Code Description

Source Code for project:

All the required libraries are listed in ‘used_library.txt’ file. We have also created a user manual to guide and assist users with installation, available at: https://docs.google.com/document/d/1wc2eVDkJMkmq5nfNCTMDtQ9TsdsvfadS/edit

Version 1 (First Presentation): https://drive.google.com/drive/u/2/folders/1FRtMW6V6ugrDGMTJhhumWpE9Ta9jXunw

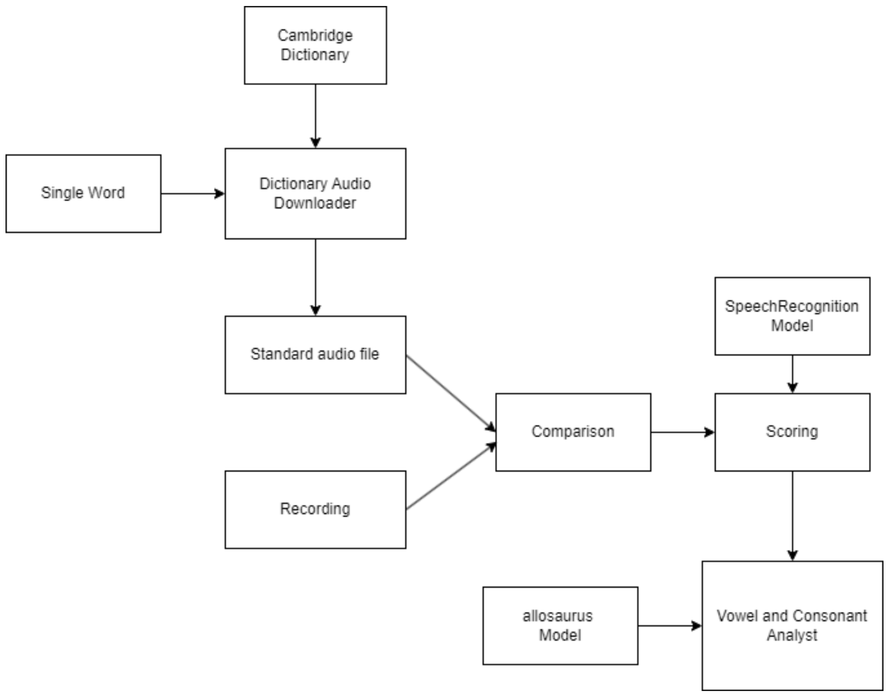
Version 2 (Final Presentation):

https://drive.google.com/drive/u/2/folders/1-00mVJJYiWr6OTw8qVXCjE3OWoeDQu9a

We also create a document for User Manual here:

https://docs.google.com/document/d/1wc2eVDkJMkmq5nfNCTMDtQ9TsdsvfadS/edit

System Pipeline:



Workflow

The system workflow can be briefly described as follows:

1. For a selected single word, download the audio file of that word from the Cambridge Dictionary using the Dictionary Audio Downloader Webpage.

2. Convert the audio file from MP3 to WAV format to use as the standard audio file.

3. Record the user's pronunciation of the same word.

4. Compare the two audio files and use the SpeechRecognition model to determine if both files contain the same word. If not, the score will be 0. Then, calculate the similarity score using Cosine similarity, which will score the user's pronunciation compared to the standard audio.

5. Finally, use the Allosaurus model to analyze the vowels and consonants in both audio files.

Pre-processing parameter:

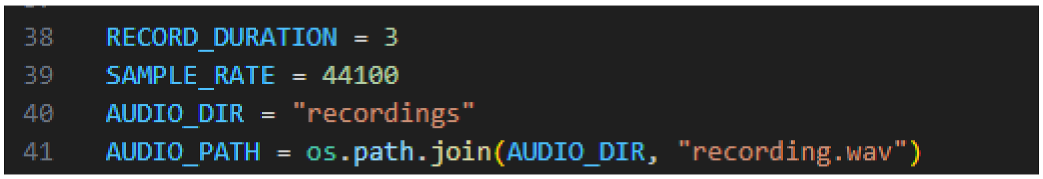
To compare the audio files, we first need to perform preprocessing and data normalization steps. This ensures that the data is easier to compare and can be processed across different bit rates and sample rates by standardizing them to a uniform level.

The preprocessing parameters include:

Choosing parameters number:

Record Duration: 3 seconds (Convert both recording audio and standard audio to 3 seconds)

Sample Rate: 44100 (This threshold is commonly used in music and aligns with the human hearing range).



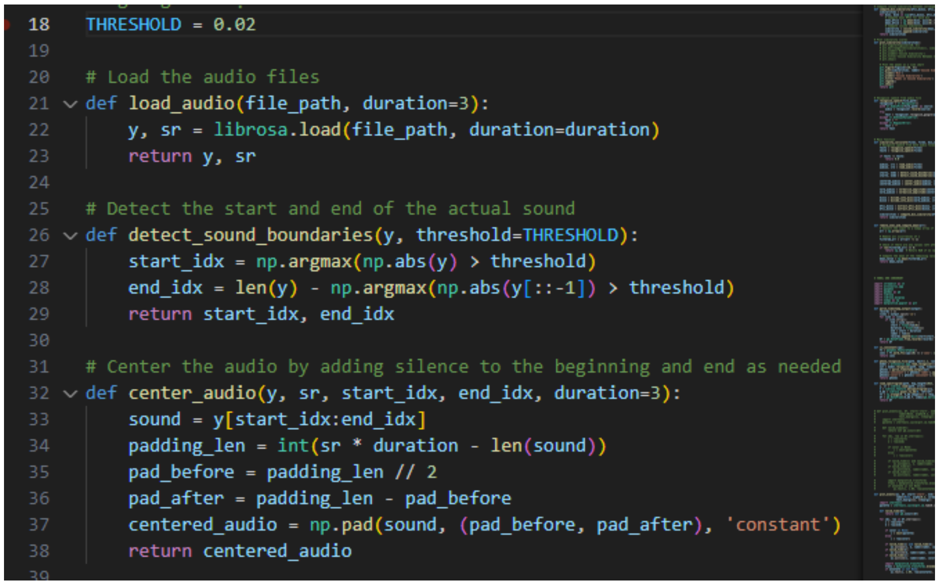
```
RECORD_DURATION = 3
SAMPLE_RATE = 44100
AUDIO_DIR = "recordings"
AUDIO_PATH = os.path.join(AUDIO_DIR, "recording.wav")
```

Comparison for 2 audio files:

To compare the two audio files, we start by loading them using the librosa library.

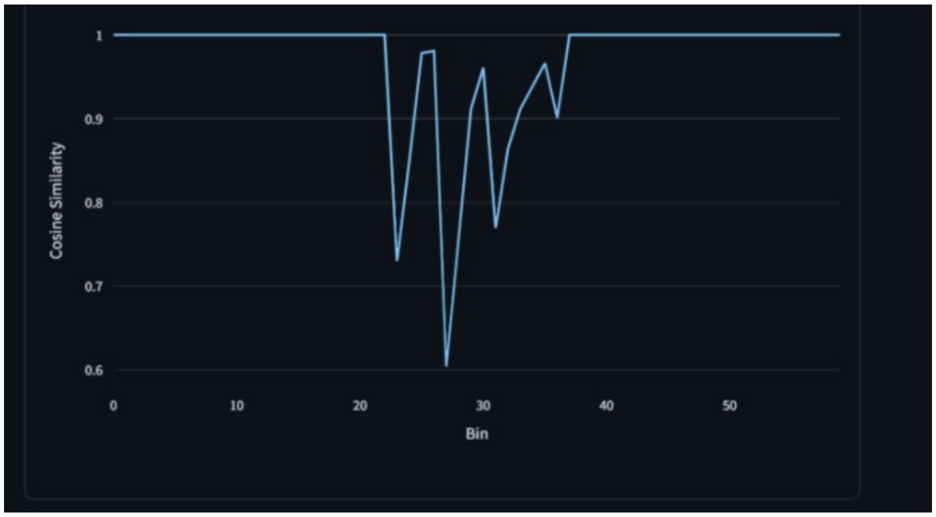
One issue with normalizing audio is that we focus on comparing individual words. Therefore, a 3-second audio clip often contains a lot of silence (segments with no sound or noise). To address this, we create a threshold: if the audio intensity (before any other normalization steps, to ensure purity) is below this threshold, we identify it as silence and set the audio intensity of these silent segments to zero.

Furthermore, to make the similarity comparison more effective (we use cosine similarity), it is beneficial for the positions of the sounds in the two audio segments to correspond as closely as possible. Therefore, after identifying the silent segments, we center the audio and add padding with silence at both ends to ensure they each have a length of 3 seconds.



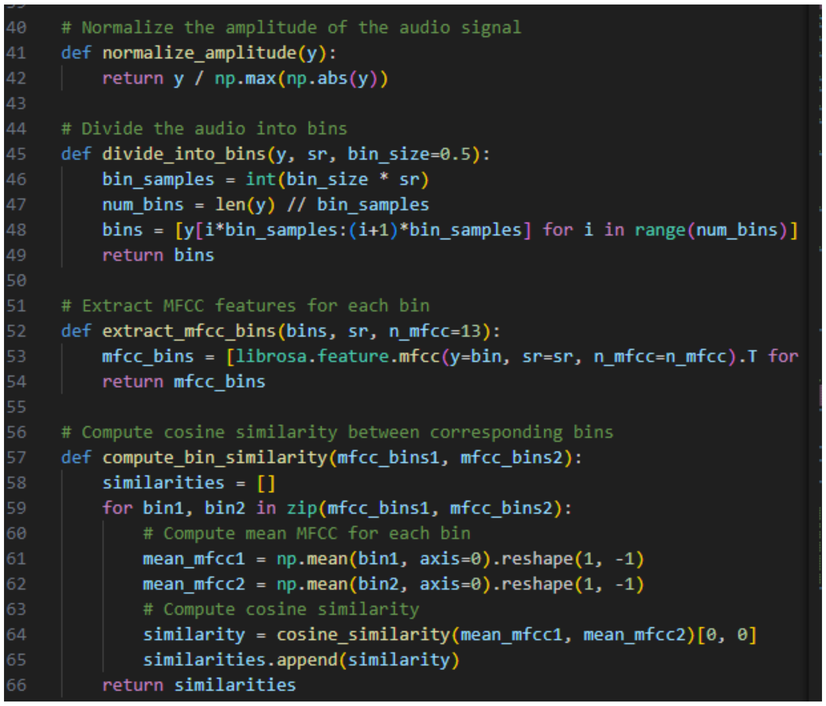
```python
THRESHOLD = 0.02

# Load the audio files
def load_audio(file_path, duration=3):
    y, sr = librosa.load(file_path, duration=duration)
    return y, sr

# Detect the start and end of the actual sound
def detect_sound_boundaries(y, threshold=THRESHOLD):
    start_idx = np.argmax(np.abs(y) > threshold)
    end_idx = len(y) - np.argmax(np.abs(y[::-1]) > threshold)
    return start_idx, end_idx

# Center the audio by adding silence to the beginning and end as needed
def center_audio(y, sr, start_idx, end_idx, duration=3):
    sound = y[start_idx:end_idx]
    padding_len = int(sr * duration - len(sound))
    pad_before = padding_len // 2
    pad_after = padding_len - pad_before
    centered_audio = np.pad(sound, (pad_before, pad_after), 'constant')
    return centered_audio
```

Load and preprocessing audio



For example, when comparing two audio files and obtaining the above chart, since the audio has been normalized, the ends are silent with a similarity score of 1. The middle part of the chart will provide insight into the similarity between the two audio segments.

After denoising and adding padding to both ends, we normalize the audio data by standardizing the amplitude and dividing the 3-second audio segment into bins (each bin is 0.05 seconds long, resulting in 60 bins). The purpose of dividing into bins is to ensure that each bin can contain an entire vowel or consonant without being affected by others when comparing and analyzing them.



```
# Normalize the amplitude of the audio signal
def normalize_amplitude(y):
    return y / np.max(np.abs(y))

# Divide the audio into bins
def divide_into_bins(y, sr, bin_size=0.5):
    bin_samples = int(bin_size * sr)
    num_bins = len(y) // bin_samples
    bins = [y[i*bin_samples:(i+1)*bin_samples] for i in range(num_bins)]
    return bins

# Extract MFCC features for each bin
def extract_mfcc_bins(bins, sr, n_mfcc=13):
    mfcc_bins = [librosa.feature.mfcc(y=bin, sr=sr, n_mfcc=n_mfcc).T for
    return mfcc_bins

# Compute cosine similarity between corresponding bins
def compute_bin_similarity(mfcc_bins1, mfcc_bins2):
    similarities = []
    for bin1, bin2 in zip(mfcc_bins1, mfcc_bins2):
        # Compute mean MFCC for each bin
        mean_mfcc1 = np.mean(bin1, axis=0).reshape(1, -1)
        mean_mfcc2 = np.mean(bin2, axis=0).reshape(1, -1)
        # Compute cosine similarity
        similarity = cosine_similarity(mean_mfcc1, mean_mfcc2)[0, 0]
        similarities.append(similarity)
    return similarities
```

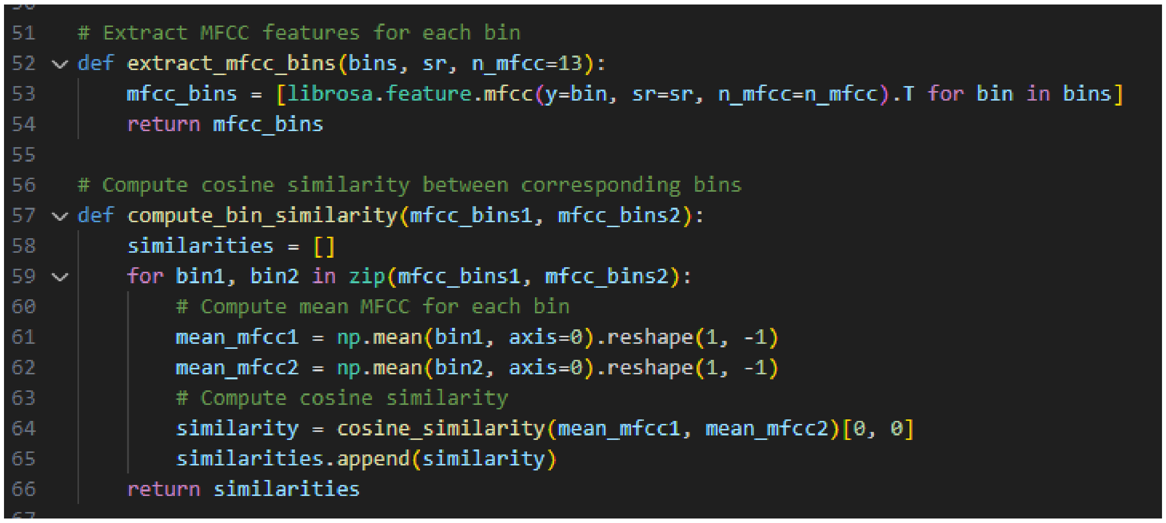
Normalize and divide into bins

Of course, these normalization steps are low-level and can be improved in the future.
After the above normalization steps, the data is ready to be fed into feature extraction modules. We use MFCC (Mel-frequency cepstral coefficients) for this purpose.

MFCC (Mel-Frequency Cepstral Coefficients) is a popular method in audio signal processing, especially for speech recognition and sound classification. MFCCs use the Mel scale, which is based on how the human ear perceives different frequencies, making them effective in capturing the characteristics of speech that humans hear and suitable for our task. Here are the key steps involved in computing MFCCs:

- Frame the signal: Divide the audio signal into small frames of about 20-40 ms.
- Apply a Hamming window: Reduce edge effects by applying a Hamming window to each frame.
- Fast Fourier Transform (FFT): Convert each frame from the time domain to the frequency domain using FFT.
- Mel filter bank: Use a set of triangular filters focused on Mel frequencies to emphasize the frequencies important to human hearing.
- Logarithm of energy: Compute the logarithm of the energy at the output of each Mel filter to mimic how humans perceive sound intensity.
- Discrete Cosine Transform (DCT): Apply DCT to transform the log Mel energies into the cepstral domain, resulting in the MFCCs.

Even though higher order coefficients represent increasing levels of spectral details, depending on the sampling rate and estimation method, 12 to 20 cepstral coefficients are typically optimal for speech analysis; and we choose 13 cepstral coefficients in our project – a common number for speech recognition tasks.



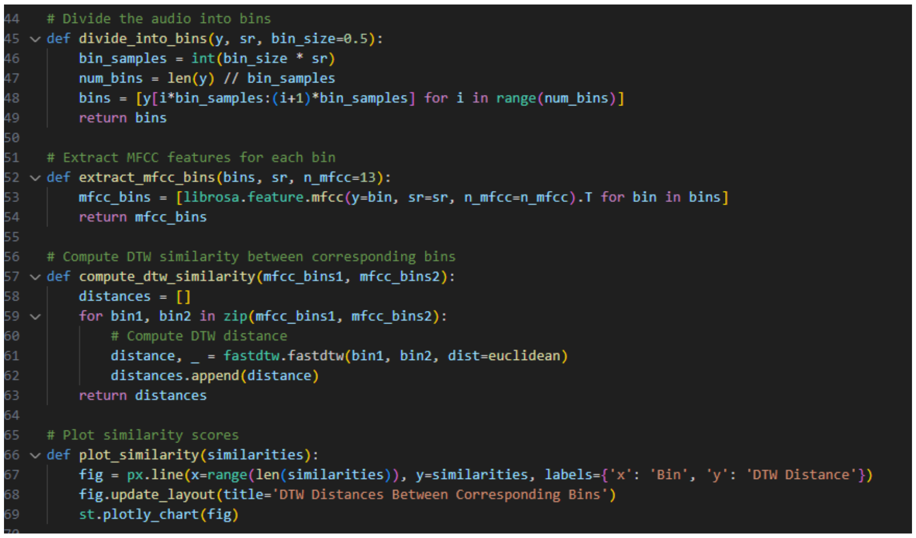
```python
# Extract MFCC features for each bin
def extract_mfcc_bins(bins, sr, n_mfcc=13):
    mfcc_bins = [librosa.feature.mfcc(y=bin, sr=sr, n_mfcc=n_mfcc).T for bin in bins]
    return mfcc_bins

# Compute cosine similarity between corresponding bins
def compute_bin_similarity(mfcc_bins1, mfcc_bins2):
    similarities = []
    for bin1, bin2 in zip(mfcc_bins1, mfcc_bins2):
        # Compute mean MFCC for each bin
        mean_mfcc1 = np.mean(bin1, axis=0).reshape(1, -1)
        mean_mfcc2 = np.mean(bin2, axis=0).reshape(1, -1)
        # Compute cosine similarity
        similarity = cosine_similarity(mean_mfcc1, mean_mfcc2)[0, 0]
        similarities.append(similarity)
    return similarities
```

Divide the audio into bins and calculate the similarity for each bin using MFCC

For comparison, we use Cosine Similarity. In the context of comparing the similarity of MFCC features: Cosine Similarity is often considered one of the optimal methods. It is effective in high-dimensional spaces and is not sensitive to the magnitude of the vectors, focusing only on the direction. This is suitable for MFCC features as they are typically normalized and directional similarity is more important.

We also experimented with other methods. One such method was DTW (Dynamic Time Warping). At its core, DTW is an algorithm designed to align and compare two time-series datasets. Unlike simpler methods that compare points based on their position in the time sequence, DTW focuses on the shape of the data. This allows it to find the optimal alignment between two time series by minimizing the distance between them, even if they are out of phase or differ in length.

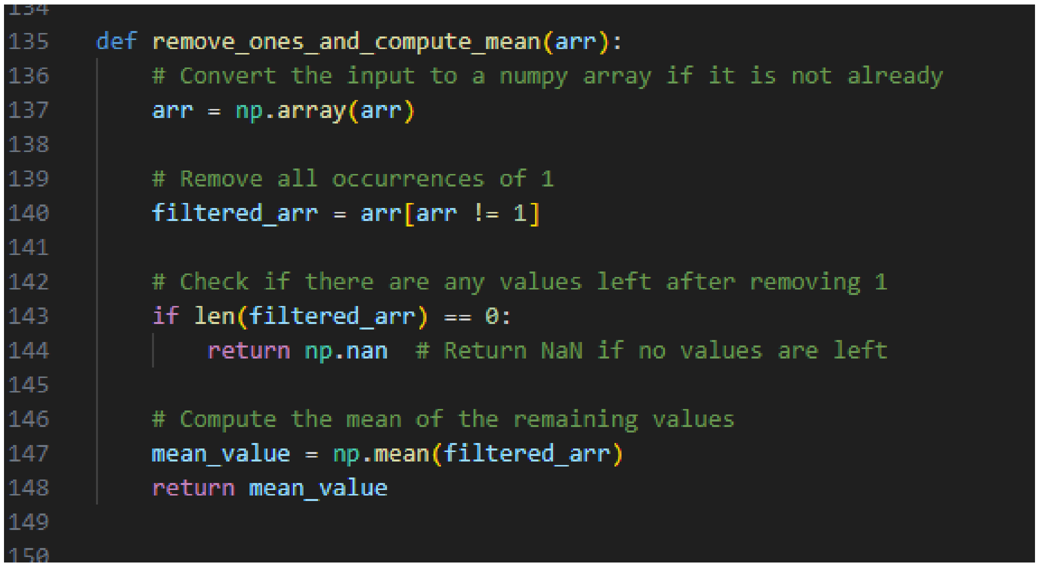
However, since we have already normalized the time domain (standardizing to 3 seconds), the advantages of DTW are not fully realized. Additionally, this method does not completely fulfill our requirements as it returns a similarity score rather than a percentage similarity between the two audio files. Therefore, we decided to use cosine similarity for the final presentation. Other comparison methods have been studied and documented in our group's Google Drive.



```python
# Divide the audio into bins
def divide_into_bins(y, sr, bin_size=0.5):
    bin_samples = int(bin_size * sr)
    num_bins = len(y) // bin_samples
    bins = [y[i*bin_samples:(i+1)*bin_samples] for i in range(num_bins)]
    return bins

# Extract MFCC features for each bin
def extract_mfcc_bins(bins, sr, n_mfcc=13):
    mfcc_bins = [librosa.feature.mfcc(y=bin, sr=sr, n_mfcc=n_mfcc).T for bin in bins]
    return mfcc_bins

# Compute DTW similarity between corresponding bins
def compute_dtw_similarity(mfcc_bins1, mfcc_bins2):
    distances = []
    for bin1, bin2 in zip(mfcc_bins1, mfcc_bins2):
        # Compute DTW distance
        distance, _ = fastdtw.fastdtw(bin1, bin2, dist=euclidean)
        distances.append(distance)
    return distances

# Plot similarity scores
def plot_similarity(similarities):
    fig = px.line(x=range(len(similarities)), y=similarities, labels={'x': 'Bin', 'y': 'DTW Distance'})
    fig.update_layout(title='DTW Distances Between Corresponding Bins')
    st.plotly_chart(fig)
```

DTW testing

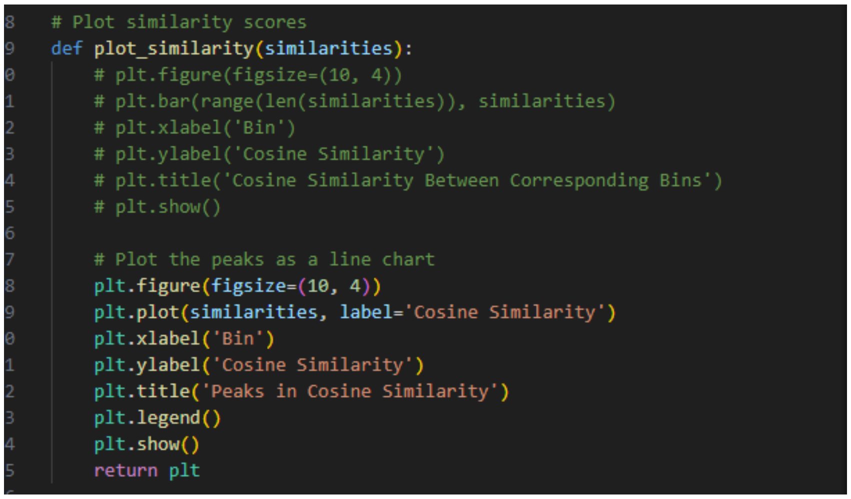
After calculating the similarity for each of the 60 bins, we compute the average value of these bins to produce the final result. However, as mentioned earlier, silent segments will yield a similarity score of 1, which biases the average towards 1. Therefore, we exclude bins with a similarity score of 1 and only average the bins with scores other than 1.



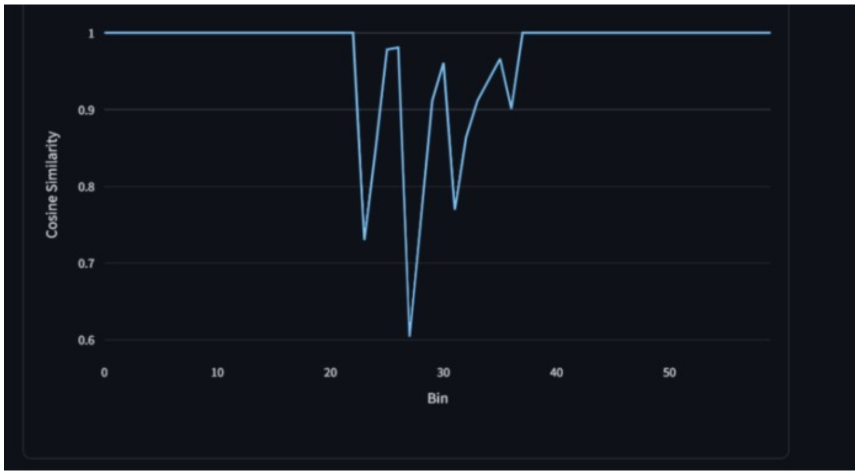
```python
def remove_ones_and_compute_mean(arr):
    # Convert the input to a numpy array if it is not already
    arr = np.array(arr)

    # Remove all occurrences of 1
    filtered_arr = arr[arr != 1]

    # Check if there are any values left after removing 1
    if len(filtered_arr) == 0:
        return np.nan  # Return NaN if no values are left

    # Compute the mean of the remaining values
    mean_value = np.mean(filtered_arr)
    return mean_value
```

Calculate mean function

Finally, we use 'matplotlib.pyplot' to plot the result.



```python
# Plot similarity scores
def plot_similarity(similarities):
    # plt.figure(figsize=(10, 4))
    # plt.bar(range(len(similarities)), similarities)
    # plt.xlabel('Bin')
    # plt.ylabel('Cosine Similarity')
    # plt.title('Cosine Similarity Between Corresponding Bins')
    # plt.show()

    # Plot the peaks as a line chart
    plt.figure(figsize=(10, 4))
    plt.plot(similarities, label='Cosine Similarity')
    plt.xlabel('Bin')
    plt.ylabel('Cosine Similarity')
    plt.title('Peaks in Cosine Similarity')
    plt.legend()
    plt.show()
    return plt
```



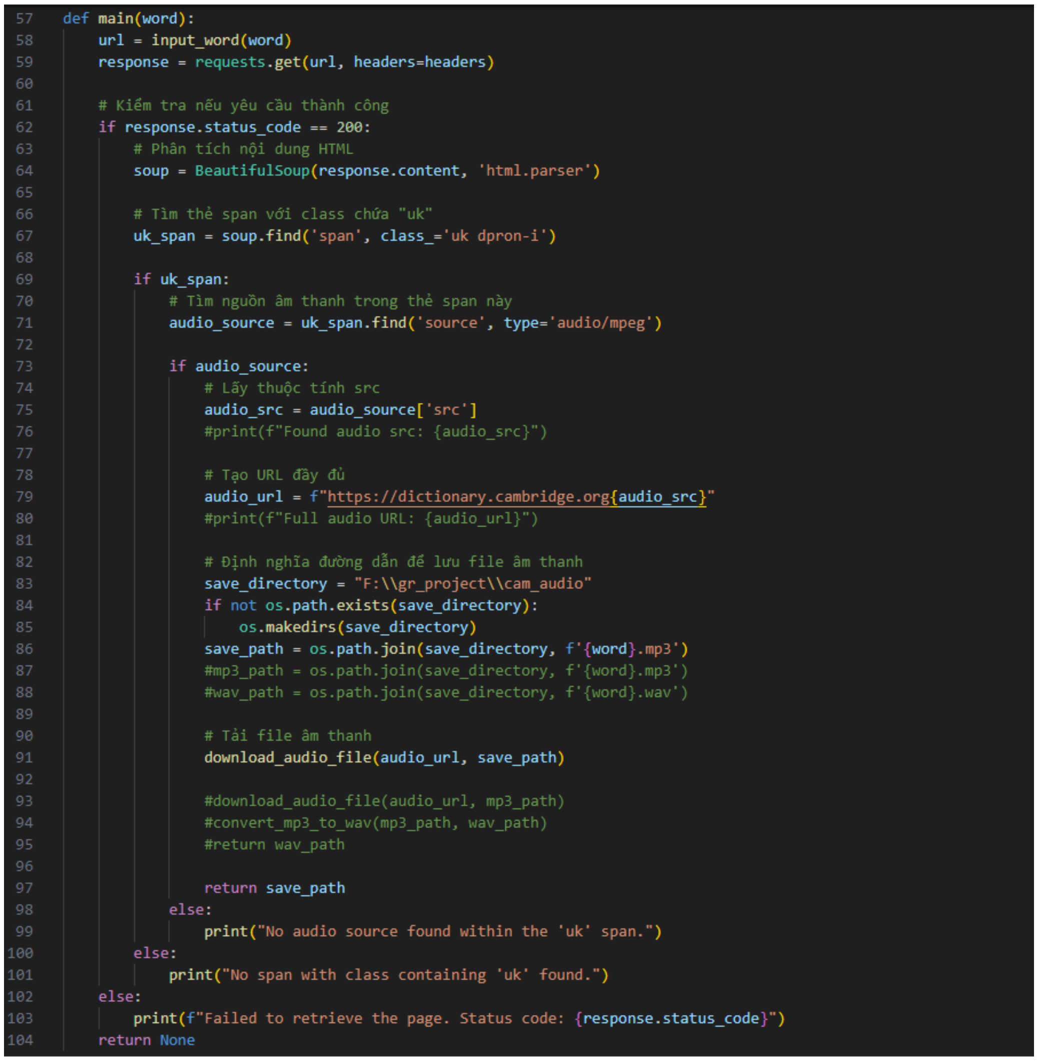
Dictionary Audio Downloader (DAD)

Downloading audio files from a dictionary is the method we use to obtain standard audio for comparison, and we believe that audio from a dictionary is sufficiently accurate to serve as the standard for users to compare with their recordings.

To achieve this, we have scraped audio data using Beautiful Soup 4 and packaged it into a web app using Streamlit. Finally, we need to convert the audio (continuing to use the librosa library) from MP3 format (the format of the audio files from the Cambridge Dictionary we selected) to WAV format to ensure compatibility with the normalization and comparison processes. We have packaged this into a web page and named it the Dictionary Audio Downloader (DAD) and all code are in file ‘get_dict.py’.

The application has two main functionalities: downloading audio files from the Cambridge Dictionary website and converting audio files from MP3 to WAV format.

To download audio files from Cambridge Dictionary, we use BeautifulSoup4 and put in function ‘main(word)’. This function is responsible for scraping a webpage to download an audio file corresponding to a word from the Cambridge Dictionary.

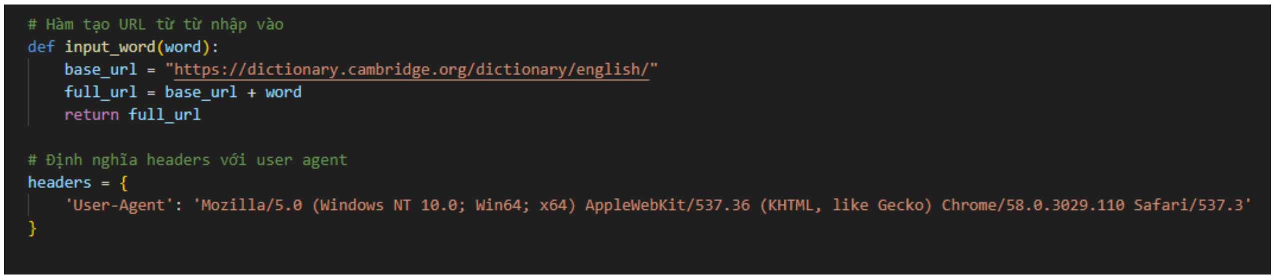


```python
def main(word):
    url = input_word(word)
    response = requests.get(url, headers=headers)

    # Kiểm tra nếu yêu cầu thành công
    if response.status_code == 200:
        # Phân tích nội dung HTML
        soup = BeautifulSoup(response.content, 'html.parser')

        # Tìm thẻ span với class chứa "uk"
        uk_span = soup.find('span', class_='uk dpron-i')

        if uk_span:
            # Tìm nguồn âm thanh trong thẻ span này
            audio_source = uk_span.find('source', type='audio/mpeg')

            if audio_source:
                # Lấy thuộc tính src
                audio_src = audio_source['src']
                #print(f"Found audio src: {audio_src}")

                # Tạo URL đầy đủ
                audio_url = f"https://dictionary.cambridge.org{audio_src}"
                #print(f"Full audio URL: {audio_url}")

                # Định nghĩa đường dẫn để lưu file âm thanh
                save_directory = "F:\\gr_project\\cam_audio"
                if not os.path.exists(save_directory):
                    os.makedirs(save_directory)
                save_path = os.path.join(save_directory, f'{word}.mp3')
                #mp3_path = os.path.join(save_directory, f'{word}.mp3')
                #wav_path = os.path.join(save_directory, f'{word}.wav')

                # Tải file âm thanh
                download_audio_file(audio_url, save_path)

                #download_audio_file(audio_url, mp3_path)
                #convert_mp3_to_wav(mp3_path, wav_path)
                #return wav_path

                return save_path
            else:
                print("No audio source found within the 'uk' span.")
        else:
            print("No span with class containing 'uk' found.")
    else:
        print(f"Failed to retrieve the page. Status code: {response.status_code}")
    return None
```

Dictionary Audio Downloader main function

The steps include the following:

- <u>Constructing the URL</u>: Generates the URL for the Cambridge Dictionary page corresponding to the input word. 'input_word(word)' appends the input word to the base URL, so we can input any single word and we will have the URL corresponding to that word at the cambridge dictionary page.

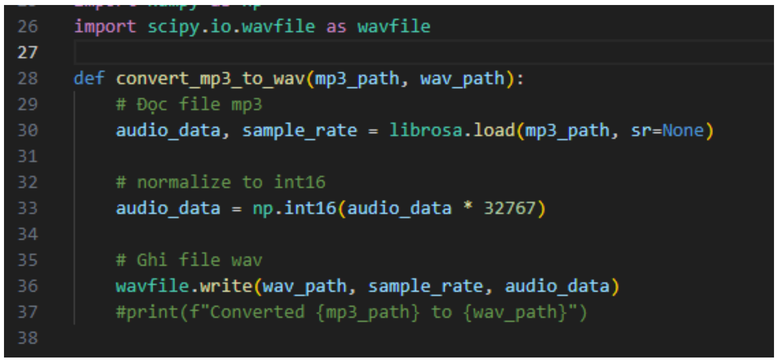


```python
# Hàm tạo URL từ từ nhập vào
def input_word(word):
    base_url = "https://dictionary.cambridge.org/dictionary/english/"
    full_url = base_url + word
    return full_url

# Định nghĩa headers với user agent
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'
}
```

- Sending an HTTP Request: Sends an HTTP GET request to the URL to retrieve the page content. ‘headers’ are included to simulate a request from a web browser (using a common User-Agent string).
- Checking the Response Status: Purpose: Checks if the HTTP request was successful (status code 200 means OK) and proceeds with parsing the content only if the request was successful.
- Parsing HTML Content: soup = BeautifulSoup(response.content, 'html.parser'). We use BeautifulSoup to parse the HTML content of the webpage.
- Finding the Span Containing Audio Source: Searches for a <span> element with the class ‘uk dpron-i’, which is likely where the UK speaker audio source is located.
- Finding the Audio Source: audio_source = uk_span.find('source', type='audio/mpeg'). If the span is found, it searches within this span for a <source> tag with type='audio/mpeg'.
- Extracting the Audio URL: Extracts the src attribute from the <source> tag to get the relative path of the audio file.
- Download and Save File: Finally, we select the file path to save the downloaded audio and return save_path

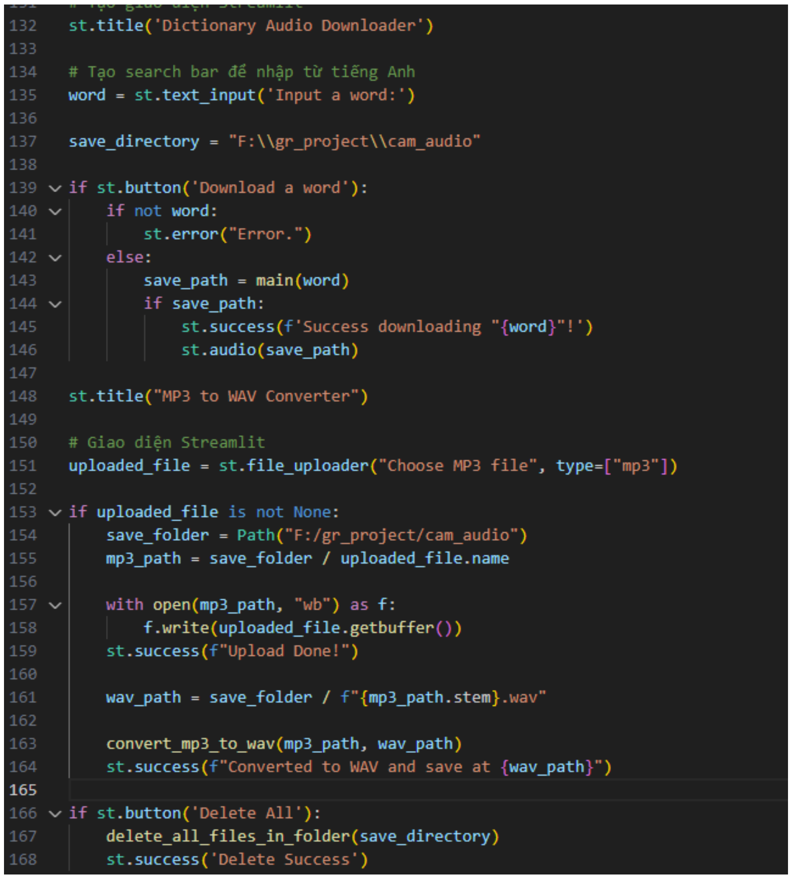
This function essentially automates the process of scraping a dictionary page for an audio pronunciation and saving it locally.

Another functionality is converting audio files from MP3 to WAV format for further processing and comparison steps. To do that, we use librosa library and normalize to int16 data type.



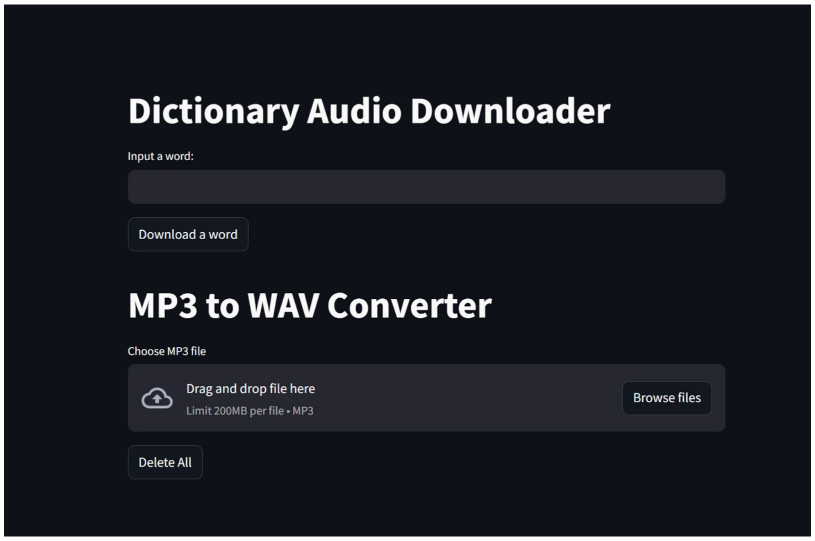
```
import numpy as np
import scipy.io.wavfile as wavfile

def convert_mp3_to_wav(mp3_path, wav_path):
    # Đọc file mp3
    audio_data, sample_rate = librosa.load(mp3_path, sr=None)

    # normalize to int16
    audio_data = np.int16(audio_data * 32767)

    # Ghi file wav
    wavfile.write(wav_path, sample_rate, audio_data)
    #print(f"Converted {mp3_path} to {wav_path}")
```

Finally, we created a basic Streamlit webpage with a text box to enter the word to be downloaded and a button to upload the file and convert it to WAV format (details can be found in the User Manual).



```python
st.title('Dictionary Audio Downloader')

# Tạo search bar để nhập từ tiếng Anh
word = st.text_input('Input a word:')

save_directory = "F:\\gr_project\\cam_audio"

if st.button('Download a word'):
    if not word:
        st.error("Error.")
    else:
        save_path = main(word)
        if save_path:
            st.success(f'Success downloading "{word}"!')
            st.audio(save_path)

st.title("MP3 to WAV Converter")

# Giao diện Streamlit
uploaded_file = st.file_uploader("Choose MP3 file", type=["mp3"])

if uploaded_file is not None:
    save_folder = Path("F:/gr_project/cam_audio")
    mp3_path = save_folder / uploaded_file.name

    with open(mp3_path, "wb") as f:
        f.write(uploaded_file.getbuffer())
    st.success(f"Upload Done!")

    wav_path = save_folder / f"{mp3_path.stem}.wav"

    convert_mp3_to_wav(mp3_path, wav_path)
    st.success(f"Converted to WAV and save at {wav_path}")

if st.button('Delete All'):
    delete_all_files_in_folder(save_directory)
    st.success('Delete Success')
```

Creating Webpage with Streamlit



DAD webpage

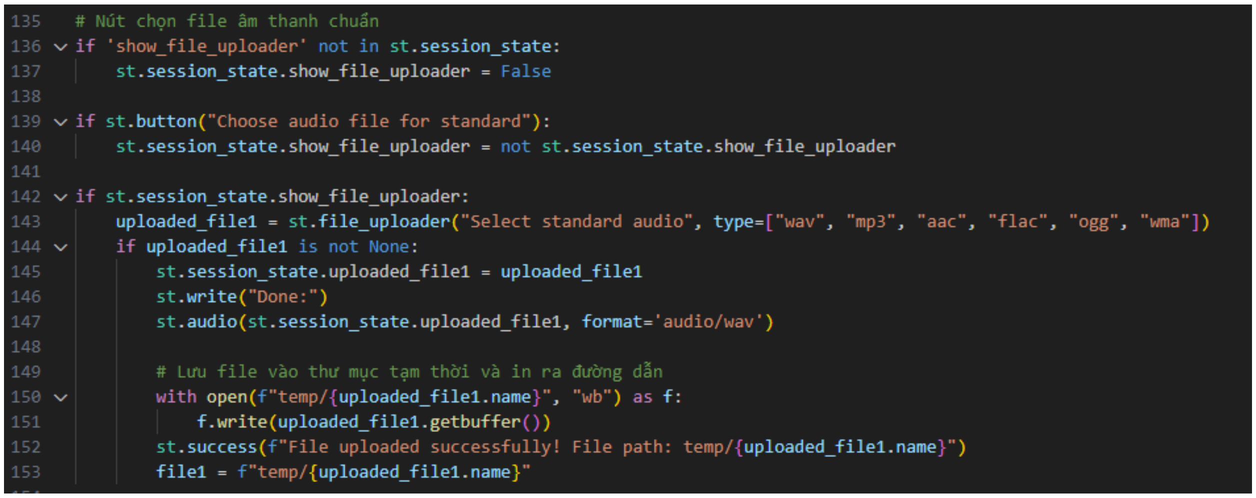
Pronunciation Analyze (PA) page

On this webpage, the objective is to compare the similarity of standard audio (obtained from the Cambridge dictionary) with user-recorded audio. Subsequently, the positions of vowels and consonants from these sounds will be analyzed to provide a visual comparison between the two audio samples. Additionally, the similarity in the sounds of these vowels and consonants will be examined. Buttons and web interfaces using Streamlit.

The webpage will include the following main features:

- Upload an audio file to be used as the standard (a wav file from the DAD page).
- Record the user's pronunciation of the word.
- Conduct a comparison and scoring.
- Analyze the score.
- Identify vowels and consonants, generate plots, and analyze the similarity in the sounds of these vowels and consonants.

<u>Upload an audio file to be used as the standard:</u>



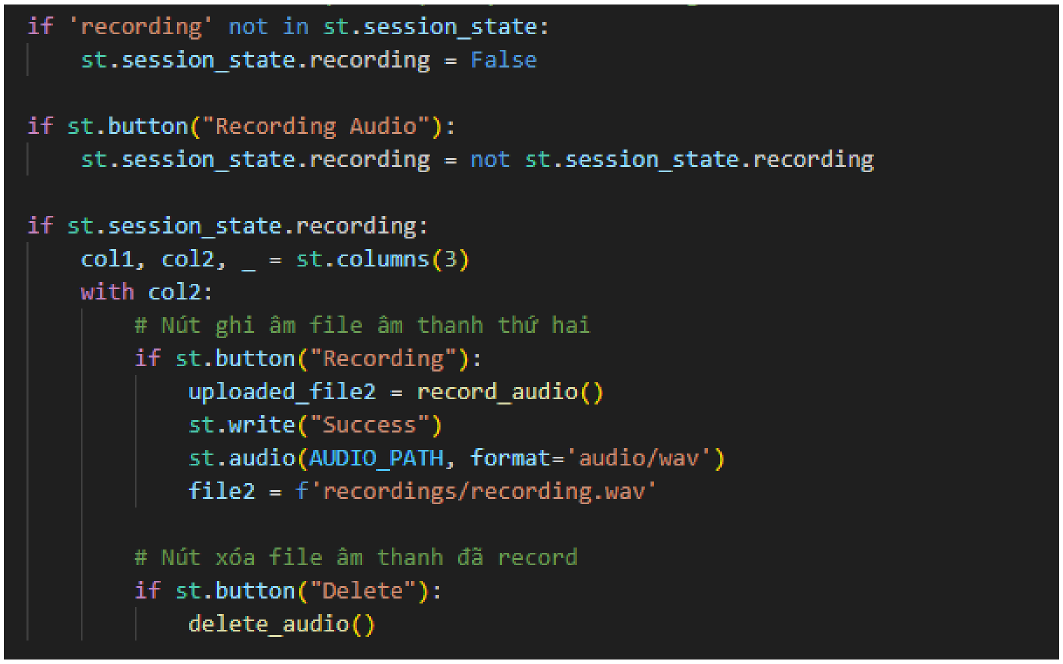
```python
# Nút chọn file âm thanh chuẩn
if 'show_file_uploader' not in st.session_state:
    st.session_state.show_file_uploader = False

if st.button("Choose audio file for standard"):
    st.session_state.show_file_uploader = not st.session_state.show_file_uploader

if st.session_state.show_file_uploader:
    uploaded_file1 = st.file_uploader("Select standard audio", type=["wav", "mp3", "aac", "flac", "ogg", "wma"])
    if uploaded_file1 is not None:
        st.session_state.uploaded_file1 = uploaded_file1
        st.write("Done:")
        st.audio(st.session_state.uploaded_file1, format='audio/wav')

        # Lưu file vào thư mục tạm thời và in ra đường dẫn
        with open(f"temp/{uploaded_file1.name}", "wb") as f:
            f.write(uploaded_file1.getbuffer())
        st.success(f"File uploaded successfully! File path: temp/{uploaded_file1.name}")
        file1 = f"temp/{uploaded_file1.name}"
```

Upload audio file function

Create session_state variables to serve as flags for the webpage. When the user clicks a button, it will transition to the upload audio file section.

Use the `file_uploader` function in Streamlit to upload the audio file selected as the standard. The user will select a .wav file downloaded from the DAD webpage, located in the cam_audio folder. Subsequently, save the audio file in a specified directory with the name of the downloaded word. Finally, assign the variable `file1` to the path of the audio file to facilitate comparison later.

Record the user's pronunciation of the word:



```python
if 'recording' not in st.session_state:
    st.session_state.recording = False

if st.button("Recording Audio"):
    st.session_state.recording = not st.session_state.recording

if st.session_state.recording:
    col1, col2, _ = st.columns(3)
    with col2:
        # Nút ghi âm file âm thanh thứ hai
        if st.button("Recording"):
            uploaded_file2 = record_audio()
            st.write("Success")
            st.audio(AUDIO_PATH, format='audio/wav')
            file2 = f'recordings/recording.wav'

        # Nút xóa file âm thanh đã record
        if st.button("Delete"):
            delete_audio()
```

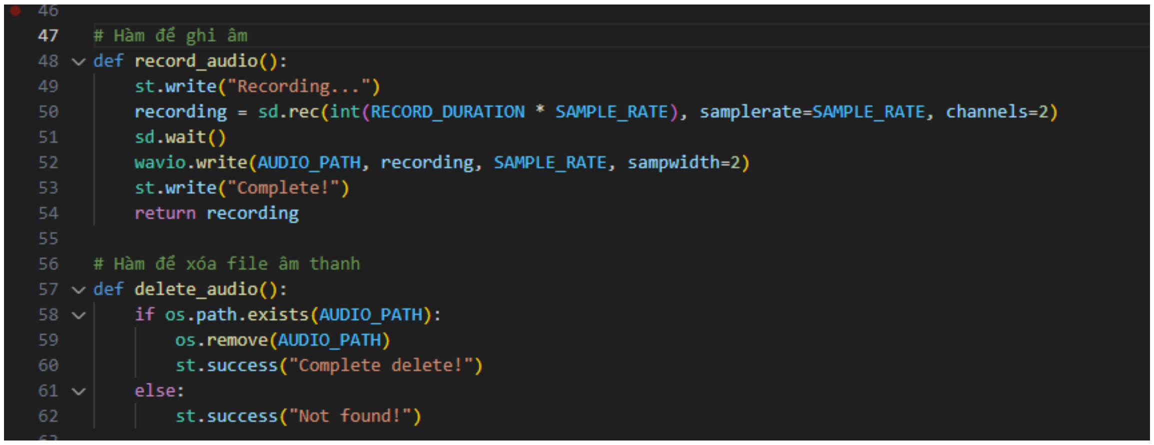
Recording function

Similar to uploading the standard audio file, create session_state variables to support the user interface.

Use the `sd` (imported as `import sounddevice as sd`) function to record an audio file with the specified parameters, such as sample rate and record duration mentioned earlier.

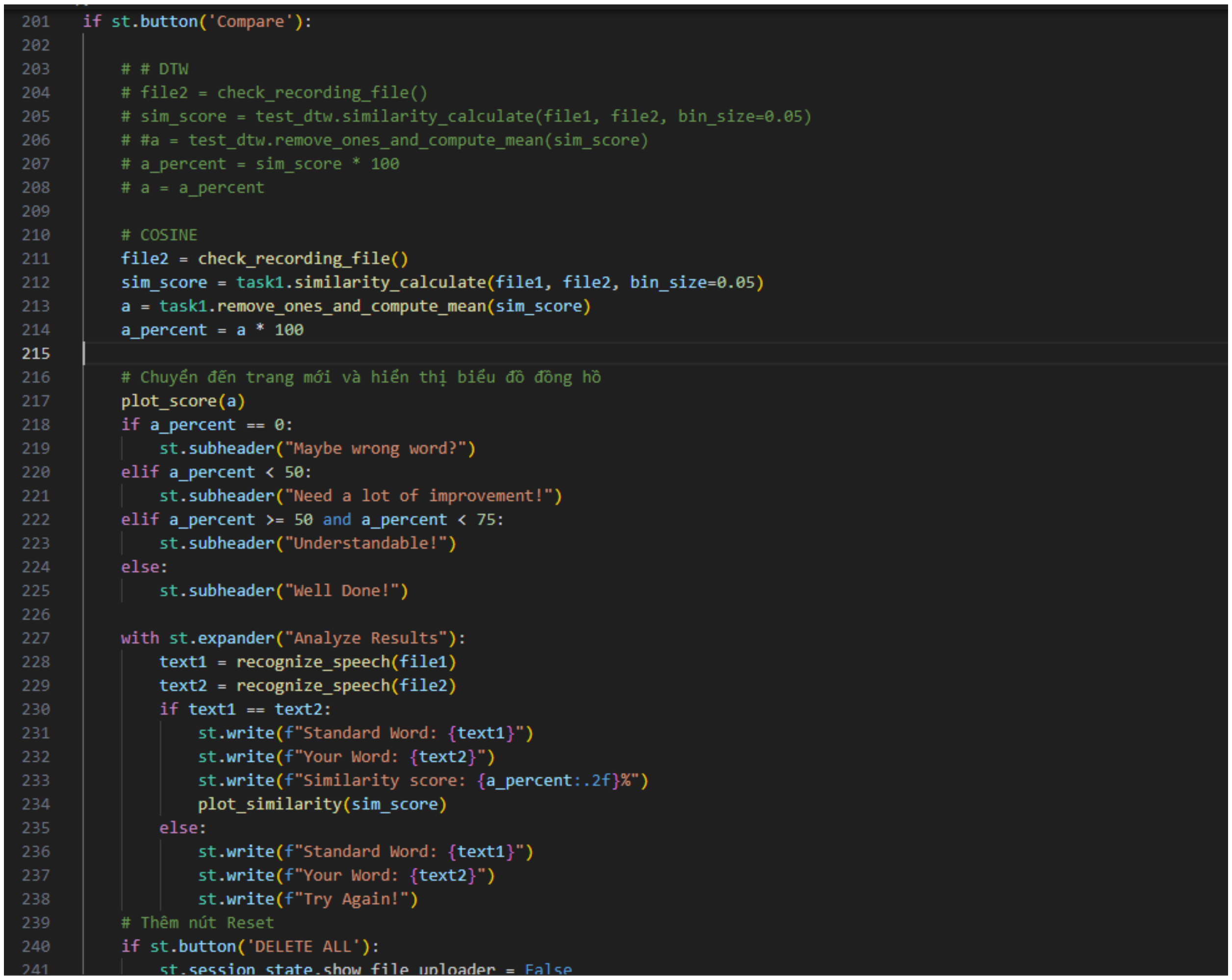
Create an additional button to delete the recorded file if the user wishes to retry. The user can listen to their recorded audio using the `st.audio` function to display an audio player.

Similarly, assign the variable `file2` to the path of the recorded file to facilitate comparison later.



```python
# Hàm để ghi âm
def record_audio():
    st.write("Recording...")
    recording = sd.rec(int(RECORD_DURATION * SAMPLE_RATE), samplerate=SAMPLE_RATE, channels=2)
    sd.wait()
    wavio.write(AUDIO_PATH, recording, SAMPLE_RATE, sampwidth=2)
    st.write("Complete!")
    return recording

# Hàm để xóa file âm thanh
def delete_audio():
    if os.path.exists(AUDIO_PATH):
        os.remove(AUDIO_PATH)
        st.success("Complete delete!")
    else:
        st.success("Not found!")
```

Conduct a comparison and scoring:



```python
if st.button('Compare'):

    # # DTW
    # file2 = check_recording_file()
    # sim_score = test_dtw.similarity_calculate(file1, file2, bin_size=0.05)
    # #a = test_dtw.remove_ones_and_compute_mean(sim_score)
    # a_percent = sim_score * 100
    # a = a_percent

    # COSINE
    file2 = check_recording_file()
    sim_score = task1.similarity_calculate(file1, file2, bin_size=0.05)
    a = task1.remove_ones_and_compute_mean(sim_score)
    a_percent = a * 100

    # Chuyển đến trang mới và hiển thị biểu đồ đồng hồ
    plot_score(a)
    if a_percent == 0:
        st.subheader("Maybe wrong word?")
    elif a_percent < 50:
        st.subheader("Need a lot of improvement!")
    elif a_percent >= 50 and a_percent < 75:
        st.subheader("Understandable!")
    else:
        st.subheader("Well Done!")

    with st.expander("Analyze Results"):
        text1 = recognize_speech(file1)
        text2 = recognize_speech(file2)
        if text1 == text2:
            st.write(f"Standard Word: {text1}")
            st.write(f"Your Word: {text2}")
            st.write(f"Similarity score: {a_percent:.2f}%")
            plot_similarity(sim_score)
        else:
            st.write(f"Standard Word: {text1}")
            st.write(f"Your Word: {text2}")
            st.write(f"Try Again!")
    # Thêm nút Reset
    if st.button('DELETE ALL'):
        st.session_state.show_file_uploader = False
```

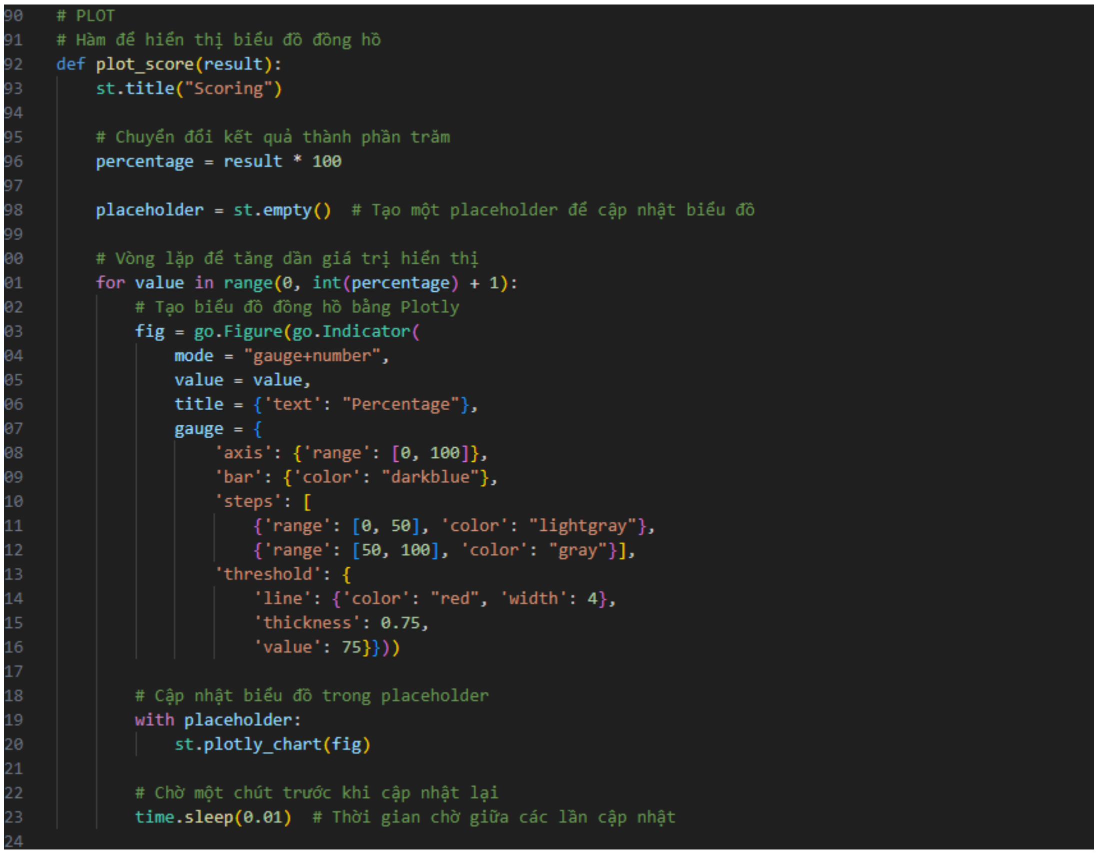
Comparison function

After assigning the variables `file1` and `file2` to the paths of the uploaded and recorded audio files respectively, we create a "Compare" button to initiate the comparison between these two audio files.

We utilize the Cosine similarity functions developed earlier (including the DTW similarity test file) to score the similarity between the two audio files.

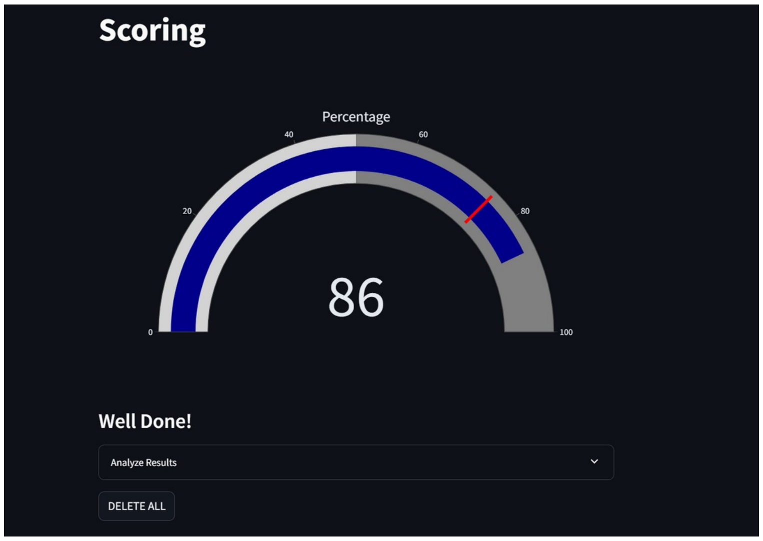
To provide a user-friendly visualization of the results based on the scored similarity, we use the Plotly library to create the interface. Threshold values and accompanying annotations can be modified in the code between lines 217 and 238. Regarding the similarity comparison, the workflow is as follows: We use the speech_recognition library to identify the word from both audio files before calculating the cosine similarity between the two sounds. If the two audio files are not recognized as the same word, the similarity score will be 0, and a message "Maybe wrong word?" will be displayed to the user.

Finally, a "Delete All" button is provided to remove all temporary audio files, allowing the user to reset and restart the entire process from the beginning.

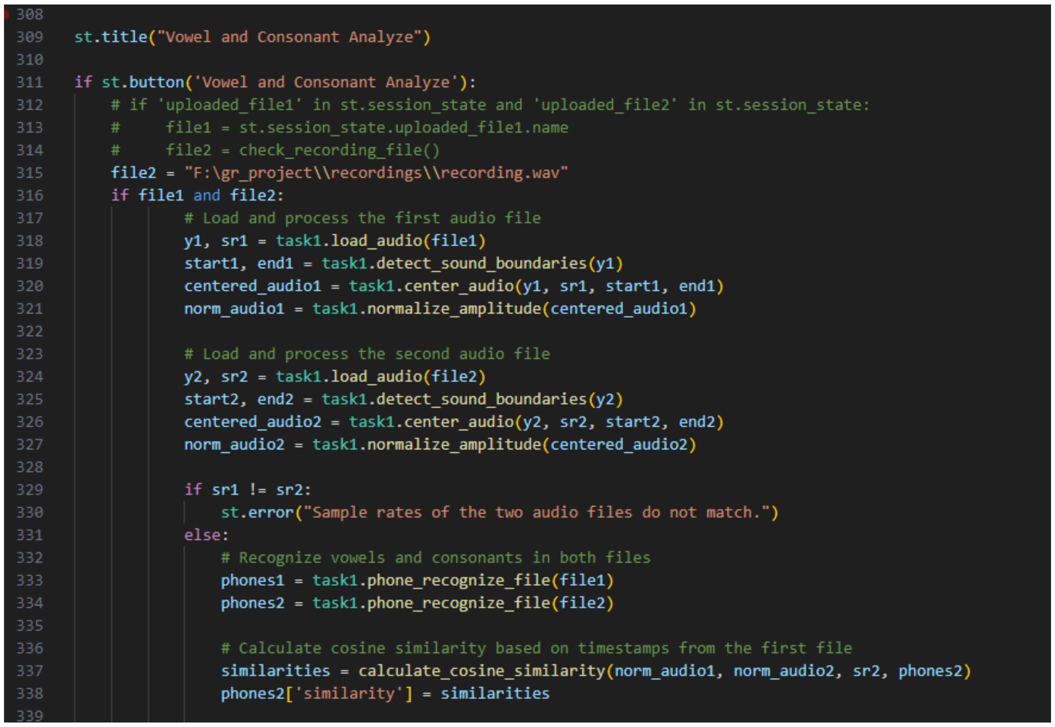


```python
# PLOT
# Hàm để hiển thị biểu đồ đồng hồ
def plot_score(result):
    st.title("Scoring")

    # Chuyển đổi kết quả thành phần trăm
    percentage = result * 100

    placeholder = st.empty()  # Tạo một placeholder để cập nhật biểu đồ

    # Vòng lặp để tăng dần giá trị hiển thị
    for value in range(0, int(percentage) + 1):
        # Tạo biểu đồ đồng hồ bằng Plotly
        fig = go.Figure(go.Indicator(
            mode = "gauge+number",
            value = value,
            title = {'text': "Percentage"},
            gauge = {
                'axis': {'range': [0, 100]},
                'bar': {'color': "darkblue"},
                'steps': [
                    {'range': [0, 50], 'color': "lightgray"},
                    {'range': [50, 100], 'color': "gray"}],
                'threshold': {
                    'line': {'color': "red", 'width': 4},
                    'thickness': 0.75,
                    'value': 75}}))

        # Cập nhật biểu đồ trong placeholder
        with placeholder:
            st.plotly_chart(fig)

        # Chờ một chút trước khi cập nhật lại
        time.sleep(0.01)  # Thời gian chờ giữa các lần cập nhật
```

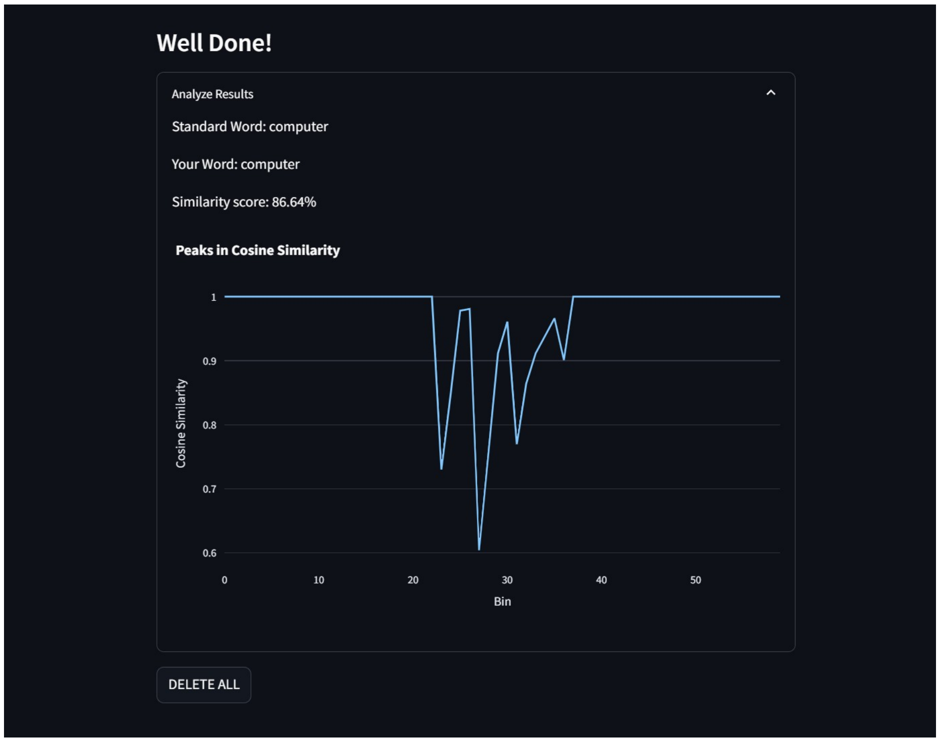
plot_score function



Analyze the score:



```python

st.title("Vowel and Consonant Analyze")

if st.button('Vowel and Consonant Analyze'):
    # if 'uploaded_file1' in st.session_state and 'uploaded_file2' in st.session_state:
    #     file1 = st.session_state.uploaded_file1.name
    #     file2 = check_recording_file()
    file2 = "F:\gr_project\\recordings\\recording.wav"
    if file1 and file2:
            # Load and process the first audio file
            y1, sr1 = task1.load_audio(file1)
            start1, end1 = task1.detect_sound_boundaries(y1)
            centered_audio1 = task1.center_audio(y1, sr1, start1, end1)
            norm_audio1 = task1.normalize_amplitude(centered_audio1)

            # Load and process the second audio file
            y2, sr2 = task1.load_audio(file2)
            start2, end2 = task1.detect_sound_boundaries(y2)
            centered_audio2 = task1.center_audio(y2, sr2, start2, end2)
            norm_audio2 = task1.normalize_amplitude(centered_audio2)

            if sr1 != sr2:
                st.error("Sample rates of the two audio files do not match.")
            else:
                # Recognize vowels and consonants in both files
                phones1 = task1.phone_recognize_file(file1)
                phones2 = task1.phone_recognize_file(file2)

                # Calculate cosine similarity based on timestamps from the first file
                similarities = calculate_cosine_similarity(norm_audio1, norm_audio2, sr2, phones2)
                phones2['similarity'] = similarities

```


Well Done!

Analyze Results

Standard Word: computer

Your Word: computer

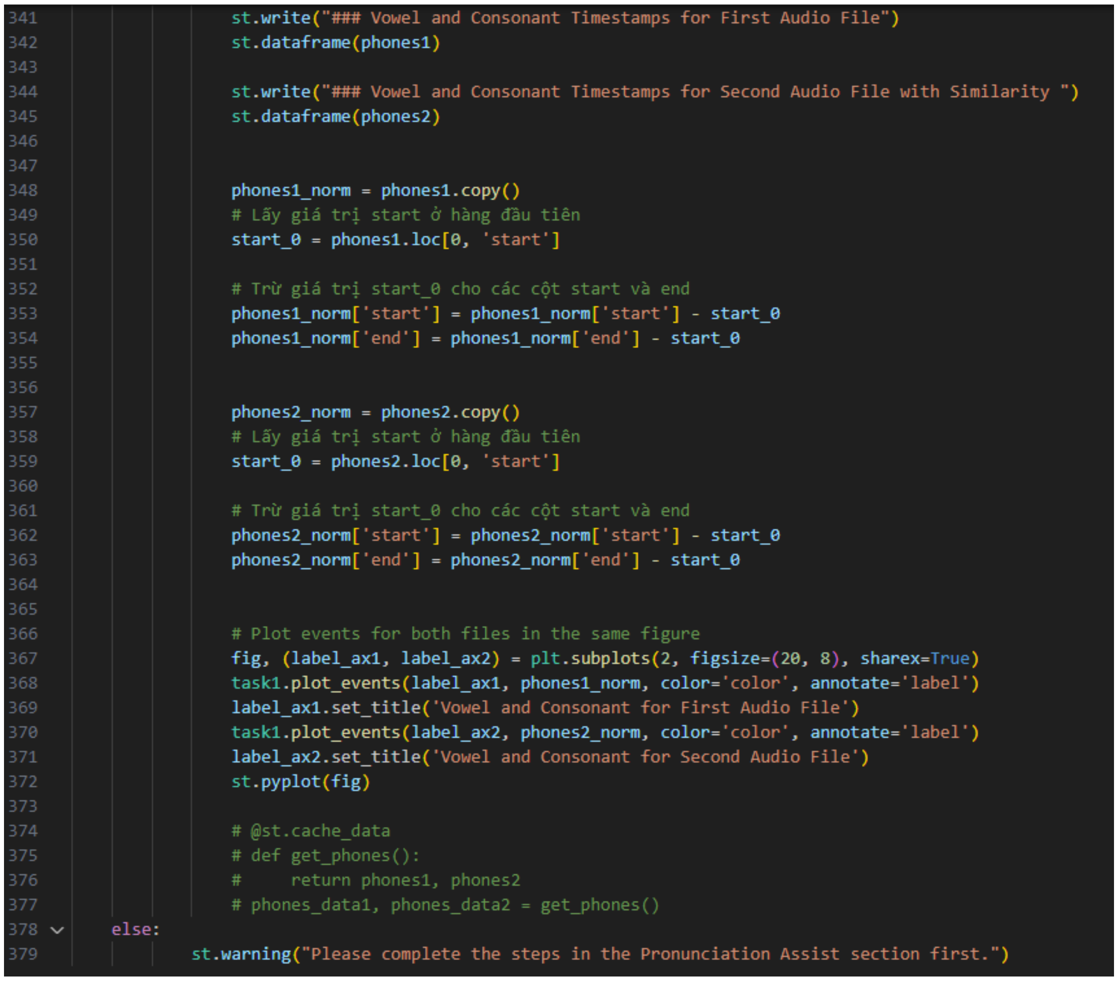
Similarity score: 86.64%

Analyze Result

After comparing and scoring the similarity between the two audio files, an additional option to analyze the results will be available.

Utilize the plotting function from the cosine similarity mentioned earlier to generate a similarity plot between the two audio files. Additionally, the recognized words (Standard Word vs. Your Word) and the similarity score will be shown after using the "Analyze Result" button.

<u>Identify vowels and consonants, generate plots, and analyze the similarity in the sounds of these vowels and consonants:</u>

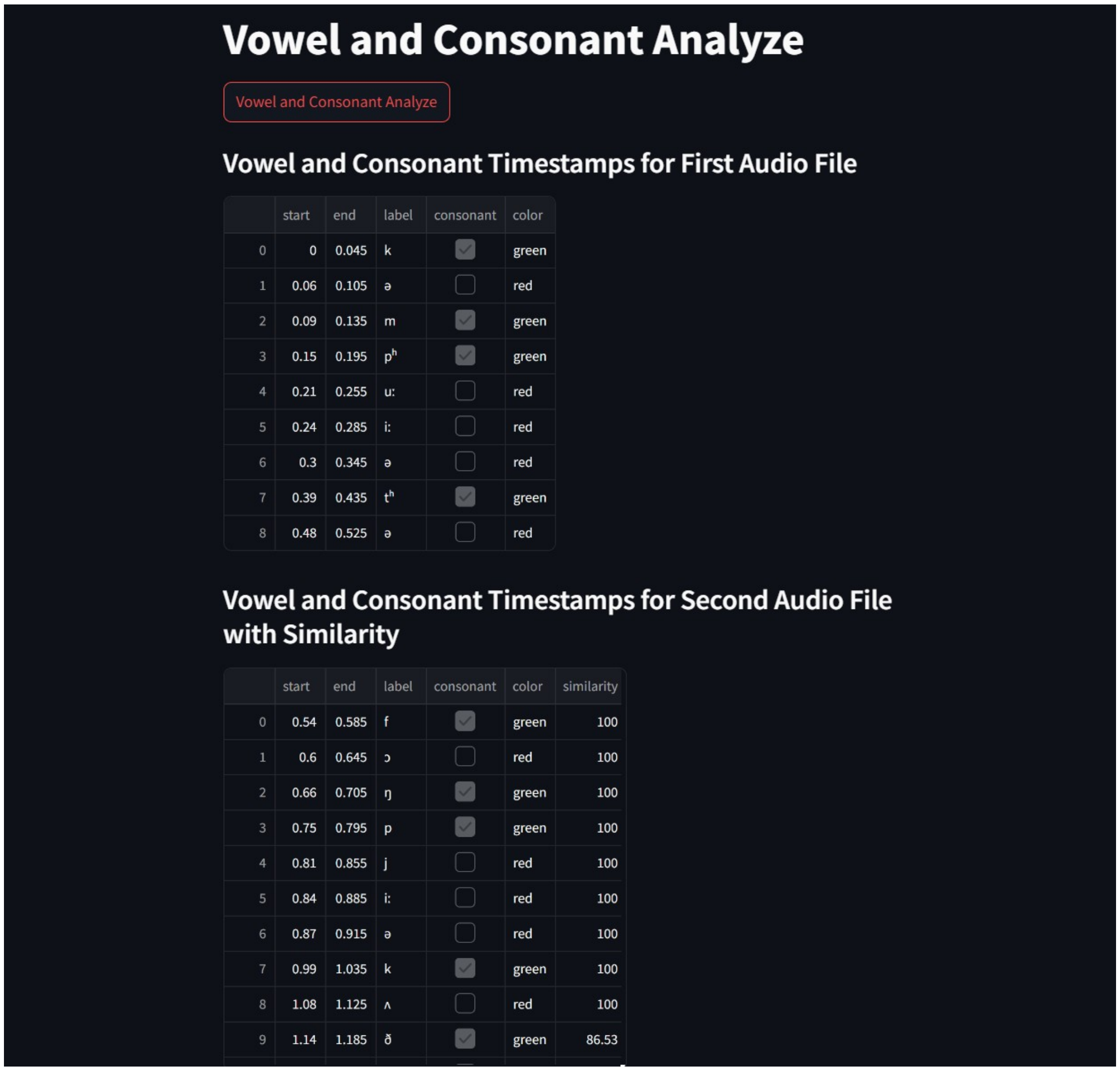


```python
                st.write("### Vowel and Consonant Timestamps for First Audio File")
                st.dataframe(phones1)

                st.write("### Vowel and Consonant Timestamps for Second Audio File with Similarity ")
                st.dataframe(phones2)


                phones1_norm = phones1.copy()
                # Lấy giá trị start ở hàng đầu tiên
                start_0 = phones1.loc[0, 'start']

                # Trừ giá trị start_0 cho các cột start và end
                phones1_norm['start'] = phones1_norm['start'] - start_0
                phones1_norm['end'] = phones1_norm['end'] - start_0


                phones2_norm = phones2.copy()
                # Lấy giá trị start ở hàng đầu tiên
                start_0 = phones2.loc[0, 'start']

                # Trừ giá trị start_0 cho các cột start và end
                phones2_norm['start'] = phones2_norm['start'] - start_0
                phones2_norm['end'] = phones2_norm['end'] - start_0


                # Plot events for both files in the same figure
                fig, (label_ax1, label_ax2) = plt.subplots(2, figsize=(20, 8), sharex=True)
                task1.plot_events(label_ax1, phones1_norm, color='color', annotate='label')
                label_ax1.set_title('Vowel and Consonant for First Audio File')
                task1.plot_events(label_ax2, phones2_norm, color='color', annotate='label')
                label_ax2.set_title('Vowel and Consonant for Second Audio File')
                st.pyplot(fig)

                # @st.cache_data
                # def get_phones():
                #     return phones1, phones2
                # phones_data1, phones_data2 = get_phones()
    else:
            st.warning("Please complete the steps in the Pronunciation Assist section first.")
```

Vowel and Consonant Analyze

Continuing with the score analysis, we add a feature for identifying vowels and consonants. We use the Allosaurus library, a pretrained universal phone recognizer (GitHub: https://github.com/xinjli/allosaurus). The model will identify vowels and consonants in the input audio file with timestamps; start and end indicate the boundaries of the sound segment containing the vowel or consonant. Labels are assigned with green for consonants and red for vowels.

Additionally, we plot two charts to allow users to compare their recorded audio with the uploaded standard audio. For the recorded audio file, we create an additional column for similarity, which calculates the similarity between the audio segment at a given position and the corresponding position in the standard audio file. This provides users with a more detailed insight into their pronunciation of vowels and consonants.



Vowel and Consonant Analyze

Vowel and Consonant Analyze

Vowel and Consonant Timestamps for First Audio File

	start	end	label	consonant	color
0	0	0.045	k	✓	green
1	0.06	0.105	ə		red
2	0.09	0.135	m	✓	green
3	0.15	0.195	pʰ	✓	green
4	0.21	0.255	u:		red
5	0.24	0.285	i:		red
6	0.3	0.345	ə		red
7	0.39	0.435	tʰ	✓	green
8	0.48	0.525	ə		red

Vowel and Consonant Timestamps for Second Audio File with Similarity

	start	end	label	consonant	color	similarity
0	0.54	0.585	f	✓	green	100
1	0.6	0.645	ɔ		red	100
2	0.66	0.705	ŋ	✓	green	100
3	0.75	0.795	p	✓	green	100
4	0.81	0.855	j		red	100
5	0.84	0.885	i:		red	100
6	0.87	0.915	ə		red	100
7	0.99	1.035	k	✓	green	100
8	1.08	1.125	ʌ		red	100
9	1.14	1.185	ð	✓	green	86.53

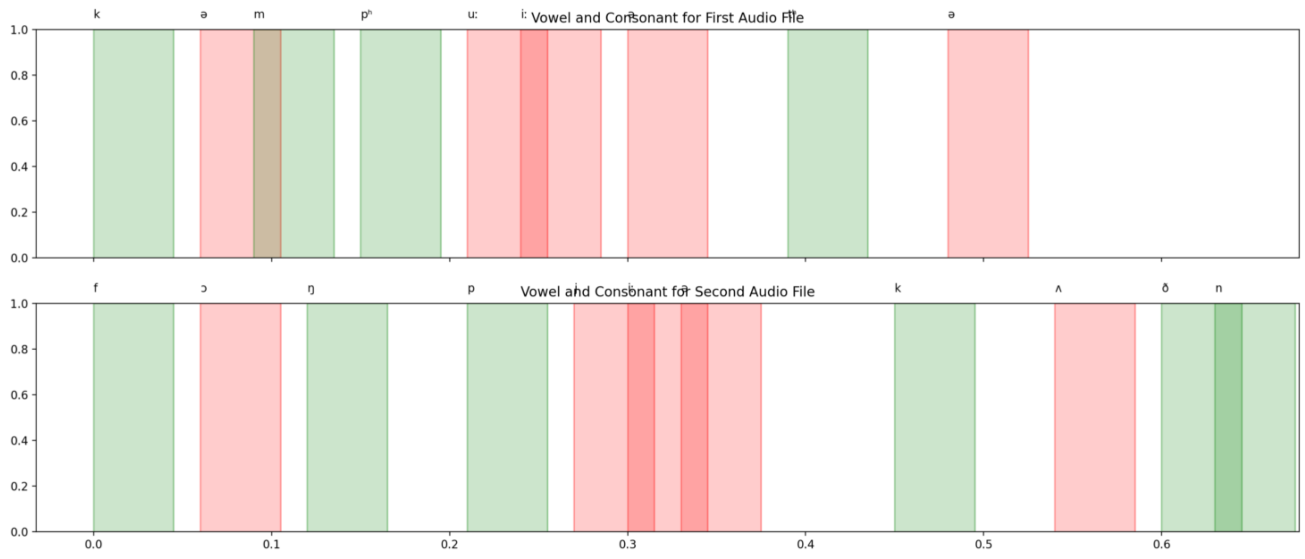


Illustration results for record audio and upload audio