
    1. Summarize the transcript in a single concise paragraph.
    2. Ignore any timestamps in your summary.
    3. Focus on the spoken content (Text) of the video.
    Note: In the transcript, "Text" refers to the spoken words in the video, and "start" indicates the timestamp when that part begi
    Please summarize the following YouTube video transcript:
    {transcript}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
    """

    # Create the PromptTemplate object with the defined template
    prompt = PromptTemplate(
        input_variables=["transcript"],
        template=template
    )

    return prompt
def create_summary_chain(llm, prompt, verbose=True):
    """
    Create an LLMChain for generating summaries.

    :param llm: Language model instance
    :param prompt: PromptTemplate instance
    :param verbose: Boolean to enable verbose output (default: True)
    :return: LLMChain instance
    """
    return LLMChain(llm=llm, prompt=prompt, verbose=verbose)
def retrieve(query, faiss_index, k=7):
    """
    Retrieve relevant context from the FAISS index based on the user's query.
    Parameters:
        query (str): The user's query string.
        faiss_index (FAISS): The FAISS index containing the embedded documents.
        k (int, optional): The number of most relevant documents to retrieve (default is 3).
    Returns:
        list: A list of the k most relevant documents (or document chunks).
```

```
        """
        relevant_context = faiss_index.similarity_search(query, k=k)
        return relevant_context
    def create_qa_prompt_template():
        """
        Create a PromptTemplate for question answering based on video content.
        Returns:
            PromptTemplate: A PromptTemplate object configured for Q&A tasks.
        """

        # Define the template string
        qa_template = """
        <|begin_of_text|><|start_header_id|>system<|end_header_id|>
        You are an expert assistant providing detailed and accurate answers based on the following video content. Your responses should
        1. Precise and free from repetition
        2. Consistent with the information provided in the video
        3. Well-organized and easy to understand
        4. Focused on addressing the user's question directly
        If you encounter conflicting information in the video content, use your best judgment to provide the most likely correct answer
        Note: In the transcript, "Text" refers to the spoken words in the video, and "start" indicates the timestamp when that part begi
        <|start_header_id|>user<|end_header_id|>
        Relevant Video Context: {context}
        Based on the above context, please answer the following question:
        {question}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
        """
        # Create the PromptTemplate object
        prompt_template = PromptTemplate(
            input_variables=["context", "question"],
            template=qa_template
        )
        return prompt_template
    def create_qa_chain(llm, prompt_template, verbose=True):
        """
        Create an LLMChain for question answering.
        Args:
            llm: Language model instance
                The language model to use in the chain (e.g., WatsonxGranite).
            prompt_template: PromptTemplate
                The prompt template to use for structuring inputs to the language model.
            verbose: bool, optional (default=True)
                Whether to enable verbose output for the chain.
        Returns:
            LLMChain: An instantiated LLMChain ready for question answering.
        """

        return LLMChain(llm=llm, prompt=prompt_template, verbose=verbose)
    def generate_answer(question, faiss_index, qa_chain, k=7):
        """
        Retrieve relevant context and generate an answer based on user input.
        Args:
            question: str
                The user's question.
            faiss_index: FAISS
                The FAISS index containing the embedded documents.
            qa_chain: LLMChain
                The question-answering chain (LLMChain) to use for generating answers.
            k: int, optional (default=3)
                The number of relevant documents to retrieve.
        Returns:
            str: The generated answer to the user's question.
        """
        # Retrieve relevant context
        relevant_context = retrieve(question, faiss_index, k=k)
        # Generate answer using the QA chain
        answer = qa_chain.predict(context=relevant_context, question=question)
        return answer
    # Initialize an empty string to store the processed transcript after fetching and preprocessing
    processed_transcript = ""
    def summarize_video(video_url):
        """
        Title: Summarize Video
        Description:
        This function generates a summary of the video using the preprocessed transcript.
        If the transcript hasn't been fetched yet, it fetches it first.
        Args:
            video_url (str): The URL of the YouTube video from which the transcript is to be fetched.
        Returns:
            str: The generated summary of the video or a message indicating that no transcript is available.
        """
        global fetched_transcript, processed_transcript


        if video_url:
            # Fetch and preprocess transcript
            fetched_transcript = get_transcript(video_url)
            processed_transcript = process(fetched_transcript)
        else:
            return "Please provide a valid YouTube URL."
        if processed_transcript:
            # Step 1: Set up IBM Watson credentials
            model_id, credentials, client, project_id = setup_credentials()
            # Step 2: Initialize WatsonX LLM for summarization
            llm = initialize_watsonx_llm(model_id, credentials, project_id, define_parameters())
            # Step 3: Create the summary prompt and chain
            summary_prompt = create_summary_prompt()
            summary_chain = create_summary_chain(llm, summary_prompt)
            # Step 4: Generate the video summary
            summary = summary_chain.run({"transcript": processed_transcript})
            return summary
        else:
            return "No transcript available. Please fetch the transcript first."
    def answer_question(video_url, user_question):
        """
        Title: Answer User's Question
```

```
        Description:
        This function retrieves relevant context from the FAISS index based on the user's query
        and generates an answer using the preprocessed transcript.
        If the transcript hasn't been fetched yet, it fetches it first.
        Args:
            video_url (str): The URL of the YouTube video from which the transcript is to be fetched.
            user_question (str): The question posed by the user regarding the video.
        Returns:
            str: The answer to the user's question or a message indicating that the transcript
                 has not been fetched.
        """
        global fetched_transcript, processed_transcript
        # Check if the transcript needs to be fetched
        if not processed_transcript:
            if video_url:
                # Fetch and preprocess transcript
                fetched_transcript = get_transcript(video_url)
                processed_transcript = process(fetched_transcript)
            else:
                return "Please provide a valid YouTube URL."
        if processed_transcript and user_question:
            # Step 1: Chunk the transcript (only for Q&A)
            chunks = chunk_transcript(processed_transcript)
            # Step 2: Set up IBM Watson credentials
            model_id, credentials, client, project_id = setup_credentials()
            # Step 3: Initialize WatsonX LLM for Q&A
            llm = initialize_watsonx_llm(model_id, credentials, project_id, define_parameters())
            # Step 4: Create FAISS index for transcript chunks (only needed for Q&A)
            embedding_model = setup_embedding_model(credentials, project_id)
            faiss_index = create_faiss_index(chunks, embedding_model)
            # Step 5: Set up the Q&A prompt and chain
            qa_prompt = create_qa_prompt_template()
            qa_chain = create_qa_chain(llm, qa_prompt)
            # Step 6: Generate the answer using FAISS index
            answer = generate_answer(user_question, faiss_index, qa_chain)
            return answer
        else:
            return "Please provide a valid question and ensure the transcript has been fetched."
    with gr.Blocks() as interface:
        gr.Markdown(
            "<h2 style='text-align: center;'>YouTube Video Summarizer and Q&A</h2>"
        )
        # Input field for YouTube URL
        video_url = gr.Textbox(label="YouTube Video URL", placeholder="Enter the YouTube Video URL")

        # Outputs for summary and answer
        summary_output = gr.Textbox(label="Video Summary", lines=5)
        question_input = gr.Textbox(label="Ask a Question About the Video", placeholder="Ask your question")
        answer_output = gr.Textbox(label="Answer to Your Question", lines=5)
        # Buttons for selecting functionalities after fetching transcript
        summarize_btn = gr.Button("Summarize Video")
        question_btn = gr.Button("Ask a Question")
        # Display status message for transcript fetch
        transcript_status = gr.Textbox(label="Transcript Status", interactive=False)
        # Set up button actions
        summarize_btn.click(summarize_video, inputs=video_url, outputs=summary_output)
        question_btn.click(answer_question, inputs=[video_url, question_input], outputs=answer_output)
    # Launch the app with specified server name and port
    interface.launch(server_name="0.0.0.0", server_port=7860)
```

Serve the application

To serve the application, paste the following into your Python terminal:

```
python3.11 ytbot.py
```

If you cannot find an open Python terminal or the buttons on the above cell do not work, you can launch a terminal by going to **Terminal > New Terminal**. However, if you launch a new terminal, do not forget to source the virtual environment you created at the beginning of the tutorial before running the above line:

```
source my_env/bin/activate # activate my_env
```

Launch the application

You are now ready to launch the served application! To launch, click the following button:

Launch Application

If the above button does not work, complete the following steps:

1. Select the Skills Network extension.
2. Click **Launch Application**.
3. Insert the port number (in this case 7860, which is the server port we put in `ytbot.py`)
4. Click **Your Application** to launch the application. **Note**: If the application does not work using **Your Application**, use the icon **Open in new browser tab**.

Test the application

To test the application, you can use the YouTube video link https://www.youtube.com/watch?v=T-D1OfcDW1M. This video offers a high-level introduction to RAG from a trusted source and can help ground the LLM's responses, reducing the likelihood of hallucinations.

Steps to generate the summary

1. **Input the video URL**: Enter the following URL into the input field labeled "YouTube Video URL":
   `https://www.youtube.com/watch?v=T-D1OfcDW1M`
2. **Summarize the video**: Click the **Summarize Video** button. The application will fetch the transcript and generate a summary based on the content of the video.
3. **View the summary**: Once the summarization is complete, the generated summary will be displayed in the **Video Summary** text box.

Example questions

After summarizing the video, you can engage further by asking specific questions:

1. **Question**: *How does one reduce hallucinations?*
   This question can't be answered accurately without context, as the term 'hallucination' can refer either to a psychological condition in humans or to the generation of false or misleading outputs by large language models (LLMs). Fortunately, in this case, we have a video transcript that provides the necessary context. To confirm this, simply paste the question into the **Ask a Question About the Video** input field and click the **Ask a Question** button.

2. **Question**: *Which problems does RAG solve, according to the video?*
   In this case we are asking for information that is specifically contained in the video. In order to obtain a context-aware response, paste the question into the **Ask a Question About the Video** input field and click the **Ask a Question** button.

Conclusion

In this lab, you explored the use of AI and NLP techniques to fetch, summarize, and ask questions about YouTube videos. You learned how to:

- Fetch a video transcript and preprocess it.
- Use AI models to generate concise summaries of the video's content.
- Retrieve relevant information based on user questions, using advanced Q&A techniques.

You've made great progress, and if you missed anything, don't worry! You can always come back and do the lab again to reinforce your understanding.

Next steps

Now that you've gained hands-on experience with video summarization and Q&A, here are some ideas for further exploration:

1. **Try asking different questions**: Experiment with asking new types of questions based on the video you already used. For instance, you can ask about specific timestamps, deeper insights on discussed topics, or further clarifications.

2. **Use a different video**: Test the application with a new video. Simply input a different YouTube URL and see how well the summarizer and Q&A tool handle new content. This will help you assess the model's adaptability to different video topics and formats.

3. **Enhance the application**: Consider adding new features such as sentiment analysis on the video transcript or enabling the tool to summarize videos in different languages.

Author(s)

Kunal Makwana

Other Contributor(s)

Ricky Shi

Wojciech "Victor" Fulmyk