UI Documentation

Tech Stack – NodeJS, NPM, React, Material UI, ElectronJS

Work Flow:

Voice is recorded using ReactMic from frontend and converted to a file. Then the file is sent to backend server for processing and a message is got from backend . The message is displayed to user and processing is finished.

The UI is in form of a chat application with a mic button

Description of components:

ChatArea (ChatArea.jsx) - A layout where all messages will be placed.

Message (message.jsx) – Design to show one message.

MicButton (micbutton.jsx) – A button to switch microphone on and off.

Navbar (navbar.jsx) – A standard MUI navbar.

RecordingArea (recordingarea.jsx)- It is the part showing and controlling ReactMic it also controls making of voice files and sending backend requests and updating messages.

Typewriter. (typewriter.jsx) – To show any text with a typing effect.

Contexts Description

ChatContext – Stores all messages along with status of input and output globally

Backend Docmentation

The backend is created with Django and follows standard Django architecture.

It has following endpoints:

1) /converttotext
   It is used to invoke and check wheather backend is working properly or not. If working properly “Message sent” is returned else Network error is thrown at frontend

```
@csrf_exempt
def converToText(req):
    if (req.method=='POST'):
        # file=req.FILES['voice']
        # print(file)
        # default_storage.save("backend/"+file.name,file)
        # backend\newvoice672.webm
        # backend\newvoice672.webm
        # res=transcribe("backend/"+file.name)
      return JsonResponse({"text":"Message sent"})
```

2) /action
   It recieves the audio file from frontend and the uses the relevant functions to first convert the audio to text commands and then Perform os operations and generates a status report which is sent to frontend.

```
@csrf_exempt
def action(req):
    if (req.method=='POST'):
        file=req.FILES.get('voice')
        print(file)
        default_storage.save("backend/"+file.name,file)
        # backend\newvoice672.webm
        # backend\newvoice672.webm
        res=transcribe("backend/"+file.name)
        # os.remove("backend/"+file.name)
    # return JsonResponse({"text":res})
        print("RESPONSE",res)
        if not res:
           res=os.environ["pwd"]+",Executed without response."
    return JsonResponse({"status":res})
```

ML Model Implementation

Initial idea was to use a Sequence Model like LSTM(Long Short-Term Memory). But due to the fact that LSTMs being sequential, take a long time to produce an output and still yield below average accuracy in Speech-to-Text conversion, Transformers became a better choice.

Transformer Model Details:

A transformer is a deep learning architecture developed by Google and based on the multi-head attention mechanism, proposed in a 2017 paper "Attention Is All You Need". Text is converted to numerical representations called tokens, and each token is converted into a vector via looking up from a word embedding table. At each layer, each token is then contextualised within the scope of the context window with other (unmasked) tokens via a parallel multi-head attention mechanism allowing the signal for key tokens to be amplified and less important tokens to be diminished. The transformer paper, published in 2017, is based on the softmax-based attention mechanism proposed by Bahdanau et. al. in 2014 for machine translation, and the Fast Weight Controller, similar to a transformer, proposed in 1992.

Transformers have the advantage of having no recurrent units, and thus requires less training time than previous recurrent neural architectures, such as long short-term memory (LSTM), and its later variation has been prevalently adopted for training large language models (LLM) on large (language) datasets, such as the Wikipedia corpus and Common Crawl.

This architecture is now used not only in natural language processing and computer vision, but also in audio and multi-modal processing. It has also led to the development of pre-trained systems, such as generative pre-trained transformers (GPTs) and BERT (Bidirectional Encoder Representations from Transformers).

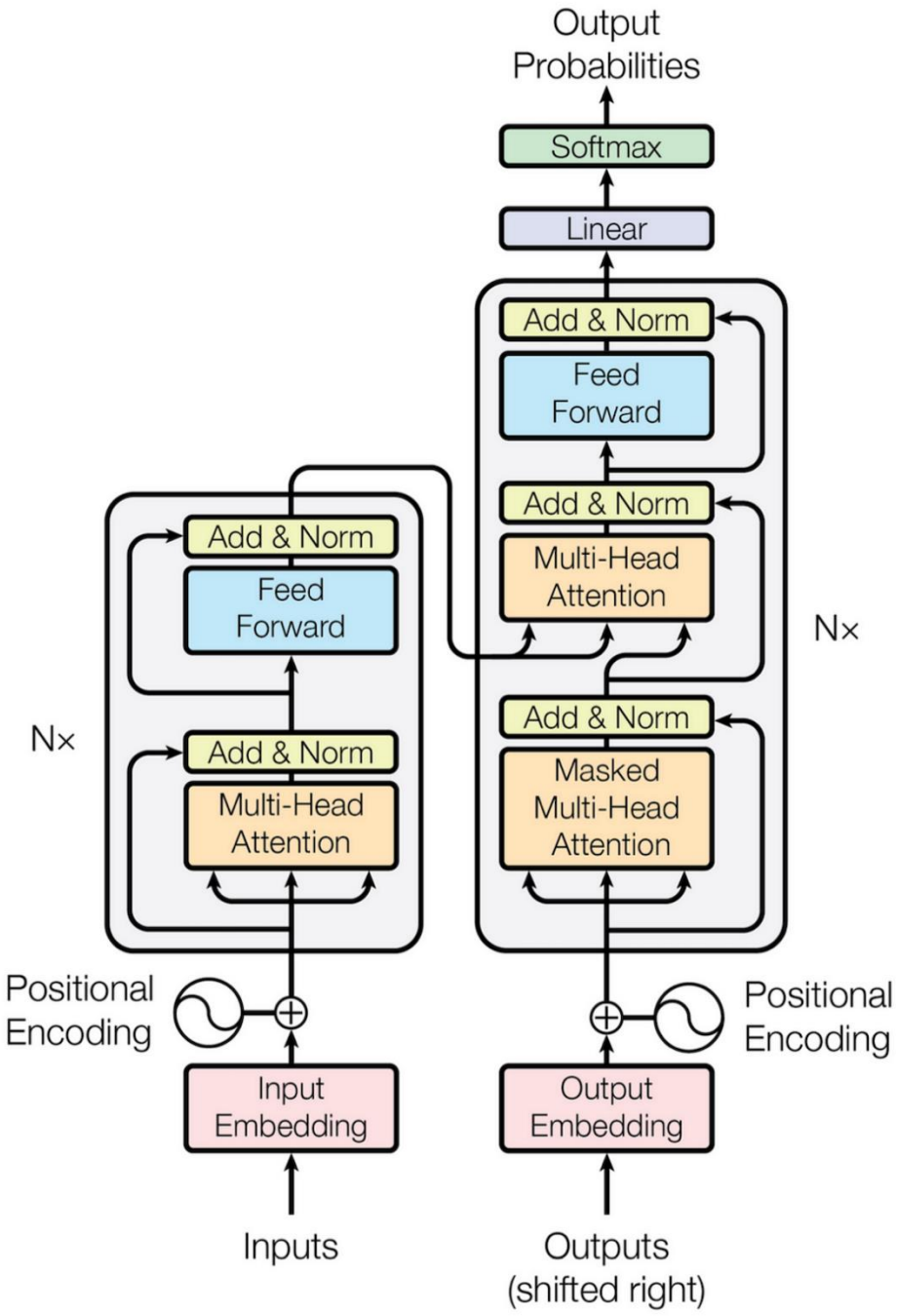

Figure 1: The Transformer - model architecture.

Transformer models are very taxing on the system and creating our own custom transformer model will lead to poor accuracy due to the lack of sufficient speech data in different accents and languages to train the model. Hence, the first approach was to try the pre-trained transformer models available on packages like Hugging Face or OpenAI. Two transformer models were tried out namely: ***Wav2Vec2*** and ***Whisper***.

The ***Wav2Vec2*** model produced an output in nearly the same time as the input recording(on Google Colab CPU) but had a WER(Word Error Rate) of around 40% - 60% which was simply not reliable enough for it to be mapped to the system call commands directly without excessive error correction.

The ***Whisper*** model on the other hand, had a respectable WER of 10% - 15%. But it takes nearly 3 times the duration of the input audio file(on Google Colab CPU) to produce the transcript. Given that an average audio input will be around 10 seconds, this was too long a delay to realistically implement in our system, especially when we have no clue about the computing capacity of the machine on which the program will be executed, which may lead to even higher delays. One option was to enable the use of GPU to predict the model output but that is not ideal since that will be machine-specific leading to vastly different efficiency on different machines.

The objective now was to use a tool which can transcribe the audio in the shortest time possible with a respectable WER. The final model implemented is ***Deepgram API***.

Tech Stack for Final ML Model:

The libraries and SDKs required to be installed on the system for the Deepgram API are as follows:

- deepgram-sdk — pip version 2.12.0
- deepgram module in python
- asyncio module in python
- json module in python
- os module in python

1. We install and import the dependencies above first.

Pre-requisites and required modules

```
! pip install deepgram-sdk==2.12.0 requests ffmpeg-python
from deepgram import Deepgram
import asyncio, json, os
```

2. We use an API key to connect to the Deepgram server and mention the file type and parameters of the model.

```
dg_key = '47751687a6d15fd18e646296e27333fdefb348dc'
dg = Deepgram(dg_key)
MIMETYPE = 'wav'
DIRECTORY = '.'

params = {
    "punctuate": True,
    "model": 'general',
    "tier": 'nova'
}
```

3. The audio file in the specified directory is opened and a json file is generated containing all the details of the audio transcript like the words in the sample, the WER of each word, timestamp of each word, etc.

```
def main():
    audio_folder = os.listdir(DIRECTORY)
    for audio_file in audio_folder:
        if audio_file.endswith(MIMETYPE):
          with open(f"{DIRECTORY}/{audio_file}", "rb") as f:
              source = {"buffer": f, "mimetype":'audio/'+MIMETYPE}
              res = dg.transcription.sync_prerecorded(source, params)
              with open(f"./{audio_file[:-4]}.json", "w") as transcript:
                  json.dump(res, transcript)
    return

main()
```

4. Only the required transcribed text is extracted from the json file and displayed as output.

```
OUTPUT = '/content/harvard.json'

def print_transcript(transcription_file):
  with open(transcription_file, "r") as file:
        data = json.load(file)
        result = data['results']['channels'][0]['alternatives'][0]['transcript']
        result = result.split('.')
        for sentence in result:
          print(sentence + '.')

print_transcript(OUTPUT)
```

The overview of the contents of the json file is as follows:


```
{"metadata": {"transaction_key": "deprecated", "request_id": "d6ef20a7-8835-4283-857a-08bfc9613ae1", "sha256": "971b4163670445c415c6b0fb681
    "created": "2024-04-03T14:47:50.628Z",
    "duration": 18.356188,
    "channels": 1,
    "models": ["aa274f3c-e8b3-456a-ac08-dfd797d45514"],
    "model_info": {"aa274f3c-e8b3-456a-ac08-dfd797d45514": {"name": "general-nova", "version": "2023-07-06.22746", "arch": "nova"}}},
    "results": {"channels": [{"alternatives": [{"transcript": "The stale smell of old beer lingers. It takes heat to bring out the odor, a
        "confidence": 0.9916992, "words": [{"word": "the", "start": 1.28, "end": 1.52, "confidence": 0.7451172, "punctuated_word": "The"},
        {"word": "smell", "start": 1.92, "end": 2.24, "confidence": 1.0, "punctuated_word": "smell"}, {"word": "of", "start": 2.24, "end":
```


Finally to implement this model, the required code was encapsulated into a function which takes the audio file path as input and returns the transcript text as output:

Speech-to-Text transcribe function

---

takes audio file path as input and returns the transcribed text as output

```
def transcribe(audio_file):

  dg_key = '47751687a6d15fd18e646296e27333fdefb348dc'
  dg = Deepgram(dg_key)
  MIMETYPE = 'webm'
  DIRECTORY = '.'

  params = {
      "punctuate": True,
      "model": 'general',
      "tier": 'nova'
  }

  if audio_file.endswith(MIMETYPE):
    with open(f"{DIRECTORY}/{audio_file}", "rb") as f:
      source = {"buffer": f, "mimetype":'audio/'+MIMETYPE}
      res = dg.transcription.sync_prerecorded(source, params)
      with open(f"./{audio_file[:-4]}.json", "w") as transcript:
        json.dump(res, transcript)

  transcript = f"./{audio_file[:-4]}.json"

  with open(transcript, "r") as file:
    data = json.load(file)
    result = data['results']['channels'][0]['alternatives'][0]['transcript']
    return result
```

Documentation

The File-Directory Management System

-e4stw1nd[Ankur Roy]

Initial Attempt(Not Used in Final Code)

Initially the plan was to store the output of the ML-model in a temporary file and then call the python file with the content of the file as parameters.

```
os.system("/usr/bin/python3 final.py < tmp.txt")
```

Inside the python file the code

```
import os
import sys
cmd=""
code=""
def joiner():
    global cmd
    m=len(sys.argv)
    for i in range(1,m):
        cmd=cmd+sys.argv[i]+" "
    cmd=cmd.lower()
def replacer():
    global cmd
    t=""
    for i in cmd:
        if (i==',' or i==";"):
            t=t+" "
        else:
            t=t+i
    cmd=t
joiner()
replacer()
```

```
def executor(cmd):
    cmd=cmd.split()
    global code
    if (cmd[0]=="read"):
        code="./read "+cmd[2]
    elif(cmd[0]=="create" and cmd[1]=="file"):
        code="./create "+cmd[2]
    elif(cmd[0]=="create" and cmd[1]=="folder"):
        code="./makedir "+cmd[2]
    elif(cmd[0]=="copy" and cmd[1]=="file"):
        code="./copy "+cmd[2]+" "+cmd[3]
    elif(cmd[0]=="copy" and cmd[1]=="folder"):
        code="./makedir "+cmd[2]
    ...[Other cases]
    os.system(code)
executor(cmd)
```

ISSUE WITH THE APPROACH

Since os.system() will launch a new sh shell by default everytime so the shell is amnesiac in nature and cannot remember the final state and directory of the system.

Rectified Approach

```
def init():
    os.environ["pwd"]="/home/kali"
    os.system("export PATH=/usr/local/sbin:/usr/local/bin:/us
init()
```

init() function initialises the environment variable “pwd” and set the path to location of the compiled codes.

```
def replacer(cmd):
    t=""
    for i in cmd:
        if (i==',' or i==";"):
            t=t+" "
        else:
            t=t+i
    cmd=t
    return(cmd)
```

replacer() replaces punctuation caused due to pause in the audio.

```
def executor(cmd):
    t=""
    cmd=replacer(cmd)
    cmd=cmd.split()
    global code
    # print("pwd:",os.environ["pwd"])
    if(len(cmd)>2):
        cmd[2]=os.environ["pwd"]+"/"+cmd[2]
    if len(cmd)>3:
        cmd[3]=os.environ["pwd"]+"/"+cmd[3]
    if (cmd[0]=="read"):
        code="/home/kali/os/OS-Project/file_bin/read "+cmd[2]
    elif(cmd[0]=="create" and cmd[1]=="file"):
        t="Copied"
        code="/home/kali/os/OS-Project/file_bin/create "+cmd[
    elif(cmd[0]=="create" and cmd[1]=="directory"):
        t="Created"
        code="/home/kali/os/OS-Project/file_bin/makedir "+cmd
    elif(cmd[0]=="copy" and cmd[1]=="file"):
        t="Copied"
        code="/home/kali/os/OS-Project/file_bin/copy "+cmd[2]
    elif(cmd[0]=="copy" and cmd[1]=="directory"):
        t="Copied"
```

```
        code="/home/kali/os/OS-Project/file_bin/copy -r "+cmd
    elif(cmd[0]=="rename" ):
        t="Renamed"
        code="/home/kali/os/OS-Project/file_bin/move "+cmd[2]
    elif(cmd[0]=="remove" and cmd[1]=="file"):
        t="Removed"
        code="/home/kali/os/OS-Project/file_bin/remove "+cmd[
    elif(cmd[0]=="remove" and cmd[1]=="directory"):
        t="Removed"
        code="/home/kali/os/OS-Project/file_bin/remove -rf "+
    elif(cmd[0]=="change"):

        os.environ["pwd"]=cmd[2]

    elif(cmd[0]=="go"):
        os.environ["pwd"]=os.environ["pwd"]+"/.."
        print("backked")
    elif(cmd[0]=="list"):
        code="/home/kali/os/OS-Project/file_bin/list "+os.env
    else:
        print(cmd)
        return('Error')
    try:
        if(code!=""):
            print(code+" pwd:"+os.environ["pwd"])
            os.system(code+" >tmp.txt")
            code=""

            return(open("tmp.txt","r").read())

        code=""

    except:
        print(code)
        code=""
        return('Error')
```

Finally executor() reads the environment variable “pwd” and converts all relative path to absolute path, executes the commands based on the case ans store the result to tmp.txt.

Instead of executing python file for every audio clip , I have integrated the file manipulation binary within the django and ML-Model function itself.

Advantages:

Removes the necessity of calling the python file every time , substantially reducing the workload.

As it stores the present working directory in an environment variable hence it ensures that the final state and directory remains consistent.