Project-3

Student Name: Hitesh Reddy Tippasani

Email: htippasani2024@fau.edu

GCP Project Name: cov1234

GCP Project ID: conv12334

GCP dashboard link:
https://console.cloud.google.com/home/dashboard?hl=en&project=conv12334

Github repository link: https://github.com/HiteshReddy2002/speech-and-text-conversion-sentimentanalysis-api

Cloud Run Service name: speech-and-text-conversion-api-api

App URL: https://speech-and-text-conversion-sentimentanalysis-api-542126986249.us-central1.run.app

Introduction

This web application is built using Flask, designed for audio processing, transcription, and sentiment analysis. It leverages Google's Generative AI model, offering functionalities for uploading audio files, performing transcription, analyzing sentiment, and saving the results in text format. The system is tailored to process .wav audio files and generate a detailed analysis, including a transcript and sentiment score, making it ideal for users needing fast audio analysis.

Goals and Objectives:

1. **Streamline Audio Transcription and Sentiment Analysis:**
    - Provide an easy-to-use interface for uploading .wav audio files, processing them with Google Cloud's Gemini API to generate accurate transcripts and sentiment analysis.
2. **Enhance Audio Processing with Gemini AI:**
    - Utilize Google's Gemini AI to deliver high-quality transcription and sentiment analysis in one integrated workflow, simplifying the user's experience.
3. **Provide Emotional Insights:**
    - Automatically categorize the sentiment of the audio content into positive, negative, or neutral categories, enhancing understanding of the context behind the transcription.

4. **Ensure Easy Accessibility and File Management:**
    - Facilitate users with a simple online interface that handles file uploads, saves results in organized directories, and provides convenient access to transcripts and sentiment analysis.

High-Level Features:

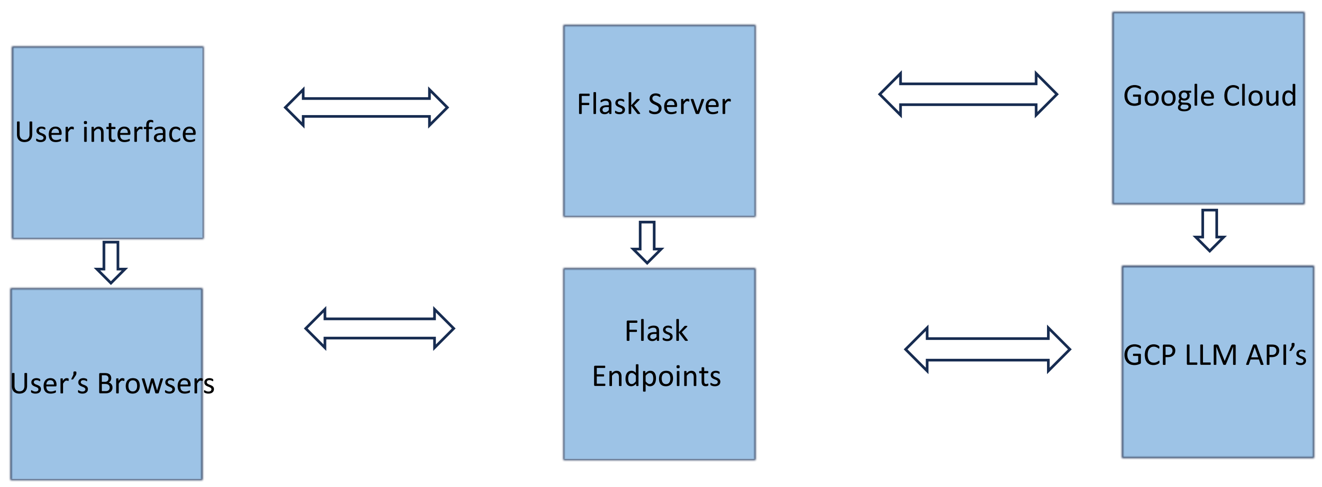
1. **Audio Upload and Processing:**
    - Users can upload .wav audio files for analysis.
    - The audio file is processed through the Gemini API to perform both transcription and sentiment analysis in a single call.
2. **Audio Transcription:**
    - The uploaded audio is transcribed into text using Google's Gemini AI.
    - The transcription is stored as a .txt file for easy reference.
3. **Sentiment Analysis:**
    - The sentiment of the transcribed audio is analyzed, and a sentiment score is provided, indicating whether the audio has a positive, negative, or neutral tone.
4. **File Management and Accessibility:**
    - All processed files (audio, transcriptions, sentiment results) are stored in well-organized directories on the server.
    - The web interface allows users to view, download, and manage these files with ease.
5. **Web Interface for File Management:**
    - Users can view a list of uploaded .wav audio files along with links to download the corresponding .txt files containing the transcriptions and sentiment analysis results.
6. **Real-time Feedback and Error Handling:**
    - The application provides immediate feedback to the user, logging any errors that occur during the upload or processing of audio files.
7. **Server and System Information:**
    - Utility routes (/ping, /info) offer users details about the server status and the health of the application, ensuring transparency and reliability.

Technical Details:

- The Flask app manages file uploads and routes to handle user requests.
- The genai library is used to interact with the Gemini AI model, which performs both transcription and sentiment analysis.

- Results are saved as .txt files, which can be accessed and downloaded by users.
- A user-friendly web interface enables the upload of .wav files and the viewing of results.

Architecture



1. User Interface:

- Users interact with the application through a web interface to upload .wav audio files, view transcriptions, analyze sentiment, and access synthesized speech results.
- The interface also allows users to view and manage their uploaded files, including downloading the generated transcription and sentiment analysis results.

2. Flask Application:

- A Python-based web framework (Flask) hosts the backend logic of the application.

- **Endpoints:**
  - /upload: Handles file uploads, triggers audio transcription using the Gemini API, and performs sentiment analysis on the uploaded audio.
  - /uploads/<filename>: Serves the uploaded audio and generated files (e.g., transcriptions and sentiment analysis results) for download.
  - /info & /ping: Provide application details and the server's status, ensuring users are informed about the state of the system.

3. File Storage:

- **uploads**: Stores user-uploaded .wav audio files.

- **analysis**: Stores sentiment analysis results and transcriptions as .txt files for future reference.

4. FFmpeg Subprocess:

- Although the code does not currently include explicit FFmpeg handling, audio processing steps can be easily integrated for compatibility with Google Cloud's Speech-to-Text API, such as converting audio files to the appropriate format (e.g., LINEAR16 with a 16kHz sample rate).

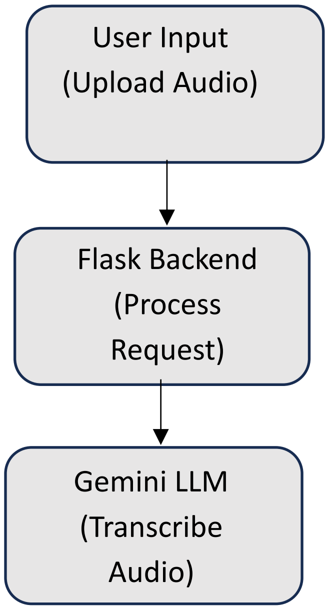
5. Google Cloud Services:

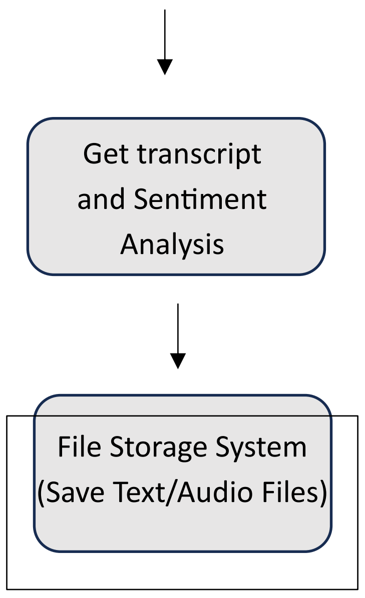
- **Gemini API** (part of Google Cloud's generative models) handles both transcription and sentiment analysis in a single request.
- The API transcribes the audio content and categorizes the sentiment as positive, negative, or neutral, providing both emotional insights and textual transcription in one response.

6. Data Flow and Interaction:

- Users upload .wav audio files via the web interface.
- The Flask app processes the files by calling the Gemini API to transcribe the audio and analyze sentiment.
- Processed data (transcriptions, sentiment analysis results) are saved as .txt files and made available for download through the web interface.
- The system stores the uploaded files and processed results in the appropriate directories for easy access and management by users.

Workflow:




Get transcript and Sentiment Analysis
File Storage System (Save Text/Audio Files)

Implementation

1. User Interface (Browser)

Code and Configuration:

- **Frontend (JavaScript - script.js)**:
    - The user interface is designed to allow users to record and upload audio directly from the browser. The recording process is controlled via buttons that trigger the MediaRecorder API.
    - **Recording Audio**: The recordButton starts recording, and the stopButton stops the recording. The MediaRecorder API handles the recording, and once the audio is recorded, it is automatically sent to the backend /upload endpoint via a fetch() request.
    - **Recording Timer**: A real-time timer is displayed to the user during the audio recording session. This is managed by the formatTime function, which calculates and updates the timer every second, showing the elapsed time in a mm:ss format.
    - **Audio Display**: After the user finishes recording, the audio file is displayed in the form of an audio player on the page, which allows users to listen to the recording before uploading it.
- **HTML Files (templates/index.html)**:
    - The HTML page is simple and user-friendly, providing all necessary elements for user interaction, including file upload, audio recording, and sentiment analysis display.
    - The page contains buttons for starting and stopping the recording (record and stop), and once the audio is uploaded, an audio player is dynamically displayed.

    - The page also displays lists of files uploaded by the user. For each uploaded file, the transcription and sentiment analysis results are linked, allowing users to easily view the processed content.

Location on Server:

- The frontend files (HTML, CSS, and JavaScript) are stored in the templates/ directory for HTML files and the static/ directory for static assets like JavaScript and CSS files. The use of Flask's url_for function helps dynamically serve these files to the browser.

2. Flask Application

Dependencies:

- **Flask**: The core Python framework used to handle HTTP requests and serve the web pages. It is responsible for routing user requests to the appropriate functions in the backend.
- **google-cloud-speech** and **google-cloud-texttospeech**: These libraries interact with Google Cloud's Speech-to-Text and Text-to-Speech APIs to perform transcription and speech synthesis respectively.
- **werkzeug**: Used for handling file uploads securely, ensuring that users can only upload safe files.
- **subprocess**: Allows the backend to interact with external programs, like FFmpeg, for preprocessing audio files before sending them to Google Cloud APIs.
- **os**: This module is used to handle file and directory operations, such as creating necessary folders for storing user-uploaded files.

Code Structure:

- **Main File (main.py)**: This file contains all the routing logic for the application. It is where Flask's app is initialized and where the different routes are defined to handle user interactions.
    - The main routes include:
        - /: Displays the home page with the user interface for uploading and recording audio.
        - **/upload**: Handles the upload of .wav files, triggers the transcription process, and performs sentiment analysis.
        - **/tts/<filename>**: Allows users to view or download synthesized speech files generated from text inputs.
        - **/uploads/<filename>**: Serves the uploaded audio files to users after they've been processed.
        - **/info & /ping**: Provide application status and health check information for developers or system administrators.
- **Functions**:

    - **upload_audio()**: Handles file uploads. It saves the uploaded file and then processes it by calling a function like convert_to_16000hz() for audio preprocessing.
    - **process_audio_with_llm()**: This function processes the uploaded audio, sends it to Google Cloud's Speech-to-Text API for transcription, and simultaneously analyzes the sentiment of the transcribed text using the Gemini API.
    - **synthesize_text()**: Uses Google Cloud's Text-to-Speech API to convert user-input text into speech and generates an .mp3 file for download.

Configuration:

- The Flask app is located in main.py, the root directory of the project.
- To start the app, run python main.py or deploy it using a WSGI server like **gunicorn** (configured in the Procfile).
- By default, the app runs on port 5000, but this can be changed in the app's configuration if needed.
- **HTTP Requests**: The application supports:
    - **POST** requests for uploading audio files and submitting text for TTS conversion.
    - **GET** requests for serving files (e.g., .wav and .mp3 files) and rendering pages for users to view processed content.

---

3. File Storage

Configuration and Structure:

- The application uses multiple directories to manage different types of files:
    - **uploads/**: Stores the user-uploaded .wav files before they are processed.
    - **analysis/**: Stores the results of sentiment analysis and transcriptions as .txt files, which are linked to the corresponding audio files for user access.
    - **tts/**: Stores the .mp3 files generated by the Text-to-Speech API.

Code Implementation:

- **os.makedirs()**: Ensures that the required directories (uploads/, analysis/, tts/) are created if they don't already exist when the app starts.
- **Dynamic File Paths**: The file paths for storing user uploads, transcription outputs, and TTS files are dynamically created using the os.path.join() method, ensuring compatibility across different operating systems.
- **Serving Files**: Flask's send_from_directory function is used to serve these files securely. For example, uploaded audio files are made available through routes like /uploads/<filename>, and users can download the transcription and sentiment analysis results at /uploads/<filename>.txt.

---

4. FFmpeg Subprocess

Code Implementation:

- **convert_to_16000hz()**: This function uses subprocess.run() to invoke FFmpeg for converting audio files to the LINEAR16 format with a 16kHz sample rate, which is the format required by the Google Cloud Speech-to-Text API.
- **FFmpeg Parameters**:
  - -i: Specifies the input audio file.
  - -acodec pcm_s16le: Converts the audio to LINEAR16 codec.
  - -ar 16000: Sets the sample rate to 16kHz.
  - -ac 1: Ensures the audio is mono-channel.

Configuration:

- FFmpeg must be installed on the server and available in the system's PATH to allow for seamless audio processing.

---

5. Google Cloud Services

APIs Used:

1. **Speech-to-Text API**:
   - The **RecognitionConfig** used in the transcribe_audio() function is configured to:
     - Use the LINEAR16 encoding to ensure compatibility with the Speech-to-Text API.
     - Set the sample rate to 16kHz, which matches the processed audio files.
     - Specify the language_code as en-US for English transcription.
2. **Text-to-Speech API**:
   - The **VoiceSelectionParams** and **AudioConfig** used in the synthesize_text() function are configured to:
     - Use the en-US language code.
     - Set the SSML gender to neutral for a balanced voice tone.
     - Generate .mp3 audio files from the provided text.
3. **Sentiment Analysis**:
   - The Gemini API handles sentiment analysis of the transcription results. It categorizes the text into positive, negative, or neutral sentiments based on the

analysis, providing users with a deeper understanding of the emotional tone of the spoken content.

Authentication and Setup:

- Google Cloud API credentials (a .json key file) must be set up in the environment via the **GOOGLE_APPLICATION_CREDENTIALS** variable to authenticate the app's access to Google Cloud services.
- API usage is billed based on the amount of audio processed and the text converted, so it is important to configure appropriate budget limits in the Google Cloud Console to avoid excessive charges.

---

6. Server and Deployment

Server Location and Deployment:

- The app is deployed using **gunicorn**, a production-ready WSGI server, as specified in the Procfile.
- The app can be hosted on any cloud service provider (AWS, Google Cloud, Heroku, etc.).

Firewall Rules:

- The app must have port 5000 open (or the appropriate port configured in your environment).
- HTTPS should be enabled for secure communication, especially if handling sensitive user data.

Deployment Steps:

1. Install all dependencies using pip install -r requirements.txt.
2. Configure Google Cloud credentials by setting the environment variable GOOGLE_APPLICATION_CREDENTIALS to the path of the credentials file.
3. Start the server using gunicorn main:app for production deployment.

---

7. Additional Configurations

Static Files:

- JavaScript and CSS files are stored in the static/ directory. These files are automatically served by Flask using the url_for('static', filename='script.js') method, ensuring they are available to the frontend.

Error Handling:

- Each route in the Flask application includes exception handling to catch errors that may arise during file uploads, transcription failures, or TTS generation. This ensures that the user experience is smooth and that meaningful error messages are provided.

Logging:

- Logs are printed to the console during development to facilitate debugging.
- For production, the logs can be redirected to files, allowing developers or system administrators to monitor application performance and catch issues early.

Pros and Cons

Pros:

1. **Simple and Modular:**
    - The application is built using Flask, which provides a lightweight framework that is easy to understand, maintain, and extend. The modular structure allows developers to add new features without disrupting the existing functionality. This modularity makes the codebase more organized and maintainable.
2. **Integration with Google Cloud APIs:**
    - The use of **Google Cloud Speech-to-Text** and **Text-to-Speech** APIs ensures high accuracy in transcription and speech synthesis. These robust, cloud-based APIs handle complex processing, which offloads resource-intensive tasks from the local server, ensuring reliable results for users.
3. **File Organization:**
    - The system uses a structured directory layout (uploads, tts, analysis), which keeps different types of files organized and easy to manage. This makes it simple to track user-uploaded content, processed audio files, and analysis outputs.
4. **Easy Deployment:**
    - The application is lightweight, meaning it can be easily deployed on cloud platforms such as **Heroku**, **AWS**, or **Google Cloud**. This flexibility allows for quick deployment and scaling, making it a good fit for small to medium-sized applications.
5. **Reusability:**
    - The code is modular and can easily be extended to support additional features. For example, adding more advanced transcription features or additional APIs for processing different file formats is straightforward.
6. **Scalable Backend Services:**
    - By offloading heavy tasks like transcription and text-to-speech generation to **Google Cloud** services, the app can scale efficiently. These cloud services are highly reliable and can handle an increased load without the application's backend becoming overwhelmed.

Cons:

1. **Scalability Issues:**
   - While the app is well-suited for smaller use cases, Flask and the use of local file storage can struggle to handle large-scale usage or multiple users simultaneously. Flask's default server is not designed for high concurrency, and local file storage can quickly become inefficient when scaling.
2. **High Latency:**
   - The **FFmpeg** processing for audio conversion, as well as the communication with external APIs (Google Cloud APIs), introduces latency in the system. This can result in delays between user actions (such as recording audio) and the system's response (like returning a transcription or synthesized speech).
3. **Single Point of Failure:**
   - The application is structured in a monolithic way, meaning that if any component (such as the transcription service, audio conversion, or Google Cloud APIs) fails, the whole system could crash. This makes the app susceptible to downtime, especially if external services experience issues.
4. **Limited Security:**
   - The application currently lacks user authentication, and files uploaded by users are publicly accessible without any security measures. This could pose risks in a production environment, especially if sensitive or private data is involved. There's also no authorization mechanism in place to restrict access to certain files or user data.
5. **High API Costs:**
   - The **Google Cloud APIs** (Speech-to-Text and Text-to-Speech) are billed based on usage, and at scale, these costs can become significant. This can limit the application's ability to scale cost-effectively if the number of users or frequency of API calls increases.

Recommendations for Improvement:

1. **Scalable Infrastructure:**
   - Transition to containerization using **Docker** and orchestration with **Kubernetes**. This would enable the app to scale efficiently as demand increases. Additionally, cloud storage solutions like **AWS S3** or **Google Cloud Storage** can be integrated to handle large file storage and backups more efficiently than local disk storage.
2. **Asynchronous Processing:**
   - To address the latency issues and offload long-running tasks, implement **Celery** and **Redis** for background task processing. This would allow the app to perform

transcription, sentiment analysis, and TTS synthesis asynchronously, ensuring a faster and smoother user experience.

3. **Security Enhancements:**
    - Implement **user authentication** and authorization to ensure that only authorized users can access specific resources. Add file upload security by validating the file types and sizes before processing them to prevent malicious uploads. Implement secure file access controls, ensuring that users can only download or access their own files.
4. **Microservices Architecture:**
    - Break the application into separate microservices (e.g., a separate service for file processing, one for transcription, and another for TTS generation). This would allow the app to be more resilient, as each service can be independently scaled or replaced without affecting the rest of the system.
5. **Cost Optimization:**
    - Monitor API usage to avoid overage charges by setting up **quotas** and **usage limits** in the **Google Cloud Console**. Additionally, look into implementing caching strategies to reduce the number of API calls for repeated tasks and minimize costs. Explore the possibility of using cheaper alternatives for specific use cases, such as open-source speech-to-text models for non-critical tasks.

Problems Encountered and Solutions

Problem 1: Not Saving Recordings

- **Description**: The application initially failed to store the audio recordings from the browser due to improper handling of the **MediaRecorder API** and form data in the frontend. This prevented the recorded audio from being uploaded and stored on the server.
- **Solution**: The issue was fixed by adjusting the JavaScript code to correctly append the audioBlob from the **MediaRecorder API** to a **FormData** object. This ensured the audio file was correctly passed to Flask's /upload endpoint for further processing. The endpoint was also verified to ensure it was properly receiving the uploaded file, allowing the recording to be stored.

Problem 2: No Transcription of the Recordings

- **Description**: After fixing the recording storage issue, the application still failed to transcribe the audio files. The problem stemmed from the incompatibility of the uploaded audio format with the **Google Speech-to-Text** API, which required specific audio formats (LINEAR16 encoding with a 16kHz sample rate).
- **Solution**: To resolve this, **FFmpeg** was integrated into the backend to convert audio files to the required **LINEAR16** format with a **16kHz** sample rate. After ensuring compatibility,

the transcription functionality started working as expected, and audio files were successfully transcribed to text.

Problem 3: Problems with Git Push

- **Description**: When trying to push the project to GitHub, large file sizes, and an improperly configured repository caused issues. Specifically, audio files were being tracked by Git, leading to problems when pushing the project to the remote repository.
- **Solution**: The repository was re-initialized after deleting unnecessary large files manually and updating the .gitignore file to ensure that audio files were not included in the repository. After making these changes, the repository was successfully pushed to GitHub using git push origin main.

Problem 4: Incorrect Sentiment Analysis Results

- **Description**: The sentiment analysis was producing incorrect results. The initial configuration of the sentiment analysis API did not yield accurate readings for the transcription text.
- **Solution**: The API parameters were adjusted, and text preprocessing techniques were fine-tuned to improve accuracy. This included cleaning and formatting the transcribed text before sending it for sentiment analysis. Once these changes were implemented, the sentiment results improved, providing more accurate insights into the tone of the transcriptions.

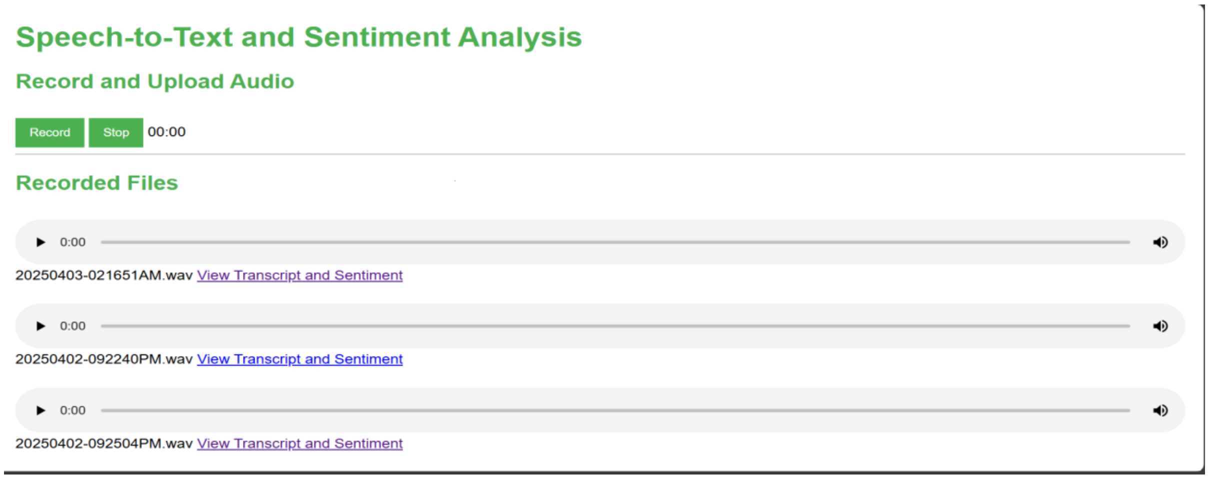
Application Instructions (Recording Only)

Step 1: Open the Application

- Open the provided link to load the web page where you can access the audio recording feature.

Step 2: Record Audio

- On the homepage, you’ll see the **Record** button to start recording your audio.
  - Click the **Record** button to begin recording.
  - Once you're done, click the **Stop** button to stop the recording.

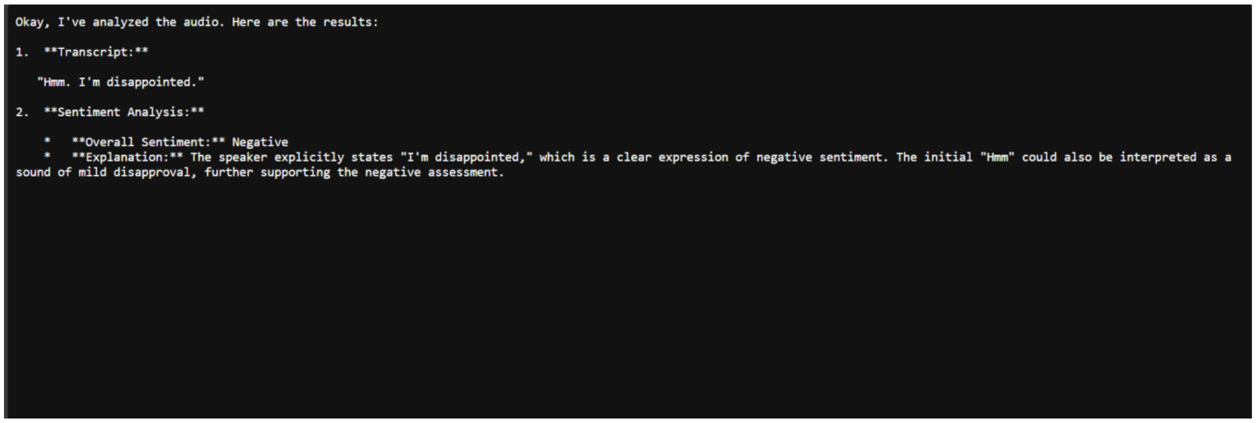

Speech-to-Text and Sentiment Analysis
Record and Upload Audio
Record
Stop
00:00
Recorded Files
0:00
20250403-021651AM.wav View Transcript and Sentiment
0:00
20250402-092240PM.wav View Transcript and Sentiment
0:00
20250402-092504PM.wav View Transcript and Sentiment

Step 3: Transcribe and Analyze Sentiment

- After stopping the recording, the application will:
    1. **Transcribe the recorded audio** to text using the **Gemini API**, which combines both transcription and sentiment analysis in a single call.
    2. **Analyze the sentiment** of the transcribed text (positive, negative, or neutral).
    3. Display the **transcription** and **sentiment analysis** results on the page. These results will be saved in a .txt file for later access.

Step 4: View Results

- Once the audio is processed, you can view the **transcription** and **sentiment analysis** results directly on the page.
    - The transcription will be displayed along with the **sentiment** score and explanation.



```
Okay, I've analyzed the audio. Here are the results:

1.  **Transcript:**

   "Hmm. I'm disappointed."

2.  **Sentiment Analysis:**

    *   **Overall Sentiment:** Negative
    *   **Explanation:** The speaker explicitly states "I'm disappointed," which is a clear expression of negative sentiment. The initial "Hmm" could also be interpreted as a
sound of mild disapproval, further supporting the negative assessment.
```

Lessons Learned

Developing Applications from Start to Finish:

1. Gained practical experience in building a full-stack web application using **Flask** for the backend and **HTML/JavaScript** for the frontend. This project demonstrated how to

seamlessly integrate both frontend and backend with external services like the **Gemini API**.

2. Learned the process of designing user interfaces that allow seamless interaction between users and backend services, such as audio recording, transcription, sentiment analysis, and file management.

Audio Processing and File Handling: 3. Gained knowledge of how to handle **audio recordings** from the browser using the **MediaRecorder API**, including converting the recordings into the necessary format and uploading them for processing. This includes the integration of **FFmpeg** for audio file formatting before passing it to the **Gemini API** for transcription and sentiment analysis. 4. Gained experience in **file storage management**, organizing files for both audio input and processed outputs (transcriptions, sentiment analysis results), and making them available for user access via Flask routes.

Using APIs: 5. Developed a deep understanding of how to integrate **cloud APIs**, specifically **Google Cloud's Gemini API** for combined transcription and sentiment analysis. 6. Learned how to handle API **authentication**, manage API keys securely, and ensure **efficient API usage** for transcription and sentiment analysis without exceeding the limits. 7. Mastered handling API responses and structuring them in a way that improves the user experience (e.g., saving transcriptions and sentiment results as downloadable files).

Solving Problems and Debugging: 8. Overcame various issues with **audio recording**, **transcription**, and **API integration**. Debugging challenges such as file conversion, handling invalid inputs, and ensuring compatibility between the frontend and backend enhanced my problem-solving abilities. 9. The integration of **FFmpeg** for audio preprocessing and handling different audio formats was crucial for ensuring smooth API communication, and resolving these issues improved the overall stability of the application.

Version Management and Implementation: 10. Gained valuable experience in **version control** using **Git**. Encountered challenges with **large files** and **repository organization**, learning the importance of using .gitignore to exclude unnecessary files (e.g., audio files) from being pushed to the repository. 11. Learned the significance of deploying applications to production-ready servers, ensuring proper setup, and keeping the repository clean for smoother team collaboration and future development.

This project provided hands-on experience in both technical aspects (API integration, audio handling, debugging) and operational practices (version control, deployment, server management). It also demonstrated the importance of managing external APIs and debugging issues in real-time, offering essential skills for building scalable and maintainable web applications.

Appendix

main.py

```
import os

from datetime import datetime

from flask import Flask, render_template, request, jsonify
```

```
import io
import google.generativeai as genai
from flask import send_from_directory
import logging
import base64

# Initialize Flask app

app = Flask(__name__)

# Configure upload folder
UPLOAD_FOLDER = 'uploads'
ALLOWED_EXTENSIONS = {'wav'}
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

# Ensure the uploads folder exists
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Set your API key (ensure it's configured correctly)
genai.configure(api_key="AIzaSyD1hQZGSUccjECeQ4kj5gtvlxXv10T4gko")

# Fetch files for display
def get_files():
    files = []
    for filename in os.listdir(UPLOAD_FOLDER):
        if allowed_file(filename) or filename.endswith('.txt'):
            files.append(filename)
    files.sort(reverse=True)
    return files


# Check if the file is allowed
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS


# Process the audio file with Gemini API
def process_audio_with_llm(file_path):
    # Read the audio file
    with io.open(file_path, 'rb') as audio_file:
        content = audio_file.read()



    model = genai.GenerativeModel('gemini-2.0-flash')


    try:
        # Define the prompt for transcription and sentiment analysis
        prompt = """
        Analyze the provided audio and provide the following:


        1. A complete transcript of the audio.
         2. A sentiment analysis of the audio, indicating the overall sentiment (positive, negative, or
neutral) and providing a brief explanation of why."""


        # Correct request format to match Gemini API structure
        response = model.generate_content(
            [
                {
                    "parts": [
            {
               "text": prompt
            },
            {
               "inline_data": {
                  "mime_type": "audio/wav",  # Specify the mime type
                  "data": content           # Send raw binary audio data directly
               }
            }
         ]
      }
   ])

# Assuming the response has 'transcript' and 'sentiment' parts
response_str = response.text
file_dir = os.path.dirname(file_path)

# Generate a filename for the response text file
response_file_path = os.path.splitext(file_path)[0] + '.txt'

# Save to text file
with open(response_file_path, 'w') as file:
   file.write(response_str)

logging.info(f"Full response saved to: {response_file_path}")

if hasattr(response, 'parts') and len(response.parts) > 1:
   transcript = response.parts[0].text if hasattr(response.parts[0], 'text') else ''
   sentiment = response.parts[1].inline_data if hasattr(response.parts[1], 'inline_data') else {}
```

```
            sentiment_score = sentiment.get('score', 0)
            sentiment_magnitude = sentiment.get('magnitude', 0)

            if not transcript:
                logging.warning(f"Failed to transcribe audio file: {file_path}")
            if sentiment_score == 0 and sentiment_magnitude == 0:
                logging.warning(f"Sentiment analysis failed for audio file: {file_path}")

            logging.info(f"Processed audio file: {file_path}")
            return transcript, sentiment_score, sentiment_magnitude

        else:
            logging.error(f"Unexpected response format for audio file: {file_path}")
            return '', 0, 0

    except Exception as e:
        logging.error(f"Error processing audio file {file_path}: {e}")
        raise  # Re-raise the exception to be handled in the route

@app.route('/')
def index():
   # files = get_files()  # List of uploaded audio files (from uploads folder)
    files = [f for f in os.listdir(UPLOAD_FOLDER) if f.endswith('.wav')]
    return render_template('index.html', files=files)

# Route to upload an audio file
@app.route('/upload', methods=['POST'])
def upload_audio():
    if 'audio_data' not in request.files:
        logging.error("No audio data in request")
        return "No audio data in request", 400

    file = request.files['audio_data']
    if file.filename == '':
        logging.error("No file selected")
        return "No file selected", 400

    # Save the uploaded audio file
    filename = datetime.now().strftime("%Y%m%d-%I%M%S%p") + '.wav'
    file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
    file.save(file_path)
    logging.info(f"Uploaded audio file: {filename}")

    try:
        process_audio_with_llm(file_path)
        return "Uploaded, transcribed, and sentiment analyzed successfully. Check the generated text file.", 200
    except Exception as e:
        logging.error(f"Error during processing: {e}")
        return f"Error during processing: {e}", 500



# Route to serve uploaded files (audio and text)
@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'], filename)

if __name__ == '__main__':
    app.run(debug=True)
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Audio Recorder and Transcription</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
    }
    table {
      width: 100%;
      border-collapse: collapse;
    }
    td {
      padding: 20px;
    }
    h1, h2 {
      color: #4CAF50;
    }
    button {
      margin-top: 10px;
      padding: 10px 15px;
      background-color: #4CAF50;
      color: white;
      border: none;
      cursor: pointer;
    }
```

```
        button:hover {
            background-color: #45a049;
        }
        audio {
            margin-top: 10px;
            width: 100%;
        }
        ul {
            list-style-type: none;
            padding: 0;
        }
        li {
            margin-bottom: 15px;
        }
    </style>
</head>
<body>

    <h1>Speech-to-Text and Sentiment Analysis</h1>
    <h2>Record and Upload Audio</h2>

    <!-- Record and Stop buttons -->
    <button id="record">Record</button>
    <button id="stop" disabled>Stop</button>
    <span id="timer">00:00</span>

    <!-- Upload form -->
    <form id="uploadForm" method="POST" enctype="multipart/form-data" action="/upload">
        <input type="hidden" name="audio_data" id="audioData">
        <!-- Dynamically filled with recorded audio data -->
    </form>


    <script src="{{ url_for('static', filename='script.js') }}"></script>
    <hr>


    <h2>Recorded Files</h2>
    <ul>
        {% for file in files %}
        <li>
            <audio controls>
                <source src="{{ url_for('uploaded_file', filename=file) }}" type="audio/wav">
                Your browser does not support the audio element.
            </audio><br>
            {{ file }}
                <a  href="{{  url_for('uploaded_file',  filename=file.replace('.wav',  '.txt'))  }}">View
Transcript and Sentiment</a>
        </li>
        {% endfor %}
    </ul>


</body>
</html>
```

requirements.txt

```
Flask==3.0.3
google-cloud-speech==2.27.0
google-cloud-texttospeech==2.17.2
gunicorn==22.0.0
google-generativeai==0.8.4
```

Procfile/Dockerfile web: gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0 main:app

script.js

```
const recordButton = document.getElementById('record');
const stopButton = document.getElementById('stop');
const audioElement = document.getElementById('audio');
const uploadForm = document.getElementById('uploadForm');
const audioDataInput = document.getElementById('audioData');
const timerDisplay = document.getElementById('timer');

let mediaRecorder;
let audioChunks = [];
let startTime;

function formatTime(time) {
  const minutes = Math.floor(time / 60);
  const seconds = Math.floor(time % 60);
  return `${minutes}:${seconds.toString().padStart(2, '0')}`;
}

recordButton.addEventListener('click', () => {
  navigator.mediaDevices.getUserMedia({ audio: true })
    .then(stream => {
      mediaRecorder = new MediaRecorder(stream);
      mediaRecorder.start();

      startTime = Date.now();
      let timerInterval = setInterval(() => {
        const elapsedTime = Math.floor((Date.now() - startTime) / 1000);
        timerDisplay.textContent = formatTime(elapsedTime);
      }, 1000);

      mediaRecorder.ondataavailable = e => {
        audioChunks.push(e.data);
      };

      mediaRecorder.onstop = () => {
        const audioBlob = new Blob(audioChunks, { type: 'audio/wav' });
        const formData = new FormData();
        formData.append('audio_data', audioBlob, 'recorded_audio.wav');

        fetch('/upload', {
            method: 'POST',
            body: formData
        })
        .then(response => {
            if (!response.ok) {
                throw new Error('Network response was not ok');
            }
            location.reload(); // Force refresh

            return response.text();
        })
        .then(data => {
            console.log('Audio uploaded successfully:', data);
            // Redirect to playback page or display success message
        })
        .catch(error => {
            console.error('Error uploading audio:', error);
        });
```

```
      };
    })
.catch(error => {
console.error('Error accessing microphone:', error);
});
recordButton.disabled = true;
stopButton.disabled = false;
});
stopButton.addEventListener('click', () => {
if (mediaRecorder) {
mediaRecorder.stop();
}
recordButton.disabled = false;
stopButton.disabled = true;
});
// Initially disable the stop button
stopButton.disabled = true;
```