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="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")
   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="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")

    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="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")
    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="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")


  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}")
```