Rapidfire.py

from collections import Counter

from transformers import pipeline

from groq import Groq

from collections import Counter

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

import json

import numpy as np

from .audio\_process import process\_audio\_upload, convert\_wav\_to\_mp3, transcribe\_audio, save\_uploaded\_file

def generate\_incomplete\_analogy():

    client = Groq(api\_key="gsk\_uGsCULmfXTX6NI2qP2hQWGdyb3FYhFZD59hstrxgvCdDkM5uFEPT")

    completion = client.chat.completions.create(

        model="llama-3.3-70b-versatile",

        messages=[

            {

                "role": "system",

                "content": (

                    "You are a helpful AI Assistant. "

                    "Generate a single incomplete analogy prompt. "

                    "For example: 'Learning is like', 'Love is like'. "

                    "Output exactly one incomplete analogy without any additional words."

                )

            },

            {

            "role": "user",

            "content": "give a random incomplete anology",

            }

        ],

        temperature=2,

        frequency\_penalty=0.0,

        max\_completion\_tokens=1024,

        top\_p=1,

        stream=True,

        stop=None,

    )

    print(completion)

    # Handling the streamed output

    response\_content = ""

    for chunk in completion:

        response\_content = response\_content+(chunk.choices[0].delta.content or "")

        print(chunk.choices[0].delta.content or "", end="")

    return response\_content

def extract\_topic(transcript):

    # Initialize english stopwords

    english\_stopwords = stopwords.words("english")

    #convert article to tokens

    tokens = word\_tokenize(transcript)

    #extract alpha words and convert to lowercase

    alpha\_lower\_tokens = [word.lower() for word in tokens if word.isalpha()]

    #remove stopwords

    alpha\_no\_stopwords = [word for word in alpha\_lower\_tokens if word not in english\_stopwords]

    #Count word

    BoW = Counter(alpha\_no\_stopwords)

    #Most common words

    return BoW.most\_common(3)

def score\_analogy\_with\_groq(transcript, incomplete\_analogy):

    """

    Use Groq to evaluate analogy relevance and creativity based on the transcript

    and the incomplete analogy prompt.

    Parameters:

        transcript (str): The transcribed text from the user's response

        incomplete\_analogy (str): The incomplete analogy prompt (e.g., "Success is like")

    Returns:

        dict: A dictionary with analogy\_relevance and creativity scores

    """

    client = Groq(api\_key="gsk\_uGsCULmfXTX6NI2qP2hQWGdyb3FYhFZD59hstrxgvCdDkM5uFEPT")

    prompt = f"""

    Below is an incomplete analogy prompt and a user's spoken response to complete it.

    Incomplete Analogy: "{incomplete\_analogy}"

    User's Response: "{transcript}"

    Please evaluate the response based on two criteria:

    1. Relevance (0-10): How well does the response connect to the analogy prompt?

       Does it create a clear and appropriate comparison?

    2. Creativity (0-10): How original, insightful, or thought-provoking is the analogy?

       Does it provide a fresh perspective or use unexpected connections?

    Return your evaluation as a JSON object with two properties:

    - analogy\_relevance: a number between 0 and 10 (with up to 2 decimal places)

    - creativity: a number between 0 and 10 (with up to 2 decimal places)

    Response must be in this exact JSON format and nothing else:

    {{

        "analogy\_relevance": 0.00,

        "creativity": 0.00

    }}

    """

    completion = client.chat.completions.create(

        model="llama-3.3-70b-versatile",

        messages=[

            {

                "role": "system",

                "content": "You are an AI assistant that evaluates analogies based on relevance and creativity. Provide numeric scores only."

            },

            {

                "role": "user",

                "content": prompt

            }

        ],

        temperature=0.8,

        max\_completion\_tokens=256

    )

    response\_content = completion.choices[0].message.content

    response\_dict = json.loads(response\_content)

    return {

        "analogy\_relevance": round(response\_dict['analogy\_relevance'], 2),

        "creativity": round(response\_dict["creativity"], 2)

    }

def process\_rapidfire\_audio(audio\_file\_path, analogy):

    result = process\_audio\_upload(audio\_file\_path)

    segments = result.get("segments", [])

    transcript = result.get("text", "")

    # Calculate word-level timestamps

    word\_timestamps = []

    previous\_word\_end = None

    gaps = []  # store gaps between consecutive words

    for segment in segments:

        seg\_start = segment["start"]

        seg\_end = segment["end"]

        seg\_text = segment["text"].strip()

        words = seg\_text.split()

        num\_words = len(words)

        if num\_words == 0:

            continue

        duration = seg\_end - seg\_start

        word\_duration = duration / num\_words

        for i, word in enumerate(words):

            word\_start = seg\_start + i \* word\_duration

            word\_end = word\_start + word\_duration

            word\_timestamps.append({

                "word": word,

                "start": round(word\_start, 2),

                "end": round(word\_end, 2)

            })

            if previous\_word\_end is not None:

                gap = word\_start - previous\_word\_end

                gaps.append(gap)

            previous\_word\_end = word\_end

    if transcript.strip() == "":

        speech\_continuity = 0

    else:

        if gaps:

            gaps\_array = np.array(gaps)

            avg\_gap = np.mean(gaps\_array)

            std\_gap = np.std(gaps\_array)

            # Penalize both large average gaps and inconsistent timing

            speech\_continuity = max(0, 10 - (avg\_gap \* 5) - (std\_gap \* 3))

        else:

            speech\_continuity = 10.0

    # Score analogy relevance and creativity using Groq

    analogy\_scores = score\_analogy\_with\_groq(transcript, analogy)

    analogy\_relevance = analogy\_scores["analogy\_relevance"]

    creativity = analogy\_scores["creativity"]

    # Extract text topic using NLP

    text\_topic = extract\_topic(transcript)

    # Calculate overall score

    overall\_rapidfire\_score = (speech\_continuity + analogy\_relevance + creativity) / 3

    metrics = {

        "speech\_continuity": round(speech\_continuity, 2),

        "analogy\_relevance": analogy\_relevance,

        "creativity": creativity,

        "overall\_rapidfire\_score": round(overall\_rapidfire\_score, 2),

        "text\_topic": text\_topic

    }

    return {

        "transcript": transcript,

        "word\_timestamps": word\_timestamps,

        "metrics": metrics,

        "generated\_analogy": analogy

    }

Triplestep.py

import os

from pydub import AudioSegment

import whisper

import random

from groq import Groq

import nltk

from sentence\_transformers import SentenceTransformer, util

from nltk.tokenize import sent\_tokenize

import numpy as np

import nltk

from nltk.tokenize import word\_tokenize

import nltk

from groq import Groq

def analyze\_distractor\_smoothness(transcript: str, distractor\_words: list) -> float:

    """

    Analyze how smoothly distractor words are integrated in the transcript.

    For each sentence containing any distractor word (case-insensitive), compute the cosine similarity

    between that sentence and its adjacent sentences (previous and next). A higher similarity indicates smoother integration.

    Returns a smoothness score between 0 and 10. If no sentence contains a distractor word, returns 10.0.

    """

    sentences = sent\_tokenize(transcript)

    if not sentences:

        return 0.0

    model = SentenceTransformer('all-mpnet-base-v2')

    distracted\_similarities = []

    for i, sentence in enumerate(sentences):

        if any(dw.lower() in sentence.lower() for dw in distractor\_words):

            neighbor\_sims = []

            # Compute similarity with previous sentence if exists

            if i > 0:

                try:

                    sim\_prev = util.cos\_sim(

                        model.encode(sentence, convert\_to\_tensor=True),

                        model.encode(sentences[i-1].strip(), convert\_to\_tensor=True)

                    ).item()

                    neighbor\_sims.append(sim\_prev)

                except Exception as e:

                    print("Error computing similarity with previous sentence:", e)

            # Compute similarity with next sentence if exists

            if i < len(sentences) - 1:

                try:

                    sim\_next = util.cos\_sim(

                        model.encode(sentence, convert\_to\_tensor=True),

                        model.encode(sentences[i+1].strip(), convert\_to\_tensor=True)

                    ).item()

                    neighbor\_sims.append(sim\_next)

                except Exception as e:

                    print("Error computing similarity with next sentence:", e)

            if neighbor\_sims:

                distracted\_similarities.append(np.mean(neighbor\_sims))

    if not distracted\_similarities:

        return 10.0

    avg\_similarity = np.mean(distracted\_similarities)

    smoothness\_score = avg\_similarity \* 10

    smoothness\_score = max(0, min(smoothness\_score, 10))

    return round(smoothness\_score, 2)

def analyze\_topic\_adherence(transcript: str, expected\_topic: str) -> float:

    """

    Analyze topic adherence by:

      1. Splitting the transcript into sentences.

      2. Computing embeddings for each non-empty sentence using SentenceTransformer.

      3. Computing the embedding for the expected topic.

      4. Calculating the cosine similarity between each sentence and the expected topic.

      5. Aggregating these similarities into a global topic adherence score (0-10).

    Returns a score between 0 and 10.

    """

    sentences = sent\_tokenize(transcript)

    if not sentences:

        return 0.0

    model = SentenceTransformer('all-mpnet-base-v2')

    try:

        topic\_embedding = model.encode(expected\_topic, convert\_to\_tensor=True)

    except Exception as e:

        print("Error encoding expected\_topic:", e)

        return 0.0

    similarities = []

    for sentence in sentences:

        sentence = sentence.strip()

        if not sentence:

            continue

        try:

            sentence\_embedding = model.encode(sentence, convert\_to\_tensor=True)

        except Exception as e:

            print("Error encoding sentence:", sentence, e)

            continue

        # Check if embedding is valid (non-empty)

        if sentence\_embedding is None or sentence\_embedding.shape[0] == 0:

            print("Empty embedding for sentence:", sentence)

            continue

        try:

            cos\_sim = util.cos\_sim(topic\_embedding, sentence\_embedding).item()

        except Exception as e:

            print("Error computing cosine similarity for sentence:", sentence, e)

            continue

        similarities.append(cos\_sim)

    if not similarities:

        return 0.0

    avg\_similarity = np.mean(similarities)

    score = avg\_similarity \* 10

    score = max(0, min(score, 10))

    return round(score, 2)

def analyze\_coherence(transcript):

    """

    Analyze speech coherence by:

      1. Segmenting the transcript into sentences.

      2. Embedding each sentence using Sentence-BERT.

      3. Computing cosine similarity between adjacent sentence embeddings.

      4. Aggregating these similarities into a global coherence score.

    Returns:

      - coherence\_score: A score (0-10) representing global coherence.

      - sentences: List of segmented sentences.

      - similarities: List of cosine similarities between adjacent sentences.

    """

    # Sentence Segmentation

    sentences = nltk.sent\_tokenize(transcript)

    if len(sentences) < 2:

        return 10.0, sentences, []

    # Embed Sentences with Sentence-BERT

    model = SentenceTransformer('all-mpnet-base-v2')

    sentence\_embeddings = model.encode(sentences, convert\_to\_tensor=True)

    # Compute Cosine Similarity Between Adjacent Sentences

    similarities = []

    for i in range(len(sentences) - 1):

        cos\_sim = util.cos\_sim(sentence\_embeddings[i], sentence\_embeddings[i+1]).item()

        similarities.append(cos\_sim)

    # Aggregate Similarities: Compute the average similarity

    avg\_similarity = np.mean(similarities)

    coherence\_score = avg\_similarity \* 10

    return coherence\_score

def save\_uploaded\_file(audio\_file\_path, destination\_path):

    with open(audio\_file\_path, 'rb') as in\_file:

        data = in\_file.read()

    with open(destination\_path, 'wb') as out\_file:

        out\_file.write(data)

def convert\_wav\_to\_mp3(wav\_file, mp3\_file="output.mp3"):

    try:

        audio = AudioSegment.from\_wav(wav\_file)

        audio.export(mp3\_file, format="mp3")

        print(f"Converted {wav\_file} to {mp3\_file}")

        return mp3\_file

    except Exception as e:

        print("Error converting WAV to MP3:", e)

        raise

def transcribe\_audio(mp3\_file):

    print("Loading Whisper model...")

    model = whisper.load\_model("small.en")

    print("Transcribing audio...")

    result = model.transcribe(mp3\_file)

    return result["text"]

def process\_audio\_upload(audio\_file\_path):

    wav\_path = "temp\_output.wav"

    mp3\_path = "temp\_output.mp3"

    try:

        print("Converting webm to wav...")

        audio\_segment = AudioSegment.from\_file(audio\_file\_path, format="webm")

        audio\_segment.export(wav\_path, format="wav")

    except Exception as e:

        print("Error converting webm to wav:", e)

        raise

    convert\_wav\_to\_mp3(wav\_path, mp3\_path)

    transcript = transcribe\_audio(mp3\_path)

    for file in [wav\_path, mp3\_path]:

        if os.path.exists(file):

            os.remove(file)

    return transcript

def generate\_topics():

    """

    Generate a main speaking topic and contextually relevant distractor words.

    The response from the model is expected to contain lines like:

        Main Topic: <topic text>

        Distractor Words: <word1>, <word2>, <word3>

    Returns:

        dict: {"main\_topic": <str>, "distractor\_words": [<str>, ...]}

    """

    client = Groq(api\_key="gsk\_uGsCULmfXTX6NI2qP2hQWGdyb3FYhFZD59hstrxgvCdDkM5uFEPT")

    completion = client.chat.completions.create(

        model="llama-3.3-70b-versatile",

        messages=[

            {

                "role": "system",

                "content": ("You are a helpful AI Assistant. You should generate and display a main speaking topic, like Cars tourism travelling  "

                            "and then generate contextually relevant distractor words related to that topic. "

                            "Please output the result in the following format exactly:\n\n"

                            "Main Topic: <your topic here>\n"

                            "Distractor Words: <word1>, <word2>, <word3>")

            },

            {

                "role": "user",

                "content": "Give a main topic and some distractor words."

            }

        ],

        temperature=2,

        frequency\_penalty=0.0,

        max\_completion\_tokens=1024,

        top\_p=1,

        stream=True,

        stop=None,

    )

    response\_content = ""

    for chunk in completion:

        # Each chunk's delta content may be None; we append if present.

        delta\_text = chunk.choices[0].delta.content or ""

        response\_content += delta\_text

        print(delta\_text, end="")

    main\_topic = ""

    distractor\_words = []

    lines = response\_content.splitlines()

    for line in lines:

        if line.lower().startswith("main topic:"):

            main\_topic = line.split(":", 1)[1].strip()

        elif line.lower().startswith("distractor words:"):

            words\_str = line.split(":", 1)[1].strip()

            distractor\_words = [w.strip() for w in words\_str.split(",") if w.strip()]

    return {"main\_topic": main\_topic, "distractor\_words": distractor\_words}

def process\_triple\_step\_audio(audio\_file\_path, main\_topic, distractor\_words):

    transcript = process\_audio\_upload(audio\_file\_path)

    # evaluation scores

    coherence\_result = analyze\_coherence(transcript)

    if isinstance(coherence\_result, tuple):

        coherence\_score = coherence\_result[0]

    else:

        coherence\_score = coherence\_result

    topic\_adherence\_score = analyze\_topic\_adherence(transcript, main\_topic)

    distraction\_handling\_score = analyze\_distractor\_smoothness(transcript,distractor\_words)

    overall\_triple\_step\_score = (coherence\_score + topic\_adherence\_score + distraction\_handling\_score) / 3.0

    result = {

        "main\_topic": main\_topic,

        "distractor\_words": distractor\_words,

        "transcript": transcript,

        "coherence\_score": coherence\_score,

        "topic\_adherence\_score": topic\_adherence\_score,

        "distraction\_handling\_score": distraction\_handling\_score,

        "overall\_triple\_step\_scrore": overall\_triple\_step\_score

    }

    return result

Conductor.py

import os

from pydub import AudioSegment

import whisper

from groq import Groq

import librosa

import numpy as np

import nltk

import joblib

def save\_uploaded\_file(audio\_file\_path, destination\_path):

    with open(audio\_file\_path, 'rb') as in\_file:

        data = in\_file.read()

    with open(destination\_path, 'wb') as out\_file:

        out\_file.write(data)

def convert\_wav\_to\_mp3(wav\_file, mp3\_file="output.mp3"):

    try:

        audio = AudioSegment.from\_wav(wav\_file)

        audio.export(mp3\_file, format="mp3")

        print(f"Converted {wav\_file} to {mp3\_file}")

        return mp3\_file

    except Exception as e:

        print("Error converting WAV to MP3:", e)

        raise

def transcribe\_audio(mp3\_file):

    print("Loading Whisper model...")

    model = whisper.load\_model("small.en")

    print("Transcribing audio...")

    result = model.transcribe(mp3\_file)

    return result["text"]

def process\_audio\_upload(audio\_file\_path):

    wav\_path = "temp\_output.wav"

    mp3\_path = "temp\_output.mp3"

    try:

        print("Converting webm to wav...")

        audio\_segment = AudioSegment.from\_file(audio\_file\_path, format="webm")

        audio\_segment.export(wav\_path, format="wav")

    except Exception as e:

        print("Error converting webm to wav:", e)

        raise

    convert\_wav\_to\_mp3(wav\_path, mp3\_path)

    transcript = transcribe\_audio(mp3\_path)

    for file in [wav\_path, mp3\_path]:

        if os.path.exists(file):

            os.remove(file)

    return transcript

def generate\_conductor\_exercise():

    """

    Generate instructions for an exercise to improve vocal variety and expression.

    The response from the model is expected to contain lines like:

        Energy Levels: <energy\_level1>, <energy\_level2>, <energy\_level3>

        Moods: <mood1>, <mood2>, <mood3>

        Improvement Suggestions: <suggestion1>, <suggestion2>, <suggestion3>

    Returns:

        dict: {

            "energy\_levels": [<str>, ...],

            "moods": [<str>, ...],

            "improvement\_suggestions": [<str>, ...]

        }

    """

    client = Groq(api\_key="gsk\_uGsCULmfXTX6NI2qP2hQWGdyb3FYhFZD59hstrxgvCdDkM5uFEPT")

    completion = client.chat.completions.create(

        model="llama-3.3-70b-versatile",

        messages=[

            {

                "role": "system",

                "content": (

                    "You are a helpful AI Assistant. Generate an exercise prompt for improving vocal variety and expression. "

                    "The exercise should guide users through different energy levels and moods, and include instructions for real-time voice analysis "

                    "to track energy levels, analyze vocal variety, provide instant feedback on mood matching, and generate personalized improvement suggestions. "

                    "Please output the result in the following format exactly:\n\n"

                    "Energy level should only be High, Medium, or Low.\n"

                    "Mood should only be Joy, Sadness, Fear, Anger, Surprise, Neutral, Disgust, or Shame.\n"

                    "Energy Levels: <energy\_level1>, <energy\_level2>, <energy\_level3>\n"

                    "Moods: <mood1>, <mood2>, <mood3>\n"

                    "Improvement Suggestions: <suggestion1>, <suggestion2>, <suggestion3>\n\n"

                    "Note: Moods must be chosen only from the following values: joy, sadness, fear, anger, surprise, neutral, disgust, shame."

                )

            },

            {

                "role": "user",

                "content": "Generate an exercise prompt for improving vocal variety and expression."

            }

        ],

        temperature=0.5,

        frequency\_penalty=0.0,

        max\_completion\_tokens=1024,

        top\_p=1,

        stream=True,

        stop=None,

    )

    response\_content = ""

    for chunk in completion:

        # Append each delta's content (if present) to the response\_content.

        delta\_text = chunk.choices[0].delta.content or ""

        response\_content += delta\_text

        print(delta\_text, end="")  # print for debugging

    energy\_levels = []

    moods = []

    improvement\_suggestions = []

    lines = response\_content.splitlines()

    for line in lines:

        if line.lower().startswith("energy levels:"):

            levels\_str = line.split(":", 1)[1].strip()

            energy\_levels = [lvl.strip() for lvl in levels\_str.split(",") if lvl.strip()]

        elif line.lower().startswith("moods:"):

            moods\_str = line.split(":", 1)[1].strip()

            moods = [m.strip() for m in moods\_str.split(",") if m.strip()]

        elif line.lower().startswith("improvement suggestions:"):

            suggestions\_str = line.split(":", 1)[1].strip()

            improvement\_suggestions = [s.strip() for s in suggestions\_str.split(",") if s.strip()]

    return {

        "energy\_levels": energy\_levels,

        "moods": moods,

        "improvement\_suggestions": improvement\_suggestions

    }

def remove\_adjacent\_duplicates(seq):

    if not seq:

        return []

    result = [seq[0]]

    for item in seq[1:]:

        if item != result[-1]:

            result.append(item)

    return result

def score\_sequence\_match(audio\_sequence, target\_sequence):

    # Remove adjacent duplicates

    audio\_sequence = remove\_adjacent\_duplicates(audio\_sequence)

    if not target\_sequence or not audio\_sequence:

        return 0.0

    # Convert categorical values to numerical

    energy\_map = {"low": 0, "medium": 1, "high": 2}

    # Convert sequences to numerical values

    num\_audio = [energy\_map.get(level.lower(), 1) for level in audio\_sequence]

    num\_target = [energy\_map.get(level.lower(), 1) for level in target\_sequence]

    # This allows for partial matching and timing flexibility

    max\_score = len(target\_sequence)

    score = 0.0

    i, j = 0, 0

    while i < len(num\_audio) and j < len(num\_target):

        # Exact match

        if num\_audio[i] == num\_target[j]:

            score += 1.0

        # Close match (off by one level)

        elif abs(num\_audio[i] - num\_target[j]) == 1:

            score += 0.5

        # Move forward in sequences

        if i < len(num\_audio) - 1 and j < len(num\_target) - 1:

            # Determine which sequence to advance

            if num\_audio[i+1] == num\_target[j]:

                i += 1

            elif num\_audio[i] == num\_target[j+1]:

                j += 1

            else:

                i += 1

                j += 1

        else:

            i += 1

            j += 1

    # Normalize to 0-10 scale

    final\_score = (score / max\_score) \* 10

    return max(round(final\_score, 2),10)

def analyze\_mood\_matches(transcript, target\_moods\_list):

    allowed\_moods = ["joy", "sadness", "fear", "anger", "surprise", "neutral", "disgust", "shame"]

    sentences = nltk.sent\_tokenize(transcript)

    results = []

    current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

    pipeline\_path = os.path.join(current\_dir, "emotion\_classifier\_pipe\_lr.pkl")

    # Verify file exists

    if not os.path.exists(pipeline\_path):

        raise FileNotFoundError(f"Classifier pipeline not found at: {pipeline\_path}")

    # Load the classifier

    with open(pipeline\_path, "rb") as pipeline\_file:

        loaded\_pipe\_lr = joblib.load(pipeline\_file)

    # Related mood groups for partial matching

    related\_moods = {

        "joy": ["surprise"],

        "sadness": ["shame", "disgust"],

        "fear": ["surprise", "shame"],

        "anger": ["disgust"],

        "surprise": ["joy", "fear"],

        "neutral": [],

        "disgust": ["anger", "sadness"],

        "shame": ["sadness", "fear"]

    }

    total\_score = 0.3

    total\_sentences = 1

    for i, sentence in enumerate(sentences):

        sentence = sentence.strip()

        if not sentence:

            continue

        total\_sentences += 1

        # Get emotion prediction for this sentence

        predicted\_emotion = loaded\_pipe\_lr.predict([sentence])[0].lower()

        # If we have a target mood for this sentence

        if target\_moods\_list and i < len(target\_moods\_list):

            expected\_mood = target\_moods\_list.lower()

            # Exact match

            if predicted\_emotion in expected\_mood:

                total\_score += 1.0

            # Related mood (partial match)

            elif predicted\_emotion in related\_moods.get(expected\_mood, []):

                total\_score += 0.5

            # Check if expected mood is in the related moods of predicted emotion

            elif expected\_mood in related\_moods.get(predicted\_emotion, []):

                total\_score += 0.3

        results.append({

            "sentence": sentence,

            "predicted\_mood": predicted\_emotion

        })

    # Normalize score to 0-10 scale

    print("MOOD sequence", results)

    mood\_score = (total\_score/ total\_sentences) \* 10

    return max(round(mood\_score, 2),10)

def get\_energy\_level\_sequence(audio\_path, sr=22050, segment\_duration=1.0):

    # Load audio

    y, sr = librosa.load(audio\_path, sr=sr)

    # Compute RMS energy over frames

    rms = librosa.feature.rms(y=y)[0]

    hop\_length = 512  # default hop\_length in librosa.feature.rms

    times = librosa.frames\_to\_time(np.arange(len(rms)), sr=sr, hop\_length=hop\_length)

    # Divide audio into segments of 'segment\_duration' seconds.

    max\_time = times[-1]

    num\_segments = int(np.ceil(max\_time / segment\_duration))

    segment\_energies = []

    for i in range(num\_segments):

        start\_time = i \* segment\_duration

        end\_time = (i + 1) \* segment\_duration

        indices = np.where((times >= start\_time) & (times < end\_time))[0]

        if len(indices) == 0:

            avg\_energy = 0.0

        else:

            avg\_energy = np.mean(rms[indices])

        segment\_energies.append(avg\_energy)

    segment\_energies = np.array(segment\_energies)

    # Define thresholds using quantiles

    low\_threshold = np.quantile(segment\_energies, 0.33)

    high\_threshold = np.quantile(segment\_energies, 0.66)

    # Map each segment's average energy to a category.

    energy\_sequence = []

    for energy in segment\_energies:

        if energy < low\_threshold:

            energy\_sequence.append("low")

        elif energy < high\_threshold:

            energy\_sequence.append("medium")

        else:

            energy\_sequence.append("high")

    return energy\_sequence

def generate\_targeted\_suggestions(energy\_score, mood\_score, audio\_sequence, target\_sequence):

    """

    Generate targeted improvement suggestions based on actual performance

    """

    suggestions = []

    # Energy-related suggestions

    if energy\_score < 5.0:

        if len(set(audio\_sequence)) <= 1:

            suggestions.append("Try using more varied energy levels - your delivery was mostly at one level")

        else:

            # Check if specific energy levels are missing

            audio\_levels = set(level.lower() for level in audio\_sequence)

            target\_levels = set(level.lower() for level in target\_sequence)

            missing\_levels = target\_levels - audio\_levels

            if "high" in missing\_levels:

                suggestions.append("Work on incorporating higher energy moments in your delivery")

            if "low" in missing\_levels:

                suggestions.append("Practice including quieter, more intimate moments in your delivery")

    # Mood-related suggestions

    if mood\_score < 5.0:

        suggestions.append("Focus on matching your vocal tone to the intended emotion")

        suggestions.append("Try exaggerating the emotional quality to make it more recognizable")

    # General suggestions if we don't have enough targeted ones

    if len(suggestions) < 2:

        general\_suggestions = [

            "Record yourself and listen back to identify subtle mood inconsistencies",

            "Try mirroring professional speakers to develop better vocal variety",

            "Work on maintaining consistent volume while varying your pitch and pace"

        ]

        # Add general suggestions until we have at least 2

        while len(suggestions) < 2 and general\_suggestions:

            suggestions.append(general\_suggestions.pop(0))

    return suggestions[:2]  # Return top 2 suggestions

def process\_conductor\_audio(audio\_file\_path, instructions, energy\_levels, moods):

    """

    Process the Conductor exercise audio with improved scoring logic

    """

    transcript = process\_audio\_upload(audio\_file\_path)

    # Get energy sequence from audio

    audio\_sequence = get\_energy\_level\_sequence(audio\_file\_path, segment\_duration=1.0)

    # Calculate energy level score

    energy\_level\_score = score\_sequence\_match(audio\_sequence, energy\_levels)

    # Calculate mood score

    mood\_match\_score = analyze\_mood\_matches(transcript, moods)

    # Generate personalized improvement suggestions based on actual scores

    improvement\_suggestions = generate\_targeted\_suggestions(

        energy\_level\_score,

        mood\_match\_score,

        audio\_sequence,

        energy\_levels

    )

    # Calculate overall score

    overall\_conductor\_score = (energy\_level\_score + mood\_match\_score) / 2

    result = {

        "transcript": transcript,

        "instructions": instructions,

        "energy\_levels": energy\_levels,

        "moods": moods,

        "energy\_level\_score": energy\_level\_score,

        "mood\_match\_score": mood\_match\_score,

        "overall\_conductor\_score": overall\_conductor\_score,

        "improvement\_suggestions": improvement\_suggestions

    }

    return result

xp\_system.py

import math

from django.db.models import Sum

from exercises.models import RapidFire, TripleStep, Conductor

def calculate\_user\_xp(user):

    rapidfire\_xp = RapidFire.objects.filter(user=user).aggregate(total=Sum('overall\_rapidfire\_score'))['total'] or 0

    triplestep\_xp = TripleStep.objects.filter(user=user).aggregate(total=Sum('overall\_triple\_step\_scrore'))['total'] or 0

    conductor\_xp = Conductor.objects.filter(user=user).aggregate(total=Sum('overall\_conductor\_score'))['total'] or 0

    total\_xp = rapidfire\_xp + triplestep\_xp + conductor\_xp

    return total\_xp

def calculate\_user\_level(xp, xp\_per\_level=50):

    level = math.floor(xp / xp\_per\_level) + 1

    return level

audio\_process.py  
import os

from pydub import AudioSegment

import whisper

import random

def save\_uploaded\_file(audio\_file\_path, destination\_path):

    with open(audio\_file\_path, 'rb') as in\_file:

        data = in\_file.read()

    with open(destination\_path, 'wb') as out\_file:

        out\_file.write(data)

def convert\_wav\_to\_mp3(wav\_file, mp3\_file="output.mp3"):

    try:

        audio = AudioSegment.from\_wav(wav\_file)

        audio.export(mp3\_file, format="mp3")

        print(f"Converted {wav\_file} to {mp3\_file}")

        return mp3\_file

    except Exception as e:

        print("Error converting WAV to MP3:", e)

        raise

def transcribe\_audio(mp3\_file):

    print("Loading Whisper model...")

    model = whisper.load\_model("small.en")

    print("Transcribing audio...")

    result = model.transcribe(mp3\_file)

    return result

def process\_audio\_upload(audio\_file\_path):

    wav\_path = "temp\_output.wav"

    mp3\_path = "temp\_output.mp3"

  try:

        print("Converting webm to wav...")

        audio\_segment = AudioSegment.from\_file(audio\_file\_path, format="webm")

        audio\_segment.export(wav\_path, format="wav")

    except Exception as e:

        print("Error converting webm to wav:", e)

        raise

    convert\_wav\_to\_mp3(wav\_path, mp3\_path)

    transcript = transcribe\_audio(mp3\_path)

    for file in [wav\_path, mp3\_path]:

        if os.path.exists(file):

            os.remove(file)

    return transcript

Emotion Detection in Text.ipynb

# EDA

import pandas as pd

import numpy as np

# Load Data Viz Pkgs

import seaborn as sns

# Load Text Cleaning Pkgs

import neattext.functions as nfx

# Load ML Pkgs

# Estimators

from sklearn.linear\_model import LogisticRegression

from sklearn.naive\_bayes import MultinomialNB

# Transformers

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score,classification\_report,confusion\_matrix

# Load Dataset

df = pd.read\_csv("emotion\_dataset\_raw.csv")

df.head()

# Value Counts

df['Emotion'].value\_counts()

# Plot

sns.countplot(x='Emotion',data=df)

# Data Cleaning

dir(nfx)

# User handles

df['Clean\_Text'] = df['Text'].apply(nfx.remove\_userhandles)

# Stopwords

df['Clean\_Text'] = df['Clean\_Text'].apply(nfx.remove\_stopwords)

df

# Features & Labels

Xfeatures = df['Clean\_Text']

ylabels = df['Emotion']

# Split Data

x\_train,x\_test,y\_train,y\_test = train\_test\_split(Xfeatures,ylabels,test\_size=0.3,random\_state=42)

# Build Pipeline

from sklearn.pipeline import Pipeline

# LogisticRegression Pipeline

pipe\_lr = Pipeline(steps=[('cv',CountVectorizer()),('lr',LogisticRegression())])

# Train and Fit Data

pipe\_lr.fit(x\_train,y\_train)

pipe\_lr

# Check Accuracy

pipe\_lr.score(x\_test,y\_test)

# Make A Prediction

ex1 = "This book was so interesting it made me happy"

pipe\_lr.predict([ex1])

# Prediction Prob

pipe\_lr.predict\_proba([ex1])

pipe\_lr.classes\_

# Save Model & Pipeline

import joblib

pipeline\_file = open("emotion\_classifier\_pipe\_lr.pkl","wb")

joblib.dump(pipe\_lr,pipeline\_file)

pipeline\_file.close()

# Load the saved pipeline

pipeline\_file = open("emotion\_classifier\_pipe\_lr.pkl", "rb")

loaded\_pipe\_lrr = joblib.load(pipeline\_file)

pipeline\_file.close()

# Example prediction

ex2 = "I feel soo sad and lonely today."

predicted\_emotion = loaded\_pipe\_lrr.predict([ex2])

predicted\_proba = loaded\_pipe\_lrr.predict\_proba([ex2])

# Output results

print(f"Predicted Emotion: {predicted\_emotion[0]}")

print(f"Prediction Probabilities: {predicted\_proba}")