End-to-End AI Voice Assistance Pipeline

Objective*:* Design a pipeline that takes a voice query command, converts it into text, uses a Large Language Model (LLM) to generate a response, and then converts the output text back into speech. The system should have low latency, Voice Activity Detection (VAD), restrict the output to 2 sentences, and allow for tunable parameters such as pitch, male/female voice, and speed.

Proposed Methodology:

Implementing both audio file or microphone recording feature for input audio. Preprocess audio using ffmpeg and webrtcvad for vad_threshold, mono audio_channel_count and 16kHZ sampling rate. Using faster-whisper for preprocessed audio transcription. Feed the transcripted text to the llama3.1:8B LLM model. Pass the generated response to edge-tts with tunable parameters such as pitch, rate, male/female voice for text-to-speech conversion.

Procedure*:*

(For Demonstration Google Colab is used here as it provides free GPU)

Step 1: Install prerequisites using pip.

```
!pip install faster-whisper ffmpeg-python webrtcvad edge-tts asyncio
```

Here, I've installed faster-whisper for transcription, ffmpeg-python & webrtcvad for audio preprocessing, edge-tts for text-to speech conversion of llama response and asyncio forayanchronous edge-tts conversion.

Step 2: Install & Setup Ollama.

```
!sudo apt-get install -y pciutils
!curl -fsSL https://ollama.com/install.sh | sh # download ollama api
from IPython.display import clear_output

# Create a Python script to start the Ollama API server in a separate thread

import os
import threading
import subprocess
import requests
import json

def ollama():
```

```
    os.environ['OLLAMA_HOST'] = '0.0.0.0:11434'
    os.environ['OLLAMA_ORIGINS'] = '*'
    subprocess.Popen(["ollama", "serve"])

ollama_thread = threading.Thread(target=ollama)
ollama_thread.start()
```

Huggingface's Llama pre-trained model crashes due to insufficient RAM on local and colab notebooks. Thus I switched to Ollama.

First, I'm installing some essential tools. pciutils helps us manage the hardware, and the next command downloads the Ollama API, which is a crucial piece for running the language model that we'll be using later. After running these installation commands, I clear the output to keep things neat.

Environment Setup: I set two environment variables. The OLLAMA_HOST variable is set to 0.0.0.0:11434, which means that the server will listen on all IP addresses of the machine (useful if we want to access it from different devices) on port 11434. The OLLAMA_ORIGINSvariable is set to *, meaning the server will accept requests from any origin, which is handy during development.

Starting the Server: The subprocess.Popen command launches the Ollama server in the background. By running this in a separate thread (ollama_thread), I make sure that this process doesn't block the rest of my code. This is crucial because we want the server to run in the background while we proceed with the rest of our pipeline.

Step 3: Install Lightrag Dependency.

```
from IPython.display import clear_output
!ollama pull llama3.1:8b
clear_output()

!pip install -U lightrag[ollama]
```

First, I'm pulling a specific version of the LLaMA model (version 3.1 with 8 billion parameters) using the ollama pull command. This means I'm downloading the model so I can use it later.After that, I'm clearing the output to keep the notebook clean.

Then, I'm making sure that the lightrag library is installed and up to date, along with its ollama dependencies, by running pip install -U lightrag[ollama]. This library will help me integrate the model I just pulled into my Python code.

Step 4: Importing Libraries.

```
from IPython.display import Javascript
from google.colab import output
from base64 import b64decode
import os
import ffmpeg
import webrtcvad
import wave
from faster_whisper import WhisperModel
from lightrag.core.generator import Generator
from lightrag.core.component import Component
from lightrag.core.model_client import ModelClient
from lightrag.components.model_client import OllamaClient, GroqAPIClient
import time
import asyncio
import random
import edge_tts
from edge_tts import VoicesManager
```

Here, I've imported all the required libraries that will be used throughout the module.

Step 5: Microphone Recording.

```
RECORD = """
const sleep  = time => new Promise(resolve => setTimeout(resolve, time))
const b2text = blob => new Promise(resolve => {
  const reader = new FileReader()
  reader.onloadend = e => resolve(e.srcElement.result)
  reader.readAsDataURL(blob)
})
var record = time => new Promise(async resolve => {
  stream = await navigator.mediaDevices.getUserMedia({ audio: true })
  recorder = new MediaRecorder(stream)
  chunks = []
  recorder.ondataavailable = e => chunks.push(e.data)
  recorder.start()
  await sleep(time)
  recorder.onstop = async ()=>{
    blob = new Blob(chunks)
    text = await b2text(blob)
```

```
    resolve(text)
  }
  recorder.stop()
})
"""

def record(sec=5):
  display(Javascript(RECORD))
  s = output.eval_js('record(%d)' % (sec*1000))
  b = b64decode(s.split(',')[1])
  audio_path = 'audio.wav'
  with open(audio_path,'wb') as f:
    f.write(b)
  return audio_path
```

In this code, I'm setting up a JavaScript function that records audio from my computer's microphone. The RECORDvariable holds the JavaScript code, which does all the work.

Here's what happens:

1. **JavaScript Code Setup:** The RECORDvariable contains a block of JavaScript code. This code will run in my browser and handle audio recording. It grabs the audio from my microphone, records it for a specific amount of time, and then converts that recording into a format that Python can handle.

2. **Recording Audio:** The record(sec=5) function in Python calls this JavaScript code. The default recording time is set to 5 seconds, but I can change it by passing a different value when I call record(sec=x).

3. **Getting the Audio Data:** Once the recording is done, the JavaScript code stops the recording and converts the audio into a Base64 string. Then, it passes that string back to my Python environment.

4. **Saving the Audio:** Back in Python, the Base64 string is decoded to get the actual audio data. This data is then saved as a .wav file named audio.wav. Now, I've got the recorded audio file saved on my machine, ready for further processing.

*Step 6:*Defining Preprocessing Audio Function.

```
def preprocess_audio(input_file, output_file):
    stream = ffmpeg.input(input_file)
```

```
    stream = ffmpeg.output(stream, output_file, ar=16000, ac=1)
    ffmpeg.run(stream)
    vad_filter('processed_audio.wav', 'vad_filtered_audio.wav')

def vad_filter(input_file, output_file, vad_threshold=0.5):
    vad = webrtcvad.Vad()
    vad.set_mode(2)  # 0: Aggressive VAD, 3: Very Aggressive

    with wave.open(input_file, 'rb') as wf:
        with wave.open(output_file, 'wb') as out:
            out.setnchannels(wf.getnchannels())
            out.setsampwidth(wf.getsampwidth())
            out.setframerate(wf.getframerate())

            frame_size = 160  # 10 ms at 16kHz
            frame = wf.readframes(frame_size)
            while frame:
                if vad.is_speech(frame, wf.getframerate()):
                    out.writeframes(frame)
                frame = wf.readframes(frame_size)
    transcript('vad_filtered_audio.wav')
```

In this code, I'm processing an audio file to make it ready for further analysis. It starts with the preprocess_audio function, where I use ffmpeg to convert the audio to a standard format—specifically, setting the audio to a 16kHz sample rate and mono channel. This makes sure that the audio is in a consistent format for later steps.

After that, the vad_filter function comes into play. Here, I'm using webrtcvad, which is a Voice Activity Detection (VAD) tool. It helps in identifying whether the audio contains speech or just noise. I set the VAD to a moderately aggressive mode (level 2), which is pretty good at filtering out non-speech parts.

In this function, I open the processed audio file and create a new file where only the speech parts will be saved. I read through the audio in small chunks (about 10 milliseconds each), and whenever the VAD detects speech, I write that part into the new file. By the end of this process, I've got a new audio file that should mostly contain just the speech segments.

Finally, I pass this filtered audio to the transcript function for further processing, like transcribing the speech to text.

*Step 7:*Preprocessed Audio Transcription.

```
def transcript(audio_file):
    model_size = "medium"
    model = WhisperModel(model_size, device="cpu", compute_type="int8")
    segments, info = model.transcribe(audio_file, beam_size=5)
    prompt= ""
    for segment in segments:
        prompt += segment.text + " "

    # Optional: Trim any extra space at the end of the prompt
    prompt = prompt.strip()
    print(prompt)
    return prompt
```

I'm loading a medium-sized Whisper model for transcribing audio, specifically set to run on the CPU with int8 precision for efficiency. Then, I'm feeding an audio file into the model to break it down into segments, each representing a chunk of transcribed text. I loop through these segments, piecing together the text from each one into a single string called prompt. After that,I trim any unnecessary spaces at the end of this string. Finally, I print out the full transcribed text and return it for further use.

*Step 8:*Defining Llama3.1:8B LLM Model.

```
def llama(prompt):
    qa_template = r"""<SYS>
    You are a helpful assistant.
    </SYS>
    User: {{input_str}}
    You:"""

    class SimpleQA(Component):
        def __init__(self, model_client: ModelClient, model_kwargs: dict):
            super().__init__()
            self.generator = Generator(
                model_client=model_client,
                model_kwargs=model_kwargs,
                template=qa_template,
            )

        def call(self, input: dict) -> str:
```

```
            return self.generator.call({"input_str": str(input)})

        async def acall(self, input: dict) -> str:
            return await self.generator.acall({"input_str": str(input)})

    model = {
        "model_client": OllamaClient(),
        "model_kwargs": {"model": "llama3.1:8b"}
    }
    qa = SimpleQA(**model)
    output=qa(f"{prompt} Respond strictly in two sentences only.")
    response = output.data
    display(f"**Answer:** {response}")
    return response
```

The llama function is designed to interact with a language model, specifically the LLaMA model, to generate a concise response to a given prompt. Here's a breakdown:

1. **Defining a Template**: I start by setting up a template (qa_template) that tells the model it's acting as a helpful assistant. This template will guide the model in framing its responses.
2. **Creating a Class (SimpleQA)**: I define a class called SimpleQA that inherits from Component. This class wraps around the Generator object, which is responsible for taking the input (in this case, a user prompt) and using the language model to generate a response.
3. **Model Setup**: I set up the LLaMA model by specifying the client (OllamaClient) and model parameters (model_kwargs). This is how I tell the SimpleQAclass which model to use and how to configure it.
4. **Generating a Response**: I create an instance of SimpleQA with the model configuration and then call it with the user's prompt. I also add a little extra instruction, asking the model to respond in two sentences only. The response generated by the model is stored in the responsevariable.
5. **Displaying the Response**: Finally, I print the response in a formatted way so that it's easy to read.
6. **Returning the Response**: The function returns the generated response, so it can be used elsewhere if needed.

In essence, this code is a neat little package that takes a prompt, sends it to a language model, and returns a succinct, assistant-like response.

*Step 9:*Text-to-Speech Function.

```
async def tts(text):
    # Generate male and female TTS responses from input text and save them
as audio files.
    voices = await VoicesManager.create()  # Await the asynchronous
creation of VoicesManager

    # Find male and female voices
    male_voice = voices.find(Gender="Male", Language="en")
    female_voice = voices.find(Gender="Female", Language="en")

    # Generate male voice response
    communicate_male = edge_tts.Communicate(text,
random.choice(male_voice)["Name"], rate="-10%", pitch="-10Hz")
    await communicate_male.save("male_response.mp3")

    # Generate female voice response
    communicate_female = edge_tts.Communicate(text,
random.choice(female_voice)["Name"], rate="-10%", pitch="-10Hz")
    await communicate_female.save("female_response.mp3")

#Text To Speech Conversion
await tts(response)
```

So in the code above, what I'm doing is setting up a function called ttsthat takes some text and then creates two different voice recordings—a male and a female version of that text.

First, I make sure to get the available voices using VoicesManager.create(), which I have to wait for because it's an asynchronous operation. Once I have those voices, I look for a male and a female voice that both speak English.

Then, I pick a random male voice from the options I found and use it to create an audio file with the text, adjusting the speech rate and pitch a bit to make it sound just right. After that, I do the same thing for the female voice.

At the end of the function, I call await tts(response) elsewhere in my script to actually run all this, which means it'll produce those two audio files and save them as "male_response.mp3" and "female_response.mp3".

Feel free to modify or tune parameters as per your likings such as pitch, rate (speed), voice, etc.. You can comment out or remove and keep only one voice male/female. This was for demonstration purposes so I've kept both.

*Step 10:*Defining main() function of the program.

```
def main():
    # Prompt the user to choose recording or file path input
    choice = input("Do you want to record from the microphone or enter an
audio file path? (Enter 'record' or 'audio_path'): ").strip().lower()

    if choice == 'record':
        print("Recording from microphone...")
        audio_path = record()  # Record for 5 seconds (default)
        print(f"Audio recorded and saved to {audio_path}")
    elif choice == 'audio_path':
        audio_path = input("Please enter the path to the audio file:
").strip()
        if not os.path.exists(audio_path):
            print("The file does not exist. Please check the path and try
again.")
            return
        print(f"Using the provided audio file: {audio_path}")
    else:
        print("Invalid choice. Please enter 'record' or 'path'.")
        return

    # Preprocess the audio file
    prompt = preprocess_audio(audio_path, 'processed_audio.wav')
    response = llama(prompt)
    print(response)

    for i in ['processed_audio.wav', 'vad_filtered_audio.wav']:
        os.remove(i)
    return response

# Run the main function
response = main()
```

This script is designed to either record audio from a microphone or process an existing audio file, transcribe the spoken words into text, and then use that text to generate a response using a language model. The response is then printed out.

Step-by-Step Walkthrough

1. **User Input:**
    - The script starts by asking me whether I want to record audio using the microphone or provide an existing audio file.
    - If I choose "record," it will record my voice for 5 seconds by default and save it as an audio file. If I choose "audio_path," I can provide the path to an existing audio file.
2. **Recording or Loading Audio:**
    - If I decide to record, the script captures my voice and saves it as audio.wav. If I provide a path, the script checks if the file exists. If the file doesn't exist, it lets me know and stops the process.
3. **Audio Preprocessing:**
    - Once the audio file is ready, the script preprocesses it, converting it to a standard format (16kHz, mono) and applying Voice Activity Detection (VAD) to filter out non-speech parts.
4. **Transcription and Response Generation:**
    - After preprocessing, the script transcribes the audio into text and then feeds that text into a language model (probably something like GPT) to generate a response based on what was said.
5. **Cleaning Up:**
    - Once everything is done, the script deletes the temporary audio files it created during processing (processed_audio.wavand vad_filtered_audio.wav)to keep things tidy.
6. **Output:**
    - Finally, the script returns the response generated by the language model and prints it out for me to see. And the generated output audio file is then generated by the tts(response) function at last when it’s called after the main() function.

The generated audio file can be found in the files section in google colab.

Latency: Suggest how to minimize the latency under (< 500 ms) using WRTC ?

-> Use GPU for faster processing. Avoid using larger or bulky models such as you can use the base-en model for speed improvement instead of large-v2 & v3 or medium in faster-whisper. Whisper has much faster implementations of itself such as whisperX jax-whisper with 70X faster performance which can be used instead. Implementing asynchronous processing for parallel execution of various tasks such as real-time audio transcription for microphone recording.

Output Restriction: Restrict the output response to a maximum of 2 sentences.

-> *output=qa(f"{prompt} Respond strictly in two sentences only.")*
Explicitly providing instruction in the prompt itself for limiting sentences in the output response.

Architechture: