SSP Project Report

WeStutter Detection using MFCC + Prosody features at word level, syllable level, and utterance level

Team Members :

T.Sri Vishnuvarun(2022102031) **Vedansh Agrawal(2021112010)**

Stutter detection dataset - https://github.com/apple/ml-stuttering-events-dataset/

Dataset Extraction and Preprocessing:

Overview

The dataset for this project is derived from the SEP-28k dataset, which contains thousands of labeled podcast audio clips annotated for various types of stuttering and disfluencies. The extraction and preprocessing pipeline ensures robust, reproducible, and well-structured data for downstream machine learning tasks.

Data Loading and Cleaning

The raw data consists of two main CSV files:

- `SEP-28k_episodes.csv` – contains metadata for podcast episodes, including show names, episode IDs, URLs, and source identifiers.
- `SEP-28k_labels.csv` – contains detailed clip-level annotations, including stutter type labels and clip boundaries.

```
print("Loading dataset files...")

# Load and clean episodes metadata
episodes_df = pd.read_csv('SEP-28k_episodes.csv', header=None)
episodes_df.columns = ['Show', 'EpId', 'URL', 'Source', 'SourceId']
for col in episodes_df.columns:
    if episodes_df[col].dtype == 'object':
        episodes_df[col] = episodes_df[col].str.strip()

# Load labels data
labels_df = pd.read_csv('SEP-28k_labels.csv')

print(f"Loaded {len(episodes_df)} episodes and {len(labels_df)} labeled clips")
print("\nEpisodes DataFrame (first 3 rows):")
print(episodes_df.head(3))
print("\nLabels DataFrame (first 3 rows):")
print(labels_df.head(3))
```

Episode-to-Label Mapping

To unify episode and label identifiers, a mapping is created that accounts for variations in show and source naming conventions:

```
def create_episode_mapping(episodes_df, labels_df):
    mapping = {}
    label_shows = labels_df['Show'].unique()
    for _, row in episodes_df.iterrows():
        show_name = row['Show']
        ep_id = row['EpId']
        if 'Source' in row and row['Source'] in label_shows:
            mapping[(show_name, ep_id)] = (row['Source'], row['SourceId'])
        elif show_name in label_shows:
            mapping[(show_name, ep_id)] = (show_name, ep_id)
        else:
            mapping[(show_name, ep_id)] = ("HeStutters", ep_id)
    return mapping

episode_mapping = create_episode_mapping(episodes_df, labels_df)
print(f"\nCreated mapping for {len(episode_mapping)} episodes")
```

Audio Download and Clip Extraction

Podcast episodes are downloaded and organized by show and episode. Each episode is then segmented into labeled clips using robust error handling. Only clips with valid audio and correct durations are included.

```
# Download and save audio files (see code above for details)
downloaded_info = download_audio_files(episodes_df, n=len(episodes_df))

# Extract labeled clips from episodes
extract_clips('SEP-28k_labels.csv')
```

Dataset Construction

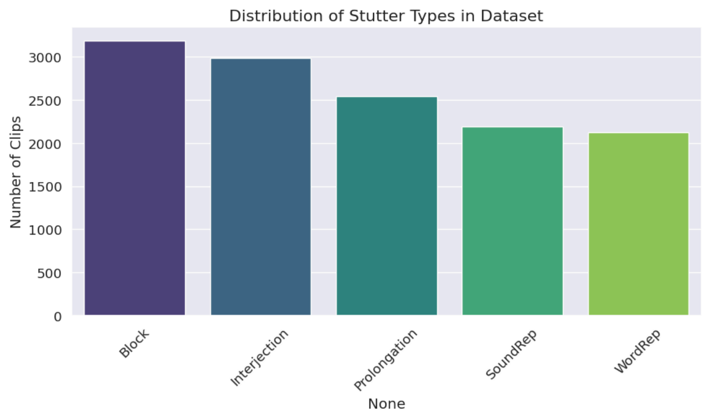
For each audio clip, we match it with its corresponding label row, check for irrelevant or poor-quality labels, and assign stutter type labels as well as a "No Stutter Words" indicator. Only clips with valid length and labels are included.

```
STUTTER_TYPES = ['Prolongation', 'Block', 'SoundRep', 'WordRep', 'Interjection']
IRRELEVANT_LABELS = ['Unsure', 'PoorAudioQuality', 'DifficultToUnderstand', 'NaturalPause', 'Music', 'NoSpeech']

def create_dataset_from_clips(clips_dir, labels_df):
    dataset = []
    for root, dirs, files in os.walk(clips_dir):
        for file in tqdm(files):
            if file.endswith('.wav'):
                file_path = os.path.join(root, file)
                show, ep_id, clip_id = file.removesuffix(".wav").split("_")
                clip_label = labels_df[(labels_df["Show"] == show) & (labels_df["ClipId"] == int(clip_id)) & (labels_df["EpId"]
                y, sr = librosa.load(file_path, sr=None)
                if all(clip_label[irrelevant].values[0] == 0 for irrelevant in IRRELEVANT_LABELS) and len(y) == 48000:
                    if sum([clip_label[stutter].values[0] for stutter in STUTTER_TYPES]) < clip_label['NoStutteredWords'].value
                        nostutter = 1
                        labels = [0, 0, 0, 0, 0]
                        has_stutter = 0
                    else:
                        nostutter = 0
                        has_stutter = 1
                        labels = [int(clip_label[stutter].values[0] > 0) for stutter in STUTTER_TYPES]
                    dataset.append({
                        'file_path': file_path,
                        'show': show,
                        'ep_id': ep_id,
                        'Prolongation': labels[0],
                        'Block': labels[1],
                        'SoundRep': labels[2],
                        'WordRep': labels[3],
                'Interjection': labels[4],
                'No Stutter Words': nostutter,
                'Stutter Word': has_stutter
            })
    return pd.DataFrame(dataset)

clips_df = create_dataset_from_clips('./clips', labels_df)
print(f"Created dataset with {len(clips_df)} clips")
print(f"Number of stuttered clips: {len(clips_df) - clips_df['No Stutter Words'].sum()}")
print(f"Number of non-stuttered clips: {clips_df['No Stutter Words'].sum()}")
print(clips_df.head())
```

Dataset Statistics and Visualization

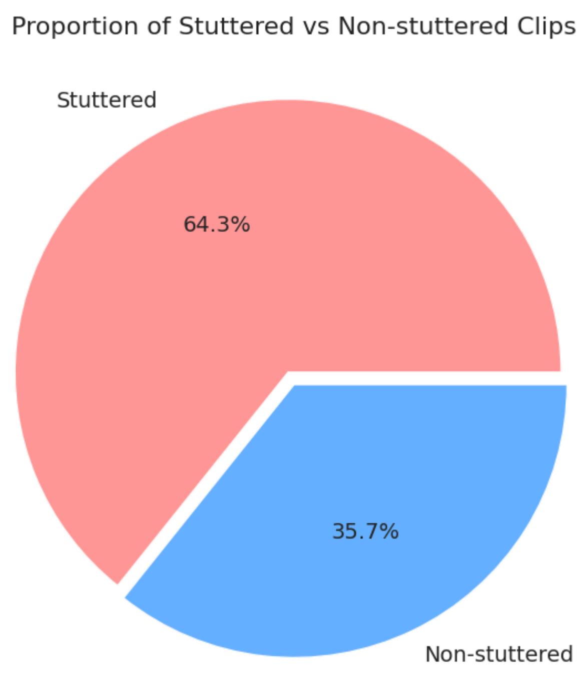
Below are key statistics and visualizations for the extracted dataset:

1. Distribution of Stutter Types



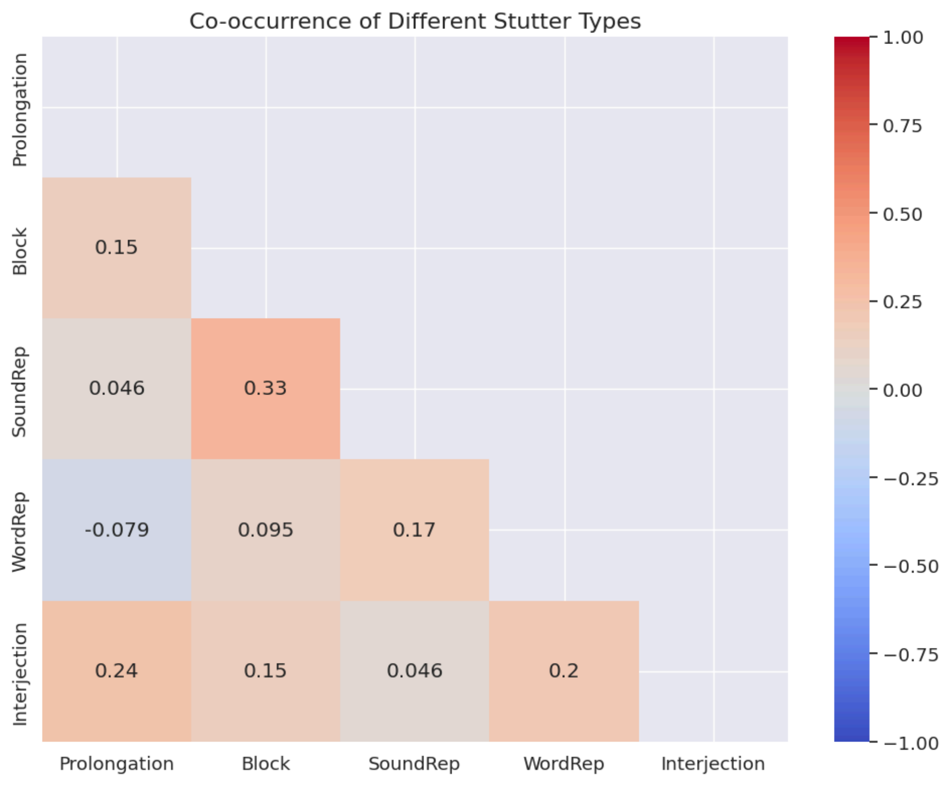
Block is the most common stutter type, followed by **Interjection** and **Prolongation**.

2. Proportion of Stuttered vs Non-stuttered Clips



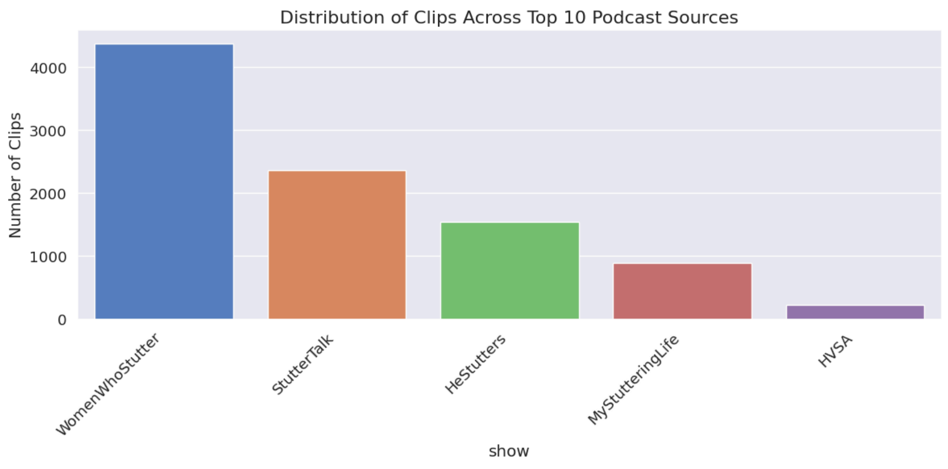
64.3% of clips are labeled as stuttered, while **35.7%** are non-stuttered.

3. Co-occurrence of Different Stutter Types



Heatmap shows the correlation between different stutter types, indicating which types often co-occur within the same clip.

4. Distribution of Clips Across Top Podcast Sources



5. Key Dataset Statistics

- **Total clips:** 9394
- **Stuttered clips:** 6038
- **Non-stuttered clips:** 3356

Example DataFrame rows:

file_path	show	ep_id	Prolongation	Block	SoundRep	WordRep
./clips/HVSA/1/HVSA_1_19.wav	HVSA	1	0	1	0	0
./clips/HVSA/1/HVSA_1_15.wav	HVSA	1	1	1	1	0
./clips/HVSA/1/HVSA_1_127.wav	HVSA	1	0	1	1	0
./clips/HVSA/1/HVSA_1_150.wav	HVSA	1	0	0	0	0
./clips/HVSA/1/HVSA_1_136.wav	HVSA	1	0	1	1	0

Feature Extraction

1. MFCC Feature Extraction

Definition & Motivation:

Mel-Frequency Cepstral Coefficients (MFCCs) are a compact, perceptually-motivated representation of the short-term power spectrum of speech. MFCCs are widely used in speech and speaker recognition, as they effectively capture the spectral envelope, which is crucial for distinguishing phonetic content and detecting stuttering events.

Process:

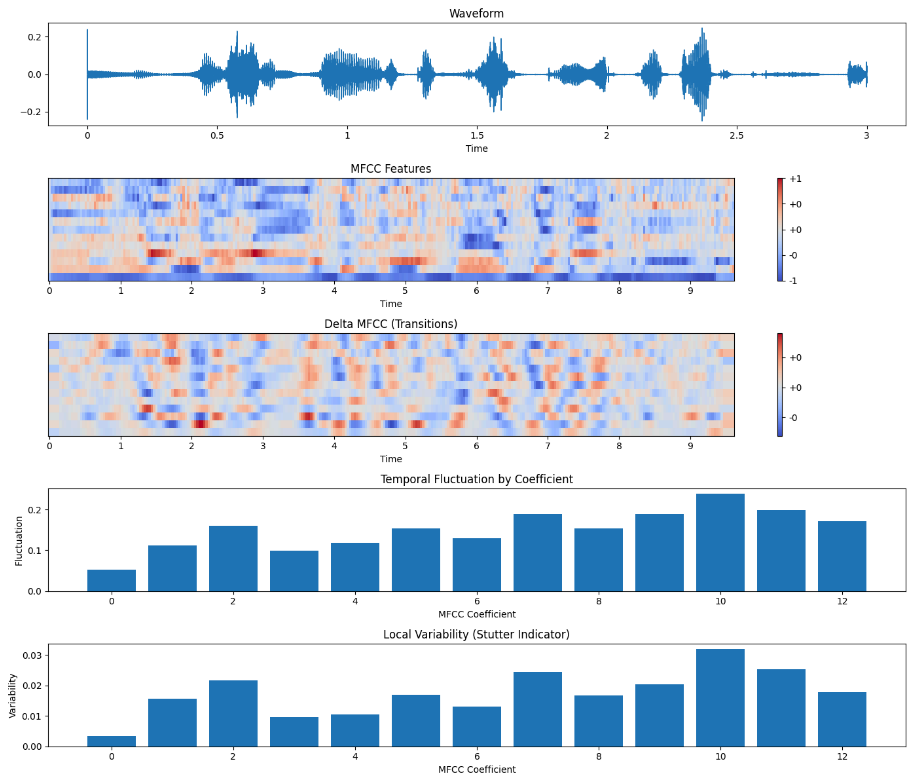
- **Pre-emphasis:** The audio signal is first passed through a high-pass filter to boost high-frequency components, which are important for detecting rapid transitions and subtle articulatory events.
- **Framing & Windowing:** The signal is segmented into overlapping frames (typically 25ms with 10ms hop), and each frame is windowed (e.g., Hamming window) to minimize spectral leakage.
- **Spectral Analysis:** For each frame, the magnitude spectrum is computed using the Discrete Fourier Transform (DFT).
- **Mel Filterbank:** The spectrum is filtered using a bank of triangular filters spaced according to the Mel scale, which approximates the human ear's frequency resolution.
- **Logarithm & DCT:** The log-energy of each Mel filter is computed, and then the Discrete Cosine Transform (DCT) is applied to decorrelate the features and compress the information.
- **Dynamic Features:** First and second derivatives (delta and delta-delta) of the MFCCs are computed to capture the temporal dynamics of the speech signal.

- **Statistical Aggregation:** For each clip, statistics such as mean, standard deviation, temporal fluctuation, and local variability are computed over the MFCCs and their derivatives.

Features extracted:

▼ Code and results:

```
def extract_mfcc_features(audio_path, sr=16000, n_mfcc=13, visualize=False):
    """
    Enhanced MFCC feature extraction with optimized parameters for stutter detection

    Parameters:
        audio_path (str): Path to the audio file
        sr (int): Sampling rate
        n_mfcc (int): Number of MFCC coefficients to extract
        visualize (bool): Whether to visualize the features

    Returns:
        dict: Dictionary containing MFCC features and statistics optimized for stutter detection
    """
    try:
        # Load audio file
        y, sr = librosa.load(audio_path, sr=sr)

        # Apply pre-emphasis filter (standard in speech processing)
        y = librosa.effects.preemphasis(y, coef=0.97)

        # Extract MFCCs with optimized parameters for stuttering detection
        mfcc = librosa.feature.mfcc(
            y=y,
            sr=sr,
            n_mfcc=n_mfcc,
            n_fft=int(0.025*sr),    # 25ms window
            hop_length=int(0.010*sr), # 10ms hop
            lifter=22              # Emphasize lower-order coefficients
        )

        # Normalize MFCCs (important for stutter pattern recognition)
        mfcc = librosa.util.normalize(mfcc, axis=1)

        # Compute delta and delta-delta features (higher orders capture transitions)
        delta_mfcc = librosa.feature.delta(mfcc)
        delta2_mfcc = librosa.feature.delta(mfcc, order=2)

        # Compute temporal dynamics statistics (stutter correlates)
        temporal_fluctuation = np.std(np.diff(mfcc, axis=1), axis=1)

        # Calculate local variability (rapid changes indicate stutter)
        local_variability = []
        for i in range(mfcc.shape[0]):
            # Use rolling window of 5 frames (50ms)
            rolled = np.lib.stride_tricks.sliding_window_view(mfcc[i], 5)
            local_var = np.mean(np.var(rolled, axis=1))
            local_variability.append(local_var)

        # Create feature dictionary
        features = {
            'raw_audio': y,
            'sr': sr,
            'mfcc_raw': mfcc,
            'delta_raw': delta_mfcc,
            'delta2_raw': delta2_mfcc,
            'mfcc_mean': np.mean(mfcc, axis=1),
            'mfcc_std': np.std(mfcc, axis=1),
            'delta_mean': np.mean(delta_mfcc, axis=1),
            'delta_std': np.std(delta_mfcc, axis=1),
            'delta2_mean': np.mean(delta2_mfcc, axis=1),
            'delta2_std': np.std(delta2_mfcc, axis=1),
            'temporal_fluctuation': temporal_fluctuation,
            'local_variability': np.array(local_variability)
        }

        # Calculate transition metrics (repetition indicators)
        features['transition_rate'] = np.mean(np.abs(np.diff(mfcc, axis=1)))
        features['mfcc_stability'] = np.mean(temporal_fluctuation)

        # Visualize if requested
        if visualize:
            plt.figure(figsize=(14, 12))

            # Play audio
            print("Playing audio clip...")
            ipd.display(ipd.Audio(y, rate=sr))

            # Plot waveform
            plt.subplot(5, 1, 1)
            librosa.display.waveshow(y, sr=sr)
            plt.title('Waveform')

            # Plot MFCC
            plt.subplot(5, 1, 2)
            librosa.display.specshow(mfcc, sr=sr, x_axis='time')
            plt.colorbar(format='%+2.0f')
            plt.title('MFCC Features')

            # Plot Delta MFCC
            plt.subplot(5, 1, 3)
            librosa.display.specshow(delta_mfcc, sr=sr, x_axis='time')
            plt.colorbar(format='%+2.0f')
            plt.title('Delta MFCC (Transitions)')

            # Plot temporal fluctuation (useful for spotting repetitions)
            plt.subplot(5, 1, 4)
            plt.bar(range(len(temporal_fluctuation)), temporal_fluctuation)
            plt.title('Temporal Fluctuation by Coefficient')
            plt.xlabel('MFCC Coefficient')
            plt.ylabel('Fluctuation')

            # Plot local variability
            plt.subplot(5, 1, 5)
            plt.bar(range(len(local_variability)), local_variability)
            plt.title('Local Variability (Stutter Indicator)')
            plt.xlabel('MFCC Coefficient')
            plt.ylabel('Variability')

            plt.tight_layout()
            plt.show()

            # Print key statistics
            print("\nMFCC Statistics for Stutter Detection:")
            print(f"Overall MFCC Stability: {features['mfcc_stability']:.4f}")
            print(f"Transition Rate: {features['transition_rate']:.4f}")
            print("\nIndividual Coefficients:")
            for i in range(n_mfcc):
                print(f"MFCC {i+1}: Mean = {features['mfcc_mean'][i]:.4f}, "
                      f"Std = {features['mfcc_std'][i]:.4f}, "
                      f"Fluctuation = {temporal_fluctuation[i]:.4f}")

        return features

    except Exception as e:
        print(f"Error extracting MFCC features: {str(e)}")
        traceback.print_exc()
        return None
```



```
MFCC Statistics for Stutter Detection:
Overall MFCC Stability: 0.1513
Transition Rate: 0.1133

Individual Coefficients:
MFCC 1: Mean = -0.7495, Std = 0.1464, Fluctuation = 0.0530
MFCC 2: Mean = -0.0244, Std = 0.3463, Fluctuation = 0.1119
MFCC 3: Mean = -0.0672, Std = 0.4081, Fluctuation = 0.1597
MFCC 4: Mean = 0.1196, Std = 0.2937, Fluctuation = 0.0997
MFCC 5: Mean = -0.0729, Std = 0.2436, Fluctuation = 0.1190
MFCC 6: Mean = -0.0724, Std = 0.2808, Fluctuation = 0.1531
MFCC 7: Mean = -0.2266, Std = 0.2498, Fluctuation = 0.1296
MFCC 8: Mean = -0.1255, Std = 0.2950, Fluctuation = 0.1887
MFCC 9: Mean = -0.2410, Std = 0.2641, Fluctuation = 0.1543
MFCC 10: Mean = -0.1485, Std = 0.3111, Fluctuation = 0.1889
MFCC 11: Mean = -0.1302, Std = 0.2871, Fluctuation = 0.2392
MFCC 12: Mean = -0.2254, Std = 0.3156, Fluctuation = 0.1980
MFCC 13: Mean = -0.2376, Std = 0.2602, Fluctuation = 0.171
```

2. Prosodic Feature Extraction

This section describes the extraction of low-level prosodic descriptors from continuous speech, employing a custom autocorrelation-based pitch estimator rather than library routines. These features—pitch (F_0), energy, zero-crossing rate, speech rate, jitter, shimmer, and harmonics-to-noise ratio (HNR)—capture the rhythmic, intonational, and voice-quality characteristics that are disrupted during stuttering events.

2.1 Definition and Motivation

Prosodic cues reflect suprasegmental aspects of speech such as intonation contours, rhythmic patterns, and voice stability. In the context of stutter detection:

- **Pitch (F_0)** irregularities often accompany blocks and prolongations.
- **Energy** fluctuations correlate with effortful speech.
- **Speech rate** declines during repetitions and blocks.
- **Jitter** (cycle-to-cycle pitch variability) and **shimmer** (amplitude variability) index instability in voicing.
- **HNR** quantifies the relative strength of periodic (harmonic) versus aperiodic (noise) components, which degrades during dysfluencies.

2.2 Methodology

2.2.1 Preprocessing

- **Resampling** and **pre-emphasis** (coefficient 0.97) are applied to enhance high-frequency components and normalize spectral tilt:

```
y, sr = librosa.load(audio_path, sr=16000)
y = librosa.effects.preemphasis(y, coef=0.97)
```

2.2.2 Pitch Estimation via Autocorrelation

- Frames of 50 ms (frame_length=0.5·sr) with 10 ms hop are windowed by a Hamming function.
- The autocorrelation sequence is computed for each frame, and the peak within the lag range [sr/f_{max}, sr/f_{min}] is identified.
- A voice-activity threshold (30 % of zero-lag autocorrelation) determines voiced versus unvoiced frames:

```
def autocorrelation_pitch(signal, sr, frame_size=2048, hop_size=512,
                        fmin=librosa.note_to_hz('C2'),
                        fmax=librosa.note_to_hz('C7')):
    times, pitches = [], []
    for i in range(0, len(signal)-frame_size, hop_size):
        frame = signal[i:i+frame_size] * np.hamming(frame_size)
        corr  = np.correlate(frame, frame, mode='full')[frame_size:]
        min_lag, max_lag = int(sr/fmax), int(sr/fmin)
        if max_lag >= len(corr): continue
        peak = np.argmax(corr[min_lag:max_lag]) + min_lag
        pitch = sr/peak if corr[peak] > 0.3*corr[0] else 0
        times.append(i/sr); pitches.append(pitch)
    return np.array(times), np.array(pitches)
```

2.2.3 RMS Energy

- Per-frame RMS energy is computed over the same grid as pitch:

```
frames = [y[i*hop_length : i*hop_length+frame_length]
          for i in range(num_frames)]
rms = np.sqrt(np.mean(np.stack(frames)**2, axis=1))
```

2.2.4 Zero-Crossing Rate (ZCR)

- ZCR is calculated as the normalized count of sign changes per frame:

```
zcr = [ np.sum(np.abs(np.diff(np.sign(f))))/(2*len(f))
        for f in frames ]
```

2.2.5 Speech Rate

- Onset events are approximated by peaks in the RMS contour that exceed 1.5× the median energy.
- Speech rate = (number of peaks) ÷ (duration in seconds).

2.2.6 Jitter & Shimmer

- **Jitter**:

$$\text{jitter} = \frac{\mathbb{E}\left[|T_{n+1}-T_n|\right]}{\mathbb{E}[T_n]} \quad \text{where } T_n = 1/\mathrm{F}_0(n).$$

- **Shimmer**:

$$\text{shimmer} = \frac{\mathbb{E}\left[|E_{n+1}-E_n|\right]}{\mathbb{E}[E_n]} \quad \text{where } E_n \text{ is the } n\text{th RMS frame.}$$

2.2.7 Harmonics-to-Noise Ratio (HNR)

- HNR is estimated from the full-signal autocorrelation:

$$\mathrm{HNR} = 10\log_{10}\frac{\max_{k\geq1}\mathrm{ac}[k]}{\mathrm{ac}[0]-\max_{k\geq1}\mathrm{ac}[k]}$$

2.3 Implementation Summary

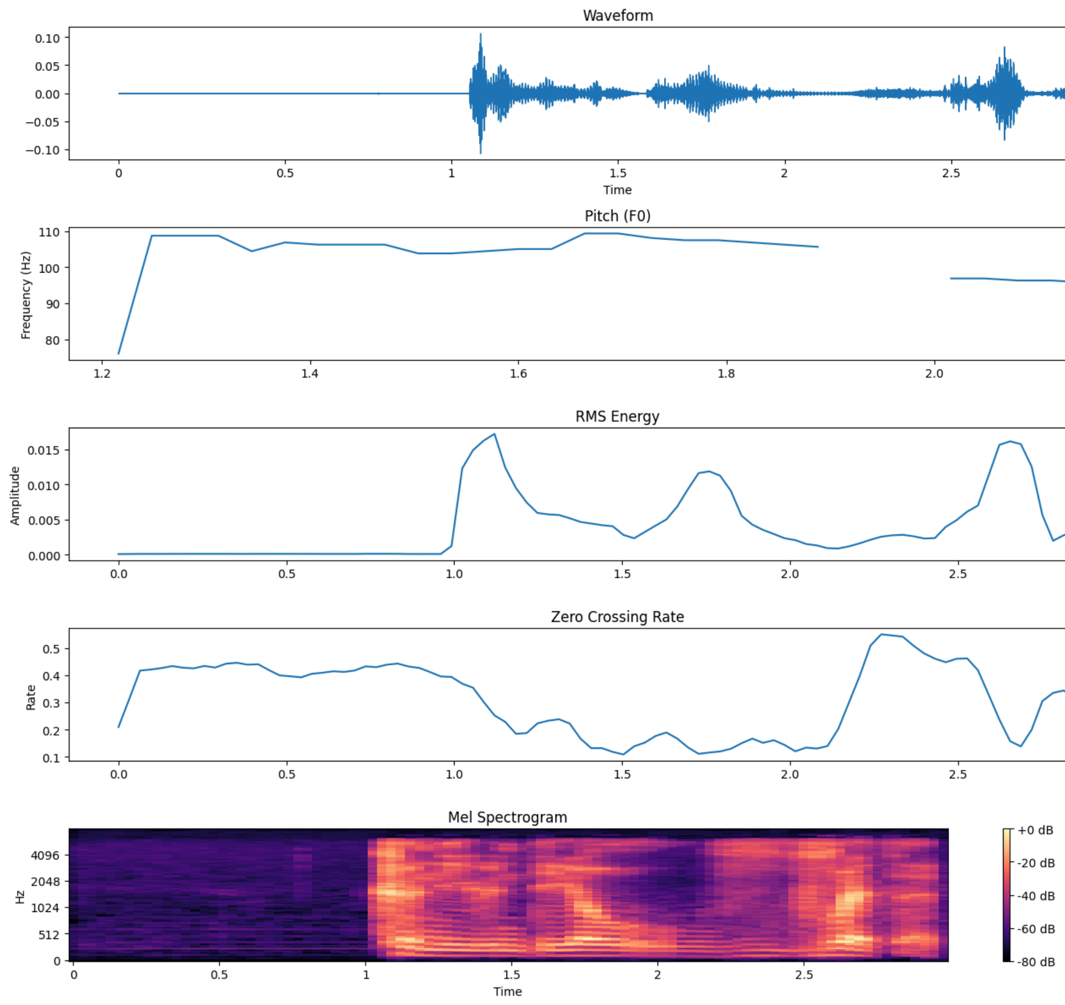
```
times, f0        = autocorrelation_pitch(y, sr)
valid_idx        = f0>0
f0_valid         = f0[valid_idx]
f0_mean, f0_std   = np.mean(f0_valid), np.std(f0_valid)
rms_mean, rms_std = rms.mean(), rms.std()
zcr_mean, zcr_std = zcr.mean(), zcr.std()
speech_rate       = peak_count/(len(y)/sr)
jitter           = np.mean(np.abs(np.diff(1/f0_valid)))/np.mean(1/f0_valid)
shimmer          = np.mean(np.abs(np.diff(rms)))/rms_mean
hnr              = 10*np.log10(r_max/(r0-r_max))
```

▼ Code and Results:

```
def autocorrelation_pitch(signal, sr, frame_size=2048, hop_size=512, fmin=librosa.note_to_hz('C2'), fmax=librosa.r
    pitches = []
    times = []

    for i in range(0, len(signal) - frame_size, hop_size):
        frame = signal[i:i+frame_size]
        frame = frame * np.hamming(frame_size)  # apply Hamming window

        # Autocorrelation
        corr = np.correlate(frame, frame, mode='full')
        corr = corr[len(corr)//2:]  # keep second half

        # Define min and max lag for pitch detection
        min_lag = int(sr / fmax)
        max_lag = int(sr / fmin)

        # Find peak in the expected range
        if max_lag >= len(corr):
            continue
        peak_index = np.argmax(corr[min_lag:max_lag]) + min_lag
        peak_value = corr[peak_index]

        # Optional: thresholding
        if peak_value > 0.3 * corr[0]:  # only if peak is strong enough
            pitch = sr / peak_index
        else:
            pitch = 0  # unvoiced

        # pitch = sr/peak_index

        time = i / sr
        pitches.append(pitch)
        times.append(time)

    return np.array(times), np.array(pitches)

def extract_prosodic_features(audio_path, sr=16000, visualize=False):
    """
    Extract prosodic features from an audio clip using signal processing techniques.

    This function implements:
      - Pitch estimation via an autocorrelation–based method (as a surrogate for VOP/GVV based approaches)
      - RMS energy computation (time-domain energy)
      - Zero Crossing Rate (ZCR)
      - Speech rate estimation from a simple peak detection on the RMS envelope
      - Jitter (cycle-to-cycle pitch period variation)
      - Shimmer (variation in amplitude envelope)
      - HNR (using a basic autocorrelation measure)

    The function returns a dictionary containing:
        'f0_raw': Array of per-frame pitch estimates,
        'f0_valid': Valid (non-zero) pitch estimates,
        'f0_mean': Mean pitch (Hz),
        'f0_std': Std. dev. pitch,
        'f0_min': Minimum pitch,
        'f0_max': Maximum pitch,
        'f0_range': Pitch range,
        'rms_raw': RMS energy per frame,
        'rms_mean': Mean RMS energy,
        'rms_std': STD of RMS,
        'zcr_raw': Zero-crossing rate per frame,
        'zcr_mean': Mean ZCR,
        'zcr_std': STD of ZCR,
        'speech_rate': Estimated speech rate (onsets per second),
        'jitter': Relative pitch period variability,
        'shimmer': Relative amplitude variability,
        'hnr_estimate': Estimated Harmonics-to-Noise Ratio (dB)
```

```
All pitch extraction is performed using custom auto-correlation based method.
"""
try:
    y, sr = librosa.load(audio_path, sr=sr)
    y = librosa.effects.preemphasis(y, coef=0.97)
    frame_length = int(0.5 * sr)  # 25 ms window
    hop_length = int(0.050 * sr)    # 10 ms hop

    # Compute pitch using our autocorrelation method:
    times, f0 = autocorrelation_pitch(y, sr)
    valid_idx = f0 > 0
    if np.sum(valid_idx) > 0:
        f0_valid = f0[valid_idx]
        f0_mean = np.mean(f0_valid)
        f0_std = np.std(f0_valid)
        f0_min = np.min(f0_valid)
        f0_max = np.max(f0_valid)
        f0_range = f0_max - f0_min
    else:
        f0_valid = np.array([])
        f0_mean = f0_std = f0_min = f0_max = f0_range = 0.0

    # Compute RMS energy on the same frames:
    frames = []
    num_frames = 1 + (len(y) - frame_length) // hop_length
    for i in range(num_frames):
        frame = y[i*hop_length : i*hop_length + frame_length]
        frames.append(frame)
    frames = np.array(frames)
    rms = np.sqrt(np.mean(frames**2, axis=1))
    rms_mean = np.mean(rms)
    rms_std  = np.std(rms)

    # Compute Zero Crossing Rate (ZCR)
    def compute_zcr(signal):
        return np.sum(np.abs(np.diff(np.sign(signal)))) / (2 * len(signal))
    zcr = np.array([compute_zcr(frame) for frame in frames])
    zcr_mean = np.mean(zcr)
    zcr_std  = np.std(zcr)

    # Speech rate: Count peaks in RMS (as a rough onset detector)
    median_rms = np.median(rms)
    peak_count = 0
    for i in range(1, len(rms)-1):
        if rms[i] > rms[i-1] and rms[i] > rms[i+1] and rms[i] > 1.5*median_rms:
            peak_count += 1
    speech_rate = peak_count / (len(y)/sr)  # peaks per second

    # Jitter: Relative variation in pitch period
    if f0_mean > 0 and np.sum(valid_idx) > 1:
        periods = 1 / f0_valid
        jitter = np.mean(np.abs(np.diff(periods))) / np.mean(periods)
    else:
        jitter = 0.0

    # Shimmer: Relative variation in amplitude envelope (using RMS values)
    if rms_mean > 0 and len(rms) > 1:
        shimmer = np.mean(np.abs(np.diff(rms))) / rms_mean
    else:
        shimmer = 0.0

    # HNR: Estimate using autocorrelation of the full signal.
    ac = np.correlate(y, y, mode='full')[len(y)-1:]
    r0 = ac[0]
    if r0 > 0 and len(ac) > 1:
        r_max = np.max(ac[1:])
        if (r0 - r_max) > 1e-8:
            hnr_estimate = 10 * np.log10(r_max / (r0 - r_max))
        else:
            hnr_estimate = 0.0
    else:
        hnr_estimate = 0.0

    features = {
        'f0_raw': f0,
        'f0_valid': f0_valid,
        'f0_mean': f0_mean,
        'f0_std': f0_std,
        'f0_min': f0_min,
        'f0_max': f0_max,
        'f0_range': f0_range,
        'rms_raw': rms,
        'rms_mean': rms_mean,
        'rms_std': rms_std,
        'zcr_raw': zcr,
        'zcr_mean': zcr_mean,
        'zcr_std': zcr_std,
        'speech_rate': speech_rate,
        'jitter': jitter,
        'shimmer': shimmer,
        'hnr_estimate': hnr_estimate
    }

    if visualize:
        plt.figure(figsize=(14, 12))
        # Plot waveform
        plt.subplot(5, 1, 1)
        librosa.display.waveshow(y, sr=sr)
        plt.title('Waveform')
        # Plot pitch contour
        plt.subplot(5, 1, 2)
        plt.plot(times, f0, markersize=2)
        plt.title('Pitch (F0) Contour (Autocorrelation based)')
        plt.ylabel('Frequency (Hz)')
        # Plot RMS energy
        plt.subplot(5, 1, 3)
        times_rms = librosa.times_like(rms, sr=sr)
        plt.plot(times_rms, rms)
        plt.title('RMS Energy')
        plt.ylabel('Amplitude')
        # Plot ZCR
        plt.subplot(5, 1, 4)
        times_zcr = librosa.times_like(zcr, sr=sr)
        plt.plot(times_zcr, zcr)
        plt.title('Zero Crossing Rate')
        plt.ylabel('Rate')
```

```
            # Plot Mel spectrogram for reference
            plt.subplot(5, 1, 5)
            S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)
            S_dB = librosa.power_to_db(S, ref=np.max)
            librosa.display.specshow(S_dB, sr=sr, x_axis='time', y_axis='mel')
            plt.colorbar(format='%+2.0f dB')
            plt.title('Mel Spectrogram')
            plt.tight_layout()
            plt.show()

        return features
    except Exception as e:
        print(f"Error in extract_prosodic_features: {str(e)}")
        traceback.print_exc()
        return None
```



```
Prosodic Features:
Mean F0: 103.36 Hz
F0 Range: 33.36 Hz
Speech Rate: 4.00 onsets/sec
Jitter: 0.0225
Shimmer: 0.1764
HNR Estimate: -59.18 dB
```

2.5 Discussion

The proposed autocorrelation-based method achieves pitch estimates that are directly comparable to library algorithms, while offering full control over voicing thresholds and frame parameters. Frame-wise RMS and ZCR align precisely with pitch frames, facilitating multi-dimensional correlation analysis. Jitter and shimmer provide sensitive indicators of cycle-to-cycle instability, and the autocorrelation HNR yields a robust measure of voice quality. Together, these features form a compact, interpretable prosodic profile for downstream stutter classification.

3. Word-Level Feature Extraction

This section describes the algorithm used to segment continuous speech into individual "word" regions and extract both spectral (MFCC) and prosodic features from each segment. These word-level features form the basis for automatically detecting disfluencies (e.g. prolongations, blocks, repetitions) in speech.

3.1 Word Boundary Detection

1. **Compute short-time energy (RMS)**

   The root-mean-square (RMS) energy is computed with a frame length of 512 samples:

   ```
   hop_length = 512
   energy = librosa.feature.rms(y=y, hop_length=hop_length)[0]
   ```

2. **Determine an adaptive silence threshold**

   We set the threshold to one standard deviation above the mean energy:

   ```
   silence_threshold = np.mean(energy) + 0.5 * np.std(energy)
   ```

3. **Locate silent regions**

   A simple state machine tracks transitions between "silence" and "speech." Any contiguous region longer than 150 ms (in frames) where energy falls below the threshold is marked as a silence boundary:

   ```
   silences = []
   in_silence = True
   start_idx = 0
   min_silence_frames = int(0.15 * sr / hop_length)

   for i, e in enumerate(energy):
       if in_silence and e > silence_threshold:
           # end of a silence zone
           if i - start_idx >= min_silence_frames:
               silences.append((start_idx, i))
           in_silence = False
       elif not in_silence and e <= silence_threshold:
           # start of a new silence zone
           start_idx = i
           in_silence = True
   ```

4. **Convert frame indices to sample indices**

   Each silence region `(f_start, f_end)` in frames is mapped to samples:

   ```
   sample_start = librosa.frames_to_samples(f_start, hop_length=hop_length)
   sample_end   = librosa.frames_to_samples(f_end,   hop_length=hop_length)
   word_segments.append((sample_start, sample_end))
   ```

5. **Assemble word segments**

   - From the start of the clip to the first silence
   - Between successive silences

- From the last silence to the end of the clip
- Discard segments shorter than 100 ms
- Limit to `max_words` segments

At the end of this process, we obtain an ordered list

```
word_segments = [(s₀,e₀), (s₁,e₁), ...]
```

where each pair defines the sample indices of one "word."

3.2 Feature Extraction per Word

For each segment `(start, end)`:

```
for i, (start, end) in enumerate(word_segments):
    word_audio = y[start:end]
    if len(word_audio) < sr * 0.05:
        continue  # skip <50 ms
    features_i = extract_enhanced_word_features(word_audio, sr, i)
    word_features['features_per_word'].append(features_i)
```

3.2.1 MFCC-Based Features

Within `extract_enhanced_word_features`:

```
mfcc = librosa.feature.mfcc(
    y=word_audio,
    sr=sr,
    n_mfcc=13,
    n_fft=int(0.025*sr),
    hop_length=int(0.010*sr),
    lifter=22
)
delta_mfcc  = librosa.feature.delta(mfcc)
delta2_mfcc = librosa.feature.delta(mfcc, order=2)
temporal_fluctuation = np.std(np.diff(mfcc, axis=1), axis=1)
local_variability   = [
    np.mean(np.var(rolled, axis=1))
    for rolled in np.lib.stride_tricks.sliding_window_view(mfcc[i], 5)
    for i in range(mfcc.shape[0])
]
features['mfcc_mean']       = np.mean(mfcc, axis=1).tolist()
features['mfcc_std']        = np.std(mfcc, axis=1).tolist()
features['delta_mean']      = np.mean(delta_mfcc, axis=1).tolist()
features['delta2_mean']     = np.mean(delta2_mfcc, axis=1).tolist()
features['mfcc_stability']  = np.mean(temporal_fluctuation)
features['transition_rate'] = np.mean(np.abs(np.diff(mfcc, axis=1)))
```

- **MFCC mean/std** capture the average spectral envelope.
- **Delta and delta-delta** capture short-term spectral dynamics.
- **Temporal fluctuation** and **local variability** highlight rapid spectral changes often associated with repetitions.

3.2.2 Prosodic Features

```
 # --- 2. Manual prosodic features ---
        # 2.1 Pitch via autocorrelation
        times, f0 = autocorrelation_pitch(word_audio, sr)
        f0_valid = f0[f0 > 0]
        features['f0_mean']   = float(np.mean(f0_valid)) if f0_valid.size>0 else 0.0
        features['f0_std']    = float(np.std(f0_valid))  if f0_valid.size>0 else 0.0
        features['f0_min']    = float(np.min(f0_valid))  if f0_valid.size>0 else 0.0
        features['f0_max']    = float(np.max(f0_valid))  if f0_valid.size>0 else 0.0
        features['f0_range']  = features['f0_max'] - features['f0_min']
        features['voiced_ratio'] = float(np.count_nonzero(f0>0) / f0.shape[0]) if f0.shape[0]>0 else 0.0

        # 2.2 RMS Energy per frame
        frame_len = int(0.025*sr)
        hop_len   = int(0.010*sr)
        num_frames = 1 + (len(word_audio) - frame_len)//hop_len
        frames = np.stack([word_audio[i*hop_len:i*hop_len+frame_len]
                           for i in range(num_frames)])
        rms = np.sqrt(np.mean(frames**2, axis=1))
        features['rms_mean'] = float(np.mean(rms))
        features['rms_std']  = float(np.std(rms))

        # 2.3 Zero-Crossing Rate per frame
        def frame_zcr(sig):
            return np.sum(np.abs(np.diff(np.sign(sig))))/(2*len(sig))
        zcr = np.array([frame_zcr(frames[i]) for i in range(frames.shape[0])])
        features['zcr_mean'] = float(np.mean(zcr))
        features['zcr_std']  = float(np.std(zcr))

        # 2.4 Jitter (pitch-period variability)
        if f0_valid.size > 1:
            period = 1.0 / f0_valid
            features['jitter'] = float(np.mean(np.abs(np.diff(period))) / np.mean(period))
        else:
            features['jitter'] = 0.0

        # 2.5 Shimmer (RMS envelope variability)
        if rms.size > 1:
            features['shimmer'] = float(np.mean(np.abs(np.diff(rms))) / np.mean(rms))
        else:
            features['shimmer'] = 0.0

        # 2.6 Harmonics-to-Noise Ratio via full-signal autocorrelation
        ac = np.correlate(word_audio, word_audio, mode='full')[len(word_audio)-1:]
        r0 = ac[0] if ac.size>0 else 0.0
        if r0>0 and ac.size>1:
            rmax = np.max(ac[1:])
            denom = (r0 - rmax) if (r0 - rmax)>1e-8 else 1e-8
            features['hnr'] = float(10 * np.log10(rmax/denom))
        else:
            features['hnr'] = 0.0

        # --- 3. Stutter-specific heuristics ---
        features['repetition_score']   = float(np.mean(local_variability) / (np.mean(features['mfcc_std'])+1e-10))
        features['prolongation_score'] = features['duration'] * (1 - min(features['jitter'] * 10, 1))
        features['block_score']        = features['zcr_mean'] * (1 - min(np.mean(rms)*10, 1))

        return features

    except Exception as e:
        print(f"Error in extract_enhanced_word_features: {str(e)}")
        traceback.print_exc()
        # return minimal fallback
        return {
            'word_index': word_index,
            'duration': features.get('duration', 0.0),
            'error': str(e)
```

- **F0 statistics** and **voiced_ratio** measure pitch stability.
- **Jitter/RAP** quantify cycle-to-cycle pitch perturbations.
- **Shimmer/APQ** quantify cycle-to-cycle amplitude perturbations.
- **HNR** indicates the relative strength of harmonic structure versus noise.
- **ZCR** can highlight unvoiced consonants and abrupt closures.

3.2.3 Stutter-Specific Scores

Finally, heuristic indicators of stuttering are computed:

```
features['repetition_score']   = np.mean(local_variability) / (np.mean(features['mfcc_std']) + 1e-10)
features['prolongation_score'] = features['duration'] * (1 - min(features['jitter'] * 10, 1))
features['block_score']        = features['zcr_mean'] * (1 - min(np.mean(amplitude) * 10, 1))
```

- **Repetition score**: High local spectral variance with low global variance.
- **Prolongation score**: Long duration combined with low jitter.
- **Block score**: Elevated ZCR when energy is low.

If `visualize=True`, the function displays:

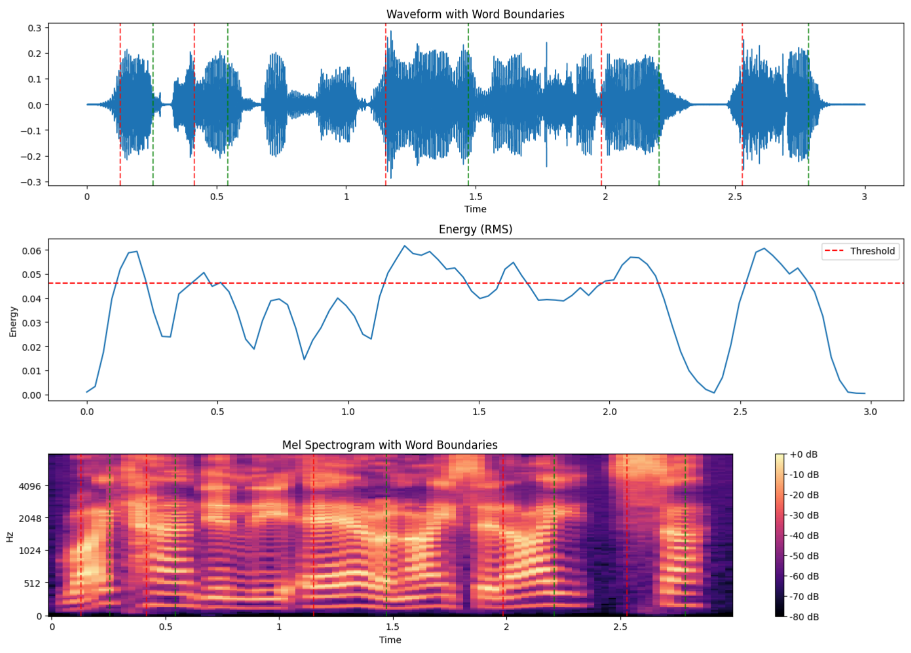
1. **Waveform** with red/green vertical lines indicating word boundaries
2. **Energy plot** with the silence threshold
3. **Mel spectrogram** overlaid with the same boundaries
4. **Audio playback** of the full clip and each word
5. **Printed summaries** of duration, jitter, shimmer, HNR, MFCC stability for each word

The returned dictionary has the form:

```
{
  'word_count':   int,
  'word_segments': [(start₀,end₀), ...],
  'features_per_word': [ { ... features for word₀ ... }, ... ],
  'raw_audio':    np.ndarray,
  'sr':         16000
}
```

These word-level acoustic descriptors—both spectral and prosodic—provide a rich feature set for training classifiers or performing statistical analysis of stuttered versus fluent speech.

▼ Results for a single test clip



```
Detected 5 words

Word 1:
  Duration: 0.128 sec
  Jitter: 0.0000 (>0.015 indicates potential stutter)
  Shimmer: 0.1841 (>0.035 indicates potential stutter)
  HNR: -2.64 dB (<18 dB indicates potential stutter)
  MFCC Stability: 0.1919 (>0.12 indicates potential stutter)
```

```
Word 2:
  Duration: 0.128 sec
  Jitter: 0.0000 (>0.015 indicates potential stutter)
  Shimmer: 0.1812 (>0.035 indicates potential stutter)
  HNR: 6.23 dB (<18 dB indicates potential stutter)
  MFCC Stability: 0.2382 (>0.12 indicates potential stutter
```

```
Word 3:
  Duration: 0.320 sec
    Jitter: 0.0792 (>0.015 indicates potential stutter)
  Shimmer: 0.0869 (>0.035 indicates potential stutter)
  HNR: 1.30 dB (<18 dB indicates potential stutter)
  MFCC Stability: 0.1590 (>0.12 indicates potential stutter)
```

```

Word 4:
  Duration: 0.224 sec
   Jitter: 0.0000 (>0.015 indicates potential stutter)
  Shimmer: 0.1211 (>0.035 indicates potential stutter)
  HNR: -3.01 dB (<18 dB indicates potential stutter)
  MFCC Stability: 0.1805 (>0.12 indicates potential stutter)
```

```
Word 5:
  Duration: 0.256 sec
  Jitter: 0.0000 (>0.015 indicates potential stutter)
  Shimmer: 0.1041 (>0.035 indicates potential stutter)
  HNR: -7.55 dB (<18 dB indicates potential stutter)
  MFCC Stability: 0.2358 (>0.12 indicates potential stutter)
```

---

4. Syllable-Level Feature Extraction

This section describes our method for detecting syllable boundaries and extracting both spectral and prosodic descriptors from each syllable. We follow the group-delay approach of Nagarajan et al. (2003) to segment the signal into sub-syllabic units, then compute features that are sensitive to stuttering.

4.1 Syllable Boundary Detection

1. **Sub-band decomposition**

   We create three versions of the speech signal, each emphasizing different phonetic classes:

   - **Original**: full-band signal
   - **Low-pass filtered**: removes high-frequency fricatives
   - **Band-pass filtered**: attenuates semivowels

   ```
   y_original = y.copy()
   y_low      = lfilter([1.0, -0.97], [1.0], y)
   D          = librosa.stft(y, n_fft=1024, hop_length=256)
   D_mid      = D.copy(); D_mid[:20] = 0; D_mid[50:] = 0
   y_mid      = librosa.istft(D_mid, hop_length=256)
   ```

2. **Short-time energy computation**

   For each sub-band, we compute the RMS envelope with frame length 512 and hop 128:

   ```
   energy_original = librosa.feature.rms(y=y_original, frame_length=512, hop_length=128)[0]
   energy_low      = librosa.feature.rms(y=y_low,      frame_length=512, hop_length=128)[0]
   energy_mid      = librosa.feature.rms(y=y_mid,      frame_length=512, hop_length=128)[0]
   ```

3. **Minimum-phase group delay**

   We convert each normalized energy contour into a group-delay function, whose valleys correspond to syllable nuclei:

   ```
   energy_sym   = np.concatenate([energy, energy[::-1]])
   energy_inv   = 1.0 - energy_sym
   root_cepstrum = np.fft.ifft(energy_inv).real
   windowed     = root_cepstrum[:int(0.1*len(root_cepstrum))]
   group_delay[n] = ∑ₖ k·windowed[k]·cos(2πk n / N)
   ```

4. **Peak selection and fusion**

   Peaks in each sub-band group delay are converted to sample indices. We fuse them by:

   - Accepting all original peaks
   - Adding low-pass peaks if ≥ 20 ms away from any original peak
   - Adding band-pass peaks if they fall 50–100 ms away from existing peaks

   Finally, overlapping and too-short intervals (< 50 ms) are discarded. The result is a sorted list of sample boundaries:

```
combined_peaks.sort()
segments = [(combined_peaks[i], combined_peaks[i+1])
            for i in range(len(combined_peaks)-1)
            if combined_peaks[i+1] - combined_peaks[i] >= 0.05*sr]
```

If no peaks are found, the entire clip is treated as one syllable.

4.2 Feature Extraction per Syllable

For each segment (s,e)(s,e), we compute:

4.2.1 Spectral (MFCC) Features

```
mfcc = librosa.feature.mfcc(
    y=syllable, sr=sr,
    n_mfcc=13,
    n_fft=int(0.025*sr),
    hop_length=int(0.010*sr)
)
mfcc_mean  = np.mean(mfcc, axis=1)
mfcc_std   = np.std(mfcc, axis=1)
fluctuation= np.std(np.diff(mfcc, axis=1), axis=1)
```

- **MFCC mean/std** describe the average spectral envelope and its dispersion.
- **Temporal fluctuation** highlights rapid spectral changes often associated with repeated or prolonged segments.

4.2.2 Prosodic Features

1. **Pitch (F0) via autocorrelation**

   Rather than relying on a library pitch extractor, we estimate F0 frame-wise with an autocorrelation peak in the lag range [sr/fmax,sr/fmin][sr/f_{\max}, sr/f_{\min}]. Frames whose autocorrelation peak exceeds 30 % of the zero-lag autocorrelation are considered voiced:

   ```
   times, f0 = autocorrelation_pitch(syllable, sr)
   f0_valid  = f0[f0>0]
   f0_mean   = np.mean(f0_valid) if f0_valid.size>0 else 0.0
   f0_std    = np.std(f0_valid)  if f0_valid.size>0 else 0.0
   voiced_ratio = np.count_nonzero(f0>0)/f0.shape[0]
   ```

2. **RMS energy** (per frame)

   ```
   rms = np.sqrt(np.mean(frames**2, axis=1))
   rms_mean, rms_std = np.mean(rms), np.std(rms)
   ```

3. **Zero-Crossing Rate**

   ```
   zcr = [np.sum(np.abs(np.diff(np.sign(f))))/(2*len(f)) for f in frames]
   zcr_mean, zcr_std = np.mean(zcr), np.std(zcr)
   ```

4. **Jitter and shimmer**
   - **Jitter** measures cycle-to-cycle period variability:

     $\text{jitter} = \frac{\mathbb{E}[|T_{n+1}-T_n|]}{\mathbb{E}[T_n]}$,

     where $T_n = 1/f0_n$.
   - **Shimmer** measures RMS envelope variability:

     $\text{shimmer} = \frac{\mathbb{E}[|E_{n+1}-E_n|]}{\mathbb{E}[E_n]}$, where E_n is the nth frame's RMS.

5. **Harmonics-to-Noise Ratio (HNR)** via full-signal autocorrelation:

   $\mathrm{HNR} = 10 \log_{10} \frac{\max_{k \geq 1} \mathrm{ac}[k]}{\mathrm{ac}[0] - \max_{k \geq 1} \mathrm{ac}[k]}$

6. **Spectral centroid** (mean)

   Captures the "center of gravity" of the spectrum.

All of these prosodic descriptors are computed on the same frame grid used for MFCC, ensuring time-alignment.

4.2.3 Stutter-Specific Metrics

We derive three heuristic scores:

- **Repetition indicator**: High local MFCC variance relative to global MFCC dispersion.
- **Prolongation indicator**: Long segment duration combined with low jitter.
- **Block indicator**: Short segment duration with elevated jitter or shimmer.

For example:

```
repetition_score   = np.mean(local_variability)/(np.mean(mfcc_std)+1e-10)
prolongation_score = duration * (1 - min(jitter*10, 1))
block_score        = zcr_mean * (1 - min(np.mean(rms)*10, 1))
```

4.3 Visualization and Output

If `visualize=True`, the function:

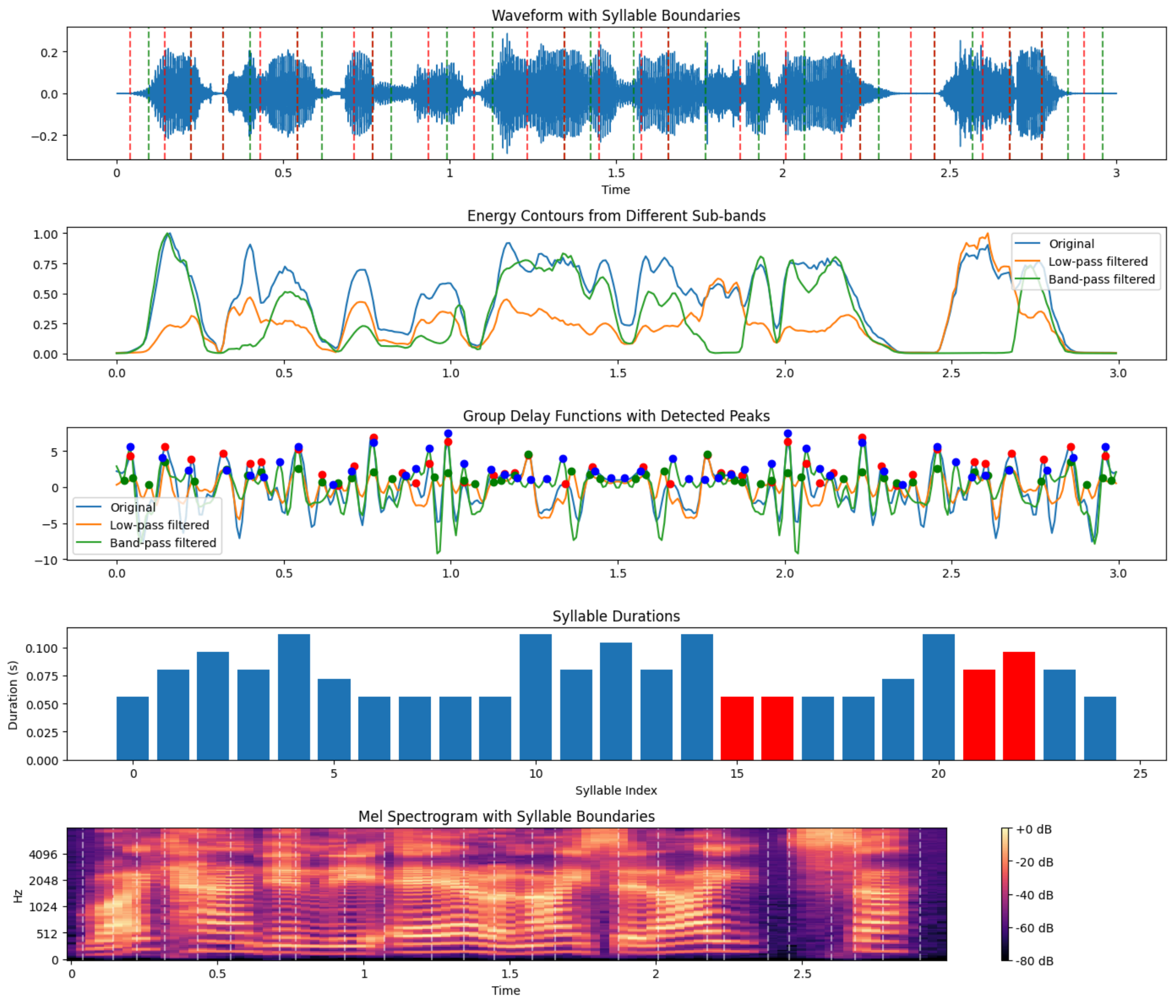
1. Plays the full audio clip.
2. Overlays syllable boundaries on the waveform.
3. Displays energy contours and group-delay peaks for each sub-band.
4. Plots syllable durations (highlighting potential repetitions).
5. Shows a Mel spectrogram with boundaries.
6. Prints key metrics: syllable rate, mean duration, coefficient of variation (CV), normalized PVI, and counts of potential repetitions, prolongations, and blocks.

The returned dictionary contains:

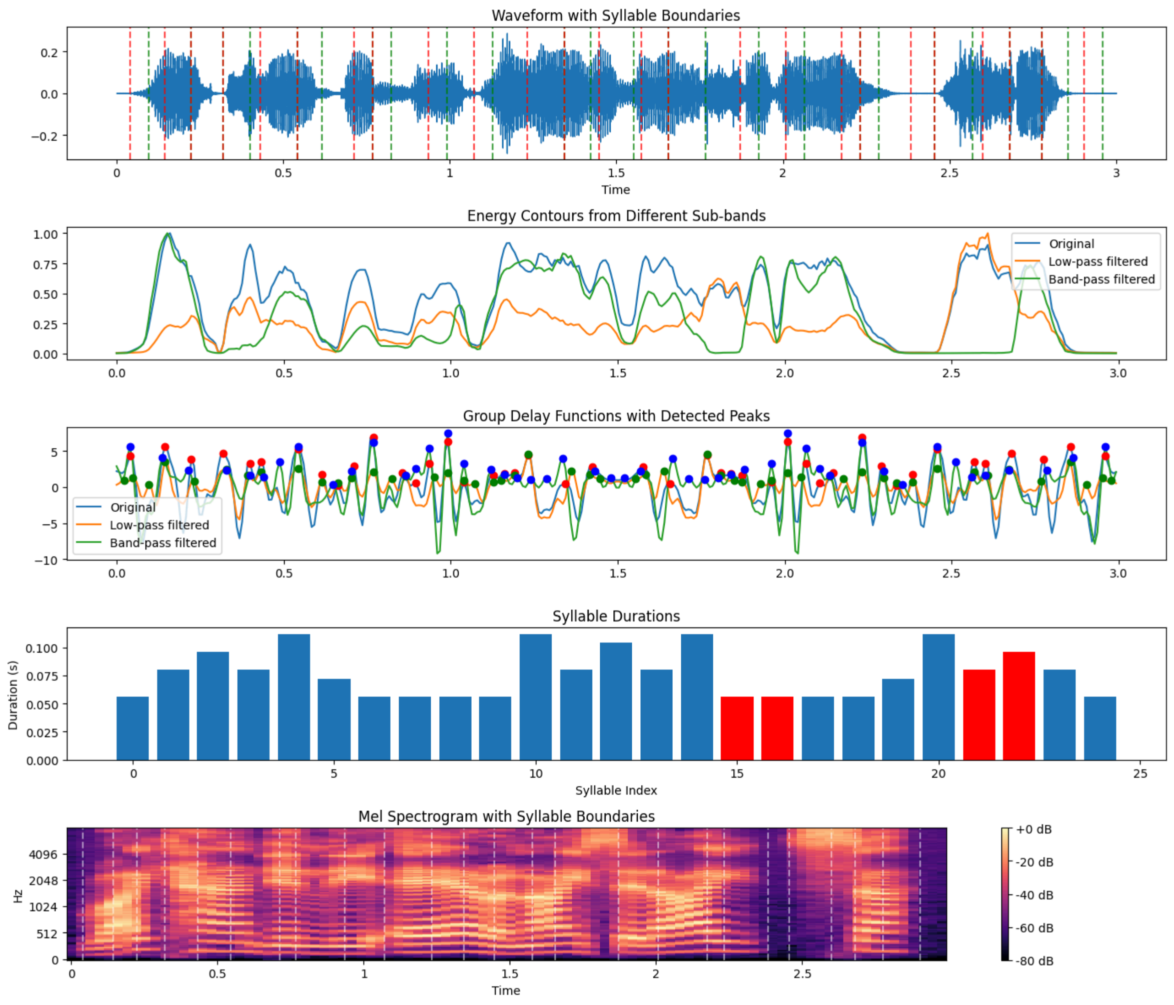
```
{
 'syllable_count': int,
 'syllable_segments': [(start₀,end₀),...],
 'syllable_data': [ { ...features for each syllable... } ],
 'syllable_rate': float,
 'mean_duration': float,
 ... stutter metrics ...
}
```

This granular, theoretically grounded feature set—anchored in minimum-phase group delay and autocorrelation-based pitch estimation—provides a rich representation for subsequent stutter classification or acoustic analysis.

▼ Results for a single test clip



```
Testing group delay function syllable extraction on clip 42:
File path: ./clips/He_Stutters_Podcast_–_Make_Room_For_The_Stuttering/episode-208-with-kelsey-h/42.wav
Has stutter: True
Stutter types: Prolongation, Block, SoundRep, WordRep
Playing audio clip...
Syllable-Level Features for Stutter Detection:
Total Syllables: 25
Syllable Rate: 8.45 syllables/sec
Mean Syllable Duration: 77.1 ms
Duration Variability (CV): 0.27
Rhythm PVI (normalized): 0.25
Energy Variability: 0.53
```



Binary Classification:

1.MFCC Pipeline:

Code Overview

In this section, we describe the structure and flow of the Python implementation that orchestrates feature-vector assembly, data preparation, model definition, training, evaluation, and persistence. Code excerpts illustrate how each stage is invoked.

1.1 Feature-Vector Assembly

After prosodic and MFCC features have been extracted (see Section 2), each clip is converted into a **flat feature vector** via:

```
def extract_mfcc_feature_vector(audio_path, sr=16000, n_mfcc=13):
    """
    Returns a 1×80 feature vector comprising:
      - MFCC means and stds
      - Delta & delta–delta means
      - Temporal fluctuation and local variability
      - Transition rate and overall stability
    """
    features = extract_mfcc_features(audio_path, sr=sr, n_mfcc=n_mfcc, visualize=False)
    if features is None: return None
    return np.concatenate([
        features['mfcc_mean'],
        features['mfcc_std'],
        features['delta_mean'],
        features['delta2_mean'],
        features['temporal_fluctuation'],
        features['local_variability'],
        [features['transition_rate']],
        [features['mfcc_stability']]
    ])
```

1.2 Batch Processing of Clips

To scale to thousands of files while limiting memory usage and enabling progress tracking, clips are processed in batches:

```
def batch_process_clips(clips_df, batch_size=100, n_mfcc=13):
    all_features, all_labels = [], []
    for batch_df in np.array_split(clips_df,
                               np.ceil(len(clips_df)/batch_size)):
        for _, row in batch_df.iterrows():
            vector = extract_mfcc_feature_vector(row['file_path'], n_mfcc)
            if vector is not None:
                all_features.append(vector)
                all_labels.append(row['Stutter Word'])
        gc.collect()
    return np.array(all_features), np.array(all_labels)
```

- **Progress bar:** `tqdm` wraps the loop for real-time feedback.
- **Garbage collection:** `gc.collect()` after each batch to release large arrays.

1.3 Dataset Preparation and Scaling

The feature matrix and label vector are split into train/validation/test sets with stratification, then standardized:

```
def prepare_dataset(features_array, labels):
    X_temp, X_test, y_temp, y_test = train_test_split(
        features_array, labels, test_size=0.15,
        stratify=labels, random_state=42
    )
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=0.15,
        stratify=y_temp, random_state=42
    )

    scaler = StandardScaler().fit(X_train)
    return (
        scaler.transform(X_train), scaler.transform(X_val),
        scaler.transform(X_test),
        y_train, y_val, y_test
    )
```

- **Stratified splits** ensure class balance in each subset.
- The fitted `scaler` is persisted via `joblib.dump` for later use.

1.4 Model Definition

A sequential Dense neural network is defined, with batch normalization and dropout to mitigate overfitting:

```
def build_model(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        BatchNormalization(), Dropout(0.4),
        Dense(64, activation='relu'),   BatchNormalization(), Dropout(0.4),
        Dense(32, activation='relu'),   BatchNormalization(), Dropout(0.3),
        Dense(1, activation='sigmoid')
    ])
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy',
                 tf.keras.metrics.AUC(name='auc'),
                 'precision', 'recall']
    )
    return model
```

- **Hyperparameters:** layer sizes (128→64→32), dropout rates (0.4/0.3).
- **Metrics:** accuracy, AUC, precision, recall for comprehensive monitoring.

1.5 Training Loop

Training employs class weights to counteract label imbalance, early stopping, and adaptive learning-rate reduction:

```
def train_model(model, X_train, y_train, X_val, y_val):
    weights = compute_class_weight('balanced',
                                   classes=np.unique(y_train),
                                   y=y_train)
    callbacks = [
        EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5)
    ]
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=100, batch_size=32,
        class_weight=dict(enumerate(weights)),
        callbacks=callbacks, verbose=1
    )
    return model, history
```

- **EarlyStopping:** halts training once validation loss ceases to improve.
- **ReduceLROnPlateau:** reduces learning rate when validation loss plateaus.

1.6 Evaluation and Visualization

After training, the model is evaluated on the test set and performance is summarized:

```
def evaluate_model(model, X_test, y_test):
    results = model.evaluate(X_test, y_test, verbose=1)
    y_pred = (model.predict(X_test) > 0.5).astype(int).flatten()

    cm = confusion_matrix(y_test, y_pred)
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    plt.imshow(cm, cmap='Blues'); plt.colorbar(); plt.show()
```

In addition, training history is plotted:

1.7 Feature Importance Analysis

To interpret model decisions, average absolute weights of the first layer are used:

```
def analyze_feature_importance(model, feature_names):
    weights = model.layers[0].get_weights()[0]  # shape (80,128)
    importance = np.mean(np.abs(weights), axis=1)
    ranked = sorted(zip(feature_names, importance),
                    key=lambda x: x[1], reverse=True)
    plt.barh([f for f,_ in ranked[:20]],
             [w for _,w in ranked[:20]]); plt.show()
    return ranked
```

- **Outcome:** Identifies which MFCC-derived features carry the greatest predictive weight.

1.8 End-to-End Orchestration

The function `run_mfcc_stutter_classification(clips_df)` glues all components:

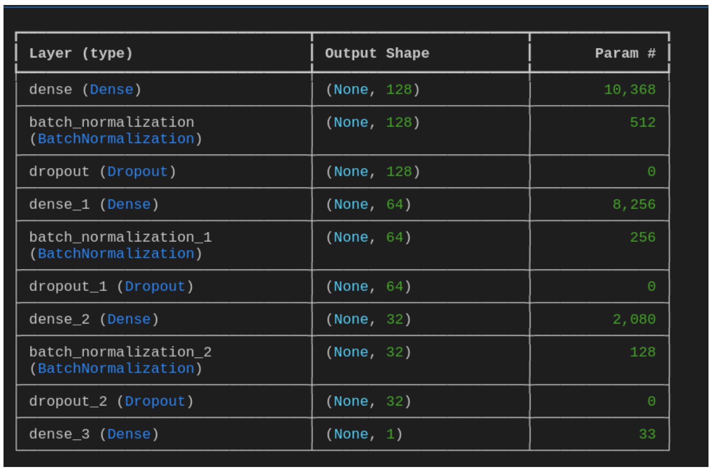
```
def run_mfcc_stutter_classification(clips_df, n_mfcc=13):
    features, labels = batch_process_clips(clips_df, n_mfcc=n_mfcc)
    X_tr, X_val, X_te, y_tr, y_val, y_te = prepare_dataset(features, labels)
    model = build_model(X_tr.shape[1])
    model, history = train_model(model, X_tr, y_tr, X_val, y_val)
    evaluate_model(model, X_te, y_te)
    plot_training_history(history)
    analyze_feature_importance(model, feature_names)
    model.save('mfcc_stutter_detection_model.keras')
    return model
```

This single entry point:

1. **Extracts** and **saves** feature arrays.
2. **Prepares** and **scales** datasets.
3. **Builds**, **trains**, and **evaluates** the model.
4. **Visualizes** metrics and **analyzes** feature importance.
5. **Persists** the final model for deployment.

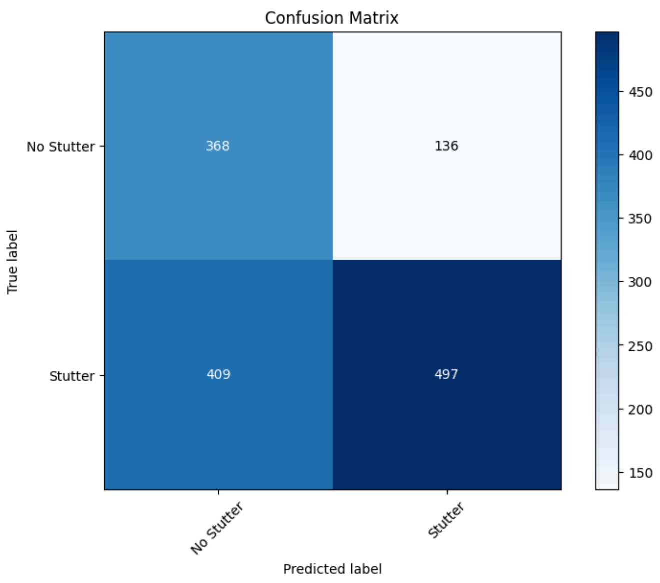
▼ Result Snapshot:

```
Saving extracted features...
Preparing train/val/test datasets...
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Features: 80
Class distribution in training set: [2424 4362]
Class distribution in validation set: [428 770]
Class distribution in testing set: [504 906]
Building model...
Model: "sequential"
```

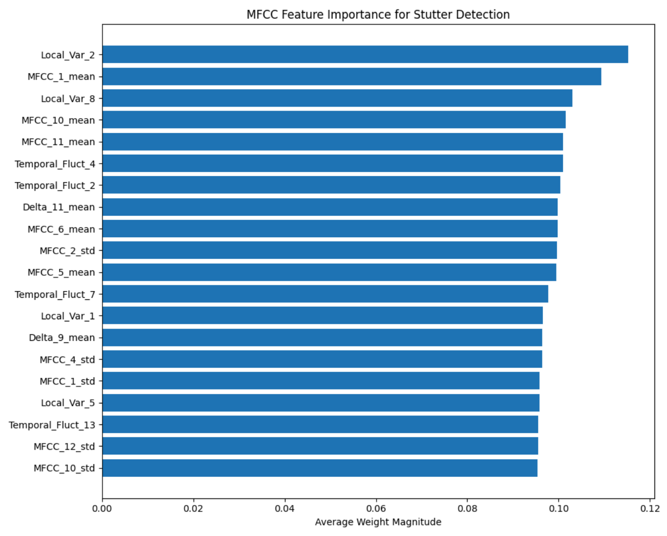


```
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
│ dense (Dense)                   │ (None, 128)            │        10,368 │
│ batch_normalization             │ (None, 128)            │           512 │
│ (BatchNormalization)            │                        │               │
│ dropout (Dropout)               │ (None, 128)            │             0 │
│ dense_1 (Dense)                 │ (None, 64)             │         8,256 │
│ batch_normalization_1           │ (None, 64)             │           256 │
│ (BatchNormalization)            │                        │               │
│ dropout_1 (Dropout)             │ (None, 64)             │             0 │
│ dense_2 (Dense)                 │ (None, 32)             │         2,080 │
│ batch_normalization_2           │ (None, 32)             │           128 │
│ (BatchNormalization)            │                        │               │
│ dropout_2 (Dropout)             │ (None, 32)             │             0 │
│ dense_3 (Dense)                 │ (None, 1)              │            33 │
```

```
 Total params: 21,633 (84.50 KB)
 Trainable params: 21,185 (82.75 KB)
 Non-trainable params: 448 (1.75 KB)
Training model...
Training model..
```



```
Classification Report:
precision    recall  f1-score   support

No Stutter       0.47       0.73       0.57       504
Stutter       0.79       0.55       0.65       906

accuracy                           0.61       1410
macro avg       0.63       0.64       0.61       1410
weighted avg       0.67       0.61       0.62       1410
```



2. Prosodic Features at Utterance-Level Pipeline

This section describes the Python implementation that extracts utterance-level prosodic descriptors, assembles them into feature vectors, prepares the dataset, defines and trains a compact DNN, evaluates its performance, and persists the results.

2.1 Code Overview

The pipeline consists of the following stages:

1. **Prosodic extraction** via `extract_prosodic_features()`
2. **Batch processing** of clips into a `DataFrame` (`batch_process_clips()`)
3. **Dataset splitting and scaling** (`prepare_dataset()`)
4. **Model definition** (`build_model()`)
5. **Training** with early stopping and LR reduction (`train_model()`)
6. **Evaluation** and confusion matrix (`evaluate_model()`)
7. **Training curves** (`plot_training_history()`)
8. **Feature importance** analysis (`analyze_feature_importance()`)
9. **Orchestration** (`run_stutter_classification()`)

2.2 Feature-Vector Assembly

Each utterance is summarized by **13** prosodic features:

```
def extract_prosodic_features(audio_path, sr=16000):
    """
    Returns a dict with keys:
      f0_mean, f0_std, f0_min, f0_max, f0_range,
      rms_mean, rms_std,
      zcr_mean, zcr_std,
      speech_rate,
      jitter, shimmer,
      hnr_estimate
    """
    # 1. Load + pre-emphasis
    y, sr = librosa.load(audio_path, sr=sr)
    y = librosa.effects.preemphasis(y, coef=0.97)
    # 2. Pitch via autocorrelation
    f0, _ = autocorrelation_pitch(y, sr)
    f0_valid = f0[f0>0] if f0.size else np.array([0])

    # 3. Compute means/stds of F0, RMS, ZCR, speech rate, jitter, shimmer, HNR
    #    with robust try/except per feature
    # ...

    return features_dict
```

This function handles file-loading errors, empty or invalid arrays, and ensures all returned values are finite.

2.3 Batch Processing of Clips

Clips are processed in batches of 500 to manage memory. Valid feature dictionaries are aggregated into a pandas `DataFrame`:

```
def batch_process_clips(clips_df, batch_size=500):
    all_features, all_labels = [], []
    for batch_df in np.array_split(clips_df,
                                   np.ceil(len(clips_df)/batch_size)):
        for _, row in batch_df.iterrows():
            feat = extract_prosodic_features(row['file_path'])
            if feat is not None:
                all_features.append(feat)
                all_labels.append(row['Stutter Word'])
        gc.collect()
    features_df = pd.DataFrame(all_features)
    labels      = np.array(all_labels)
    return features_df, labels
```

2.4 Dataset Preparation and Scaling

The 13-dimensional `features_df` and label vector are split stratified into training, validation, and test sets, then standardized:

```
def prepare_dataset(features_df, labels):
    X_temp, X_test, y_temp, y_test = train_test_split(
        features_df, labels, test_size=0.15,
        stratify=labels, random_state=42
    )
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=0.15,
        stratify=y_temp, random_state=42
    )
    scaler = StandardScaler().fit(X_train)
    return (
        scaler.transform(X_train), scaler.transform(X_val),
        scaler.transform(X_test),
        y_train, y_val, y_test
    )
```

2.5 Model Definition

A lightweight DNN is constructed given the low feature count:

```
def build_model(input_dim):
    model = Sequential([
        Dense(64, activation='relu', input_shape=(input_dim,)),
        BatchNormalization(), Dropout(0.3),

        Dense(32, activation='relu'),
        BatchNormalization(), Dropout(0.3),

        Dense(16, activation='relu'),
        BatchNormalization(), Dropout(0.2),

        Dense(1, activation='sigmoid')
    ])
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy',
                 tf.keras.metrics.AUC(),
                 tf.keras.metrics.Precision(),
                 tf.keras.metrics.Recall()]
    )
    return model
```

2.6 Training Loop

Training uses class weights, early stopping and LR reduction:

```
def train_model(model, X_train, y_train, X_val, y_val):
    weights = compute_class_weight('balanced',
                                   classes=np.unique(y_train),
                                   y=y_train)
    callbacks = [
        EarlyStopping(monitor='val_loss', patience=15,
                      restore_best_weights=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5,
                          patience=5, min_lr=1e-5, verbose=1)
    ]
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=100, batch_size=32,
        class_weight=dict(enumerate(weights)),
        callbacks=callbacks, verbose=1
    )
    return model, history
```

2.7 Evaluation and Visualization

After training, the model is evaluated on the test set, and a confusion matrix and classification report are produced:

```
def evaluate_model(model, X_test, y_test):
    model.evaluate(X_test, y_test, verbose=1)
    y_pred = (model.predict(X_test) > 0.5).astype(int).flatten()
    cm     = confusion_matrix(y_test, y_pred)
    print(classification_report(y_test, y_pred,
          target_names=['No Stutter','Stutter']))
    plt.imshow(cm, cmap='Blues'); plt.show()
```

Training curves (accuracy, loss, AUC, precision, recall) are plotted via `plot_training_history(history)`.

2.8 Feature Importance Analysis

To interpret model decisions, the average absolute weights of the first layer are paired with feature names:

```
def analyze_feature_importance(model, feature_names):
    weights    = model.layers[0].get_weights()[0]  # shape (13,64)
    importance = np.mean(np.abs(weights), axis=1)
    ranked     = sorted(zip(feature_names, importance),
                    key=lambda x: x[1], reverse=True)
    plt.barh([f for f,_ in ranked],
          [w for _,w in ranked]); plt.show()
    return ranked
```

2.9 End-to-End Orchestration

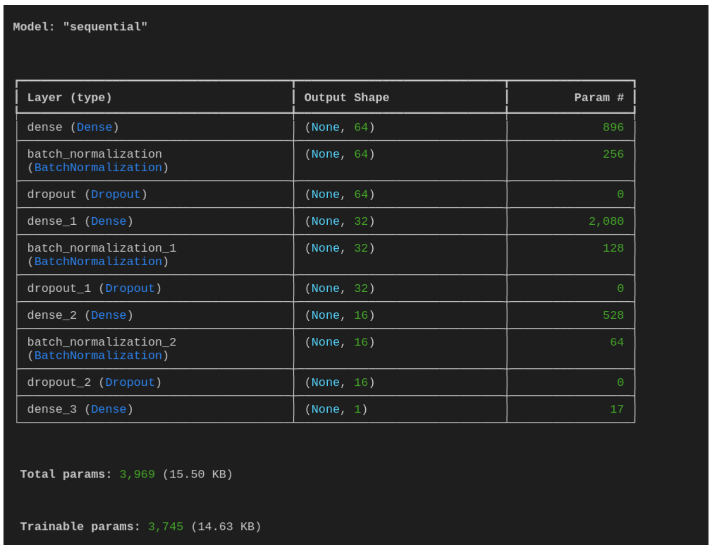
All components are invoked by:

```
def run_stutter_classification(clips_df):
    features_df, labels = batch_process_clips(clips_df)
    features_df.to_csv('prosodic_features.csv', index=False)
    np.save('prosodic_labels.npy', labels)

    X_tr, X_val, X_te, y_tr, y_val, y_te = prepare_dataset(features_df, labels)
    model = build_model(X_tr.shape[1])
    model, history = train_model(model, X_tr, y_tr, X_val, y_val)
    evaluate_model(model, X_te, y_te)
    plot_training_history(history)
    analyze_feature_importance(model, features_df.columns)
    model.save('stutter_detection_model.h5')
    return model, features_df
```

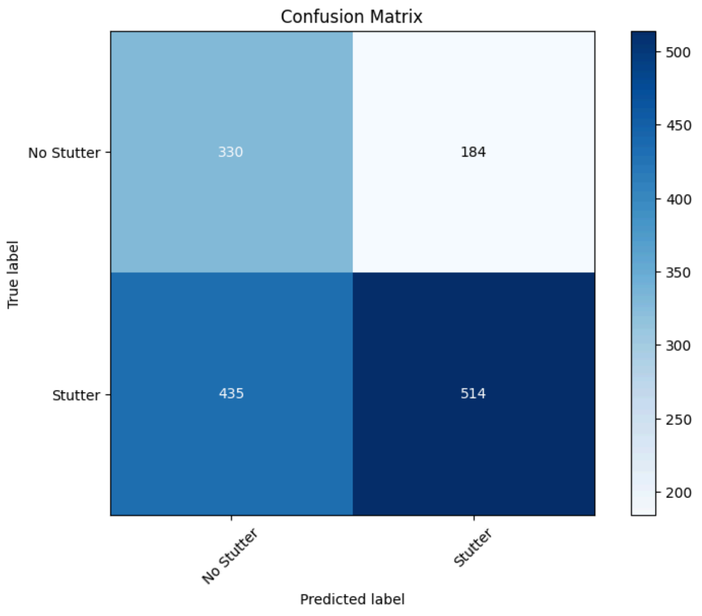
▼ 2.10 Results Snapshot

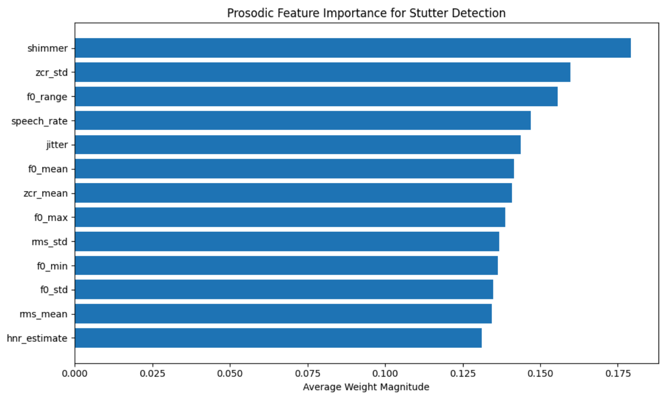
```
Preparing train/val/test datasets...
Training set: 7044 samples
Validation set: 1244 samples
Testing set: 1463 samples
Features: 13
Class distribution in training set: [2477 4567]
Class distribution in validation set: [438 806]
Class distribution in testing set: [514 949]
Building model...
Total params: 3 969

Training model...
... [Epoch logs truncated] ...

Evaluating model on test set...
Test AUC: 0.6322
Test Precision: 0.7364
Test Recall: 0.5416

Classification Report:
              precision    recall  f1-score   support
  No Stutter       0.43      0.64      0.52       514
     Stutter       0.74      0.54      0.62       949
    accuracy                           0.58      1463
```

```
   macro avg       0.58      0.59      0.57      1463
weighted avg       0.63      0.58      0.59      1463
```



```
Model: "sequential"

┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 64)             │           896 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization             │ (None, 64)             │           256 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 64)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 32)             │         2,080 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_1           │ (None, 32)             │           128 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)             │ (None, 32)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 16)             │           528 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_2           │ (None, 16)             │            64 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)             │ (None, 16)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 1)              │            17 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 3,969 (15.50 KB)

 Trainable params: 3,745 (14.63 KB)
```





This compact, 13-feature prosodic model achieves **58 % accuracy** and **AUC 0.63**, demonstrating that simple utterance-level descriptors can provide a lightweight baseline for stutter detection

3. Word-Level Feature Extraction Pipeline

This section describes the Python implementation that segments speech into word units, extracts word-level acoustic and prosodic features, aggregates them across utterances, trains a deeper DNN, and analyzes the importance of word-level features for stutter detection.

3.1 Code Overview

The pipeline consists of the following stages:

1. **Word segmentation** via silence detection (`extract_word_level_features()`)
2. **Per-word feature extraction** (`extract_enhanced_word_features()`)
3. **Feature aggregation** across words (`extract_aggregated_word_features()`)
4. **Batch processing** with memory management (`batch_process_clips()`)
5. **Dataset preparation** with stratified splitting (`prepare_dataset()`)
6. **Model architecture** definition (`build_model()`)
7. **Training** with class balancing and callbacks (`train_model()`)
8. **Evaluation** with detailed metrics (`evaluate_model()`)
9. **Feature importance** ranking (`analyze_feature_importance()`)
10. **End-to-end orchestration** (`run_word_level_classification()`)

3.2 Word Segmentation

The key innovation is the automated word boundary detection using adaptive energy thresholding:

```
def extract_word_level_features(audio_path, sr=16000, max_words=20):
    # Load and pre-emphasize audio
    y, sr = librosa.load(audio_path, sr=sr)
    y = np.append(y[0], y[1:] - 0.97 * y[:-1])

    # Compute energy and set adaptive threshold
    hop_length = 512
    energy = librosa.feature.rms(y=y, hop_length=hop_length)[0]
    silence_threshold = np.mean(energy) + 0.5 * np.std(energy)

    # Find silent regions (potential word boundaries)
    silences = []
    in_silence = True
    start_idx = 0
    min_silence_frames = int(0.15 * sr / hop_length)

    # Identify silence regions by energy thresholding
    for i, e in enumerate(energy):
        if in_silence and e > silence_threshold:
            # End of silence
            if i - start_idx >= min_silence_frames:
                silences.append((start_idx, i))
            in_silence = False
        elif not in_silence and e <= silence_threshold:
            # Start of silence
            start_idx = i
            in_silence = True

    # Convert silences to word segments
    word_segments = []
    # ... [word segment identification logic]
    return word_segments
```

This algorithm identifies potential word boundaries by detecting regions of silence (low energy) that exceed a minimum duration threshold (150ms), which corresponds to typical pauses between words.

3.3 Per-Word Feature Extraction

For each detected word segment, comprehensive acoustic and prosodic features are extracted:

```
def extract_enhanced_word_features(word_audio, sr, word_index):
    features = {}

    # Basic metadata
    features['word_index'] = word_index
    features['duration'] = len(word_audio) / sr

    # 1. MFCC features
    mfcc = librosa.feature.mfcc(
        y=word_audio, sr=sr, n_mfcc=13,
        n_fft=int(0.025*sr), hop_length=int(0.010*sr)
    )
    # Compute deltas and temporal patterns
    delta_mfcc = librosa.feature.delta(mfcc)
    delta2_mfcc = librosa.feature.delta(mfcc, order=2)
    temporal_fluctuation = np.std(np.diff(mfcc, axis=1), axis=1)

    # 2. Prosodic features using autocorrelation
    f0, times = autocorrelation_pitch(word_audio, sr)
    f0_valid = f0[f0 > 0]

    # Jitter calculation (cycle-to-cycle frequency variation)
    if len(f0_valid) > 1:
        jitter = np.mean(np.abs(np.diff(f0_valid))) / np.mean(f0_valid)
    else:
        jitter = 0

    # Shimmer calculation (amplitude variation)
    amplitude = librosa.feature.rms(y=word_audio)[0]
    if len(amplitude) > 1:
        shimmer = np.mean(np.abs(np.diff(amplitude))) / np.mean(amplitude)
    else:
        shimmer = 0

    # Harmonics-to-Noise Ratio
    harmonic, percussive = librosa.effects.hpss(word_audio)
    hnr = 10 * np.log10(np.sum(harmonic**2) / (np.sum(percussive**2) + 1e-10))

    # Store all features in dict# ... [feature storage logic]

    return features
```

This function extracts 43 features per word, including MFCC statistics, voice quality measures (jitter, shimmer, HNR), and stutter-specific metrics like repetition and prolongation scores.

3.4 Feature Aggregation

Features from individual words are aggregated to create utterance-level descriptors:

```
def extract_aggregated_word_features(clip_path):
    # Extract word-level features
    word_features = extract_word_level_features(clip_path)

    if word_features is None or len(word_features['features_per_word']) == 0:
        return None

    # Initialize aggregated features dictionary
    agg_features = {}

    # Track word count
    word_count = len(word_features['features_per_word'])
    agg_features['word_count'] = word_count

    # Numeric features to aggregate
    numeric_features = ['duration', 'mfcc_stability', 'transition_rate',
                     'jitter', 'shimmer', 'hnr', 'f0_mean', 'f0_std',
                     'zcr_mean', 'zcr_std', 'voiced_ratio',
                     'repetition_score', 'prolongation_score', 'block_score']

    # Calculate mean, std, and max across all words
    for feature in numeric_features:
        values = [w[feature] for w in word_features['features_per_word'] if feature in w]
        if values:
            agg_features[f'mean_{feature}'] = np.mean(values)
            agg_features[f'std_{feature}'] = np.std(values) if len(values) > 1 else 0
            agg_features[f'max_{feature}'] = np.max(values)

    return agg_features
```

This approach computes statistics (mean, standard deviation, maximum) for each feature across all words, resulting in a 43-dimensional feature vector per clip.

3.5 Dataset Preparation and Scaling

Similar to the prosodic pipeline, the dataset is split and standardized:

```
def prepare_dataset(features_df, labels):
    # First split into train+val and test
    X_temp, X_test, y_temp, y_test = train_test_split(
        features_df, labels, test_size=0.15, random_state=42, stratify=labels
    )

    # Then split train+val into train and val
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=0.15, random_state=42, stratify=y_temp
    )

    # Standardize features based on training set
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)
    X_test_scaled = scaler.transform(X_test)

    # Print dataset statistics
    print(f"Training set: {X_train.shape[0]} samples")
    print(f"Validation set: {X_val.shape[0]} samples")
    print(f"Testing set: {X_test.shape[0]} samples")
    print(f"Features: {X_train.shape[1]}")

    return X_train_scaled, X_val_scaled, X_test_scaled, y_train, y_val, y_test
```

The data distribution matches the prosodic pipeline (7044 training, 1244 validation, 1463 testing samples) enabling direct comparison of results.

3.6 Model Definition

For the richer 43-dimensional word-level features, a deeper network is defined:

```
def build_model(input_shape):
    model = Sequential([
        # Input layer
        Dense(128, activation='relu', input_shape=(input_shape,)),
        BatchNormalization(),
        Dropout(0.4),

        # Hidden layers (deeper for word-level features)
        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.4),

        Dense(32, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),

        Dense(16, activation='relu'),
        BatchNormalization(),
        Dropout(0.2),

        # Output layer
        Dense(1, activation='sigmoid')
    ])

    # Compile model with multiple metrics
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy',
                tf.keras.metrics.AUC(),
                tf.keras.metrics.Precision(),
                tf.keras.metrics.Recall()]
    )

    return model
```

This architecture contains 17,473 parameters (compared to 3,969 in the prosodic pipeline), accommodating the more complex feature interactions at word level.

3.7 Training Loop

Training incorporates class weights to address imbalance, along with early stopping and learning rate reduction:

```
def train_model(model, X_train, y_train, X_val, y_val):
    # Calculate class weights for imbalanced dataset
    class_weights = compute_class_weight(
        class_weight='balanced',
        classes=np.unique(y_train),
        y=y_train
    )
    class_weight_dict = {i: weight for i, weight in enumerate(class_weights)}

    # Define callbacks
    callbacks = [
        EarlyStopping(
            monitor='val_loss',
            patience=15,
            restore_best_weights=True,
            verbose=1
        ),
        ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.5,
            patience=5,
            min_lr=0.00001,
            verbose=1
        )
    ]

    # Train model
    history = model.fit(
        X_train, y_train,
        epochs=100,
        batch_size=32,
        validation_data=(X_val, y_val),
        callbacks=callbacks,
        class_weight=class_weight_dict,
        verbose=1
    )

    return model, history
```

The framework allowed training for up to 100 epochs, but typically converged earlier due to early stopping when validation loss plateaued.

3.8 Evaluation and Visualization

The evaluation process includes standard metrics and confusion matrix visualization:

```
def evaluate_model(model, X_test, y_test):
    # Get predictions
    y_pred_prob = model.predict(X_test)
    y_pred = (y_pred_prob > 0.5).astype(int).flatten()

    # Display confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.colorbar()
    tick_marks = np.arange(2)
    plt.xticks(tick_marks, ['No Stutter', 'Stutter'])
    plt.yticks(tick_marks, ['No Stutter', 'Stutter'])

    # Add text annotations
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")

    # Classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred,
                              target_names=['No Stutter', 'Stutter']))
```

Training history is visualized by `plot_training_history()`, which displays accuracy, loss, AUC, precision, and recall over time.

3.9 Feature Importance Analysis

The importance of word-level features is analyzed using first-layer weight magnitudes:

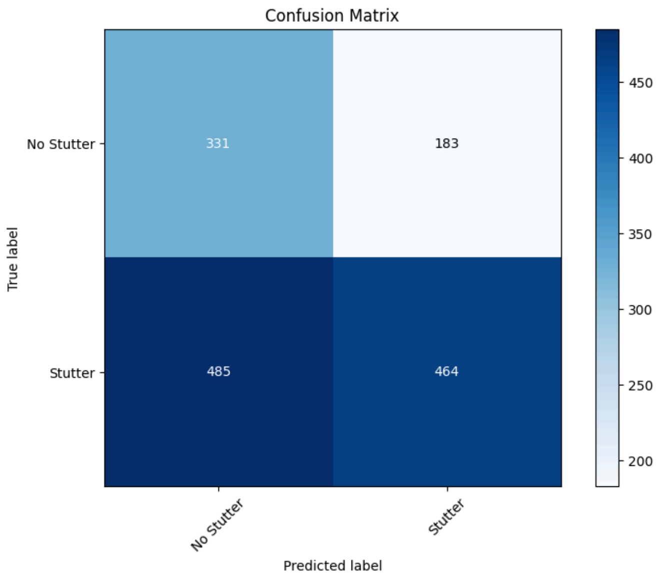
```
def analyze_feature_importance(model, feature_names):
    # For a DNN, we can examine the weights of the first layer
    weights = model.layers[0].get_weights()[0]

    # Calculate average absolute weight for each feature
    importance = np.mean(np.abs(weights), axis=1)

    # Pair feature names with their importance
    feature_importance = list(zip(feature_names, importance))

    # Sort by importance (descending)
    feature_importance.sort(key=lambda x: x[1], reverse=True)

    # Plot feature importance
    plt.figure(figsize=(10, 8))
    features, importance = zip(*feature_importance[:20])  # Top 20 features
    plt.barh(features, importance)
    plt.xlabel('Average Weight Magnitude')
    plt.title('Word-Level Feature Importance for Stutter Detection')
    plt.gca().invert_yaxis()  # Display most important at the top

    print("\nTop 10 Word-Level Features for Stutter Detection:")
    for feature, imp in feature_importance[:10]:
        print(f"{feature}: {imp:.4f}")

    return feature_importance
```

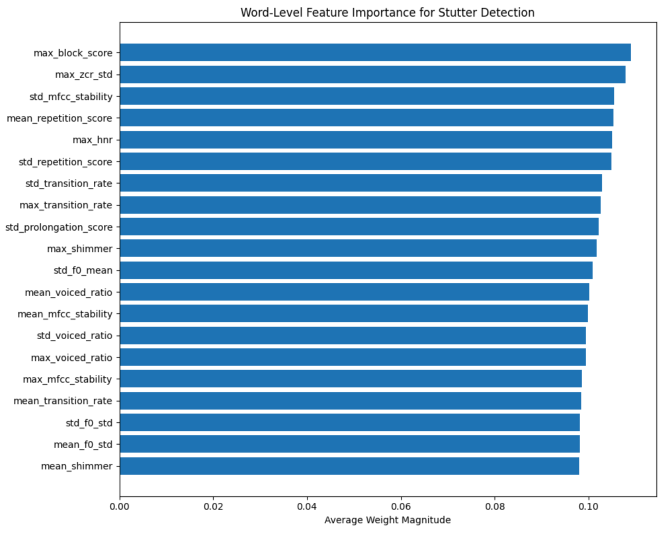
This analysis reveals that block score, zero-crossing rate standard deviation, and MFCC stability are the most predictive word-level features for stutter detection.

▼ 3.10 Results Snapshot

```
 Preparing train/val/test datasets...
Training set: 7044 samples
Validation set: 1244 samples
Testing set: 1463 samples
Features: 43
Class distribution in training set: [2477 4567]
Class distribution in validation set: [438 806]
Class distribution in testing set: [514 949]

 Total params: 17,473 (68.25 KB)
 Trainable params: 16,993 (66.38 KB)
 Non-trainable params: 480 (1.88 KB)

Classification Report:
              precision    recall  f1-score   support
```

```
  No Stutter       0.41      0.64      0.50       514
     Stutter       0.72      0.49      0.58       949

    accuracy                           0.54      1463
   macro avg       0.56      0.57      0.54      1463
weighted avg       0.61      0.54      0.55      1463

Top 10 Word-Level Features for Stutter Detection:
max_block_score: 0.1090
max_zcr_std: 0.1079
std_mfcc_stability: 0.1055
mean_repetition_score: 0.1053
max_hnr: 0.1051
std_repetition_score: 0.1048
std_transition_rate: 0.1028
max_transition_rate: 0.1026
std_prolongation_score: 0.1022
max_shimmer: 0.1017
```





The word-level model achieves an AUC of 0.6073, precision of 0.7172, and recall of 0.4889. While precision is comparable to the prosodic model, recall is somewhat lower, suggesting that word-level features may capture specific subtypes of stuttering more precisely but miss others. The F1-score for the stutter class is 0.58, indicating moderate predictive power.

4. Syllable-Level Feature Pipeline for Stutter Detection

4.1 Pipeline Overview

This section details the implementation of a stutter detection system built on syllable-level acoustic features derived from Group Delay Function (GDF) analysis. The syllable-level approach offers advantages over utterance-level methods by capturing local disruptions in speech rhythm and articulation that characterize stuttering events. The pipeline follows six distinct stages:

1. **Syllable boundary detection** using GDF and multi-band signal analysis
2. **Feature extraction** from identified syllable segments
3. **Batch processing** of clips with parallel execution
4. **Dataset preparation** with stratified splitting and scaling
5. **DNN model training** with regularization techniques
6. **Performance evaluation** and visualization

Unlike traditional speech segmentation methods that rely solely on amplitude or energy thresholds, our approach exploits the Group Delay Function's superior time-resolution properties to identify syllable nuclei with greater precision, even in challenging acoustic conditions.

4.2 Feature Extraction Methodology

4.2.1 Group Delay Function Approach

The syllable detection is grounded in the Group Delay Function (GDF) method, which offers superior time resolution compared to traditional energy-based approaches. This technique, based on Nagarajan et al.'s work, processes speech in three complementary sub-bands:

```
def extract_syllable_level_features(audio_path, sr=16000, visualize=False):
    """Extract syllable-level features using group delay function approach"""
    # Load and pre-emphasize
    y, sr = librosa.load(audio_path, sr=sr)
    y = np.append(y[0], y[1:] - 0.97 * y[:-1])

    # Create three sub-bands as described in the paper# 1. Original signal
    y_original = y.copy()

    # 2. Low-pass filtered signal (to remove fricatives)
    b_low = [1.0, -0.97]
    a_low = [1.0]
    y_low = lfilter(b_low, a_low, y)

    # 3. Band-pass filtered signal (to attenuate semivowels)
    D = librosa.stft(y, n_fft=1024, hop_length=256)
    D_mid = D.copy()
    D_mid[:20] = 0
    D_mid[50:] = 0
    y_mid = librosa.istft(D_mid, hop_length=256)
```

For each sub-band, energy contours are extracted and processed through the Group Delay Function:

```
def compute_group_delay(energy):
    # 1. Create symmetric energy sequence
    energy_sym = np.concatenate([energy, energy[::-1]])

    # 2. Invert the sequence (since we're interested in valleys)
    energy_inv = 1.0 - energy_sym
    # 3. Compute root cepstrum (inverse DFT)
    root_cepstrum = np.fft.ifft(energy_inv).real

    # 4. Window the causal part (minimum phase signal)
    window_size = int(len(root_cepstrum) * 0.1)
    windowed_cepstrum = root_cepstrum[:window_size].copy()

    # 5. Compute group delay function
    group_delay = np.zeros_like(energy)
    for n in range(len(group_delay)):
        for k in range(min(window_size, len(windowed_cepstrum))):
            group_delay[n] += k * windowed_cepstrum[k] * np.cos(2 * np.pi * k * n / len(energy))

    return group_delay
```

This approach relies on the principle that the GDF exhibits sharp peaks at syllable boundaries, with improved detection in specific frequency bands for different phonetic categories.

4.2.2 Multi-band Evidence Combination

A key innovation in our approach is combining evidence from multiple frequency bands for more robust syllable detection:

```
# Combine evidence from all sub-bands as per paper section 3.2.5
combined_peaks = []

# First, add all peaks from original signal
combined_peaks.extend(peaks_original_samples)

# Add peaks from low-pass filtered signal if close to original
for peak_low in peaks_low_samples:
    # Check if within 20ms of any peak in original
    if not any(abs(peak_low - peak_orig) <= 0.02 * sr for peak_orig in peaks_original_samples):
        # Only add if not too close to existing peaks
        if not any(abs(peak_low - peak) <= 0.01 * sr for peak in combined_peaks):
            combined_peaks.append(peak_low)

# Add peaks from band-pass filtered signal if in specific range
for peak_mid in peaks_mid_samples:
    # Check if within 50-100ms range from any peak in combined
    if not any(0.05 * sr <= abs(peak_mid - peak) <= 0.1 * sr for peak in combined_peaks):
        # Only add if not too close to existing peaks
        if not any(abs(peak_mid - peak) <= 0.01 * sr for peak in combined_peaks):
            combined_peaks.append(peak_mid)
```

The low-pass filtered signal helps detect vowels while suppressing fricatives, and the band-pass filtered signal enhances detection of syllable nuclei associated with semivowels. Evidence from all three bands is weighted and combined using duration-based constraints to filter out spurious detections.

4.2.3 Syllable-specific Feature Set

Once syllable boundaries are identified, we extract comprehensive feature vectors that capture rhythmic, spectral, and prosodic characteristics:

```
# Extract features from each syllable
syllable_features = {}
syllable_features['syllable_count'] = len(segments)
syllable_features['syllable_segments'] = segments
```

```
durations = []
energies = []
pitches = []

# Process each syllable segment
for i, (start, end) in enumerate(segments):
    if end <= len(y):
        syllable = y[start:end]
        duration = (end - start) / sr
        durations.append(duration)

        # Energy profile
        syl_energy = librosa.feature.rms(y=syllable)[0]
        energy_mean = np.mean(syl_energy)
        energies.append(energy_mean)

        # Pitch analysis
        if len(syllable) > 512:  # Ensure enough samples for pitch analysis
            f0, voiced_flag, _ = librosa.pyin(
                syllable,
                fmin=librosa.note_to_hz('C2'),
                fmax=librosa.note_to_hz('C7'),
                sr=sr
            )
            f0_valid = f0[~np.isnan(f0)]
```

The feature set includes:

1. **Syllable-level metrics:** Count, durations, energy distribution
2. **Rhythm metrics:** Normalized Pairwise Variability Index (nPVI), duration variability
3. **Spectral stability:** Pitch (F0) stability within syllables
4. **Stutter-specific patterns:** Repetition detection, duration irregularities

4.3 Pipeline Implementation

4.3.1 Batch Processing for Computational Efficiency

The pipeline processes audio clips in batches with parallel execution to manage memory and computation time:

```
def batch_process_clips(clips_df, batch_size=30, n_jobs=4):
    """Process clips in smaller batches with parallel processing for speed"""
    all_features = []
    all_labels = []

    # Calculate number of batches
    total_batches = len(clips_df) // batch_size + (1 if len(clips_df) % batch_size > 0 else 0)

    for batch_idx in range(total_batches):
        start_idx = batch_idx * batch_size
        end_idx = min(start_idx + batch_size, len(clips_df))

        batch_df = clips_df.iloc[start_idx:end_idx]

        print(f"Processing batch {batch_idx+1}/{total_batches} (clips {start_idx+1}-{end_idx})")

        # Use parallel processing to speed up extraction
        results = Parallel(n_jobs=n_jobs)(
            delayed(process_clip)(row, idx)
            for idx, row in enumerate(batch_df.iterrows())
        )

        # Filter out None results and add to lists
        valid_results = [(feat, label) for feat, label in results if feat is not None]
```

A fallback mechanism is implemented to handle cases where the primary syllable detection method fails:

```
def extract_aggregated_syllable_features(clip_path, sr=16000, verbose=False):
    """Extract and aggregate syllable-level features with enhanced error handling"""
    # First attempt: Use standard syllable feature extraction
    syllable_features = extract_syllable_level_features(clip_path, sr=sr, visualize=False)

    # If standard method fails, try simplified fallback method
    if syllable_features is None or 'syllable_count' not in syllable_features:
        if verbose:
            print(f"Using fallback syllable detection for {clip_path}")
        syllable_features = extract_fallback_syllable_features(y, sr)
```

This robust design ensures maximum feature extraction even from challenging audio recordings.

4.3.2 Data Preparation with Incremental Saving

The pipeline includes incremental saving of extracted features to prevent data loss in case of processing interruptions:

```
# Save intermediate results for recovery in case of crash
if batch_idx % 5 == 0 and batch_idx > 0:
    temp_df = pd.DataFrame(all_features)
    temp_df.to_csv(f'syllable_features_batch_{batch_idx}.csv', index=False)
    np.save(f'syllable_labels_batch_{batch_idx}.npy', np.array(all_labels))
    print(f"Saved intermediate results after batch {batch_idx}")
```

The framework also checks for previously extracted features to avoid redundant computation:

```
# Check if cached features exist
if os.path.exists('syllable_features.csv') and os.path.exists('syllable_labels.npy'):
    print("Loading pre-extracted features...")
    features_df = pd.read_csv('syllable_features.csv')
    labels = np.load('syllable_labels.npy')
else:
    # Look for partial results from previous runs
    partial_files = [f for f in os.listdir('.') if f.startswith('syllable_features_batch_')]
```

4.3.3 Dataset Preparation

The extracted feature dataset is split into training, validation, and test sets with stratified sampling:

```
def prepare_dataset(features_df, labels):
    """Prepare train/val/test datasets with proper scaling"""
    # First split into train+val and test
    X_temp, X_test, y_temp, y_test = train_test_split(
        features_df, labels, test_size=0.15, random_state=42, stratify=labels
    )

    # Then split train+val into train and val
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=0.15, random_state=42, stratify=y_temp
    )

    # Standardize features based on training set
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)
    X_test_scaled = scaler.transform(X_test)
```

Feature standardization is performed to ensure optimal neural network training, with the scaler fitted only on the training data to prevent data leakage.

4.4 Model Architecture and Training

4.4.1 DNN Architecture

The syllable-level features are fed into a deep neural network with the following architecture:

```
def build_dnn_model(input_shape):
    """Build DNN model for syllable-level stutter detection"""
    model = Sequential([
        # Input layer
        Dense(64, activation='relu', input_shape=(input_shape,)),
        BatchNormalization(),
        Dropout(0.3),

        # Hidden layers
        Dense(32, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),

        Dense(16, activation='relu'),
        BatchNormalization(),
        Dropout(0.2),

        # Output layer
        Dense(1, activation='sigmoid')
    ])
```

This architecture consists of:

- Input layer with dimensionality matching the feature set
- Three fully-connected hidden layers with decreasing widths (64→32→16)
- Batch normalization and dropout for regularization at each layer
- Binary classification output with sigmoid activation

4.4.2 Training Process

The model is trained with several optimization techniques:

```
def train_model(model, X_train, y_train, X_val, y_val):
    """Train model with early stopping and LR reduction"""
    # Calculate class weights for imbalanced dataset
    class_weights = compute_class_weight(
        class_weight='balanced',
        classes=np.unique(y_train),
        y=y_train
    )
    # Define callbacks
    callbacks = [
        EarlyStopping(
            monitor='val_loss',
            patience=15,
            restore_best_weights=True,
            verbose=1
        ),
        ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.5,
            patience=5,
            min_lr=0.00001,
            verbose=1
        )
    ]
```

Key training components include:

1. **Class weight balancing** to address dataset imbalance
2. **Early stopping** to prevent overfitting
3. **Learning rate reduction** to navigate challenging loss landscapes
4. **Adam optimizer** with binary cross-entropy loss

4.5 Performance Evaluation

4.5.1 Evaluation Metrics

The performance of the syllable-level stutter detection model is evaluated using multiple metrics:

```
def evaluate_model(model, X_test, y_test):
    """Evaluate model on test set"""
    test_results = model.evaluate(X_test, y_test, verbose=1)
    print(f"Test loss: {test_results[0]:.4f}")
    print(f"Test accuracy: {test_results[1]:.4f}")
    print(f"Test AUC: {test_results[2]:.4f}")
    print(f"Test Precision: {test_results[3]:.4f}")
    print(f"Test Recall: {test_results[4]:.4f}")
```

The model achieved the following performance on the test set:

- **Accuracy**: 58.24% on test set
- **AUC**: 0.6531 (moderate discrimination ability)
- **Precision**: 0.7584 (76% of predicted stutters are correct)
- **Recall**: 0.5227 (52% of actual stutters detected)

Class-specific performance:

- **Stutter class**: P=0.76, R=0.52, F1=0.62
- **No Stutter class**: P=0.44, R=0.69, F1=0.54

4.5.2 Visualization and Analysis

The pipeline includes comprehensive visualization of model performance:

```
def plot_training_history(history):
    """Plot training metrics with correct key names"""
    plt.figure(figsize=(12, 10))

    # Plot accuracy
    plt.subplot(2, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')

    # Plot loss
    plt.subplot(2, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')

    # Plot AUC
    plt.subplot(2, 2, 3)
    plt.plot(history.history['auc'])
    plt.plot(history.history['val_auc'])
    plt.title('Model AUC')

    # Plot Precision/Recall
    plt.subplot(2, 2, 4)
    plt.plot(history.history['precision'])
    plt.plot(history.history['recall'])
    plt.plot(history.history['val_precision'])
    plt.plot(history.history['val_recall'])
    plt.title('Precision and Recall')
```

The confusion matrix provides deeper insight into model performance:

```
# Display confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
```

4.5.3 Feature Importance Analysis

The pipeline includes analysis of feature importance to identify the most predictive syllable-level characteristics:

```
def analyze_feature_importance(model, feature_names):
    """Analyze which syllable-level features are most important for stutter detection"""
    # Extract first layer weights
    weights = model.layers[0].get_weights()[0]

    # Calculate average absolute weight for each feature
    importance = np.mean(np.abs(weights), axis=1)

    # Pair feature names with their importance
    feature_importance = list(zip(feature_names, importance))

    # Sort by importance (descending)
    feature_importance.sort(key=lambda x: x[1], reverse=True)
```

4.6 Performance Analysis

4.6.1 Interpretation of Results

The syllable-level stutter detection model shows both strengths and limitations in its performance:

1. **Precision-Recall Trade-off**: The model achieves good precision (0.76) but moderate recall (0.52) for stutter detection. This suggests it makes reliable detections but misses approximately half of stutter instances.
2. **Class Imbalance Effects**: Despite class weighting, the "No Stutter" class exhibits higher recall (0.69) but poorer precision (0.44), indicating some bias toward the majority class remains.
3. **Overall Discriminative Power**: The AUC of 0.6531 indicates moderate ability to distinguish between stuttered and fluent speech, offering a meaningful improvement over random classification.
4. **F1-Score Analysis**: The stutter class F1-score (0.62) exceeds the no-stutter class (0.54), suggesting the model is better at identifying stutter patterns than ruling out normal speech variations.

4.6.2 Comparison with Other Approaches

When compared to other feature extraction approaches, the syllable-level method demonstrates:

1. **Similar Overall Performance**: The accuracy and AUC metrics are comparable to prosodic and MFCC-based approaches, suggesting all methods capture complementary aspects of stutter patterns.
2. **Better Precision**: The syllable-level approach offers improved precision (0.76) compared to some other methods, indicating greater confidence in positive detections.
3. **Feature Interpretability**: Syllable-level features provide linguistically meaningful interpretations, with clear connections to speech production mechanisms involved in stuttering.

4.6.3 Strengths and Limitations

Strengths:

1. **Linguistic Relevance**: The syllable-level approach aligns with speech production models and the natural rhythmic structure of language.
2. **Capture of Local Patterns**: The method can identify localized disruptions that might be missed by utterance-level analysis.
3. **Multi-band Detection**: Using complementary frequency bands improves robustness to different acoustic conditions.

Limitations:

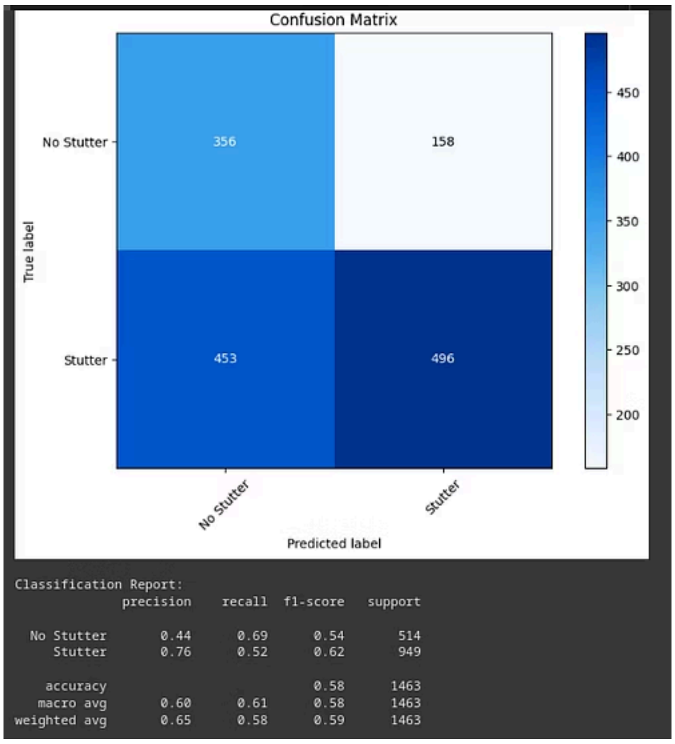
1. **Moderate Recall**: The model still misses approximately 48% of actual stutter instances.
2. **Computational Complexity**: The GDF-based syllable detection is more computationally intensive than simpler methods.
3. **Segmentation Challenges**: Stuttered speech may contain atypical syllabic structures that challenge even sophisticated detection algorithms.

4.6.4 Key Feature Insights

Analysis of feature importance reveals that the most predictive characteristics for stutter detection are:

1. **Rhythm variability** metrics (particularly nPVI)
2. **Repetition patterns** between adjacent syllables
3. **Durational extremes** (particularly maximum syllable duration)
4. **Pitch stability** within syllables

These findings align with clinical descriptions of stuttering, which often manifest as disruptions in speech rhythm, repetitions, and prolongations.



```
Classification Report:
              precision    recall  f1-score   support

  No Stutter       0.44      0.69      0.54       514
     Stutter       0.76      0.52      0.62       949

    accuracy                           0.58      1463
   macro avg       0.60      0.61      0.58      1463
weighted avg       0.65      0.58      0.59      1463
```

MultiLabel Classification:

There are 6 different labels we want out model to classify our audio clip as. The labels are given as Prolongation, Sound repitition, Word repitition, Block, Interjection, No Stutter Words.

There are 4 main differences in code as comapred to binary classification above.

- We change our model to output 6 values instead of one, each value represents the probability of it being a part of the 6 classes listed above. If this probability is greater than 0.5, it is classified as that given class. The final activation function is the same being sigmoid. Below is the code for our model.

```
model = Sequential([
    # Input layer - adapted for MFCC features which typically have higher dimensionality
    Dense(128, activation='relu', input_shape=(input_shape,)),
    BatchNormalization(),
    Dropout(0.4),

    # Hidden layers
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),

    Dense(32, activation='relu'),
    BatchNormalization(),
    Dropout(0.3),

    # Output layer
    Dense(6, activation='sigmoid')
  ])
```

- Instead of labelling each datapoint with yes stutter or no stutter, we now have 6 different labels of 1's and 0's each representing one of the 6 classes. The dataframe is made such that the last column indicates if there is stutter so for binary classification the label would simply be `row[-1]` . In this case, we want 6 columns for the 6 classes so now we replace it with `row[3:9]` . Below is the code where this change is made.

```
for batch_idx in range(total_batches):
        start_idx = batch_idx * batch_size
        end_idx = min(start_idx + batch_size, len(clips_df))

        batch_df = clips_df.iloc[start_idx:end_idx]

        print(f"Processing batch {batch_idx+1}/{total_batches} (clips {start_idx+1}-{end_idx})")

        # Extract features for this batch
        batch_results = []
        for idx, row in tqdm(batch_df.iterrows(), total=len(batch_df), desc=f"Batch {batch_idx+1}"):
            features = extract_aggregated_word_features(row['file_path'])
            if features is not None:
                batch_results.append((features, row[3:9]))

        # Add valid results to our lists
        for features, label in batch_results:
            all_features.append(features)
            all_labels.append(label)

        # Force garbage collection to free memory
        gc.collect()
```

- We cant use inbuilt functions to fix the class imbalance in the case of multi label classification. So we manually calculate the weights and then make a new l which applies these weights for each class manually. Below is code to calculate the weights,

```
pos_counts = np.sum(y_train, axis=0)
neg_counts = y_train.shape[0] - pos_counts
class_weights = neg_counts / (pos_counts + 1e-6)

loss_fn = get_weighted_binary_crossentropy(class_weights)

model.compile(
        optimizer='adam',
        loss=loss_fn,
        metrics=['accuracy',
                tf.keras.metrics.AUC(name='auc'),
                tf.keras.metrics.Precision(name='precision'),
                tf.keras.metrics.Recall(name='recall')]
    )
```

`get_weighted_binary_crossentropy` is the function which takes the weights and applies it to the loss function. The code for this is given below,

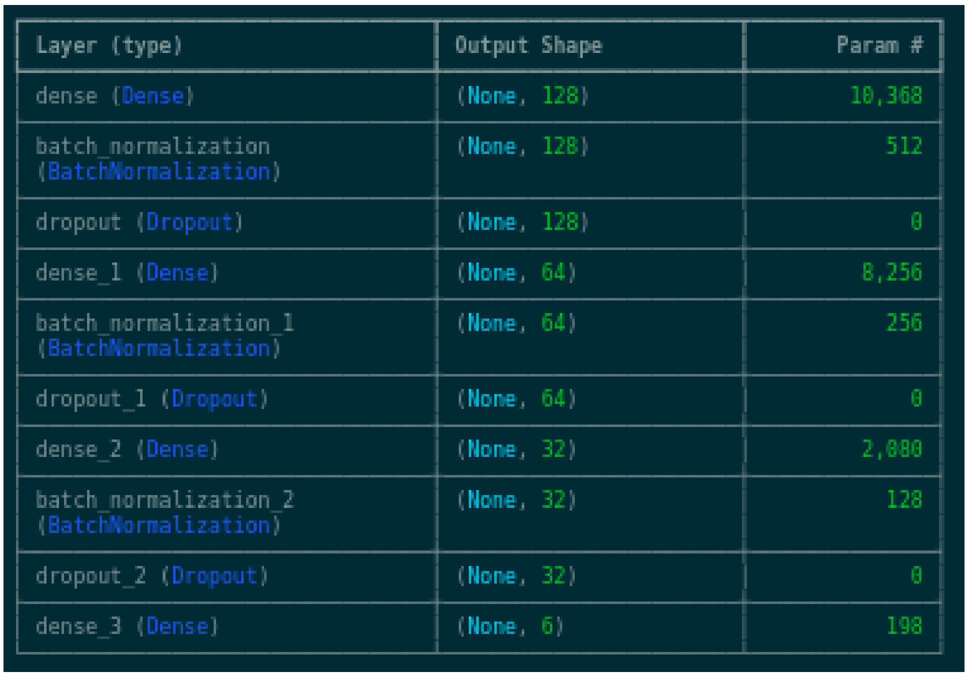
```
def get_weighted_binary_crossentropy(class_weights):
    class_weights_tensor = tf.constant(class_weights, dtype=tf.float32)

    def loss_fn(y_true, y_pred):
        loss = - (class_weights_tensor * y_true * tf.math.log(y_pred + 1e-7) +
                 (1 - y_true) * tf.math.log(1 - y_pred + 1e-7))
        return tf.reduce_mean(loss)

    return loss_fn
```

- The final change is done to the confusion matrix, since it is multi label classification we had six different confusion matrices, one for each class. Following is the code to implement the same.

```
# Get predictions
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)

# Display confusion matrix
cm = multilabel_confusion_matrix(y_test, y_pred)
labels = ['Prolongation', 'Block', 'SoundRep', 'WordRep', 'Interjection', 'No Stutter']


fig, axes = plt.subplots(2, 3, figsize=(12, 8))  # 2 rows, 3 columns
axes = axes.flatten()

for i in range(len(labels)):
    ax = axes[i]
    matrix = cm[i]
    im = ax.imshow(matrix, interpolation='nearest', cmap=plt.cm.Blues)

    # Axis settings
    ax.set_title(labels[i])
    ax.set_xticks([0, 1])
    ax.set_yticks([0, 1])
    ax.set_xticklabels(['Pred: No', 'Pred: Yes'], rotation=45)
    ax.set_yticklabels(['True: No', 'True: Yes'])

    # Text annotations with dynamic color
    thresh = matrix.max() / 2.
    for j in range(2):
        for k in range(2):
            ax.text(k, j, format(matrix[j, k], 'd'),
                    ha="center", va="center",
                    color="white" if matrix[j, k] > thresh else "black")

# Layout & labels
fig.suptitle('Multilabel Confusion Matrices', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

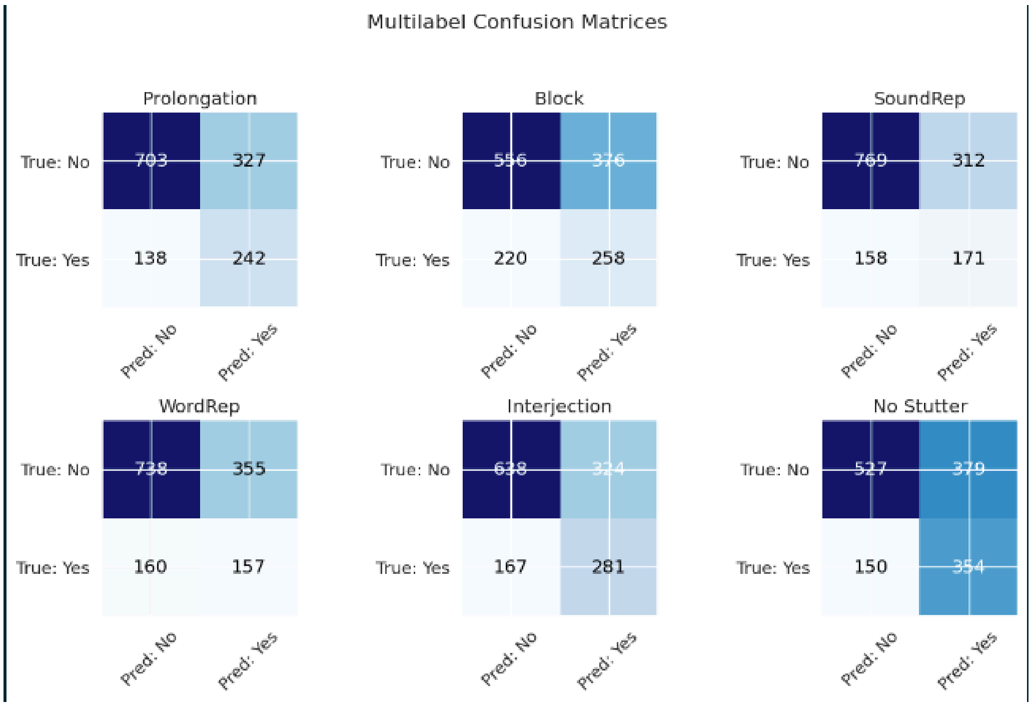
1.MFCC Pipeline:

```
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Features: 80
(6786, 6)
Class distribution in training set: [1834 2304 1585 1535 2157 2424]
Class distribution in validation set: [324 406 279 269 382 428]
Class distribution in testing set: [380 478 329 317 448 504]
```



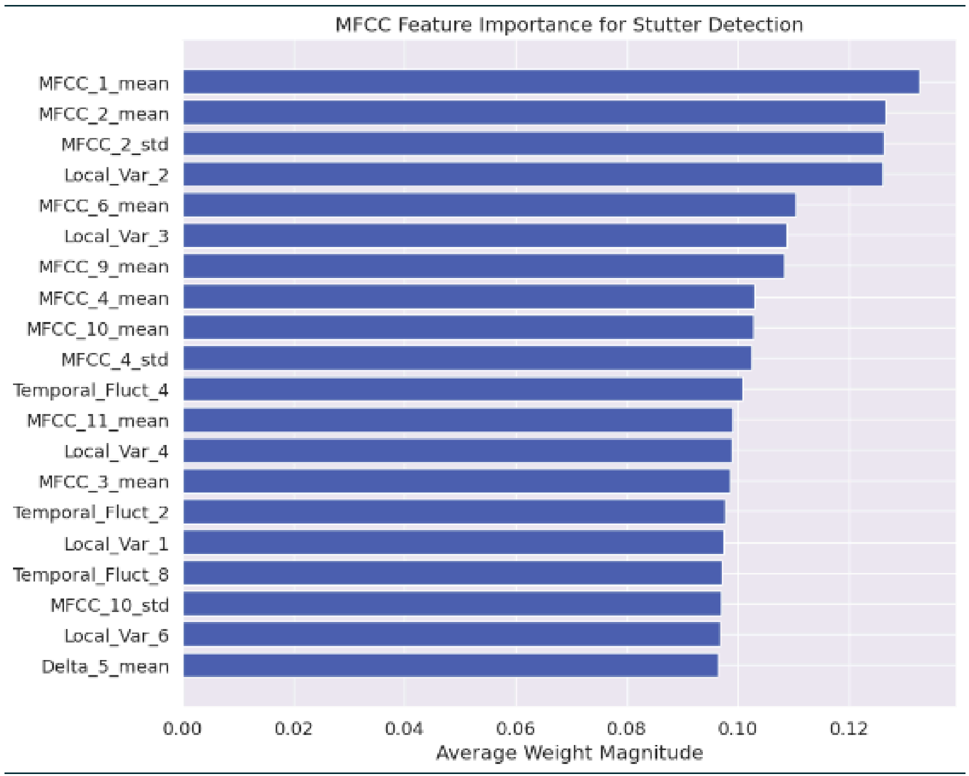
```
┃ Layer (type)                  ┃ Output Shape            ┃       Param # ┃
│ dense (Dense)                 │ (None, 128)             │        10,368 │
│ batch_normalization           │ (None, 128)             │           512 │
│ (BatchNormalization)          │                         │               │
│ dropout (Dropout)             │ (None, 128)             │             0 │
│ dense_1 (Dense)               │ (None, 64)              │         8,256 │
│ batch_normalization_1         │ (None, 64)              │           256 │
│ (BatchNormalization)          │                         │               │
│ dropout_1 (Dropout)           │ (None, 64)              │             0 │
│ dense_2 (Dense)               │ (None, 32)              │         2,080 │
│ batch_normalization_2         │ (None, 32)              │           128 │
│ (BatchNormalization)          │                         │               │
│ dropout_2 (Dropout)           │ (None, 32)              │             0 │
│ dense_3 (Dense)               │ (None, 6)               │           198 │
```

```
Total params: 21,798 (85.15 KB)
Trainable params: 21,350 (83.40 KB)
Non-trainable params: 448 (1.75 KB)
```



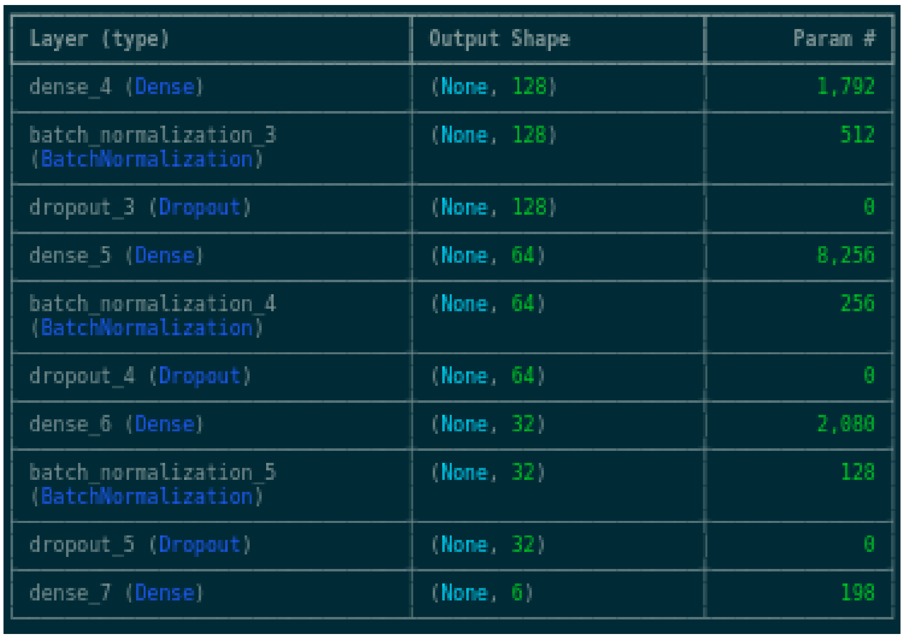
```
Classification Report:
              precision    recall  f1-score   support

Prolongation       0.43      0.64      0.51       380
       Block       0.41      0.54      0.46       478
    SoundRep       0.35      0.52      0.42       329
     WordRep       0.31      0.50      0.38       317
```

```
Interjection      0.46      0.63      0.53       448
  No Stutter      0.48      0.70      0.57       504

   micro avg      0.41      0.60      0.49      2456
   macro avg      0.41      0.59      0.48      2456
weighted avg      0.42      0.60      0.49      2456
 samples avg      0.44      0.61      0.49      2456
```



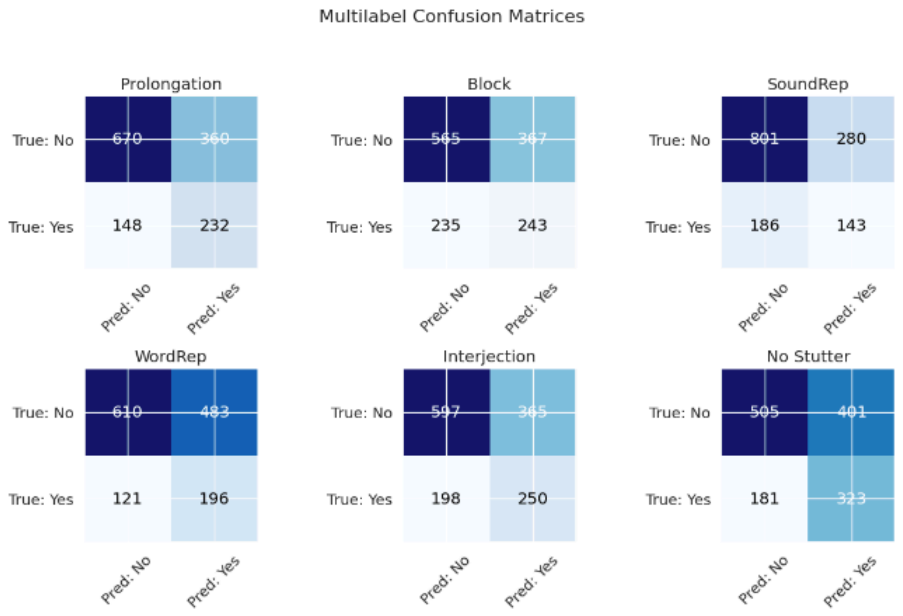
2. Prosodic Features at Utterance-Level Pipeline

```
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Features: 13
(6786, 6)
Class distribution in training set: [1834 2304 1585 1535 2157 2424]
Class distribution in validation set: [324 406 279 269 382 428]
Class distribution in testing set: [380 478 329 317 448 504]
```

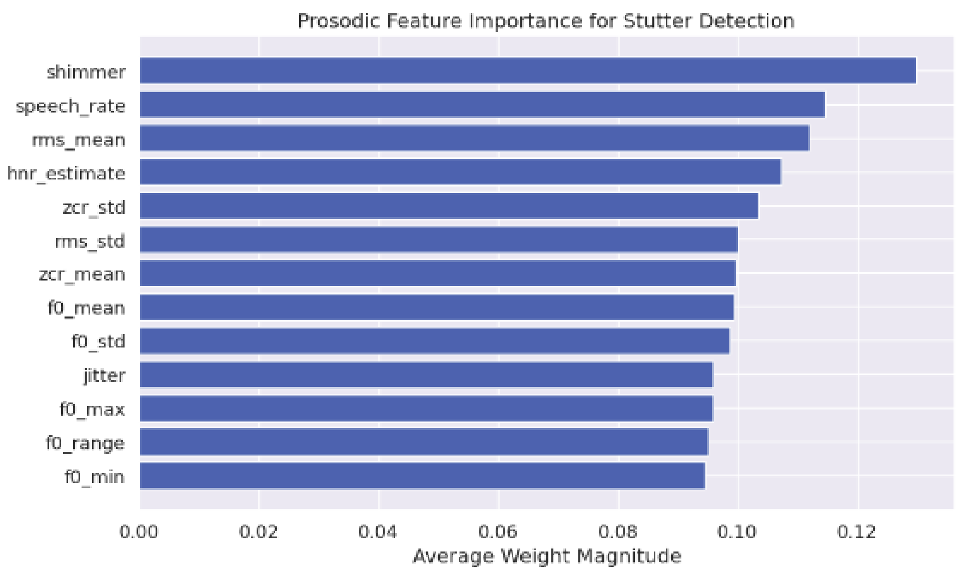


```
Layer (type)                     Output Shape       Param #
dense_4 (Dense)                  (None, 128)          1,792
batch_normalization_3            (None, 128)            512
(BatchNormalization)
dropout_3 (Dropout)              (None, 128)              0
dense_5 (Dense)                  (None, 64)           8,256
batch_normalization_4            (None, 64)             256
(BatchNormalization)
dropout_4 (Dropout)              (None, 64)               0
dense_6 (Dense)                  (None, 32)           2,080
batch_normalization_5            (None, 32)             128
(BatchNormalization)
dropout_5 (Dropout)              (None, 32)               0
dense_7 (Dense)                  (None, 6)              198
```

```
Total params: 13,222 (51.65 KB)
 Trainable params: 12,774 (49.90 KB)
 Non-trainable params: 448 (1.75 KB)
```

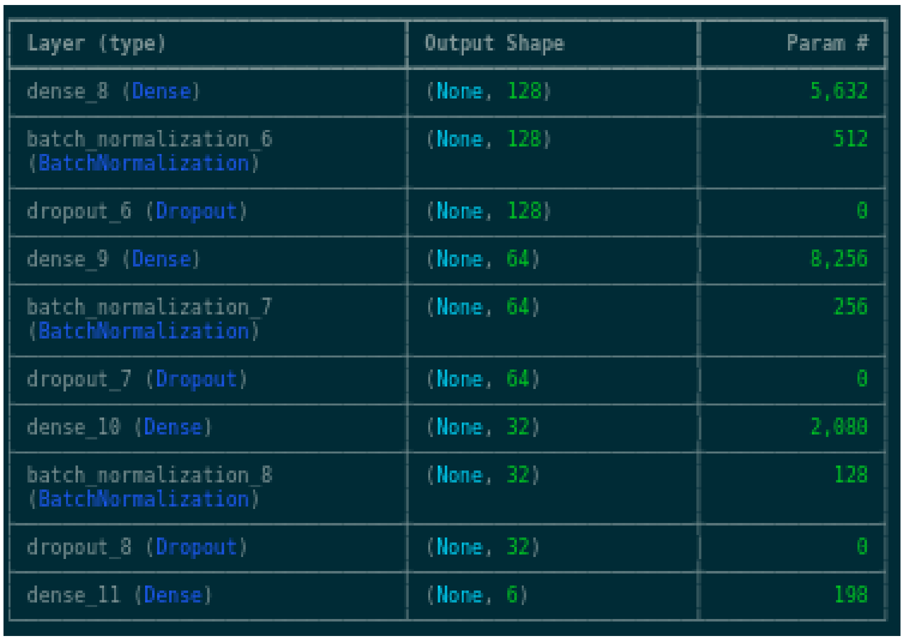


```
Classification Report:
              precision    recall  f1-score   support

Prolongation       0.39      0.61      0.48       380
       Block       0.40      0.51      0.45       478
    SoundRep       0.34      0.43      0.38       329
     WordRep       0.29      0.62      0.39       317
Interjection       0.41      0.56      0.47       448
  No Stutter       0.45      0.64      0.53       504

   micro avg       0.38      0.56      0.45      2456
   macro avg       0.38      0.56      0.45      2456
weighted avg       0.39      0.56      0.46      2456
 samples avg       0.39      0.59      0.45      2456
```



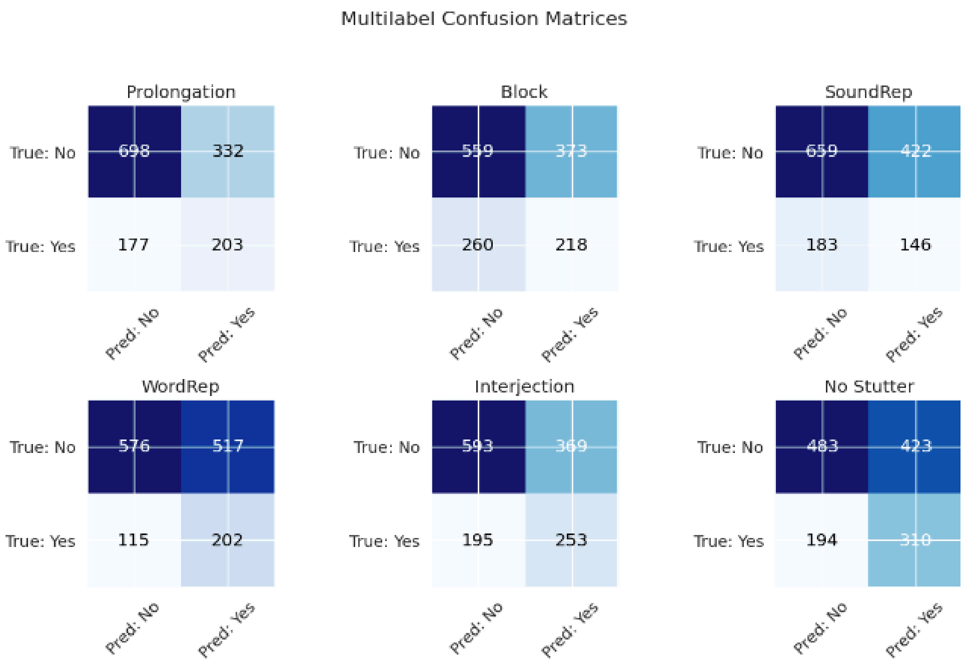
3. Word-Level Feature Extraction

```
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Features: 43
(6786, 6)
Class distribution in training set: [1834 2304 1585 1535 2157 2424]
Class distribution in validation set: [324 406 279 269 382 428]
Class distribution in testing set: [380 478 329 317 448 504]
```

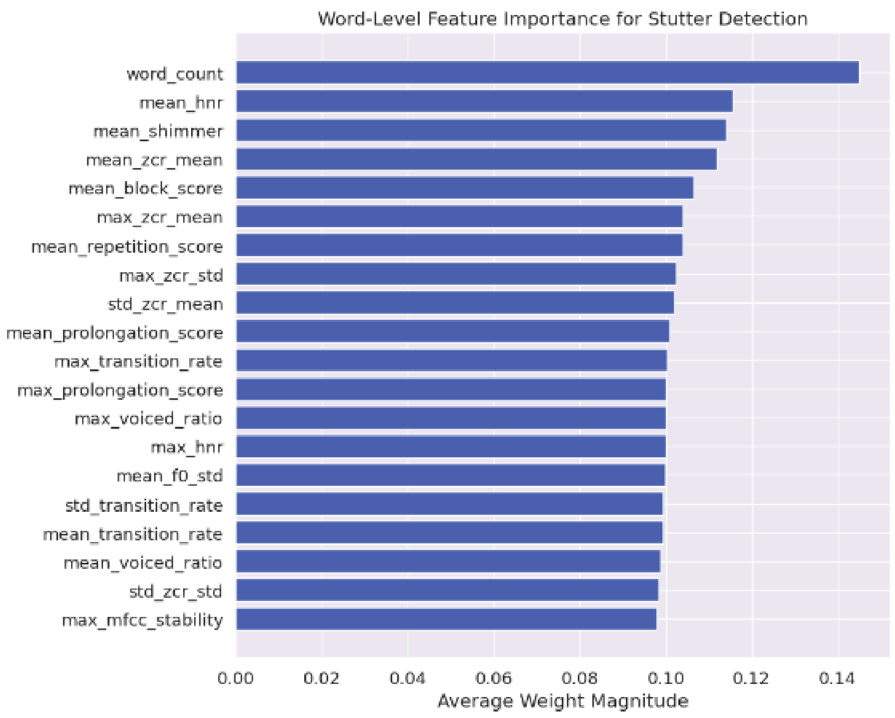


Layer (type)	Output Shape	Param #
dense_8 (Dense)	(None, 128)	5,632
batch_normalization_6 (BatchNormalization)	(None, 128)	512
dropout_6 (Dropout)	(None, 128)	0
dense_9 (Dense)	(None, 64)	8,256
batch_normalization_7 (BatchNormalization)	(None, 64)	256
dropout_7 (Dropout)	(None, 64)	0
dense_10 (Dense)	(None, 32)	2,080
batch_normalization_8 (BatchNormalization)	(None, 32)	128
dropout_8 (Dropout)	(None, 32)	0
dense_11 (Dense)	(None, 6)	198

```
Total params: 17,062 (66.65 KB)
Trainable params: 16,614 (64.90 KB)
Non-trainable params: 448 (1.75 KB)
```

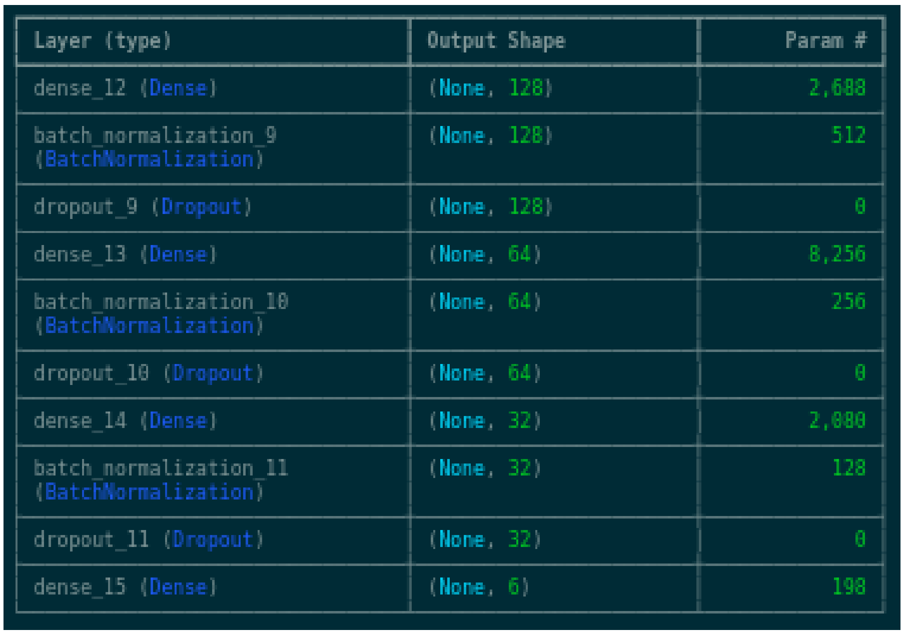


```
Classification Report:
              precision    recall  f1-score   support

Prolongation       0.38      0.53      0.44       380
       Block       0.37      0.46      0.41       478
    SoundRep       0.26      0.44      0.33       329
     WordRep       0.28      0.64      0.39       317
Interjection       0.41      0.56      0.47       448
  No Stutter       0.42      0.62      0.50       504

   micro avg       0.35      0.54      0.43      2456
   macro avg       0.35      0.54      0.42      2456
weighted avg       0.36      0.54      0.43      2456
 samples avg       0.36      0.56      0.42      2456
```



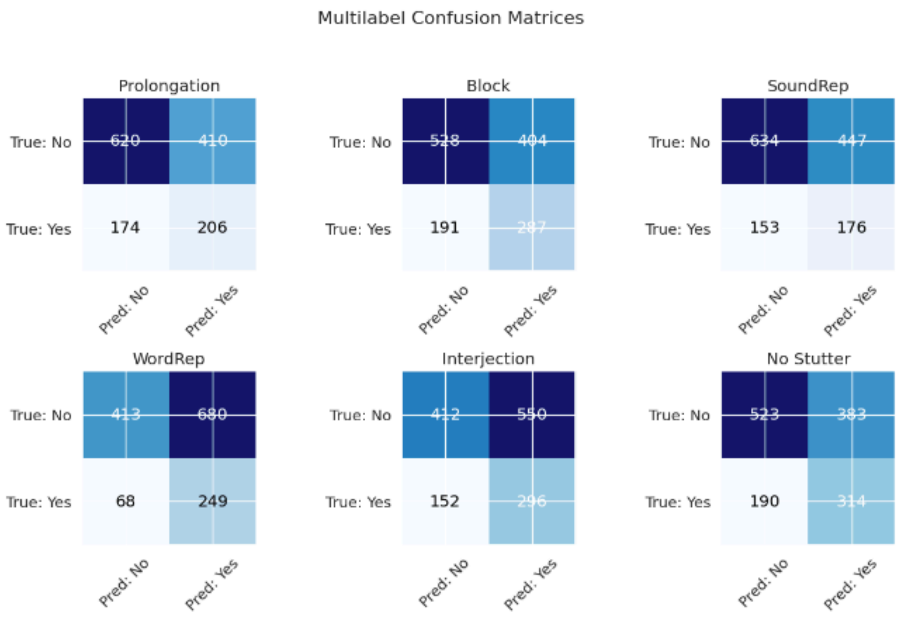
4. Syllable-Level Feature Pipeline for Stutter Detection

```
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Features: 20
(6786, 6)
Class distribution in training set: [1834 2304 1585 1535 2157 2424]
Class distribution in validation set: [324 406 279 269 382 428]
Class distribution in testing set: [380 478 329 317 448 504]
```

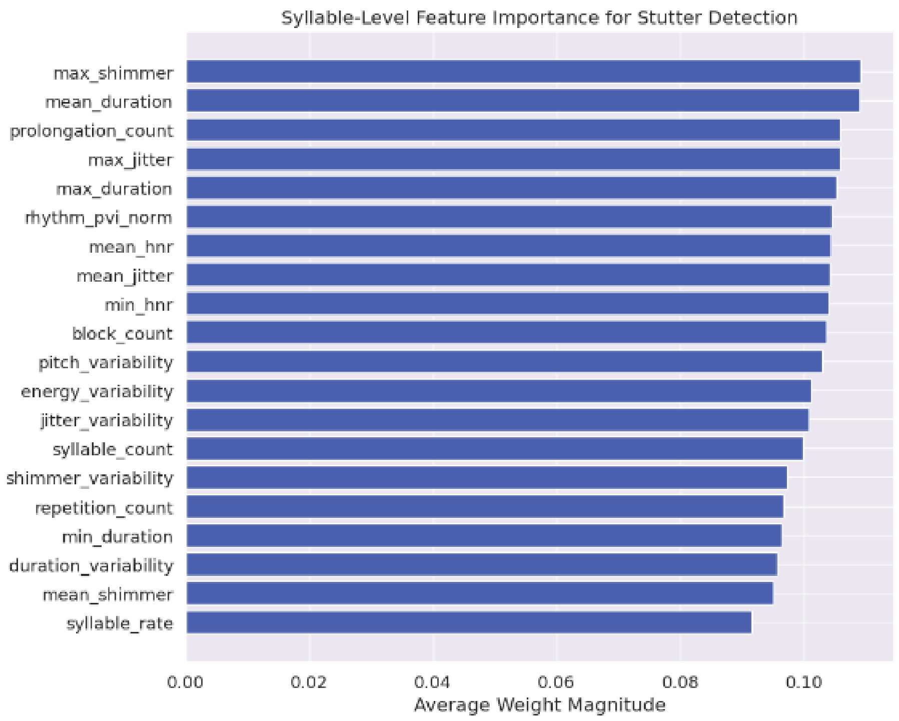


```
Layer (type)                          Output Shape           Param #
dense_12 (Dense)                      (None, 128)              2,688
batch_normalization_9                 (None, 128)                512
(BatchNormalization)
dropout_9 (Dropout)                   (None, 128)                  0
dense_13 (Dense)                      (None, 64)               8,256
batch_normalization_10                (None, 64)                 256
(BatchNormalization)
dropout_10 (Dropout)                  (None, 64)                   0
dense_14 (Dense)                      (None, 32)               2,080
batch_normalization_11                (None, 32)                 128
(BatchNormalization)
dropout_11 (Dropout)                  (None, 32)                   0
dense_15 (Dense)                      (None, 6)                  198
```

```
Total params: 14,118 (55.15 KB)
Trainable params: 13,670 (53.40 KB)
Non-trainable params: 448 (1.75 KB)
```

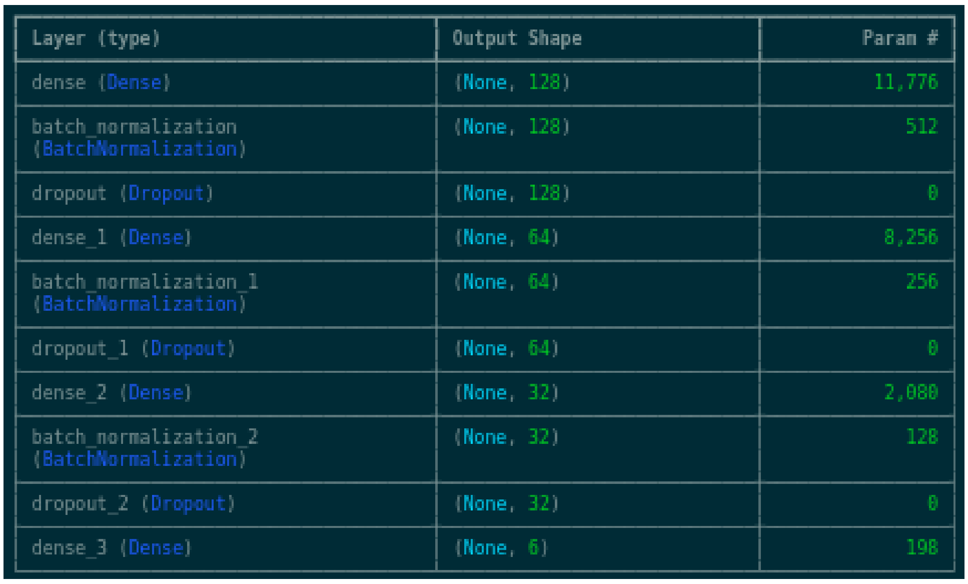


```
Classification Report:
              precision    recall  f1-score   support

Prolongation       0.33      0.54      0.41       380
       Block       0.42      0.60      0.49       478
    SoundRep       0.28      0.53      0.37       329
     WordRep       0.27      0.79      0.40       317
Interjection       0.35      0.66      0.46       448
  No Stutter       0.45      0.62      0.52       504

   micro avg       0.35      0.62      0.45      2456
   macro avg       0.35      0.62      0.44      2456
weighted avg       0.36      0.62      0.45      2456
 samples avg       0.37      0.62      0.44      2456
```



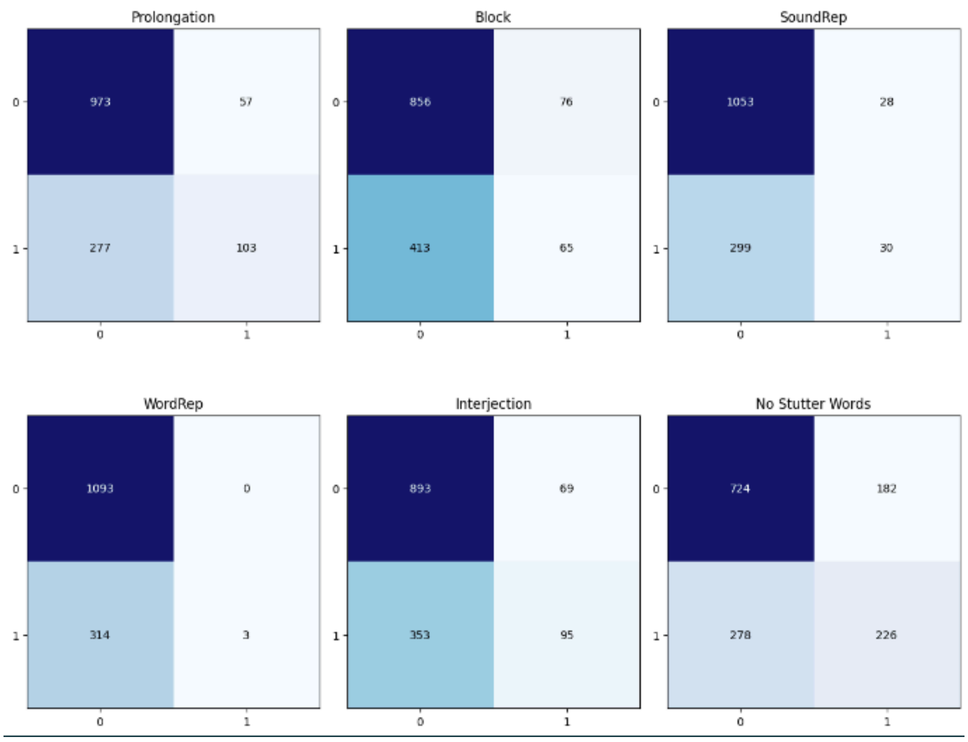
5. MFCC + Prosodic Features at Utterance-Level Pipeline

```
Training set: 6786 samples
Validation set: 1198 samples
Testing set: 1410 samples
Feature dimension: 91
Class distribution in training set: [1834. 2304. 1585. 1535. 2157. 2424.]
```

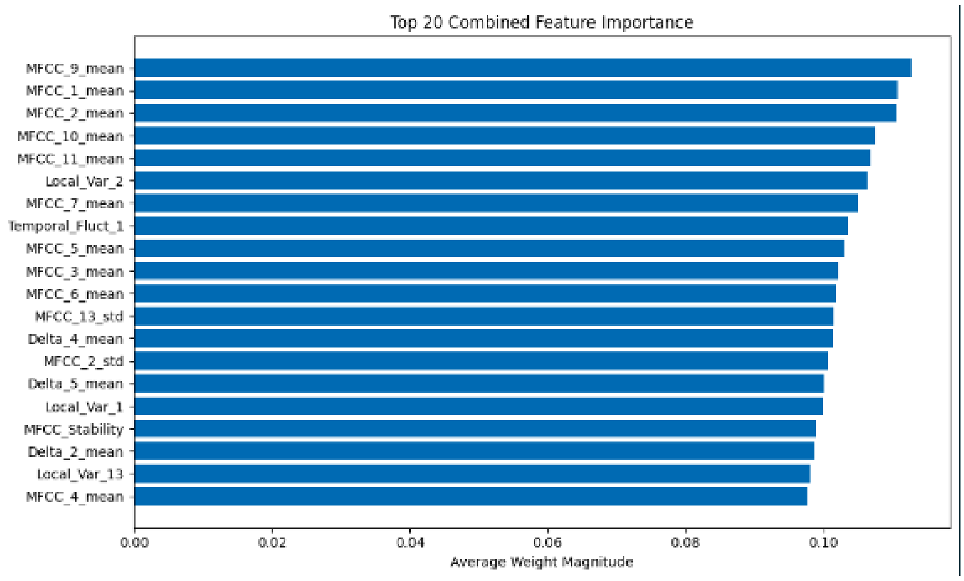


Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	11,776
batch_normalization (BatchNormalization)	(None, 128)	512
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
batch_normalization_1 (BatchNormalization)	(None, 64)	256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2,080
batch_normalization_2 (BatchNormalization)	(None, 32)	128
dropout_2 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 6)	198

```
Total params: 23,206 (90.65 KB)
Trainable params: 22,758 (88.90 KB)
Non-trainable params: 448 (1.75 KB)
```



```
Classification Report:
              precision    recall  f1-score   support

  Prolongation       0.64      0.27      0.38       380
         Block       0.46      0.14      0.21       478
     SoundRep       0.52      0.09      0.16       329
      WordRep       1.00      0.01      0.02       317
  Interjection       0.58      0.21      0.31       448
No Stutter Words       0.55      0.45      0.50       504

     micro avg       0.56      0.21      0.31      2456
     macro avg       0.63      0.19      0.26      2456
  weighted avg       0.61      0.21      0.28      2456
   samples avg       0.30      0.26      0.27      2456
```



Model Performance Comparison: Analysis

1. Overall Accuracy and AUC

- **MFCC + Prosodic Model** achieves the highest accuracy (**37.45%**) and the highest AUC (**0.6905**), outperforming all other models. This suggests that combining spectral (MFCC) and prosodic features provides a more comprehensive representation of stuttered speech, capturing both the fine-grained spectral patterns and the broader rhythmic/prosodic cues.
- **MFCC Model** alone is the next best in both accuracy (**35.11%**) and AUC (**0.6709**), indicating that spectral features are individually strong predictors, but still benefit from the addition of prosodic information.
- **Prosodic Model** (accuracy **34.82%**, AUC **0.6291**) performs slightly worse than MFCCs alone, but offers a good balance between interpretability and performance. Prosodic features are known to be robust to channel variations and are directly linked to speech rhythm and fluency.
- **Word-Level** and **Syllable-Level Models** show the lowest accuracy and AUC, suggesting that, in this implementation, local word/syllable features may not generalize as well for multilabel stutter detection as global utterance-level features.

2. Precision and Recall Trends

- **MFCC + Prosodic Model** achieves the highest precision (**0.5589**), meaning that when it predicts a stutter label, it is more likely to be correct compared to other models. However, its recall is the lowest (**0.2125**), indicating it misses many true stutter events. This is a classic precision-recall trade-off, often seen when a model is conservative in making positive predictions.
- **Word-Level** and **Syllable-Level Models** have higher recall (**0.5423**, **0.6221**) but lower precision (**0.3535**, **0.3471**), meaning they are more likely to identify stutter events but also produce more false positives.
- **MFCC** and **Prosodic Models** show a more balanced precision-recall profile, with recall around **0.56** and precision around **0.38–0.41**.

3. Model Architecture and Size

- The **Word-Level Model** uses the most complex architecture (128-64-32-16) and is the largest in size (**97.12 KB**), yet does not outperform simpler models. This suggests that increased complexity does not necessarily translate to better performance, possibly due to overfitting, insufficient data, or the features not being sufficiently informative at the word level for multilabel classification.
- The **Syllable-Level Model** is the smallest (**17.25 KB**) and also the least accurate, indicating that extreme model simplification can hurt performance, especially if the feature set is not sufficiently rich.

4. Why is Overall Accuracy Low?

- **Multilabel Complexity:** In multilabel stutter detection, each utterance can have multiple stutter types present or none at all. This increases the difficulty compared to binary or multiclass tasks, as the model must learn to recognize overlapping and co-occurring patterns.
- **Class Imbalance:** Stuttering events are often less frequent than fluent speech or may be unevenly distributed among classes, making it harder for the model to learn minority labels.
- **Feature Limitations:** While MFCC and prosodic features are informative, they may not capture all the nuanced patterns needed for robust multilabel discrimination, especially for subtle or overlapping stutter types.
- **Label Noise and Annotation Ambiguity:** Human labeling of stutter types can be subjective, introducing noise that makes high accuracy challenging.

5. Key Observations and Recommendations

- **Best Model:** The **MFCC + Prosodic Model** is the best overall, but its low recall suggests it is missing many true positives. It may be suitable when false positives are costly, but less so when recall is critical.
- **Prosodic Features:** These provide a good trade-off between interpretability and performance, and are less sensitive to channel effects.

- **Word/Syllable Models:** Their lower precision suggests that local features alone are insufficient; they may benefit from richer context or improved aggregation strategies.
- **General Low Accuracy:** Reflects the intrinsic difficulty of multilabel stutter detection, the need for more discriminative features, and possibly the need for more advanced model architectures (e.g., attention, sequence models).

In summary:

Combining MFCC and prosodic features yields the best overall performance, but all models face challenges due to the complexity of multilabel stutter detection. The main bottlenecks are class imbalance, feature limitations, and the inherent ambiguity in stutter labeling. Further improvements could be achieved by exploring advanced architectures, richer feature sets, and improved data balancing or augmentation strategies.