1. **Fast Candidate Retrieval:**
   Applied **Chromaprint and DejaVu-based audio fingerprinting** to efficiently compare the query song against a large music corpus and retrieve the **Top-K most similar candidate tracks**, enabling scalable and noise-robust pre-filtering.

2. **Segment-Level Analysis:**
   Partitioned both query and candidate songs into **overlapping temporal segments** to support **local similarity analysis** and enable detection of **partial reuse or containment** rather than relying on global song similarity.

3. **Multimodal Feature Extraction:**
   Extracted complementary **melodic (pitch movement), rhythmic (inter-onset timing), and harmonic (chroma pitch-class profiles)** features from each segment to capture multiple musical dimensions.

4. **Multimodal Bipartite Graph Matching:**
   Constructed an **implicit bipartite graph** between query and candidate segments, computed **fused multimodal similarity scores** using **DTW and cosine similarity**, and applied the **Hungarian algorithm** to obtain optimal segment-to-segment matches.

5. **Containment Detection & Report Generation:**
   Aggregated temporally aligned segment matches to detect **containment similarity**, localize reused regions, and generate a **structured report** detailing reuse duration, similarity strength, matched segment count, and query–candidate time intervals.

EXPLANATION — WITH EXAMPLE

Is part of Song A copied inside Song B?

STEP 0: INPUT SONGS

Example:

- **Song A (Query)** – 120 seconds
- **Song B (Candidate)** – 150 seconds

STEP 1: CUT BOTH SONGS INTO SMALL PIECES

We cut both songs into **3-second overlapping pieces**.

Example (Song A)

```
A0: 0 – 3 sec
A1: 1.5 – 4.5 sec
A2: 3 – 6 sec
A3: 4.5 – 7.5 sec
...
```

Same for Song B:

```
B0, B1, B2, B3, ...
```

Each piece = **one segment**

STEP 2: EXTRACT FEATURES FROM EACH PIECE

From **each 3-second piece**, we extract **3 numeric descriptions**.

2.1 MELODY (tune movement)

What we do

- Track pitch over time
- Convert pitch to numbers
- Store **pitch differences**

Example

Actual singing:

```
low → higher → much higher → lower
```

Stored as numbers:

```
[+2.1, +4.7, -6.8]
```

So melody = **list of numbers**

Why?

- Same tune in different key still matches

Step 1: Track pitch over time

Imagine we sample the singing every moment:

```
Time:    t1   t2   t3   t4
Pitch:  200  250  400  300   (Hz)
```

This is what **pitch tracking** does.

Step 2: Convert pitch to numbers (still simple)

Now we **don't care about exact pitch**.

We care about **how it moves**.

So we calculate:

```
change1 = 250 - 200 = +50
change2 = 400 - 250 = +150
change3 = 300 - 400 = -100
```

Now we store:

```
[+50, +150, -100]
```

That is **melody** in your system.

Step 3: Why do we store differences, not pitch itself?

This is VERY important.

Example 1: Original song

```
200 → 250 → 400 → 300
Differences: +50, +150, -100
```

Example 2: Same tune, different key (higher singer)

```
300 → 350 → 500 → 400
Differences: +50, +150, -100
```

Differences are identical

2.2 RHYTHM (beat timing)

What we do

- Detect beat times
- Compute gaps between beats
- Normalize them

Example

How much time between one beat and the next?

Beats at:

```
0.2s, 0.6s, 1.8s, 2.2s
```

Gaps:

```
[0.4, 1.2, 0.4]
```

Normalized rhythm:

```
[0.7, 2.1, 0.7]
```

Rhythm = **list of numbers**

Why?

- Tempo changes don't matter

Step 1: Detect beat times

The computer listens and finds **when strong beats happen**.

Example:

```
Beat 1 at 0.2 seconds
Beat 2 at 0.6 seconds
Beat 3 at 1.8 seconds
Beat 4 at 2.2 seconds
```

Think of someone tapping a table.

Step 2: Compute gaps between beats

Now we calculate:

> "How much time between one beat and the next?"

```
0.6 - 0.2 = 0.4 sec
1.8 - 0.6 = 1.2 sec
2.2 - 1.8 = 0.4 sec
```

So gaps become:

```
[0.4, 1.2, 0.4]
```

This already shows the rhythm:

```
short – long – short
```

Step 3: Normalize the gaps (THIS IS IMPORTANT)

Different songs have different speeds.

So instead of raw time, we compare **relative timing**.

Example

Average gap:

```
(0.4 + 1.2 + 0.4) / 3 ≈ 0.67
```

Now divide each gap by average:

```
0.4 / 0.67 ≈ 0.7
1.2 / 0.67 ≈ 2.1
0.4 / 0.67 ≈ 0.7
```

So rhythm becomes:

```
[0.7, 2.1, 0.7]
```

What does this REALLY mean?

It means:

```
short beat → long beat → short beat
```

Even if the song is:

- fast
- slow
- medium

This **pattern stays same**.

Why rhythm is stored as numbers?

Because the computer compares **patterns**, not tempo.

Example:

Song 1 (slow)

`0.6s → 1.8s → 0.6s`

Song 2 (fast)

`0.3s → 0.9s → 0.3s`

After normalization, both become:

`[0.7, 2.1, 0.7]`

So they match.

Why DTW is used here?

Because:

- Some beats may be stretched
- Some may be skipped
- Some may be repeated

DTW lets:

```
short, long, short
```

match:

```
short, short, long, short
```

2.3 HARMONY (background notes)

What we do

- Compute chroma features
- Get 12 values (one per musical note)

Example

```
[C, C#, D, D#, E, F, F#, G, G#, A, A#, B]
```

Stored as:

```
[0.3, 0.1, 0.5, ..., 0.2]
```

Harmony = **12 numbers**

Why?

- Represents chord feeling

STEP 3: NOW WE COMPARE SEGMENTS

Take **one piece from Song A** and **one piece from Song B**.

Example:

```
A2 ↔ B7
```

STEP 4: HOW EXACTLY DO WE COMPARE? (IMPORTANT)

4.1 Melody comparison → DTW on numbers

Input:

```
A2 melody = [2.1, 4.7, -6.8]
B7 melody = [2.0, 5.0, -6.5, -0.2]
```

DTW checks:

> Can these two number sequences be aligned even if one is longer?

If yes → high score

4.2 Rhythm comparison → DTW on numbers

Input:

```
A2 rhythm = [0.7, 2.1, 0.7]
B7 rhythm = [0.8, 2.0, 0.6]
```

DTW checks:

> Do beat patterns look similar?

4.3 Harmony comparison → cosine similarity

Input:

```
A2 harmony = [0.3, 0.1, 0.5, ...]
B7 harmony = [0.28, 0.12, 0.49, ...]
```

Cosine checks:

> Are these 12-number patterns pointing in same direction?

STEP 5: COMBINE THE THREE SCORES

We compute:

```
final_similarity =
  0.5 × melody_score
+ 0.3 × rhythm_score
+ 0.2 × harmony_score
```

This gives **ONE number** between 0 and 1.

STEP 6: BUILD THE BIPARTITE GRAPH (THIS IS THE KEY)

We repeat this comparison for **every A segment vs every B segment**.

We store results in a table:

```
       B0    B1    B2    B3
A0     0.1   0.2   0.8   0.1
A1     0.2   0.9   0.3   0.2
A2     0.7   0.3   0.1   0.2
```

This table **IS the bipartite graph**:

- Rows = Song A pieces
- Columns = Song B pieces
- Values = similarity (edge weight)

STEP 7: APPLY HUNGARIAN ALGORITHM

Problem:

> Choose best matches so one piece matches only one piece.

Hungarian algorithm chooses:

```
A0 → B2
A1 → B1
A2 → B0
```

This avoids:

- One segment matching many others
- Greedy mistakes

STEP 8: KEEP ONLY STRONG MATCHES

We keep only matches where:

```
similarity ≥ threshold
```

These become `"matches"` in your JSON.

STEP 9: DETECT CONTAINMENT

Now we check **time order**.

If matches:

- are consecutive in Song A
- and consecutive in Song B

Then:

Song A content exists inside Song B

We compute:

- start time
- end time
- duration
- average similarity

STEP 10: FINAL REPORT GENERATED

Example report:

```
Query time: 12s – 30s
Candidate time: 45s – 63s
Duration: 18s
Segments matched: 12
Avg similarity: 0.82
```

This is what you finally show.

Abstract

Music plagiarism detection is a challenging task due to variations in key, tempo, instrumentation, and partial reuse of musical content. Traditional audio fingerprinting techniques are effective for fast retrieval of similar recordings but lack the ability to localize and explain reused musical segments. In this work, we propose a **multimodal bipartite graph-based framework** for fine-grained music containment detection. The system operates in two stages:

an initial candidate filtering stage using audio fingerprinting to retrieve a small set of potentially similar songs, followed by a detailed segment-level analysis. Songs are segmented into overlapping temporal windows, and **melodic, rhythmic, and harmonic features** are extracted from each segment. A bipartite graph is constructed between query and candidate segments, where edge weights are computed using a fused multimodal similarity measure combining Dynamic Time Warping and cosine similarity. Optimal segment correspondences are obtained using the Hungarian algorithm, and temporally consistent matches are aggregated to detect and localize partial musical reuse. Experimental results demonstrate that the proposed approach effectively identifies containment similarity while providing interpretable evidence such as reuse duration, matched segments, and temporal alignment, making it suitable for explainable music plagiarism analysis.

---

1. Introduction

The rapid growth of digital music repositories has increased the need for reliable and scalable music plagiarism detection systems. Unlike exact duplication, music plagiarism often involves **partial reuse**, melodic transformation, tempo variation, or harmonic modification, making simple waveform or fingerprint-based matching insufficient. While audio fingerprinting methods such as Chromaprint and DejaVu enable efficient large-scale similarity search, they primarily focus on identifying

near-duplicate recordings and do not provide detailed explanations of similarity or localization of reused content.

Music plagiarism analysis requires not only determining whether two songs are similar, but also **identifying which parts are reused, for how long, and based on which musical attributes**. This necessitates a local and interpretable comparison strategy rather than a single global similarity score. Segment-level analysis has therefore emerged as a promising direction, as it allows detection of localized similarities that may be obscured in full-song comparisons.

In this paper, we present a **multimodal bipartite graph matching approach** for music containment detection. Following an initial candidate selection stage using audio fingerprinting, both query and candidate songs are divided into overlapping temporal segments. For each segment, three complementary musical modalities are extracted: **melody**, represented as pitch movement; **rhythm**, represented as normalized inter-onset intervals; and **harmony**, represented using chroma-based pitch class profiles. These modalities capture different aspects of musical similarity and provide robustness against common transformations such as key transposition and tempo changes.

A bipartite graph is implicitly constructed between query and candidate segments, with edge weights defined by a fused multimodal similarity score. To ensure structured and optimal matching, the Hungarian algorithm is employed to compute one-to-one segment correspondences. Finally, matched segments are temporally aggregated to detect containment similarity and generate an interpretable report describing the reused region. The proposed framework emphasizes

explainability, localization, and modularity, making it well-suited for practical music plagiarism detection scenarios.