PROJECT REPORT ON

VOICE RECOGNITION-SURVEILLANCE

BY:

NAME : MAHIN RAJAN

ROLL NUMBER:2K22/EC/143

Introduction

Voice recognition technology has evolved as a pivotal innovation in securing and interacting with modern systems. This project, **Voice Recognition - Application Surveillance**, integrates advanced signal processing techniques with voice recognition algorithms to create a robust and efficient surveillance system. The goal is to develop a system that identifies potential security threats by analyzing voice signals for specific words and patterns associated with unauthorized or malicious activities.

CODE IMPLEMETATION:

Copy code

```
# Step 1: Upload Audio File
uploaded = files.upload()
filename = list(uploaded.keys())[0]
sample_rate, data = wavfile.read(filename)

# Print File Details
print(f"File Name: {filename}")
print(f"Sample Rate: {sample_rate} Hz")
print(f"Data Shape: {data.shape}")

# Extract Signal
if len(data.shape) > 1:
    signal = data[:, 0]
else:
    signal = data

# Step 2: Homomorphic Transformation
def homomorphic_transformation(signal):
    spectrum = np.fft.fft(signal)
    magnitude = np.abs(spectrum)
    log_spectrum = np.log(magnitude + 1e-10)
    cepstrum = np.fft.ifft(log_spectrum).real
    return cepstrum

cepstrum = homomorphic_transformation(signal)
```

Copy code

```
# Step 3: LVQ Quantization
def lvq_quantization(cepstrum, n_clusters=8):
    cepstrum = cepstrum.reshape(-1, 1)
    kmeans = MiniBatchKMeans(n_clusters=n_clusters, random_state=0)
    kmeans.fit(cepstrum)
    quantized_values = kmeans.cluster_centers_[kmeans.predict(cepstrum)].flatten()
    return quantized_values, kmeans.cluster_centers_

n_clusters = 8
quantized_cepstrum, cluster_centers = lvq_quantization(cepstrum, n_clusters=n_clusters)
print(f"Cluster Centers:\n{cluster_centers.flatten()}")

# Step 4: Signal and Noise Separation
def separate_signal_and_noise(quantized_data, noise_threshold):
    signal_mask = quantized_data > noise_threshold
    noise_mask = ~signal_mask
    signal_part = quantized_data * signal_mask
    noise_part = quantized_data * noise_mask
    return signal_part, noise_part

noise_threshold = 0.05
signal_data, noise_data = separate_signal_and_noise(quantized_cepstrum, noise_threshold)
```

```python
# Step 5: Visualization
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(signal, color='gray')
plt.title('Original Signal')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')

plt.subplot(3, 1, 2)
plt.plot(quantized_cepstrum, color='blue')
plt.title('Quantized Cepstrum')
plt.xlabel('Quefrency Index')
plt.ylabel('Amplitude')

plt.subplot(3, 1, 3)
plt.plot(signal_data, color='green', label='Signal')
plt.plot(noise_data, color='red', label='Noise')
plt.title('Signal and Noise Separation')
plt.xlabel('Quefrency Index')
plt.ylabel('Amplitude')
plt.legend()
plt.tight_layout()
plt.show()

# Step 6: Brownian Motion Analysis
def plot_brownian_like_signal(signal_data, title="Signal resembling Brownian Motion"):
    if np.max(signal_data) - np.min(signal_data) < 1e-3:
        signal_data += np.random.normal(scale=0.01, size=signal_data.shape)
    cumulative_signal = np.cumsum(signal_data)
    plt.figure(figsize=(10, 5))
    plt.plot(cumulative_signal, color='b', label='Cumulative Sum (Brownian-like)')
    plt.title(title)
    plt.xlabel('Sample Index')
    plt.ylabel('Amplitude')
    plt.legend()
    plt.grid()
    plt.show()
```

```python
# Step 7: Maximum Likelihood Analysis
def distance(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def angle_between_vectors(v1, v2):
    dot_product = np.dot(v1, v2)
    magnitude_v1 = np.linalg.norm(v1)
    magnitude_v2 = np.linalg.norm(v2)
    cos_theta = dot_product / (magnitude_v1 * magnitude_v2)
    return np.degrees(np.arccos(np.clip(cos_theta, -1.0, 1.0)))

indices = [100, 300, 700]
cumsum_signal = np.cumsum(signal_data)
points = [(idx, cumsum_signal[idx]) for idx in indices]

sides = [distance(points[i], points[(i + 1) % len(points)]) for i in range(len(points))]
angles = [
    angle_between_vectors(
        np.array([points[(i + 1) % len(points)][0] - points[i][0], points[(i + 1) % len(po
        np.array([points[(i - 1) % len(points)][0] - points[i][0], points[(i - 1) % len(po
    )
    for i in range(len(points))
]

print("Triangle Side Lengths:", sides)
print("Triangle Angles (degrees):", angles)

def maximum_likelihood_analysis(sides, angles):
    side_mean, side_std = norm.fit(sides)
    angle_mean, angle_std = norm.fit(angles)
    return {
        "Side Mean": side_mean,
        "Side Std Dev": side_std,
        "Angle Mean": angle_mean,
        "Angle Std Dev": angle_std,
    }

ml_results = maximum_likelihood_analysis(sides, angles)
print("ML Analysis Results:", ml_results)
```

```python
# Step 8: Speech-to-Text and Code Word Detection
!pip install pydub SpeechRecognition

def convert_to_wav(file_path):
    audio = AudioSegment.from_file(file_path)
    audio = audio.set_channels(1).set_frame_rate(16000).set_sample_width(2)
    audio.export("converted_audio.wav", format="wav")
    return "converted_audio.wav"

converted_file = convert_to_wav(filename)

CODE_WORDS = {"hijack", "tiger", "emergency", "help"}

def detect_code_words(text, code_words):
    return [word for word in code_words if word in text.lower()]

def process_audio_file(file_path):
    recognizer = sr.Recognizer()
    with sr.AudioFile(file_path) as source:
        audio = recognizer.record(source)
    text = recognizer.recognize_google(audio)
    print(f"Transcribed Text: {text}")
    detected = detect_code_words(text, CODE_WORDS)
    if detected:
        print(f"⚠️ Code Words Detected: {', '.join(detected)}")
    else:
        print("No code words detected.")

process_audio_file(converted_file)
```

Implementation

The implementation of the **Voice Recognition - Application Surveillance** system involves various stages, leveraging advanced signal processing, machine learning, and speech recognition techniques. Below are the details of each step:

1. Importing and Preprocessing the Audio Signal

The system begins with importing the audio file (e.g., .wav) and performing preprocessing to enhance the signal for analysis:

- **Homomorphic Transformation:**
    - The Fourier Transform of the signal is computed to separate the convolutional components.

- Logarithmic transformation is applied to the magnitude spectrum to simplify the signal's structure.
- The Inverse Fourier Transform is used to compute the cepstrum, which is a critical feature for analysis.

2. Quantization Using Learning Vector Quantization (LVQ)

- The cepstral coefficients are quantized using MiniBatch K-Means, a lightweight clustering algorithm.
- This reduces the dimensionality and groups the data into n_clusters (default: 8).
- Quantized values are used as features for separating meaningful signals from noise.

3. Signal and Noise Separation

The quantized data is separated into:

- **Signal Component:** Clean audio data representing the meaningful voice signal.
- **Noise Component:** Unwanted background noise or irrelevant signals.
  A threshold-based mask is used to distinguish between the two components.

4. Statistical Analysis of the Signal

- The clean signal is modeled as **Brownian Motion** by calculating its cumulative sum.

- **Triangle Analysis:**
  - Three points on the signal are selected to form a triangle.
  - **Distances (sides)** and **angles** of the triangle are calculated.
  - Statistical properties of these features are extracted to identify patterns in the signal.
- **Maximum Likelihood Estimation (MLE):**
  - Side lengths and angles are modeled using normal distributions.
  - The system identifies the most likely values based on maximum likelihood.

5. Converting Audio to Text

- Audio files are converted to .wav format for uniformity using the pydub library.
- The **SpeechRecognition** library is used to process the .wav file and transcribe the spoken words into text.

6. Keyword Detection for Security Surveillance

- A predefined set of "alert words" (e.g., *hijack, emergency, help*) is stored.
- The transcribed text is scanned for the presence of these keywords.
- If a match is found, the system raises an **alert** to indicate a potential threat.

7. Visualization and Debugging

- The system provides clear visualizations for debugging and monitoring:
    - The original signal waveform.
    - The quantized cepstrum.
    - The separated signal and noise components.
    - Cumulative Brownian-like signal representation.

Output and Alerts

The system outputs the transcribed text and alerts if any code words are detected, ensuring real-time surveillance and monitoring of potential threats.

Homomorphic Transformation

Homomorphic transformation is a signal processing technique used to analyze and separate signals into their additive and multiplicative components. This is particularly useful for speech processing, where the goal is to separate the source (excitation) and system (vocal tract filter) components. The transformation leverages logarithmic and Fourier domain operations to achieve this decomposition.

Mathematical Explanation

1. **Input Signal Representation**: Let the input signal be represented as the convolution of two components:

where:

   - Source signal (excitation)
   - Impulse response of the system (vocal tract filter)

2. **Fourier Transform**: The convolution in the time domain translates to multiplication in the frequency domain:

3. **Logarithmic Transformation**: Taking the logarithm of the magnitude spectrum converts multiplication into addition:

Here, represents the cepstral domain, which separates the additive components.

4. **Inverse Fourier Transform (Cepstrum Calculation)**: Applying the Inverse Fourier Transform (IFT) to the logarithmic spectrum retrieves the cepstrum.

The cepstrum contains:

   - Low quefrency components corresponding to the system .
   - High quefrency components corresponding to the source .

Key Steps in Homomorphic Transformation

1. Compute the Fourier Transform of the input signal :
2. Calculate the magnitude spectrum and apply the logarithm:
3. Perform the Inverse Fourier Transform on to obtain the cepstrum:
4. Separate the source and system components using low-pass filtering in the cepstral domain.

Applications

Homomorphic transformation is widely used in:

- Speech recognition and enhancement.
- Audio signal processing.
- Biomedical signal analysis (e.g., ECG decomposition).

- This technique is crucial for preprocessing in the project as it simplifies the analysis and feature extraction of voice signals by isolating relevant components efficiently.

Cepstral Coefficients and Quantization Using Learning Vector Quantization (LVQ)

Cepstral Coefficients

Cepstral coefficients are derived from the cepstrum, which is obtained by applying the homomorphic transformation to a signal. These coefficients capture the spectral properties of a signal and are widely used in speech and audio signal processing for feature extraction.

Mathematical Derivation

1. **Cepstrum Computation**: The cepstrum is calculated as the Inverse Fourier Transform (IFT) of the logarithmic magnitude spectrum of a signal:

where is the Fourier transform of the input signal .

2. **Cepstral Coefficients**: The low-order terms of the cepstrum, typically denoted as cepstral coefficients, are extracted to represent the signal's system (e.g., vocal tract characteristics in speech).

Importance

- The coefficients are compact, representing the essential spectral features.
- Robust to noise and distortions, making them suitable for classification tasks.

Quantization of Cepstral Coefficients Using LVQ

Quantization reduces the dimensionality of the cepstral coefficients for efficient representation while preserving the most relevant features. Learning Vector Quantization (LVQ) is an unsupervised learning technique used for clustering and quantizing these coefficients.

Steps for LVQ Quantization

1. **Input Data Representation**: The cepstral coefficients are reshaped into a one-dimensional vector for processing.
2. **Clustering with MiniBatchKMeans**: LVQ uses clustering to group similar data points:

where is the number of clusters, and each represents a cluster center.

3. **Update Rule**: The algorithm minimizes the quantization error by updating cluster centers iteratively.
    - Learning rate.
    - Cepstral coefficient vector.
4. **Quantized Output**: Each coefficient is assigned to the nearest cluster center.

Advantages of LVQ in Quantization

- **Dimensionality Reduction**: Reduces the data size while preserving important features.

- **Noise Robustness**: Focuses on cluster centers, making the representation less sensitive to outliers.
- **Efficient Computation**: MiniBatchKMeans, used in LVQ, is computationally efficient and scalable.

Application in the Project

In this project:

1. The cepstral coefficients derived from the homomorphic transformation were quantized using LVQ.
2. The quantized values provided a compact and noise-robust representation of the signal.
3. The separation of signal and noise was performed on the quantized data for further analysis.

This process ensures efficient preprocessing and feature extraction for subsequent steps in voice recognition and alert detection.

OUTPUT:

```
Cluster Centers:
[-1.12658144e-04  4.88087721e-04 -5.50136752e-04  7.56442374e-05
  8.39856994e-04 -3.06827331e-04  2.65471277e-04 -1.10504496e-03]
```

Signal and Noise Separation

Separating the signal from the noise is an essential step in the pre-processing pipeline of the project. This ensures that only the meaningful parts of the signal are retained for further analysis while the noisy components are excluded. The separation is performed by analyzing the quantized cepstral coefficients obtained after the homomorphic transformation and quantization. Below is an explanation of the methodology used in this project.

1. Quantized Cepstral Coefficients

The quantized cepstral coefficients represent the transformed signal data, which is compact and efficient for analysis. These coefficients are processed to identify and segregate the signal and noise components.

2. Threshold-Based Separation

The project employs a simple thresholding technique to distinguish between signal and noise. The steps are as follows:

1. **Definition of Threshold**:
    - A noise_threshold is defined, which acts as the boundary value to separate the signal and noise. Any quantized data value above this threshold is considered part of the signal, while values below it are classified as noise.
2. **Signal and Noise Masks**:
    - Two binary masks are created:
        - **Signal Mask**: Identifies all data points greater than the threshold.
        - **Noise Mask**: Identifies all data points less than or equal to the threshold.
3. **Application of Masks**:
    - The signal and noise components are extracted by multiplying the quantized data with their respective masks. This ensures that only relevant components are retained in each category.

Mathematical Representation

Let the quantized data be denoted as , where is the index of the data point.

1. Define the noise threshold:
2. Create masks:
3. Separate signal and noise components:

Here, and represent the signal and noise components, respectively.

Code Implementation

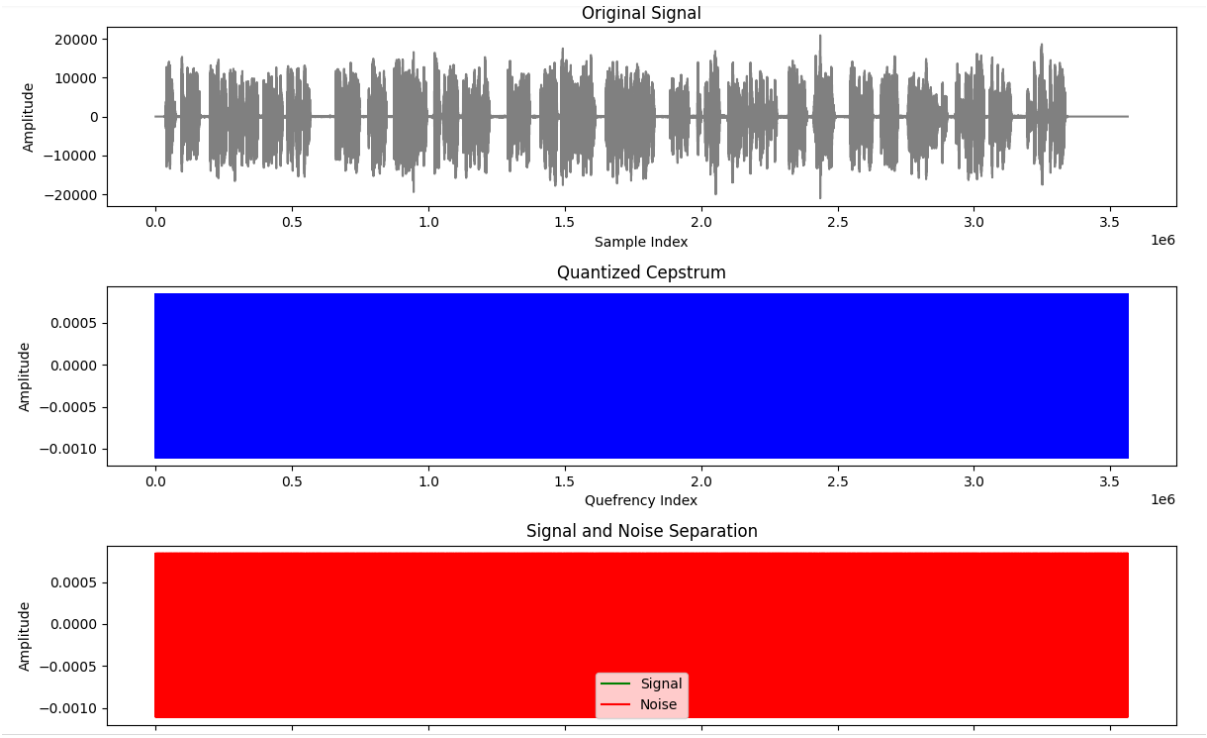
The following Python code implements the above logic:

```
# Define threshold
noise_threshold = 0.05

# Create signal and noise masks
signal_mask = quantized_data > noise_threshold
noise_mask = ~signal_mask

# Separate signal and noise components
signal_part = quantized_data * signal_mask
noise_part = quantized_data * noise_mask
```

OUTPUT:



Visualization

To validate the separation, the project visualizes:

1. **Original Signal**: Displays the input signal.
2. **Quantized Cepstrum**: Shows the transformed and quantized data.
3. **Signal and Noise Separation**: Plots the extracted signal and noise components for comparison.

This separation process is crucial for filtering out irrelevant noise and retaining meaningful signal components for further processing, such as Brownian motion analysis and maximum likelihood estimation.

Brownian Motion Analysis and Maximum Likelihood Function

Brownian Motion Analysis

The project employs Brownian motion analysis as a part of its signal processing workflow to further analyze the extracted signal component. Brownian motion, characterized by its random and continuous nature, serves as a model for understanding the behavior of the processed signal.

Methodology

The key steps involved in Brownian motion analysis are as follows:

1. **Cumulative Sum of the Signal**:
    - The signal data, after being separated from noise, is cumulatively summed to mimic the random walk nature of Brownian motion.
    - Let the signal data be denoted as , where represents the index. The cumulative signal is given by:
2. **Randomness Adjustment**:
    - If the signal appears too flat, randomness is introduced by adding a small amount of Gaussian noise to the data:

where is a normal distribution with zero mean and variance .

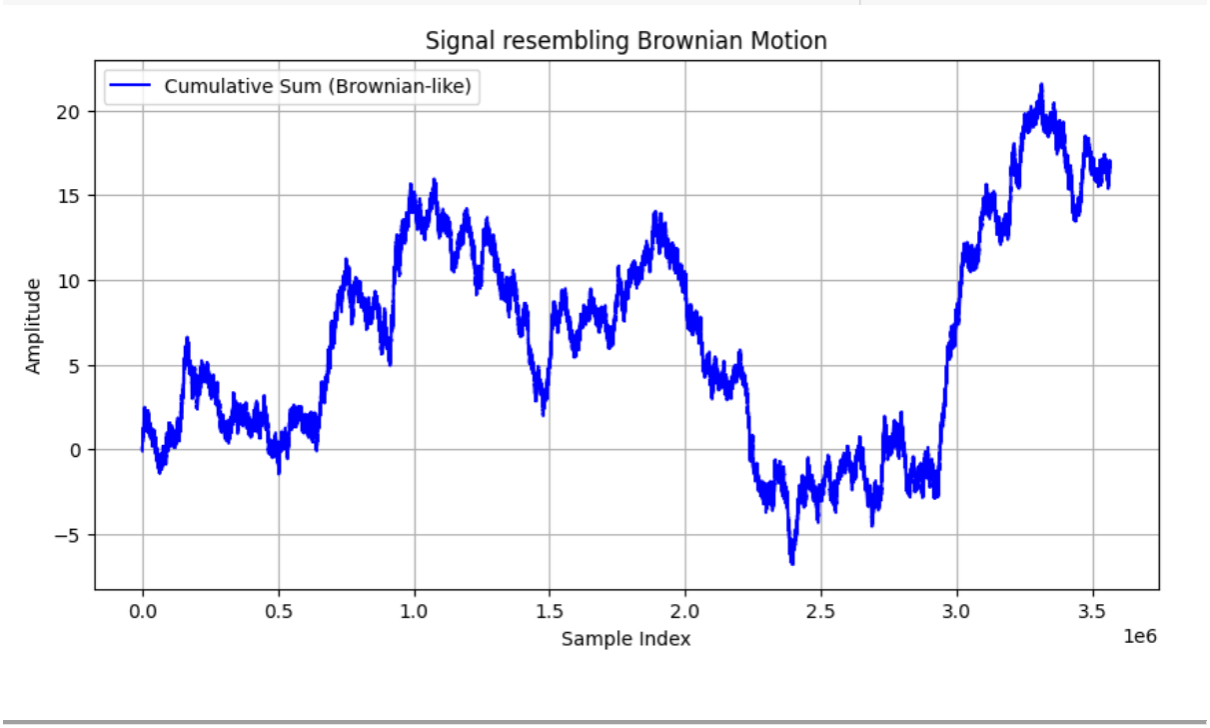
3. **Visualization**:
    - The cumulative signal is plotted to visually verify its Brownian motion characteristics.

Code Implementation

Plot Brownian-like signal

```
# Plot Brownian-like signal
cumulative_signal = np.cumsum(signal_data)
plt.figure(figsize=(10, 5))
plt.plot(cumulative_signal, color='b', label='Cumulative Sum (Br
plt.title("Signal resembling Brownian Motion")
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()
```

OUTPUT:



Maximum Likelihood Analysis

Maximum likelihood estimation (MLE) is used to determine the statistical characteristics of the triangular features formed by sampling three points in the signal. These features include the side lengths and angles of the triangle.

Methodology

The process is as follows:

1. **Triangle Construction**:
    - Three points are selected from the cumulative signal to form a triangle. The points are indexed as .
    - The side lengths of the triangle are calculated using the Euclidean distance:
2. **Angle Calculation**:
    - The angles between the triangle sides are computed using vector dot products:

where and are vectors representing triangle sides.

3. **MLE for Sides and Angles**:
    - The side lengths and angles are modeled using a Gaussian distribution, and their parameters (mean and standard deviation) are estimated using MLE.
    - For side lengths :

The same formula applies to angles .

Code Implementation

```
from scipy.stats import norm

# Fit normal distributions to sides and angles
side_mean, side_std = norm.fit(sides)
angle_mean, angle_std = norm.fit(angles)

# Calculate maximum likelihood values
max_likelihood_sides = norm.pdf(sides, loc=side_mean, scale=side
max_likelihood_angles = norm.pdf(angles, loc=angle_mean, scale=a

# Results
print("Maximum Likelihood Analysis Results:")
print(f"Side Mean: {side_mean:.2f}, Side Std Dev: {side_std:.2f}
print(f"Angle Mean: {angle_mean:.2f} degrees, Angle Std Dev: {an
```

OUTPUT:

```
Maximum Likelihood Analysis Results:
Side Mean: 400.00, Side Std Dev: 163.30
Most Likely Side Length: 400.00
Angle Mean: 120.00°, Angle Std Dev: 84.84°
Most Likely Angle: 179.99°
```

Summary

The combination of Brownian motion analysis and maximum likelihood estimation provides a comprehensive understanding of the signal's statistical properties. The Brownian motion analysis visualizes the signal's behavior, while MLE quantifies the likelihood of specific geometric features, aiding in the identification of signal patterns and anomalies.

Detecting Code Words in Audio Transcriptions

Purpose:

This step aims to analyze audio recordings and identify predefined "code words" (such as *hijack*, *tiger*, *emergency*, and *help*) that could indicate critical or emergency situations. The goal is to automate audio surveillance and provide timely alerts if any of these words are detected.

Process:

1. **Setup Code Words**:
   A list of sensitive words (CODE_WORDS) is predefined. These words represent critical signals to detect in the audio transcription.
2. **Audio Loading and Recognition**:
   - The uploaded audio file is processed using the SpeechRecognition library.
   - The file is read and converted into text using Google's speech-to-text API.

1. **Text Matching**:
   - The transcribed text is checked against the CODE_WORDS list.
   - If any of the code words are found in the text, they are flagged.

2. **Alerts**:
   - If code words are detected, an alert is generated, displaying the detected words.
   - If no code words are present, the script confirms that the audio recording is safe.

Key Functions:

Detect_code_words(text, code_words):

- This function searches the transcribed text for the presence of any code words.
- Returns a list of detected words.

Process_audio_file(file_path):

- Loads and transcribes the audio file.
- Detects and flags any critical words present in the transcription.
- Provides user feedback through alerts or confirmation of no detection.

OUTPUT:

```
Loading audio file...
Processing the audio...
Transcribed Text: tiger hijack
⚠️ Alert: Code words detected!
Detected Words: hijack, tiger
```

Significance:

This step demonstrates the practical use of audio signal processing for real-world scenarios, like security and emergency monitoring. It integrates audio transcription, natural language processing, and alert mechanisms, providing valuable insights into audio surveillance applications.
