Song Search using CLAP

HW 25

December 2, 2025

CLAP Overview

CLAP (Contrastive Language-Audio Pretraining) - Audio-text embedding framework

- **Architecture:** HTSAT audio encoder + BERT text encoder
- **Output:** 512-dimensional aligned embeddings
- **Capability:** Zero-shot audio-text semantic search
- **Selected Model:** `music_audioset_epoch_15_esc_90.14.pt`
  - 71% GTZAN accuracy (best for music)
  - 2.35 GB, trained on ~4M samples

Workflow:

1. Load audio at 48kHz
2. Generate audio & text embeddings via CLAP
3. Compute cosine similarity (dot product)
4. Rank by similarity score (-1 to +1)

Implementation

Two Modes:

- **Single Analysis:** `single_analysis.py` - One song vs one query
- **Batch Analysis:** `multiple_analysis.py` - All songs vs all queries

Architecture:

- `clap_analysis.py` - Core library (shared)
- Automatic model download & caching
- Parallel processing across CPU cores
- CSV output with detailed metrics

Similarity Thresholds:

- $> 0.3 =$ HIGH (strong match)
- $> 0.15 =$ MODERATE (partial match)
- $\leq 0.15 =$ LOW (no match)

Overlapping Segments & Optimization

Problem: CLAP processes max 10s per inference

Solution: Overlapping segment analysis

- 10s segments with 50% overlap (5s hop)
- Ensures full song coverage
- Average score across all segments

Performance Optimization:

```
# BEFORE: 17x redundant work
for song in songs:
    for query in queries:
        analyze(song, query)  # 270s per song

# AFTER: Analyze once, compare many
for song in songs:
    embeddings = get_embeddings(song)  # 18s
    for query in queries:
        compare(embeddings, query)  # 0.15s each
```

Result: ˜15-17x speedup (270s $\rightarrow$ 20s per song)

Deterministic Results

Goal: Reproducible scores across runs

Implementation:

```
# Random seeds
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)

# PyTorch determinism
torch.backends.cudnn.deterministic = True
torch.use_deterministic_algorithms(True)

# Environment variables
os.environ['PYTHONHASHSEED'] = '42'
os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':4096:8'
```

Validation: Exact same scores across multiple runs (verified)

Query Wording Sensitivity

Finding: Exact wording dramatically affects results

Example - "vocalist" vs "voice":

Query	Score	Label
female vocalist	0.328	HIGH
female voice	0.062	LOW
Difference	-0.267	**5× lower!**

Best Practices:

- Use music industry terms: "vocalist" not "voice"
- Keep queries simple: avoid complex compounds
- Test synonyms to find best wording
- Genre terms work well: "rock music", "classical music"

Song-to-Song Similarity

Feature: Compare songs to each other using audio embeddings

How It Works:

```
# During analysis: store average embedding per song
avg_embedding = np.mean(segment_embeddings, axis=0)
avg_embedding /= np.linalg.norm(avg_embedding)  # normalize
song_embeddings[filename] = avg_embedding

# After all songs: compute pairwise similarities
for song1, song2 in all_pairs:
    similarity = np.dot(song_embeddings[song1],
                        song_embeddings[song2])
```

Key Points:

- Same metric as text queries (cosine similarity)
- No re-analysis needed (uses cached embeddings)
- Negligible computation (~0.001s for 10 songs)

AI Support in Development

GitHub Copilot (Claude Sonnet 4.5) contributions:

- **Determinism fix:** Comprehensive seed initialization (1 iteration)
- **Performance:** 15x speedup via embedding caching (2 iterations)
- **Architecture:** Clean library/CLI separation (1 iteration)
- **Query analysis:** Discovered "vocalist" vs "voice" pattern
- **Documentation:** Created all technical docs

Success Pattern:

- Clear, specific requests $\rightarrow$ Single-iteration solutions
- Performance challenges $\rightarrow$ Iterative refinement
- Example-based communication $\rightarrow$ Faster results

Key Takeaway: Most requests resolved in 1 iteration when clearly specified