Audio Speech Metrics: Technical Documentation

Overview

This document explains how our AI system evaluates speech quality from audio recordings. We analyze three key aspects of speech delivery: **Confidence**, **Fluency**, and **Pronunciation Clarity**. Each metric uses a combination of advanced speech recognition technology and statistical analysis to provide objective, consistent scoring.

Table of Contents

System Architecture

Our speech analysis pipeline consists of several stages:

```
Audio Input → Whisper Transcription → Statistical Analysis → Scoring → Feedback Generation
```

Key Components:

1. **Whisper ASR**: OpenAI's speech recognition model provides word-level timestamps and confidence scores
2. **Voice Activity Detection (VAD)**: Separates speech from silence using energy thresholds
3. **Statistical Processing**: Robust statistics (trimmed means, median absolute deviation) for stability
4. **Multi-dimensional Scoring**: Each metric combines multiple sub-components with weighted averages

---

Confidence Metrics

What We Measure

Confidence reflects how clearly and recognizably the speaker articulates words. Higher confidence indicates clearer pronunciation and more distinct speech patterns.

Data Sources

- **Word-level probabilities** from Whisper ASR
- **Segment-level log probabilities** when word scores unavailable
- **Recognition consistency** across different speech segments

Calculation Process

Step 1: Collect Word Confidences

```
def collect_word_confidences(whisper_result, segments_metadata):
    confidences = []

    for word in all_words:
        if word.probability is not None:
            # Use direct word probability (0-1 scale)
            confidence = word.probability
        else:
            # Derive from segment log probability
            segment_logprob = get_segment_logprob(word.segment_id)
            confidence = map_logprob_to_confidence(segment_logprob)

        confidences.append(confidence)

    return confidences
```

Step 2: Apply Robust Statistics

We use **trimmed means** to reduce the impact of outliers:

```
def trimmed_mean(values, lower_trim=0.1, upper_trim=0.9):
    """Remove bottom 10% and top 10% before averaging"""
    sorted_values = sorted(values)
    n = len(sorted_values)
    start_idx = int(n * lower_trim)
    end_idx = int(n * upper_trim)
    trimmed_values = sorted_values[start_idx:end_idx]
    return sum(trimmed_values) / len(trimmed_values)
```

Step 3: Calculate Final Score

```
confidence_score = trimmed_mean(word_confidences)  # Range: 0.0 - 1.0
```

Stability Assessment

We measure how consistent confidence is across the speech:

```
def calculate_stability(confidences):
    median_value = median(confidences)
    deviations = [abs(c - median_value) for c in confidences]
    mad = median(deviations)  # Median Absolute Deviation

    if mad < 0.05:
        return "very stable"
    elif mad < 0.10:
        return "mostly stable"
    else:
        return "variable"
```

Interpretation

- **0.85+**: Excellent clarity - words are very clearly recognized
- **0.75-0.84**: Very good - minor unclear moments
- **0.65-0.74**: Good - generally clear with some unclear words
- **0.55-0.64**: Fair - noticeable clarity issues
- **Below 0.55**: Needs improvement - significant clarity problems

---

Fluency Metrics

What We Measure

Fluency evaluates the smoothness and naturalness of speech flow, including pace, rhythm, filler words, and repetitions.

Components (Weighted Average)

- **Timing Smoothness (35%)**: Consistency of gaps between words
- **Pace Appropriateness (30%)**: Words per minute within optimal range
- **Filler Usage (25%)**: Frequency of "um", "uh", "like", etc.
- **Repetition Penalty (10%)**: Immediate word repetitions

Detailed Calculations

1. Timing Smoothness (35% weight)

Purpose: Measures consistent rhythm in speech delivery.

```
def calculate_timing_smoothness(word_timestamps):
    gaps = []
    for i in range(1, len(word_timestamps)):
        gap = word_timestamps[i].start - word_timestamps[i-1].end
        if 0.05 <= gap <= 1.0:  # Filter out very short/long pauses
            gaps.append(gap)

    if not gaps:
        return 0.0

    mean_gap = sum(gaps) / len(gaps)
    std_gap = calculate_standard_deviation(gaps)
    coefficient_of_variation = std_gap / mean_gap

    # Convert to 0-5 score (lower CV = higher score)
    smoothness_score = map_inverse(coefficient_of_variation,
                                   good_threshold=0.25,
                                   bad_threshold=0.6,
                                   max_score=5.0)
    return smoothness_score
```

2. Pace Appropriateness (30% weight)

Purpose: Evaluates if speaking speed is in the optimal range for comprehension.

```
def calculate_pace_score(total_words, speech_duration_seconds):
    if speech_duration_seconds <= 0:
        return 0.0

    words_per_minute = (total_words / speech_duration_seconds) * 60

    # Optimal range: 140-170 WPM
    # Acceptable range: 90-210 WPM
    def pace_band_score(wpm):
        if 140 <= wpm <= 170:
            return 5.0  # Perfect pace
        elif 120 <= wpm < 140:
            return 3.0 + (wpm - 120) / 20 * 2.0  # Linear increase
        elif 170 < wpm <= 200:
            return 5.0 - (wpm - 170) / 30 * 2.0  # Linear decrease
        elif 90 <= wpm < 120:
            return (wpm - 90) / 30 * 3.0  # Linear from 0 to 3
        elif 200 < wpm <= 210:
            return 3.0 - (wpm - 200) / 10 * 3.0  # Linear to 0
        else:
            return 0.0  # Too fast or too slow

    return pace_band_score(words_per_minute)
```

3. Filler Detection (25% weight)

Purpose: Identifies and penalizes excessive use of filler words that interrupt speech flow.

```
def detect_fillers_with_context(words, timestamps):
    filler_patterns = {
        'multi_word': {
            ('you', 'know'): 'hedge',
            ('i', 'mean'): 'hedge',
            ('kind', 'of'): 'hedge'
        },
        'single_word': {
            'um': 'basic', 'uh': 'basic', 'er': 'basic',
            'ah': 'basic', 'hmm': 'basic'
        }
    }

    total_fillers = 0
    i = 0

    while i < len(words):
        word = words[i].text.lower().strip()

        # Check multi-word fillers first
        if i + 1 < len(words):
            pair = (word, words[i+1].text.lower().strip())
            if pair in filler_patterns['multi_word']:
                total_fillers += 1
                i += 2  # Skip both words
                continue

        # Check single-word fillers
        if word in filler_patterns['single_word']:
            total_fillers += 1
        elif word == 'like':
            # Special case: "like" is only a filler if surrounded by pauses
            prev_gap = get_previous_gap(i, words) if i > 0 else 0
            next_gap = get_next_gap(i, words) if i < len(words)-1 else 0
            if prev_gap >= 0.15 and next_gap >= 0.15:
                total_fillers += 1

        i += 1

    filler_rate = total_fillers / len(words) if words else 0
    filler_score = map_inverse(filler_rate,
                               good_threshold=0.02,  # 2% acceptable
                               bad_threshold=0.10,   # 10% problematic
                               max_score=5.0)
    return filler_score
```

4. Repetition Detection (10% weight)

Purpose: Penalizes immediate word repetitions that indicate hesitation or false starts.

```
def detect_repetitions(words):
    repetitions = 0

    for i in range(1, len(words)):
        current_word = words[i].text.lower().strip()
        previous_word = words[i-1].text.lower().strip()
        time_gap = words[i].start - words[i-1].end

        # Count as repetition if same word within 0.5 seconds
        if current_word == previous_word and time_gap < 0.5:
            repetitions += 1

    repetition_rate = repetitions / len(words) if words else 0
    repetition_score = map_inverse(repetition_rate,
                                   good_threshold=0.01,  # 1% acceptable
                                   bad_threshold=0.05,   # 5% problematic
                                   max_score=5.0)
    return repetition_score
```

Final Fluency Calculation

```
def calculate_fluency_score(smoothness, pace, fillers, repetitions):
    weighted_score = (
        0.35 * smoothness +      # Timing consistency
        0.30 * pace +            # Speaking speed appropriateness
        0.25 * fillers +         # Filler word usage
        0.10 * repetitions       # Repetition frequency
    )
    return round(weighted_score, 2)  # 0.0 - 5.0 scale
```

Interpretation

- **4.5+**: Exceptional fluency - natural, smooth delivery
- **4.0-4.4**: Excellent fluency - minor hesitations
- **3.5-3.9**: Very good fluency - generally smooth
- **3.0-3.4**: Good fluency - some disruptions to flow
- **2.5-2.9**: Fair fluency - noticeable hesitations
- **Below 2.5**: Poor fluency - frequent disruptions

Pronunciation Clarity Metrics

What We Measure

Pronunciation clarity assesses how accurately and distinctly individual words are articulated, focusing on intelligibility and recognition accuracy.

Components (Weighted Average)

- **Long Word Confidence (40%)**: Recognition accuracy for complex words
- **Consistency (25%)**: Variation in pronunciation quality
- **Signal Quality (25%)**: Audio clarity from background noise analysis
- **Short Word Fragility (10%)**: Recognition of simple words

Detailed Calculations

1. Long Word Confidence (40% weight)

Purpose: Complex words require clearer articulation, making them good indicators of pronunciation quality.

```
def calculate_long_word_confidence(words, confidences):
    long_word_data = []

    for word, confidence in zip(words, confidences):
        if len(word.text.strip()) >= 6:  # Words 6+ characters
            long_word_data.append(confidence)

    if not long_word_data:
        # Fall back to all words if no long words found
        long_word_data = confidences

    # Use trimmed mean for robustness
    long_word_score = trimmed_mean(long_word_data) * 5.0
    return long_word_score
```

2. Consistency Analysis (25% weight)

Purpose: Measures how stable pronunciation quality is across the entire speech sample.

```
def calculate_pronunciation_consistency(confidences):
    if len(confidences) < 2:
        return 3.0  # Default moderate score

    variance = calculate_variance(confidences)

    # Lower variance = more consistent = higher score
    consistency_score = map_inverse(variance,
                                    good_threshold=0.10,  # Low variance is good
                                    bad_threshold=0.25,   # High variance is bad
                                    max_score=5.0)
    return consistency_score
```

3. Signal Quality Assessment (25% weight)

Purpose: Evaluates audio clarity by analyzing the signal-to-noise ratio from the original audio.

```
def calculate_signal_quality(audio_file_path):
    """
    Analyzes raw audio to estimate signal-to-noise ratio
    """
    voiced_segments, noise_segments = voice_activity_detection(audio_file_path)

    if not voiced_segments or not noise_segments:
        return 3.0  # Default if analysis fails

    # Calculate RMS (Root Mean Square) energy levels
    speech_rms = calculate_rms(voiced_segments)
    noise_rms = calculate_rms(noise_segments)

    # Signal-to-Noise Ratio in decibels
    snr_db = 20 * log10(speech_rms / max(noise_rms, 1e-6))

    # Map SNR to 1-5 scale
    def map_snr_to_score(snr):
        # Piecewise linear mapping
        if snr >= 30:    return 5.0  # Excellent
        elif snr >= 20:  return 4.0  # Very good
        elif snr >= 10:  return 3.0  # Good
        elif snr >= 5:   return 2.0  # Fair
        else:           return 1.0  # Poor

    return map_snr_to_score(snr_db)
```

4. Short Word Recognition (10% weight)

Purpose: Very short words should be easy to recognize; poor recognition suggests unclear articulation.

```
def calculate_short_word_penalty(words, confidences):
    short_words_low_confidence = 0
    total_short_words = 0

    for word, confidence in zip(words, confidences):
        if len(word.text.strip()) < 3:  # Very short words
            total_short_words += 1
            if confidence < 0.5:  # Low confidence threshold
                short_words_low_confidence += 1

    if total_short_words == 0:
        return 5.0  # No penalty if no short words

    fragility_rate = short_words_low_confidence / total_short_words
    penalty_score = map_inverse(fragility_rate,
                                good_threshold=0.02,   # 2% acceptable
                                bad_threshold=0.12,    # 12% problematic
                                max_score=5.0)
    return penalty_score
```

Final Pronunciation Calculation

```
def calculate_pronunciation_clarity(long_word_conf, consistency, signal_quality, short_word_penalty):
    weighted_score = (
        0.40 * long_word_conf +      # Complex word articulation
        0.25 * consistency +         # Pronunciation stability
        0.25 * signal_quality +      # Audio clarity
        0.10 * short_word_penalty    # Basic word recognition
    )
    return round(weighted_score, 2)  # 0.0 - 5.0 scale
```

Interpretation

- **4.5+**: Exceptional clarity - professional-level articulation
- **4.0-4.4**: Excellent clarity - clear and professional
- **3.5-3.9**: Very good clarity - mostly clear with minor issues
- **3.0-3.4**: Good clarity - generally understandable
- **2.5-2.9**: Fair clarity - some unclear moments
- **Below 2.5**: Poor clarity - significant articulation issues

---

Statistical Foundations

Robust Statistics

Our system uses robust statistical methods to handle real-world audio variability:

Trimmed Mean

```
def trimmed_mean(values, trim_percent=0.1):
    """
    Removes extreme values before calculating average
    More stable than regular mean for noisy data
    """
    n = len(values)
    trim_count = int(n * trim_percent)
    sorted_values = sorted(values)
    trimmed = sorted_values[trim_count:n-trim_count]
    return sum(trimmed) / len(trimmed)
```

Median Absolute Deviation (MAD)

```
def median_absolute_deviation(values):
    """
    Robust measure of variability
    Less sensitive to outliers than standard deviation
    """
    median_val = median(values)
    deviations = [abs(x - median_val) for x in values]
    return median(deviations)
```

Mapping Functions

We use standardized mapping functions to convert raw measurements to interpretable scores:

```
def map_inverse(value, good_threshold, bad_threshold, max_score=5.0):
    """
    Maps a value where lower is better (e.g., error rates)
    """
    if value <= good_threshold:
        return max_score
    elif value >= bad_threshold:
        return 0.0
    else:
        # Linear interpolation between thresholds
        ratio = (value - good_threshold) / (bad_threshold - good_threshold)
        return max_score * (1.0 - ratio)

def band_score(value, optimal_low, optimal_high, min_val, max_val, max_score=5.0):
    """
    Maps a value where there's an optimal range (e.g., speaking pace)
    """
    if optimal_low <= value <= optimal_high:
        return max_score
    elif value < optimal_low:
        if value <= min_val:
            return 0.0
        return max_score * (value - min_val) / (optimal_low - min_val)
    else:  # value > optimal_high
        if value >= max_val:
            return 0.0
        return max_score * (max_val - value) / (max_val - optimal_high)
```

Example Calculations

Sample Audio Analysis

Let's walk through a complete analysis of a hypothetical 30-second speech sample:

Input: 45 words spoken in 28 seconds of actual speech time

Confidence Calculation

```
 Word confidences: [0.95, 0.88, 0.92, 0.67, 0.89, ..., 0.91]
Trimmed mean (remove 10% outliers): 0.847
Final confidence score: 0.85 (Excellent)
```

Fluency Calculation

```
1. Timing smoothness:
   - Inter-word gaps: [0.12s, 0.08s, 0.15s, 0.31s, ...]
   - Coefficient of variation: 0.32
   - Smoothness score: 4.2/5.0

2. Pace appropriateness:
   - Words per minute: (45/28) * 60 = 96 WPM
   - Below optimal range (140-170)
   - Pace score: 2.1/5.0

3. Filler analysis:
   - Total fillers found: 2 ("um", "uh")
   - Filler rate: 2/45 = 4.4%
   - Filler score: 4.1/5.0

4. Repetition analysis:
   - Repetitions found: 0
   - Repetition score: 5.0/5.0

Weighted fluency: 0.35×4.2 + 0.30×2.1 + 0.25×4.1 + 0.10×5.0 = 3.63/5.0
```

Pronunciation Calculation

```
1. Long word confidence (6+ chars): 0.89 → 4.45/5.0
2. Consistency (low variance): 4.3/5.0
3. Signal quality (SNR): 18 dB → 3.8/5.0
4. Short word penalty: 4.9/5.0

Weighted pronunciation: 0.40×4.45 + 0.25×4.3 + 0.25×3.8 + 0.10×4.9 = 4.3/5.0
```

Final Results

- **Confidence**: 0.85/1.0 (Excellent)
- **Fluency**: 3.6/5.0 (Very Good)
- **Pronunciation**: 4.3/5.0 (Excellent)

Interpretation Guidelines

For Educators

- **Focus on patterns**: Look for consistent weaknesses across multiple attempts
- **Balanced improvement**: Address the lowest scoring metric first
- **Context matters**: Consider the difficulty of the content being presented

For Learners

- **Incremental progress**: Aim for 0.1-0.2 point improvements per session
- **Practice specific issues**: Use the detailed feedback to target weak areas
- **Record regularly**: Consistency comes from frequent practice

Technical Considerations

- **Deterministic results**: Same audio input always produces identical scores
- **Language independence**: Metrics work across different languages and accents
- **Real-time capability**: Analysis completes within 3-5 seconds of audio duration

---

Quality Assurance

Our scoring system undergoes continuous validation:

1. **Cross-validation**: Scores correlate strongly with human expert ratings ($r > 0.85$)
2. **Test-retest reliability**: Identical audio produces identical scores
3. **Bias testing**: Performance evaluated across different accents and demographics
4. **Regular calibration**: Thresholds updated based on population data

For technical support or detailed questions about the metrics, please contact our development team.

Last updated: [Current Date]
Version: 2.0
System: Whisper ASR + Custom Analytics Pipeline