MUSICAL SIGNATURE IDENTIFICATION AND PLAGIARISM DETECTION THROUGH FEATURE EXTRACTION AND ANALYSIS

BY

SEOW YI XUAN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

JUN 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to sincerely express my gratitude and appreciation to my supervisor, Prof. Ts. Liew Soung Yue, and my moderator, Ts. Wong Chee Siang, for granting me the opportunity to explore and contribute to the field study on musical signature identification and plagiarism detection. Their invaluable guidance and constructive advice have been instrumental in helping me overcome challenges and successfully complete this project. I am truly grateful for their continuous support and encouragement.

In addition, I would like to extend my heartfelt thanks to my family and friends for their love, support, and unwavering encouragement throughout the course of this journey.

ABSTRACT

Music information retrival (MIR) systems are introduced to counter musical borrowing and unintentional plagiarism in the present music industry. Music plagiarism detection is a complex task, as similarity may arise not only from melodic lines but also from rhythm, harmony, and timbre. This system introduces an approach that integrates multi-dimensional audio features with segment-based analysis to improve detection accuracy and interpretability. Extracted features include harmonic descriptors (CENS, CQT, tonnetz, harmonic n-grams), rhythmic descriptors (tempogram, onset autocorrelation), timbral descriptors (MFCC), and spectral texture measures (spectral contrast, centroid, rolloff, bandwidth, flatness). Database songs are segmented into overlapping windows of varying lengths, while query tracks are segmented consistently, allowing robust local-to-local comparisons. A weighted similarity model balances contributions from all features and normalizes results even when certain descriptors are unavailable. Potential plagiarism is flagged when segment similarities exceed adaptive thresholds, and is presented in a ranked list. To support expert evaluation, the system employs large language models (LLMs) to generate textual analyses of the top matches, highlighting how melodic, rhythmic, harmonic, and timbral evidence supports or weakens plagiarism claims. The aim is to provide music experts with clear, evidence-driven insights, enabling more efficient and transparent decision-making.

Area of Study : Musical Information Retrival (MIR), Computational Musicology

Keywords : Music plagiarism detection, audio similarity, feature extraction, rhythm analysis, harmony analysis, timbre analysis, spectral features, large language models

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

η	Positive constant

LIST OF ABBREVIATIONS

API	Application Programming Interface
MIR	Music Information Retrival
FFT	Fast Fourier transform
STMSP	Short-Time Mean-Square Power
CENS	Chroma Energy distribution Normalized Statistics
CLP	Chroma-Log-Pitch
M	Length
BPM	Beats Per Minute
CQT	Constant-Q Transform
MFCC	Mel-frequency cepstral coefficients
OS	Operating System
LLM	Large Language Model
SR	Sampling rate
JSON	JavaScript Object Notation

Chapter 1

Introduction

In this chapter, the background and motivation of the project, the contributions to the field, and the outline of the musical signature identification and plagiarism detector are presented.

1.1 Problem Statement and Motivation

The music industry faces significant challenges due to the lack of an efficient and accurate method for detecting plagiarism, particularly for record labels. Current practices often depend on manual analysis by music experts, a procedure that requires substantial time, is prone to subjectivity, and is labor-intensive. This subjectivity leads to differing interpretations of what constitutes plagiarism. The line between influence and plagiarism in music is often blurred. With one listener viewing a song as a blatant copy, while another may argue it as a homage or source of creative inspiration as artists frequently draw inspiration from one another, reinforcing the idea of music as a collaborative and ever-evolving art form. Additionally, cultural and historical contexts can further complicate matters, as similarities in music may arise from deep rooted cultural influences rather than intentional copying. The absence of standardized criteria for defining music plagiarism exacerbates this issue. Thus, the need for Music Information Retrival (MIR) systems is at its demanding. The complexity of musical elements adds another layer of difficulty in detecting plagiarism. Music comprises various components such as melody, harmony, rhythm, lyrics, and production techniques, all of which can be subtly altered to create distinct differences. Even slight changes in melody or rhythm can make a song feel unique, despite underlying similarities. This complexity demands sophisticated tools and deep musical expertise, as identifying plagiarism requires more than just recognizing surface-level resemblances. While current technologies can capture and encode the unique characteristics of audio files, the methods for comparing these audio signatures often lack the precision and sophistication necessary to discern subtle similarities or differences. This limitation is particularly problematic in the context of music plagiarism, where even minor overlaps in melodic or rhythmic elements can lead to disputes over intellectual property. Furthermore, copyright law does not explicitly regulate the criteria for determining music plagiarism, necessitating the involvement of music experts in court to assess the degree of similarity between songs. Accusations of plagiarism must be supported by detailed evidence that clearly demonstrates the extent of the similarities [1]. In summary, the primary challenges for creators and record labels are the high demands of plagiarism detection systems and the need for comprehensive analysis to protect intellectual property. The aim of this project is to propose a more advanced system for identifying musical signatures and detecting plagiarism in audio recordings. Addressing this longstanding issue in the music industry is complex, as it's nearly impossible to create a song without incorporating some elements that resemble existing works. However, recognizing and protecting intellectual property is essential. It's crucial to appreciate the unique value of creative works and ensure that they are safeguarded against unauthorized use. The blurry line between copying and inspiration also necessitates an objective mediator to make fair and rational comparisons. While musical experts will still be needed to make the final determination on whether songs are plagiarized, this system can provide the necessary technical support to assist in these decisions. Additionally, the system can assist artists in disputes by identifying earlier works that may have influenced both parties. When an artist is accused of copying another, the system can verify whether similar audio exists that was released prior to either artist's work. By cataloging these earlier songs alongside their release dates, the system can provide evidence that the accused artist may have unknowingly drawn from a common source. In such cases, this information could help mitigate the severity of penalties by demonstrating that the similarities were not the result of direct copying, but rather unintentional overlap with pre-existing works. In short, the system will be designed to continually learn and adapt by storing more fingerprints, thereby expanding its database and improving its accuracy over time. This will make it a valuable tool for artists, producers, and legal professionals in maintaining the integrity of musical works.

1.2 Objectives

The main objective of this project is to develop a sophisticated system capable of identifying musical signatures and detecting plagiarism in audio recordings. The system is intended not only to detect direct duplication but also to capture subtle similarities that may occur in harmony, melody, rhythm, or timbre, providing a fairer and more comprehensive framework for plagiarism evaluation. To achieve this overarching goal, several sub-objectives are defined, each addressing different aspects of accuracy, efficiency, and practical applicability in real-world music industry contexts.

A primary sub-objective is to improve the precision of similarity measurements between audio segments. Music plagiarism is rarely a matter of identical copying; it often involves transposition, tempo adjustments, or partial duplication of motifs. Therefore, the system is designed to be robust against tempo and pitch variations by incorporating beat-based segmentation and key-invariant similarity measures. By employing a combination of cosine similarity for spectral and timbral features, key-invariant cosine similarity for harmonic descriptors, and Jaccard similarity for motif and chord progressions, the system can identify overlaps even when they are disguised through key changes or tempo modifications. This refinement reduces both false positives, where unrelated songs are incorrectly flagged, and false negatives, where genuine cases of plagiarism might otherwise be overlooked.

Another important sub-objective is to enhance the efficiency of audio processing and comparison. Since music databases can grow to thousands of songs, the system must handle large volumes of audio data without sacrificing speed or accuracy. To meet this challenge, preprocessing pipelines are optimized with segment limits per song and overlap strategies that balance coverage and performance. Feature extraction is streamlined to store descriptors in compact numerical and symbolic formats, enabling faster retrieval and comparison. The comparison engine is further optimized to rank potential candidates using confidence scoring that incorporates average similarity, maximum similarity, and segment coverage. This ensures that results remain both accurate and scalable, making the system practical for large-scale use.

In addition, the project aims to improve the interpretability of results. Many plagiarism detection systems produce numerical outputs that may be difficult for non-technical users to interpret. To address this, the proposed system integrates a reporting mechanism that generates ranked results along with similarity statistics. It further enhances usability by incorporating textual analysis via an external API, which translates raw scores into meaningful narratives that highlight harmonic, melodic, or rhythmic overlaps. This feature provides music experts with contextual insights, helping them to make more informed decisions in plagiarism claims.

It is important to note that this project does not attempt to separate vocal and instrumental components from the audio signal. Instead, the analysis treats the audio as a whole, considering both vocals and instrumentation as part of the musical signature. This simplifies the processing pipeline while maintaining its ability to detect meaningful similarities across complete recordings. Although this choice narrows the system's scope, it ensures that the project remains focused on overall similarity detection, providing a robust and reliable framework without overcomplicating the implementation.

In summary, the objectives of this system are centered on improving precision, efficiency, and interpretability in music plagiarism detection. By combining robust feature-based analysis with efficient data handling and user-friendly reporting, the system offers a comprehensive solution for safeguarding the creative rights of artists. Its design not only addresses the immediate challenge of detecting plagiarism but also lays the groundwork for future developments, such as the integration of deep learning models, polyphonic analysis, and more sophisticated interfaces to further strengthen its role in the music industry.

1.3 Project Scope and Direction

This project aims to address the issue of music plagiarism by developing a system that can systematically analyze full-length audio recordings and compare them against a database of existing songs. The scope of the system extends from the very first stage of preprocessing raw audio files to the final step of generating plagiarism reports that include both quantitative scores and textual interpretations. Users are able to upload a database folder of reference tracks as well as a query audio file. The system first validates the audio format to ensure consistency, then processes the inputs through normalization, resampling, and beat tracking so that differences in recording quality, tempo, or loudness do not distort the results. Both query and database songs are segmented into beat windows of fixed lengths, with different overlap settings to maximize coverage while maintaining efficiency.

From these standardized segments, the system extracts a comprehensive set of features that capture multiple dimensions of music. These include chroma features and constant-Q transform representations for harmony, MFCCs for timbral characteristics, spectral descriptors such as centroid and contrast, rhythmic features like tempograms and onset autocorrelations, tonal descriptors through the tonnetz representation, and higher-level symbolic patterns such as motifs and chord progressions. Together, these descriptors form a detailed "musical fingerprint" of each audio segment, which is stored in a feature database for future comparisons.

When a query song is submitted, its features are compared against those of the database using different similarity measures tailored to the nature of each descriptor. Cosine similarity is applied to MFCCs and spectral vectors, while chroma vectors are evaluated using key-invariant cosine similarity to account for pitch shifts. Symbolic features such as motifs and harmonic progressions are assessed with Jaccard similarity to measure structural overlaps. The results of these comparisons are aggregated through a weighted scoring scheme that balances the importance of timbral, harmonic, rhythmic, and symbolic information. Segments that achieve similarity scores above a defined threshold are flagged as matches. At the song level, additional decision rules are applied such as requiring a minimum of three matching segments or coverage of at least 15% of the query to confirm plagiarism and avoid false positives.

Once the decision module identifies plagiarism candidates, the system ranks them based on average similarity, maximum similarity, and match ratio, producing a confidence score that reflects the likelihood of plagiarism. The user is then presented with a ranked list of songs most similar to the query, along with similarity metrics and supporting details. The evaluation of system correctness relies on a structured indexing approach established during the categorization of audio files. If the query and database entries share the same index, this indicates a ground truth plagiarism pair. To enhance interpretability, the system also compiles the top matches into a compact summary and sends a request to an external API for textual analysis. This step generates natural-language explanations that highlight harmonic, melodic, or rhythmic overlaps, ensuring that the output is not just numbers but also meaningful insights that music experts can use in their evaluation.

The direction of this project is to create a reliable and efficient tool that bridges computational analysis with practical usability. By combining feature-based similarity detection with interpretive reporting, the system supports music experts in making fairer and more informed judgments regarding plagiarism claims. Although this implementation focuses on full audio signals rather than separating vocals and instruments, it establishes a strong foundation for further development. Future extensions could involve integrating deep learning models for improved feature extraction, polyphonic analysis for richer comparisons, and the design of a user-friendly graphical interface to make the system accessible to a wider range of users, from researchers to industry professionals.

1.4 Contributions

This project contributes to the field of music plagiarism detection through the development of a robust framework that combines multiple levels of musical analysis. By integrating a wide range of features including chroma, MFCCs, spectral, rhythmic, tonal descriptors, motifs, and harmonic progressions, the system goes beyond simple spectral fingerprinting methods and captures both the fine-grained and higher-level structural elements of music. This comprehensive approach provides a more complete musical signature, making the detection process more accurate and versatile.

Another contribution of this work is the implementation of adaptive similarity measures. Different comparison methods are applied depending on the feature type: cosine similarity for timbral and spectral vectors, key-invariant cosine similarity for chroma-based harmony descriptors, and Jaccard similarity for motifs and chord progressions. This design ensures that the system remains robust against transformations such as pitch shifts, tempo changes, and altered harmonic structures, which are commonly used to disguise plagiarism.

The system also advances plagiarism detection by introducing structured decision rules. Instead of relying solely on raw similarity values, it applies conditions such as minimum segment matches and coverage thresholds to strengthen the reliability of results. These rules are further complemented by sorting model that rank results according to average similarity, maximum similarity, and match ratio, ultimately reducing both false positives and false negatives in plagiarism detection.

Efficiency and scalability form another key contribution of the system. The preprocessing pipeline is optimized with carefully designed segmentation strategies that balance coverage and computational load. By setting limits on the number of segments per song and controlling overlap percentages, the system is able to handle large-scale databases without compromising performance. This ensures that the framework is not only accurate but also practical for real-world applications where thousands of songs may need to be compared.

A distinctive feature of this project is the enhancement of interpretability through textual analysis. Beyond numerical similarity scores and rankings, the system automatically generates human-readable explanations using an external API. These explanations provide contextual insights into the detected similarities, highlighting harmonic, melodic, or rhythmic overlaps in a way that is meaningful to music experts. This contribution bridges the gap between technical outputs and expert judgment, allowing plagiarism detection results to be both precise and interpretable.

Finally, the project contributes by laying a foundation for future research and practical applications. The current framework can be expanded with deep learning-based feature extraction methods, polyphonic separation techniques for more detailed analysis, and the integration of user-friendly interfaces to increase accessibility. Together, these contributions make the system a valuable step forward in both academic research and industry practices related to protecting creative rights and maintaining fairness in the music industry.

1.5 Report Organization

This report is organised into 6 chapters: Chapter 1 Introduction, Chapter 2 Literature Review, Chapter 3 System Methodology, Chapter 4 System Design, Chapter 5 System Implementation, Chapter 6 System Evaluation and Discussion, Chapter 7 Conclusion and Recommendation. Chapter 1 introduces the project, covering the problem statement and motivation, project background, scope, objectives, contributions, and the organization of the report. Chapter 2 provides a literature review on audio plagiarism detection, analyzing the strengths and limitations of existing approaches. Chapter 3 discusses the overall system methodology employed in the project, while Chapter 4 focuses on the detailed design of the system. Chapter 5 presents the implementation process, highlighting issues and challenges encountered. Chapter 6 evaluates the system's performance and discusses the results. Finally, Chapter 7 concludes the report and offers recommendations for future work in related areas.

Chapter 2
Literature Review

This chapter presents the research and findings on previous works on Musical Feature Extraction and Plagiarism Detection.

2.1 Musical Feature/Signature Extraction Methods

In this section, the research focus in audio feature extraction area is presented. The most common practice in extracting audio fingerprints is to identify the frequency components of an audio. This approach focuses on capturing dominant peaks of an audio segment. The process begins with the digitalization of the audio signal. This digital audio is then broken down into small overlapping segments or frames, typically using a technique called windowing. Each frame usually spans a few milliseconds. The system in [2] employs a Hamming window where each frame is 512 points long and displaced by half its length. The process is followed by Fourier Transform, which converts the time-domain signal into the frequency domain. Fast Fourier transform (FFT) is used in paper [3] and [4]. This conversion produces a spectrogram, which visually displays how the frequency content of an audio signal evolves over time. The spectrogram contains a vast amount of data, but not all of it is necessary for fingerprinting. The key is to extract specific features that are unique and robust to distortions like noise or compression and the common features includes peaks and hashing. Peaks are prominent points in the spectrogram, often representing the strongest frequencies over specific time intervals. In [2], the peak points are extracted by Gaussian function to simulate the masking effect on the audio signal and the masking thresholds are calculated and updated for each frame. Candidate peak points are identified by comparing the energy values with thresholds across frames. Finally, parameters like the number of peak points per frame, the Gaussian curve's standard deviation, and the attenuation factor are adjusted to determine the final peak points. Whereas, the combination of these peaks or landmarks is used to generate a unique identifier or hash [3] [4].

On the other hand, to represent the harmonic content of an audio signal, chroma features of an audio is extracted. This provides more information about the audio in the aspects of chords and pitch class energy. According to [5], the first step involves decomposing the audio signal into

88 frequency bands corresponding to the musical notes A0 to C8, which are equivalent to MIDI pitches 21 to 108. Subsequently, the short-time mean-square power (STMSP) for each subband is computed. This is done by squaring the subband signal and convolving it with a rectangular window of 200 milliseconds, using a 50% overlap. The resulting STMSPs capture the localized energy content within each frequency band over short time intervals.[6] Next, the STMSP values are aggregated into chroma classes where each chroma class corresponds to a pitch class regardless of octave. For instance, all energy values for pitches such as A0, A1, ..., A7 are summed to compute the energy for the chroma class A. The process generates a 12-dimensional vector per time window, indicating how energy is distributed across the twelve pitch classes. Finally, each 12-dimensional chroma vector is normalized by normalizing each element with respect to the sum of all twelve elements, resulting in a vector that reflects the relative energy distribution across pitch classes. This normalization step ensures consistency across different time frames and facilitates meaningful comparisons.

However, if the dynamic range of the chromagram values is very wide, chroma features that are extracted can be converted into Chroma Energy distribution Normalized Statistics (CENS) to enhance the recognition of pitch-related content and musical patterns. A key aspect of computing CENS feature is the application of quantization, which follows logarithmically chosen thresholds to introduce a form of compression similar to Chroma-Log-Pitch (CLP[η]) features. Additionally, these features incorporate temporal smoothing which is achieved through averaging feature vectors over a sliding window and the size of this window directly influences the resulting feature representation. However, when downsampling is not applied, the feature extraction depends entirely on the CENS representation itself. [7]

According to [8], Mel-frequency cepstral coefficients (MFCCs) primarily encode timbral characteristics by representing the spectral envelope, or the overall shape of the spectrum, rather than exact pitch, harmony, or melodic detail. Consequently, they are particularly effective for detecting similarities between audio signals that share comparable instrumentation or production styles. Spectral features such as spectral contrast, centroid, bandwidth, roll-off, and flatness further describe the distribution and structure of frequency components, thereby capturing perceptual attributes like brightness and sharpness, which are essential for timbre differentiation and genre classification [9]. To analyze rhythmic similarity, tempograms are employed, as they represent local periodicities and patterns of beat strength over time [10]. In addition, melodic fragments or motifs can be identified through the extraction of pitch n-grams, which requires transcribing the melody into a sequence of pitches [11]. Finally, plagiarism involving harmonic progressions can be examined by recognizing chords or harmonic content over time and extracting sequences of chords for comparison.

2.2 Musical Feature/Signature Comparison Methods

In [12], the frequency-based fingerprints are compared using the Smith-Waterman algorithm, a well-known technique in bioinformatics to measure the similarity between segments of a test lead sheet and preexisting lead sheets in the database. To reduce false positives, the system includes a negative filter database that stores common musical elements deemed non-plagiaristic, such as typical chord progressions or melodies, and filters them out during analysis. However, the method of comparing the 2D array of peak points in the clip to each song is very slow. Instead, a more efficient method involves using a hash table [4]. Pairs of points, consisting of an anchor point and a target point, are generated in a way that preserves their relative positions. Then, a hash is generated based on the frequencies of the points and the time difference between them. Pairs for each anchor point will be formed with points in a defined target zone. This hash table uses the bin number of the anchor point for indexing, with a value calculated from the bin number of the target point and the time difference. When an audio clip is compared against the database, the hash table from the clip is matched against the hash tables from songs. If matching elements are encountered, a counter gets incremented. To provide further resolution, a separate table stores the offset times in frames of the anchor points from the start of a song. The shared elements then give a histogram of the offset times. A matching song would give a spike in this histogram and hence show an overlap with the clip, while an incorrect song would give scattered and inconsistent offset times. Similarly, the system in [2] uses a HashMap to convert audio fingerprint information into a unique key value through a hash function. The hash function assigns bits to represent frequency, time difference, and frequency difference, ensuring each key value is unique. Each key is associated with a value containing the audio ID, frequency, and timestamp of peak points. To improve retrieval efficiency, the system employs inverted indexing, where each hash table entry references a structure storing the number of peak points and a pointer to their corresponding ID, frequency, and time information. The index values range up to $2^{24} - 1$ to maintain uniqueness. For fingerprint matching, the system accounts for potential distortions in audio caused by external factors. The matching algorithm allows for a certain range of variation in frequency, time difference, and frequency difference between peak points. Matches are identified when these variations fall within predefined ranges. If the number of matching peak points in the dataset exceeds a threshold, the audio is considered a candidate match. Finally, the system determines if an audio clip is infringing by setting a threshold for the number of matched peak points. [12] Next, [13] presents a method that uses a database of CD audio recordings, which may include multiple interpretations of the same musical piece. For simplicity, all audio recordings in the database are concatenated into a single continuous document, while a supplementary structure is maintained to track the boundaries between individual recordings. The query is a short segment of audio, generally 10–30 seconds long. During feature extraction, both the database and each query is represented as a sequence of CENS feature vectors. The query's feature sequence is then slid across all possible subsequences of the database feature sequence with the same length, and a similarity score is computed for each comparison. The similarity is measured using a distance function which calculates the cosine similarity between the vectors:

$$d_M(X,Y) := 1 - \frac{1}{M}\sum_{m-1}^{M}\langle X_m, Y_m\rangle \qquad (1)$$

Zero distance indicates exact similarity, and a distance of one indicates total dissimilarity.[14] The best match is identified by minimizing the similarity score, and a neighborhood of length *M* around the best match is excluded from subsequent comparisons. The operation repeats until the required number of matches is obtained or the similarity criterion is met.

2.3 Summary and Analysis of the Existing Methods

Frequency peaks and chroma features offer distinct advantages and limitations when used as audio fingerprints. Frequency peaks excel at capturing precise spectral details, making them highly effective for speech recognition, instrument identification, and analyzing specific sound components. However, they are sensitive to noise and variations in recording quality, which can lead to inconsistencies. On the other hand, chroma features focus on harmonic structures and pitch class distributions, making them ideal for music retrieval, chord recognition, and tonal analysis. They are more robust against timbral changes and instrument variations but may struggle with fine spectral details and distinguishing non-melodic content such as speech. While frequency peaks provide accuracy in detecting unique audio signatures, chroma features enhance musical pattern recognition. Timbre-oriented descriptors such as MFCCs and spectral features excel at distinguishing instrumentation, brightness, and production style, making them effective when plagiarism involves replicating sound texture; however, their reliance on absolute spectral content renders them weak for capturing melodic or harmonic similarity, particularly under transposition. Rhythm-related features, including tempograms and onset-based measures, are adept at detecting periodicities and groove patterns, offering strong indicators when rhythmic imitation occurs, but they are highly sensitive to tempo fluctuations and expressive timing, which limits robustness. Symbolic representations such as motifs (pitch n-grams) and harmonic progressions (chord n-grams) are far more effective in uncovering plagiarism at the melodic and harmonic levels, since they capture structural aspects of music rather than surface-level acoustics. Yet, both approaches depend on reliable transcription or chord recognition, and their performance declines in noisy or polyphonic contexts. Tonal representations, such as the Tonnetz, complement these symbolic approaches by explicitly modeling harmonic space, though they too suffer from instability in complex or noisy audio[15]. Taken together, timbral and spectral features provide surface-level cues, rhythmic features capture temporal regularities, and symbolic or tonal features address structural similarity, but none is sufficient on its own; their overlapping strengths and vulnerabilities highlight the necessity of integrating multiple feature types for reliable plagiarism detection.

Cosine similarity is well-suited for fine-grained comparisons of normalized feature vectors, such as chroma features, by measuring the angle between them. It provides a continuous similarity score, making it useful for ranking matches and detecting subtle melodic differences. It is also robust to amplitude variations since it ignores vector magnitudes. However, cosine similarity is computationally expensive, especially for large audio databases, and it is sensitive to key or pitch changes unless transposition-invariant features are used. In contrast, hashing methods such as landmark or jumpcode hashing offer fast and scalable lookup by converting audio features into compact, discrete representations. These methods are highly efficient for large-scale retrieval and are typically robust to distortions like noise, tempo shifts, and key changes. However, they may lose detailed information, leading to false positives or negatives, and do not provide a smooth similarity measure. Additionally, effective hashing requires careful feature engineering, and very short queries may not generate reliable matches. In summary, cosine similarity excels in precision and interpretability for smaller datasets, while hashing offers speed, scalability, and robustness for large-scale audio retrival tasks.

Chapter 3

System Approach

3.1 System Design Diagram

3.1.1 System Architecture Diagram

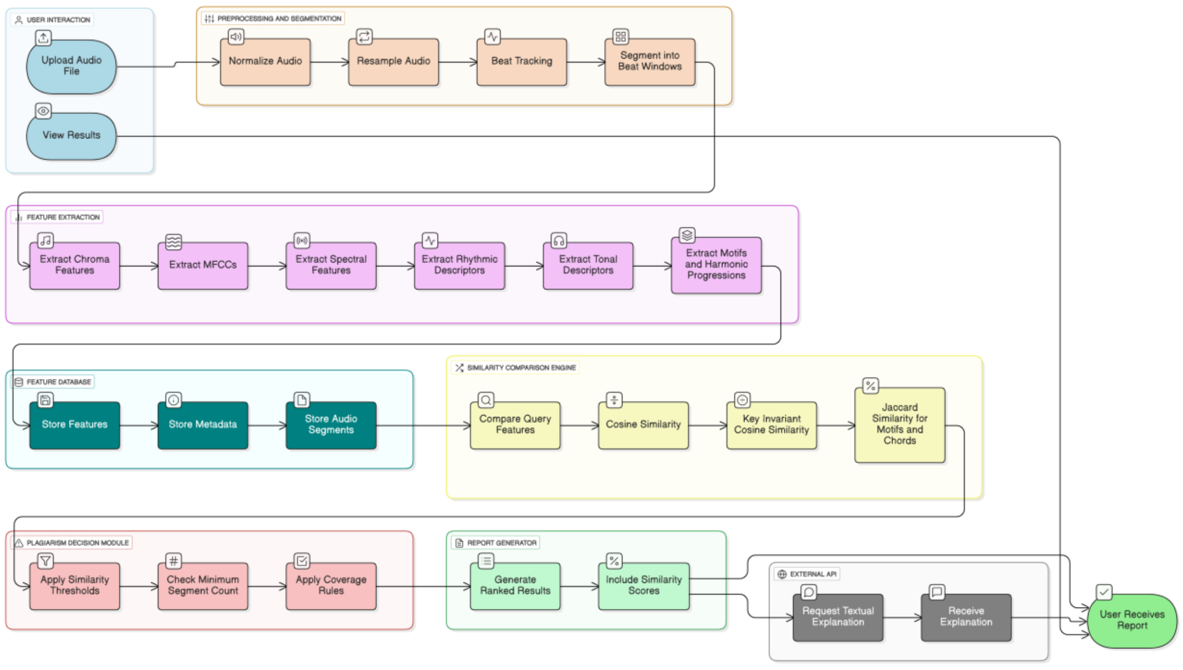

USER INTERACTION
Upload Audio File
View Results
PREPROCESSING AND SEGMENTATION
Normalize Audio
Resample Audio
Beat Tracking
Segment into Beat Windows
FEATURE EXTRACTION
Extract Chroma Features
Extract MFCCs
Extract Spectral Features
Extract Rhythmic Descriptors
Extract Tonal Descriptors
Extract Motifs and Harmonic Progressions
FEATURE DATABASE
Store Features
Store Metadata
Store Audio Segments
SIMILARITY COMPARISON ENGINE
Compare Query Features
Cosine Similarity
Key Invariant Cosine Similarity
Jaccard Similarity for Motifs and Chords
PLAGIARISM DECISION MODULE
Apply Similarity Thresholds
Check Minimum Segment Count
Apply Coverage Rules
REPORT GENERATOR
Generate Ranked Results
Include Similarity Scores
EXTERNAL API
Request Textual Explanation
Receive Explanation
User Receives Report

Figure 3.1.1: Architecture Diagram of the System

The system architecture diagram shows how the system works from the moment a user uploads an query audio file and databse folder until the results are returned. The process begins in the user interaction layer, where the user can provide either a folder of database audio files or a query song for testing, and later view the final report. Once an audio file is uploaded, it goes through the preprocessing and segmentation stage. Here, the system normalizes and resamples the audio to make sure all files follow the same format, applies beat tracking to detect the rhythm, and then divides the track into smaller beat-based windows. This ensures that both database and query songs are processed consistently and fairly before any comparisons take place.

After that, the segmented audio enters the feature extraction module, where the system pulls out different musical characteristics. These features include chroma to capture harmonic content, MFCCs to describe timbre, spectral features to represent energy distribution, rhythmic descriptors for tempo and beats, tonal features for harmonic relationships, and higher-level structures like motifs and chord progressions. Together, these features give the system a rich musical fingerprint for each segment of audio.

The processed features are stored in the feature database, which organizes them into feature files, metadata records, and segment references for efficient retrieval. When a query is submitted, the system moves to the similarity comparison engine, where the extracted query features are matched against the database. Different similarity methods are used here: cosine similarity for feature vectors like MFCCs and spectral descriptors, a key-invariant cosine similarity to handle pitch changes in chroma features, and Jaccard similarity to check overlaps in motifs and harmonic progressions.

Once comparisons are complete, the plagiarism decision module evaluates the results by checking if similarity thresholds are exceeded, whether enough segments match, and if the coverage across the query is sufficient. If the conditions are met, the song is flagged as a potential plagiarism candidate. The findings are then passed to the report generator, which creates a ranked list of results with similarity scores for the user. Finally, the system calls on an external API to provide a written explanation of the top matches, giving more context about harmonic, melodic, rhythmic, and timbral similarities. This helps the results feel less like raw numbers and more like evidence that experts can interpret.

3.1.2 Use Case Diagram and Description

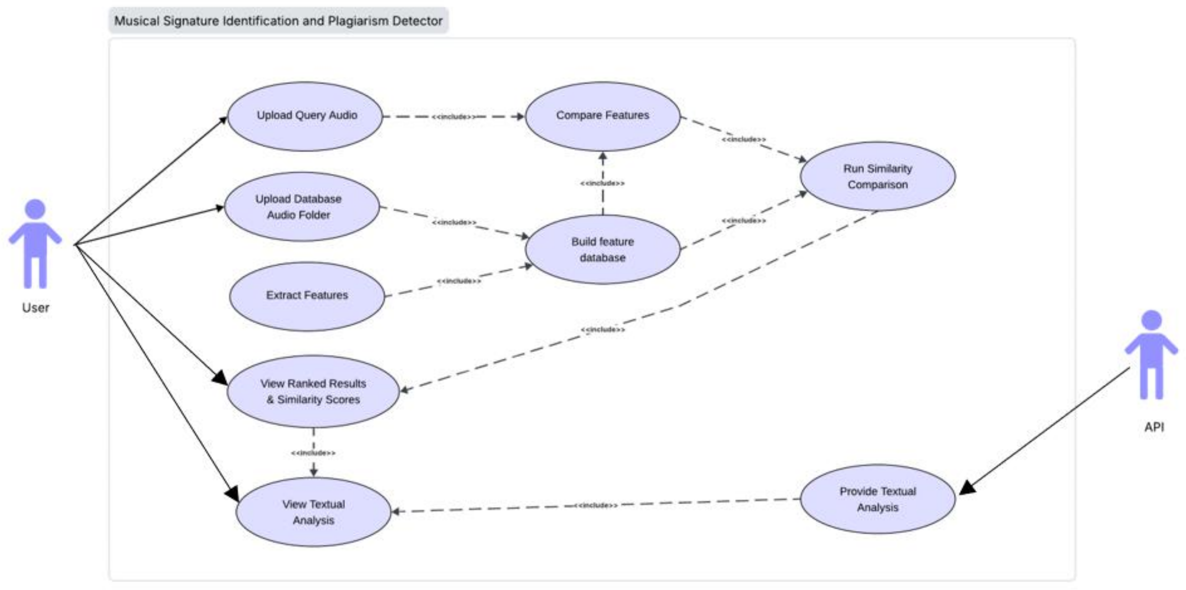

Musical Signature Identification and Plagiarism Detector
User
API
Upload Query Audio
Compare Features
Run Similarity Comparison
Upload Database Audio Folder
Build feature database
Extract Features
View Ranked Results & Similarity Scores
View Textual Analysis
Provide Textual Analysis
<<include>>

Figure 3.1.2: Use Case Diagram of the System

The use case diagram illustrates how different actors interact with the Music Plagiarism Detection System and the functions that the system provides. The main actor is the user, who plays a central role in supplying input and receiving output from the system. The user can upload audio files either as database songs or as query tracks to be tested for plagiarism. When database audio is uploaded, the system processes the songs and builds a feature database by extracting musical characteristics such as chroma, MFCCs, spectral, rhythmic, tonal features, motifs, and harmonic progressions. This ensures that a structured reference collection is available for later comparisons.

When a query audio file is submitted, the system processes it in a similar manner by extracting features and then comparing them against the stored database features. The system runs similarity measurements across multiple dimensions of music, including timbre, harmony, rhythm, and symbolic structures, and produces ranked results with similarity scores. These results are presented back to the user in the form of a plagiarism report. At this stage, the user may also request an extended explanation of the results. For this purpose, the system communicates with an external API, which acts as a secondary actor. The API provides interpretive textual analysis of the top matches, highlighting areas of similarity in terms of harmony, rhythm, and melodic content.

By modeling these interactions, the use case diagram shows both the essential functions of the system such as uploading audio, building the database, running comparisons, and viewing results as well as requesting detailed textual explanations. This provides a clear picture of how the user and external systems collaborate with the plagiarism detector to complete the overall process of identifying potential plagiarism in music recordings.

3.1.3 Activity Diagram

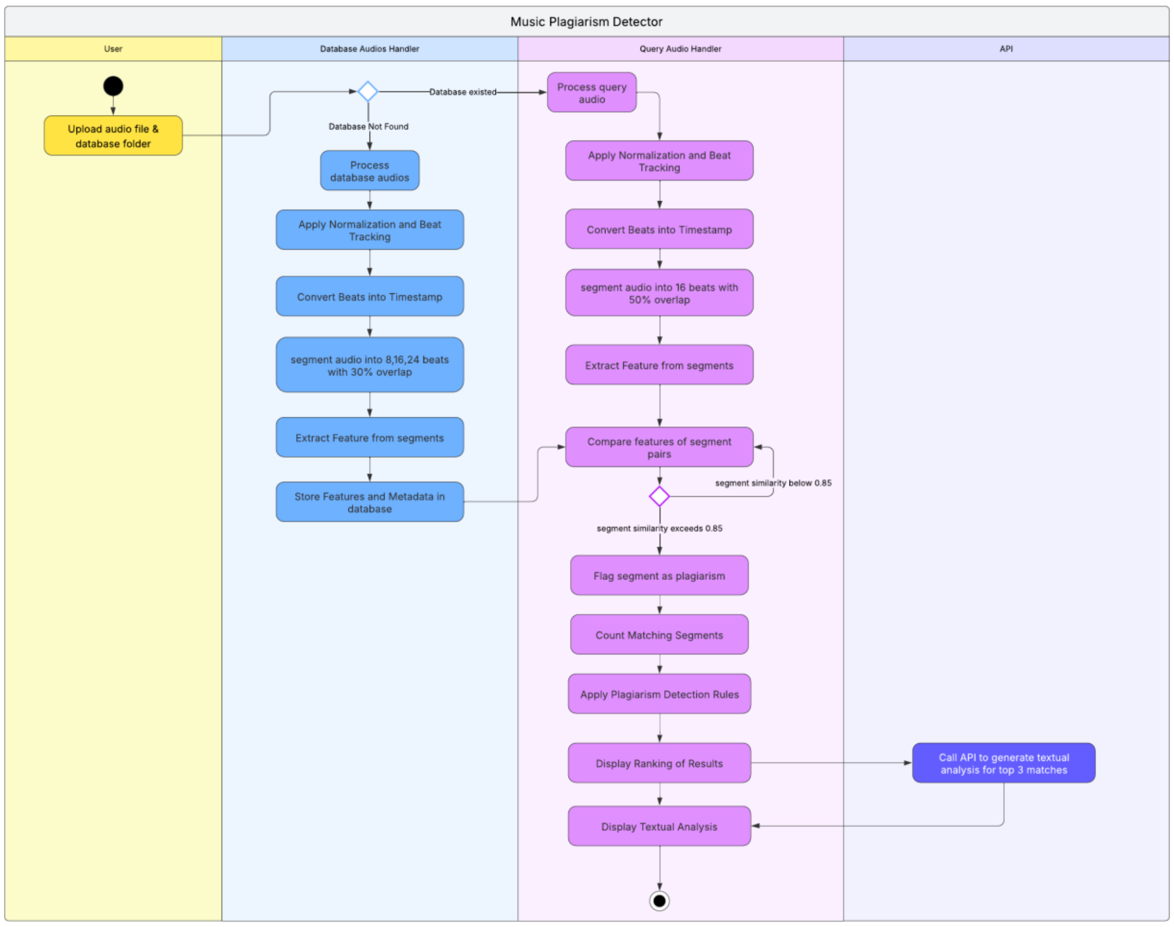

Music Plagiarism Detector
User
Database Audios Handler
Query Audio Handler
API
Upload audio file & database folder
Database existed
Database Not Found
Process database audios
Apply Normalization and Beat Tracking
Convert Beats into Timestamp
segment audio into 8,16,24 beats with 30% overlap
Extract Feature from segments
Store Features and Metadata in database
Process query audio
Apply Normalization and Beat Tracking
Convert Beats into Timestamp
segment audio into 16 beats with 50% overlap
Extract Feature from segments
Compare features of segment pairs
segment similarity below 0.85
segment similarity exceeds 0.85
Flag segment as plagiarism
Count Matching Segments
Apply Plagiarism Detection Rules
Display Ranking of Results
Call API to generate textual analysis for top 3 matches
Display Textual Analysis

Figure 3.1.3: Activity Diagram of the System

The activity diagram describes the sequence of operations that take place within the system, from the moment an audio file is uploaded until the results are delivered. The process begins when the user provides an input, which may be either a database audio file or a query audio file. Once the file is received, the system applies preprocessing steps to standardize the audio by normalizing the signal, resampling it to a fixed sampling rate, and ensuring consistent quality across all files. Following this, the system performs segmentation, dividing the audio into beat-based windows of different lengths depending on whether it is part of the database or a query.

After segmentation, the system extracts a wide range of musical features that represent different aspects of the recording. These include chroma features to capture harmonic content, MFCCs to represent timbral qualities, spectral and rhythmic descriptors to analyze energy and tempo, tonal features for harmonic relationships, and higher-level patterns such as motifs and chord progressions. If the input file belongs to the database, the extracted features are stored alongside metadata and audio segments to form a structured reference collection. If the input file is a query, the extracted features are used immediately for similarity comparison with the stored database features.

The system then evaluates the query by comparing its features with those of the database using multiple similarity measures. Once similarity scores are obtained, they are processed by the decision module, which applies thresholds, segment counts, and coverage checks to determine whether potential plagiarism is present. Based on these results, the system generates a report containing ranked matches and similarity statistics. The process concludes with the report being presented to the user, and allows for additional interpretive analysis through an external service that provides textual explanations of the top matches. Overall, the activity diagram outlines the logical flow of the system, capturing both the main pathway of operations and the conditional paths that depend on the type of input provided.

Chapter 4
System Design

4.1 System Block Diagram

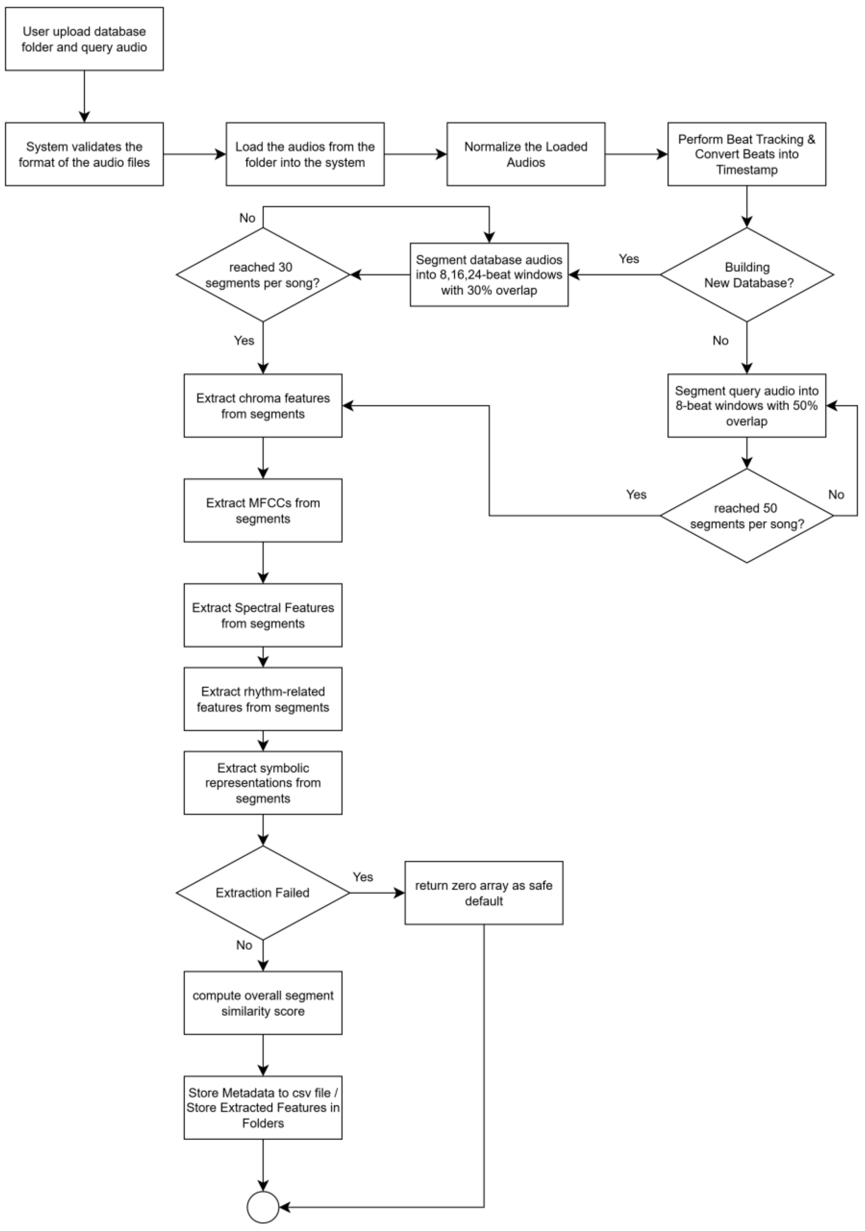

User upload database folder and query audio
System validates the format of the audio files
Load the audios from the folder into the system
Normalize the Loaded Audios
Perform Beat Tracking & Convert Beats into Timestamp
Building New Database?
Yes
Segment database audios into 8,16,24-beat windows with 30% overlap
reached 30 segments per song?
No
Yes
No
Segment query audio into 8-beat windows with 50% overlap
reached 50 segments per song?
Yes
No
Extract chroma features from segments
Extract MFCCs from segments
Extract Spectral Features from segments
Extract rhythm-related features from segments
Extract symbolic representations from segments
Extraction Failed
Yes
return zero array as safe default
No
compute overall segment similarity score
Store Metadata to csv file / Store Extracted Features in Folders

Figure 4.1.1: Block Diagram of Audio Preprocessing of the System

The audio preprocessing stage begins once the user uploads both the database folder and the query audio file into the system. Before any further processing, the system first validates the format of the uploaded files to ensure they conform to the required audio specifications, such as sampling rate, file type, and channel configuration. This step prevents errors during feature extraction and maintains consistency across all inputs. Once validated, the system proceeds to load the audio files into memory, where they undergo normalization. Normalization adjusts the amplitude of the signals, ensuring that volume differences between recordings do not interfere with subsequent analysis.

After normalization, the system applies beat tracking to detect tempo and rhythmic structure, and converts the beats into timestamped markers. These markers provide the framework for dividing audio files into beat-based windows, which form the basis for feature extraction. For database audio, segmentation is performed at multiple levels—8, 16, and 24 beats per window with a 30% overlap between segments. This approach increases robustness by capturing musical variations at different temporal resolutions. For query audio, segmentation is slightly different: the audio is divided into 16-beat windows with a higher overlap of 50%, providing more granular coverage to ensure accurate matching against the database. Both database and query segmentation processes also incorporate segment limits which are 30 segments per song for the database and 50 for the query to optimize computational efficiency and prevent excessive processing time.

Through these steps, the preprocessing module standardizes and structures the raw audio files, transforming them into well-aligned segments that are ready for the feature extraction stage. This ensures that comparisons between query and database songs are carried out on a consistent basis, regardless of variations in recording quality, tempo, or length.

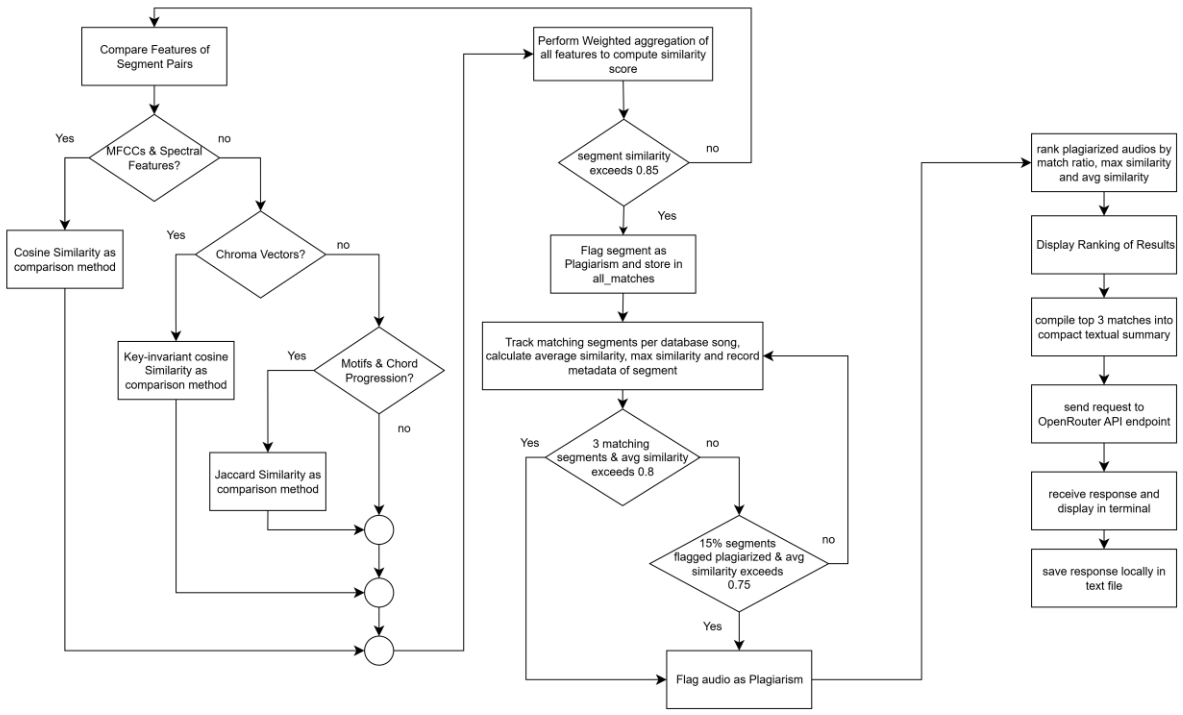

Compare Features of Segment Pairs
MFCCs & Spectral Features?
Yes
no
Cosine Similarity as comparison method
Chroma Vectors?
Key-invariant cosine Similarity as comparison method
Motifs & Chord Progression?
Jaccard Similarity as comparison method
Perform Weighted aggregation of all features to compute similarity score
segment similarity exceeds 0.85
Flag segment as Plagiarism and store in all_matches
Track matching segments per database song, calculate average similarity, max similarity and record metadata of segment
3 matching segments & avg similarity exceeds 0.8
15% segments flagged plagiarized & avg similarity exceeds 0.75
Flag audio as Plagiarism
rank plagiarized audios by match ratio, max similarity and avg similarity
Display Ranking of Results
compile top 3 matches into compact textual summary
send request to OpenRouter API endpoint
receive response and display in terminal
save response locally in text file

Figure 4.1.2: Block Diagram of Audio Comparison and Result Reporting of the System

The comparison and plagiarism detection stage begins once features have been extracted from both the query and database audio segments. At this point, the system compares pairs of segments across multiple feature types. For timbral and spectral features such as MFCCs and spectral descriptors, cosine similarity is applied to measure how closely the feature vectors align. When comparing chroma vectors, the system makes use of a key-invariant cosine similarity, which ensures that harmonic similarities are detected even if the music has been transposed into a different key. For symbolic features such as motifs and chord progressions, Jaccard similarity is used to measure the degree of overlap between the sets of symbolic patterns. This multi-method approach allows the system to capture different aspects of musical similarity in a robust manner.

Once these feature-level comparisons are completed, the results are aggregated. A weighted scheme is applied so that each feature type contributes appropriately to the overall similarity score, preventing any single descriptor from dominating the result. If the similarity score between a pair of segments exceeds a threshold of 0.85, the segment is flagged as plagiarised and stored in a record of matches. The system then keeps track of matching segments for each database song, calculating average similarity, maximum similarity, and other metadata to build a detailed profile of potential overlaps.

Plagiarism decisions are based on clear rules to avoid false positives. A database song is considered a plagiarism candidate if it contains at least three matching segments with an average similarity greater than 0.8, or if more than 15% of the query segments are flagged with an average similarity above 0.75. This two-tiered rule ensures that both concentrated overlaps and widespread partial copying can be detected. Once a song meets these conditions, it is flagged as plagiarised.

The system then ranks plagiarised candidates by combining three key factors: the match ratio, the maximum similarity score, and the average similarity across matching segments. Songs that show stronger and more consistent similarities are placed higher in the ranking. The ranked results are displayed to the user, providing a clear overview of which database songs most closely match the query. The evaluation of system correctness relies on a structured indexing approach established during the categorization of audio files. If the query and database entries share the same index, this indicates a ground truth plagiarism pair.

To make the results easier to interpret, the system automatically generates textual explanations. It compiles the top three matches into a compact summary and sends a request to the OpenRouter API. The API processes this input and returns a natural-language analysis highlighting where and how similarities occur. For example, pointing out repeated chord progressions or shared rhythmic motifs. The generated explanation is displayed in the system interface and also saved locally as a text file for future reference. This final step ensures that the results are not just numerical but also come with meaningful insights, making it easier for music experts to evaluate plagiarism claims.

4.2 System Components Specifications

4.2.1 Hardware Requirements

The required hardware consists of a standard PC or laptop equipped with an operating system such as Windows, Linux, or macOS.

4.2.2 Software Requirements

Python of version 3.8 and above is required for installation as the programming language of the system. Libraries needed including:

- Librosa (for audio feature extraction: chroma, MFCC, spectral)
- NumPy (for feature arrays, vector normalization)
- Pandas (for database management)
- SoundFile (for writing audio segments)

- OpenRouter API (LLM-based textual analysis of similarity)

4.3 Circuits and Components Design

4.3.1 Class

- EnhancedPlagiarismDetector class

This is the main controller class of the system, responsible for handling the overall workflow of plagiarism detection. It defines the initialization parameters such as the target BPM, sampling rate, and directory paths for storing audio segments, features, and query files. The class also provides functions for database construction, query processing, similarity computation, plagiarism decision-making, and textual analysis generation.

4.3.2 Methods

The core functionality of the system is implemented through the following methods:

- extract_multi_features(y, sr)

Extracts multiple audio features from an input signal, including chroma (CENS, CQT), MFCCs (static, delta, delta2), spectral descriptors (contrast, centroid, bandwidth, rolloff, flatness), tonal features (tonnetz), rhythmic features (tempogram, onset autocorrelation), and symbolic representations such as motifs and harmonic progressions.

- extract_motifs(y, sr, n=4)

Identifies melodic motifs by generating pitch n-grams from the chroma representation of the signal. Returns a set of melodic patterns for similarity analysis.

- compare_motifs(motifs1, motifs2)

Compares two motif sets using Jaccard similarity to measure the overlap of melodic n-grams between query and database segments.

- extract_harmonic_progression(y, sr, n=4)

Detects chord sequences from the chroma representation by matching frames against major and minor chord templates. Produces smoothed chord progressions and chord n-grams of various lengths.

- compare_harmonic_progressions(ngrams1, ngrams2)

Evaluates the similarity of chord progressions between two segments using Jaccard similarity.

- process_audio_optimized(audio_path, is_query=False, max_segments=50)

Handles audio segmentation and feature extraction. For database audio, segments of 8, 16, and 24 beats are created with 30% overlap. For query audio, only 16-beat segments are used with 50% overlap. Extracted features and metadata are saved in structured directories.

- compute_similarity_enhanced(features1, features2, paths1, paths2)

Computes the overall similarity score between a query segment and a database segment by aggregating weighted feature similarities. Includes key-invariant cosine similarity for chroma, cosine similarity for MFCCs and spectral features, and Jaccard similarity for motif and harmonic progression sets.

- build_database_optimized(rebuild=True, max_segments_per_song=30)

Constructs the feature database by processing all songs in the dataset, segmenting them, and extracting features. The database is saved in enhanced_music_database.csv along with feature and audio segment files.

- detect_plagiarism_enhanced(query_path, database_csv, similarity_threshold, plagiarism_threshold, min_matching_segments)

Executes the plagiarism detection pipeline for a given query. Compares query segments against the database, applies decision rules, and determines whether plagiarism is present.

- generate_report(matches_df, plagiarism_results, query_name)

Summarizes and ranks the results of plagiarism detection. Generates a human-readable report with statistics such as average similarity, maximum similarity, and matching segment ratios.

- analyze_top_matches(matches_df, top_3, api_key, model, save_path)

Integrates with the OpenRouter API to generate textual analysis of similarities between the query and top candidate songs, highlighting harmonic, rhythmic, timbral, and melodic evidence.

4.3.3 Database

The system maintains a structured database that stores extracted features and metadata for all songs in the reference collection. Each audio file is preprocessed, segmented, and stored as follows:

- Numerical features (e.g., chroma, MFCC, spectral, tonal, rhythmic descriptors) are compressed and saved in .npz format.
- Symbolic features (motifs and harmonic progressions) are stored as .txt files for set-based analysis.
- Segmented audio clips are saved in .wav format for reference and validation.
- Metadata for each segment, including song name, segment ID, segment length, duration, and file paths, is stored in a centralized CSV file (enhanced_music_database.csv).

This structured design ensures that the database can be efficiently searched and updated, while maintaining a clear separation between numerical and symbolic features for flexible comparison.

4.3.4 External API and Model Integration

The system integrates with the Meta-Llama/LLaMA-3.3-8B-Instruct:Free model, which is hosted through OpenRouter. This model is a lightweight and efficient variant of the larger LLaMA 3.3 family, optimized for quick response times while maintaining strong instruction-following capabilities. It is specifically designed to generate high-quality textual outputs, making it suitable for tasks such as producing interpretive analyses of harmonic progressions, rhythmic patterns, and melodic overlaps in music plagiarism detection.

To access this model, the system sends requests through the OpenRouter API, which acts as a middleware that standardizes communication with multiple providers. Requests are structured as JSON objects and sent using an OpenAI-compatible completion endpoint. A typical request includes the model name (meta-llama/llama-3.3-8b-instruct:free), the input prompt containing summaries of top plagiarism candidates, and configuration parameters such as maximum tokens or temperature. The request is authenticated using an API key provided by OpenRouter. Once submitted, OpenRouter routes the request to the appropriate model provider, ensuring load balancing and fallback in case of service interruptions. The response is returned in JSON format, containing the model's generated text. The system then extracts this output and incorporates it into the plagiarism report as a human-readable explanation, highlighting similarities in harmony, rhythm, and melody. This process bridges raw numerical results with expert-level narrative analysis, enhancing the interpretability of the detection results.

4.4 System Components Interaction Operations

The execution of the system is initiated through the *main()* function, which instantiates the *EnhancedPlagiarismDetector* class with the designated database directory. Upon initialization, the system first invokes the *build_database_optimized()* method to preprocess the reference audio files, segment them into beat-based windows, and extract relevant features for storage in the database. When a query audio file is provided, the *detect_plagiarism_enhanced()* method is executed. This method internally calls *process_audio_optimized()* to normalize, segment, and extract features from the query, followed by *compute_similarity_enhanced()* to perform a weighted comparison of features between query and database segments. The resulting similarity scores are consolidated, and plagiarism detection rules are applied to determine potential infringements. Subsequently, the *generate_report()* method produces a structured ranking of candidate songs, detailing similarity metrics and detection outcomes. For enhanced interpretability, the system may also utilize the *analyze_top_matches()* method, which integrates with the OpenRouter API to access the Meta-Llama/LLaMA-3.3-8B-Instruct:Free model, providing expert-level textual analysis of harmonic, rhythmic, and melodic similarities. This systematic invocation of class methods ensures a coherent and reproducible workflow, from database construction to plagiarism reporting.

Chapter 5
System Implementation

5.1 Hardware Setup

The hardware involved in this project is laptop. A laptop issued for the process of loading and processing audio files from datasets to obtain the audio fingerprint, then it will also be used for comparison on the audio fingerprints in the database. Lastly, textual analysis of the result will be displayed through the screen monitor of the laptop.

Table 5.1 Specifications of laptop

Description	**Specifications**
Model	HP Pavilion Laptop 15-eg0106TX
Processor	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
Operating System	Windows 10
Graphic	NVIDIA® GeForce® MX450 (2 GB GDDR5 dedicated)
Memory	8 GB DDR4-3200 MHz RAM (1 x 8 GB)
Storage	512 GB PCIe® NVMe™ M.2 SSD

5.2 Software Setup

The proposed system is developed using Python 3.11 due to its strong ecosystem for scientific computing and audio processing. Several open-source libraries are integrated to perform different stages of the workflow. Librosa is the core library for audio signal processing, enabling feature extraction such as chroma, MFCC, spectral, tonal, and rhythmic descriptors. NumPy is used for efficient numerical computation and vector manipulation, while Pandas manages tabular data storage and retrieval for the music database. SoundFile (PySoundFile) supports reading and writing audio segments in .wav format. To evaluate similarities, scikit-learn provides tools for cosine similarity and feature normalization. For saving extracted features, compressed storage is handled through NumPy's .npz format, and metadata is maintained in .csv files. Additionally, the system integrates with the OpenRouter API to utilize large language models (LLMs) for generating textual explanations of musical similarities.

These libraries collectively form the backbone of the system, allowing robust, scalable, and reproducible analysis of music plagiarism.

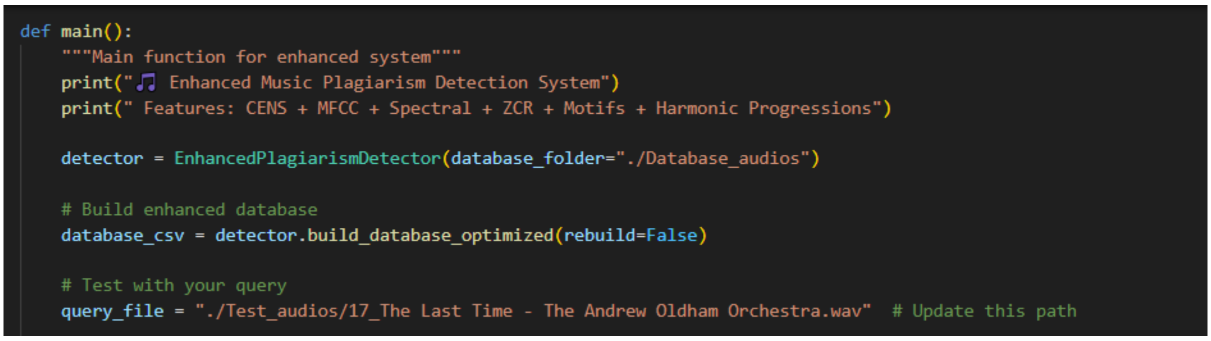
5.3 Setting and Configuration

The system requires specific configurations to ensure consistent preprocessing, feature extraction, and similarity comparison across both the database and query audio files. The sampling rate was fixed at 22,050 Hz to standardize all audio inputs, while normalization was applied to reduce amplitude variations. The system was configured to operate at a target tempo of 120 BPM, which allows query and database segments to be aligned consistently. For segmentation, database audio was divided into segments of 8, 16, and 24 beats with a 30% overlap, whereas query audio was segmented into 16-beat windows with a 50% overlap to increase matching opportunities.

A predefined feature extraction pipeline was enabled, consisting of chroma features (CENS, CQT), MFCCs (static, delta, and delta2), spectral descriptors (centroid, bandwidth, rolloff, flatness, and contrast), rhythmic features (tempogram and onset autocorrelation), tonal features (tonnetz), and symbolic representations (motifs and harmonic progressions). Similarity thresholds were configured such that a segment-level similarity score of 0.85 was required to consider two segments as matching. At the decision level, a song was flagged for plagiarism if at least three matching segments were detected, or if 15% of the query segments overlapped with the database song.

The system directories were also organized systematically to separate data flow. Database audio and features were stored under the ./Database_audios, ./segments, and ./features folders, while query audio and features were stored under the ./Test_audios, ./query_segments, and ./query_features folders. Metadata and extracted features were indexed in enhanced_music_database.csv for efficient retrieval. Finally, the system was configured to integrate with the OpenRouter API, requiring an API key for generating textual analyses of detected similarities.

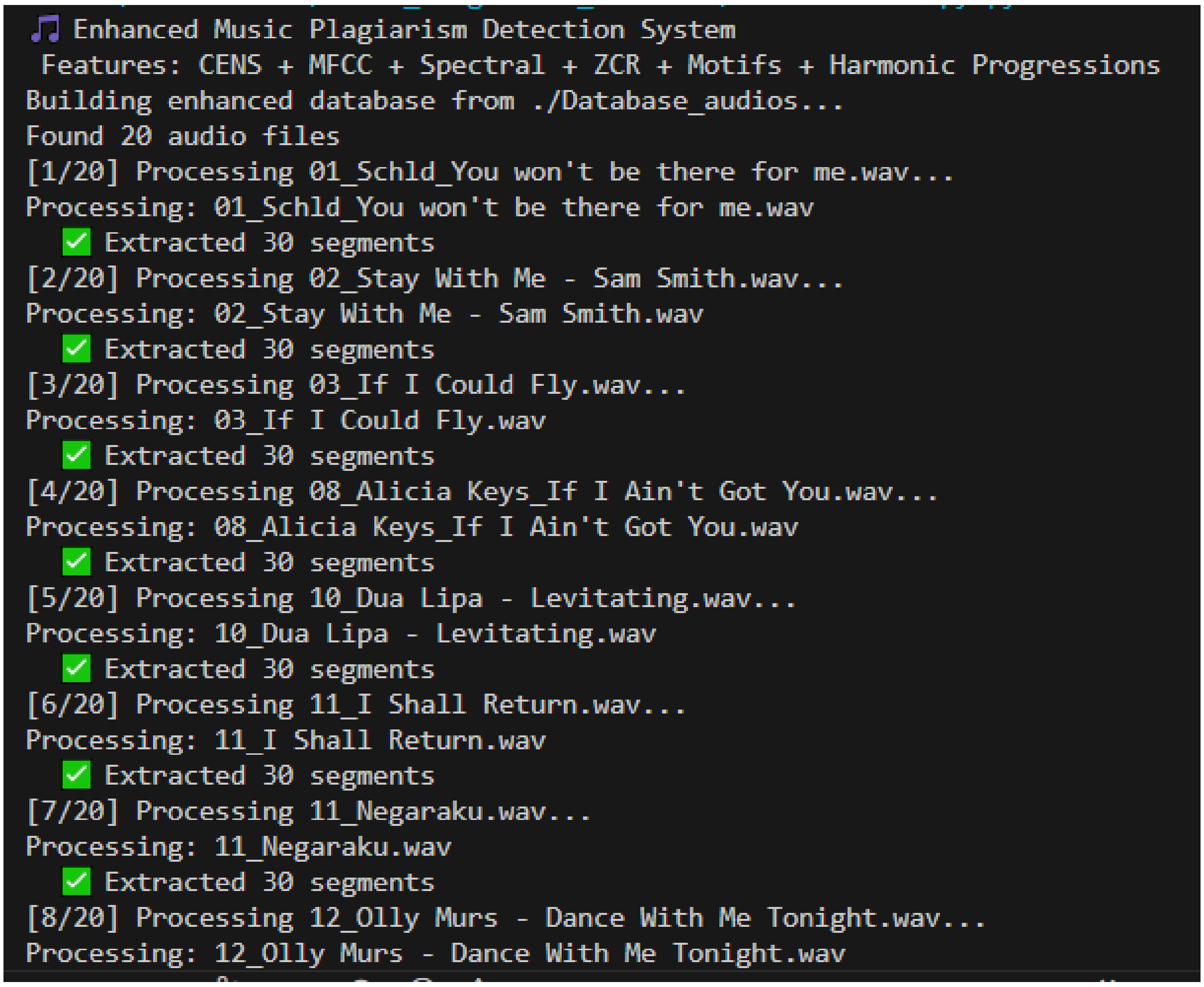
5.4 System Operation



```
def main():
    """Main function for enhanced system"""
    print("🎵 Enhanced Music Plagiarism Detection System")
    print(" Features: CENS + MFCC + Spectral + ZCR + Motifs + Harmonic Progressions")

    detector = EnhancedPlagiarismDetector(database_folder="./Database_audios")

    # Build enhanced database
    database_csv = detector.build_database_optimized(rebuild=False)

    # Test with your query
    query_file = "./Test_audios/17_The Last Time - The Andrew Oldham Orchestra.wav"  # Update this path
```

Figure 5.4.1: System Screenshot for uploading query audio file and database folder

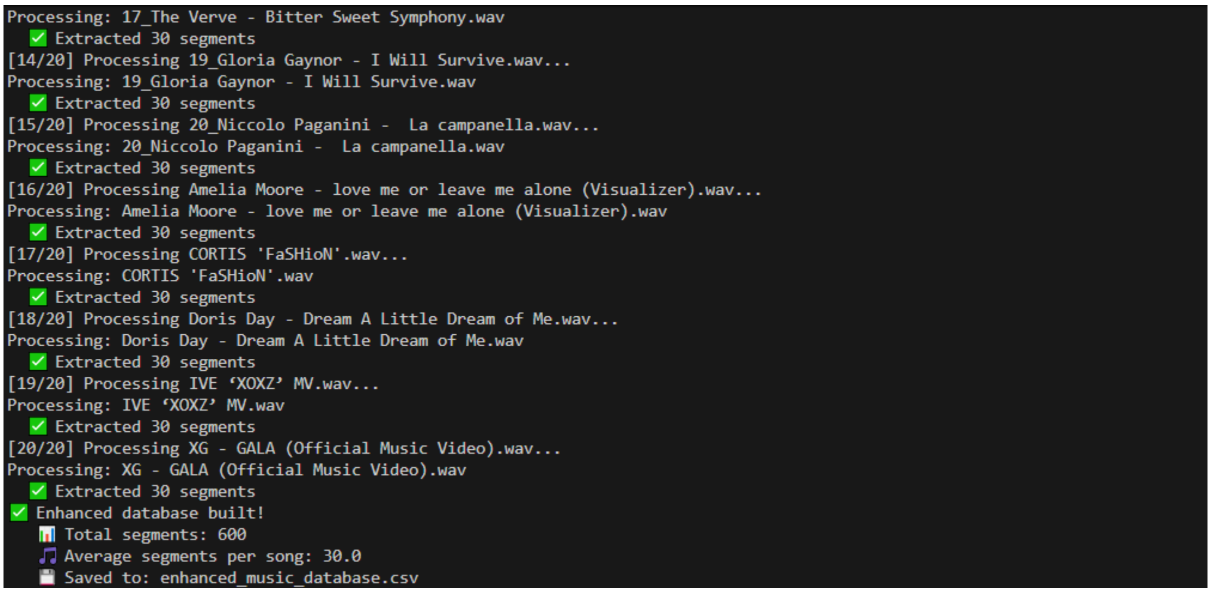
Figure 5.4.1 shows the system interface where the user uploads the query audio and the database folder for plagiarism detection. Once uploaded, these inputs are passed into the preprocessing pipeline, where they are prepared for segmentation, feature extraction, and subsequent similarity analysis. This step forms the starting point of the entire detection process, ensuring that both the query and database are consistently formatted before comparison.



```
🎵 Enhanced Music Plagiarism Detection System
 Features: CENS + MFCC + Spectral + ZCR + Motifs + Harmonic Progressions
Building enhanced database from ./Database_audios...
Found 20 audio files
[1/20] Processing 01_Schld_You won't be there for me.wav...
Processing: 01_Schld_You won't be there for me.wav
  ✅ Extracted 30 segments
[2/20] Processing 02_Stay With Me - Sam Smith.wav...
Processing: 02_Stay With Me - Sam Smith.wav
  ✅ Extracted 30 segments
[3/20] Processing 03_If I Could Fly.wav...
Processing: 03_If I Could Fly.wav
  ✅ Extracted 30 segments
[4/20] Processing 08_Alicia Keys_If I Ain't Got You.wav...
Processing: 08_Alicia Keys_If I Ain't Got You.wav
  ✅ Extracted 30 segments
[5/20] Processing 10_Dua Lipa - Levitating.wav...
Processing: 10_Dua Lipa - Levitating.wav
  ✅ Extracted 30 segments
[6/20] Processing 11_I Shall Return.wav...
Processing: 11_I Shall Return.wav
  ✅ Extracted 30 segments
[7/20] Processing 11_Negaraku.wav...
Processing: 11_Negaraku.wav
  ✅ Extracted 30 segments
[8/20] Processing 12_Olly Murs - Dance With Me Tonight.wav...
Processing: 12_Olly Murs - Dance With Me Tonight.wav
```

Figure 5.4.2: System Screenshot for Building Database

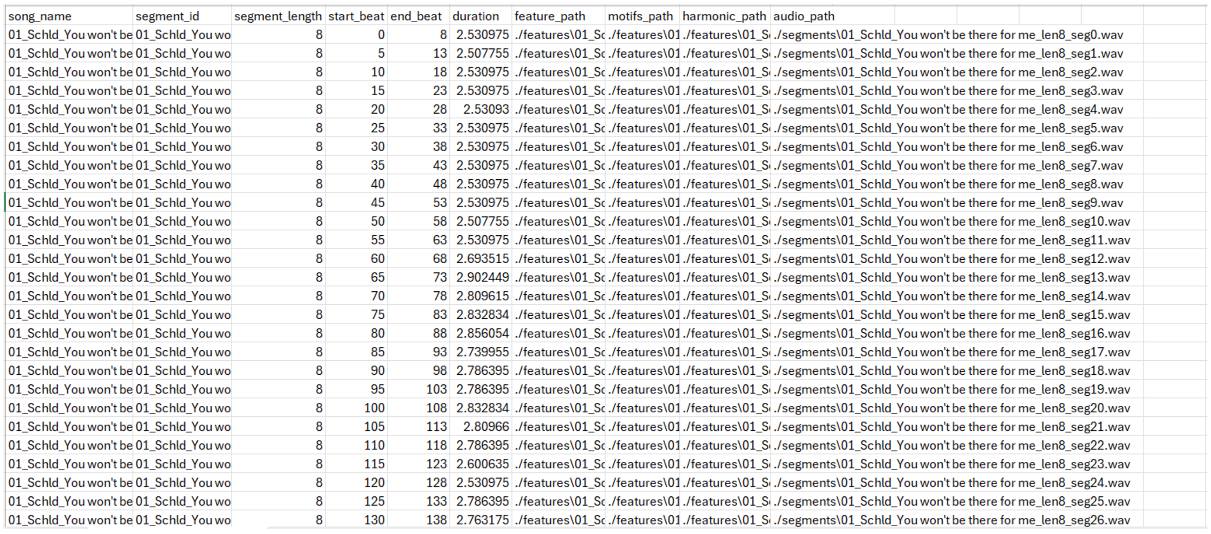
Figure 5.4.2 illustrates the process of building the feature database within the Enhanced Music Plagiarism Detection System. In this stage, the system begins by scanning the specified folder ./Database_audios and identifying the available audio files, in this case detecting a total of 20 songs. Each file is processed sequentially, as shown in the log output. For example, the first song, “01_Schld_You won’t be there for me.wav”, is divided into 30 beat-based segments, from which multiple features are extracted, including CENS, MFCCs, spectral descriptors, zero-crossing rate, motifs, and harmonic progressions. The same procedure is then applied to the subsequent songs, such as “02_Stay With Me – Sam Smith.wav”. These extracted features and their associated metadata are stored in the database, creating a structured reference collection that will later be used for similarity comparison against query audio files. This stage ensures that the system has a well-prepared and feature-rich database that forms the backbone of the plagiarism detection process.



```
Processing: 17_The Verve - Bitter Sweet Symphony.wav
  ✅ Extracted 30 segments
[14/20] Processing 19_Gloria Gaynor - I Will Survive.wav...
Processing: 19_Gloria Gaynor - I Will Survive.wav
  ✅ Extracted 30 segments
[15/20] Processing 20_Niccolo Paganini -  La campanella.wav...
Processing: 20_Niccolo Paganini -  La campanella.wav
  ✅ Extracted 30 segments
[16/20] Processing Amelia Moore - love me or leave me alone (Visualizer).wav...
Processing: Amelia Moore - love me or leave me alone (Visualizer).wav
  ✅ Extracted 30 segments
[17/20] Processing CORTIS 'FaSHioN'.wav...
Processing: CORTIS 'FaSHioN'.wav
  ✅ Extracted 30 segments
[18/20] Processing Doris Day - Dream A Little Dream of Me.wav...
Processing: Doris Day - Dream A Little Dream of Me.wav
  ✅ Extracted 30 segments
[19/20] Processing IVE ‘XOXZ’ MV.wav...
Processing: IVE ‘XOXZ’ MV.wav
  ✅ Extracted 30 segments
[20/20] Processing XG - GALA (Official Music Video).wav...
Processing: XG - GALA (Official Music Video).wav
  ✅ Extracted 30 segments
✅ Enhanced database built!
   📊 Total segments: 600
   🎵 Average segments per song: 30.0
   💾 Saved to: enhanced_music_database.csv
```

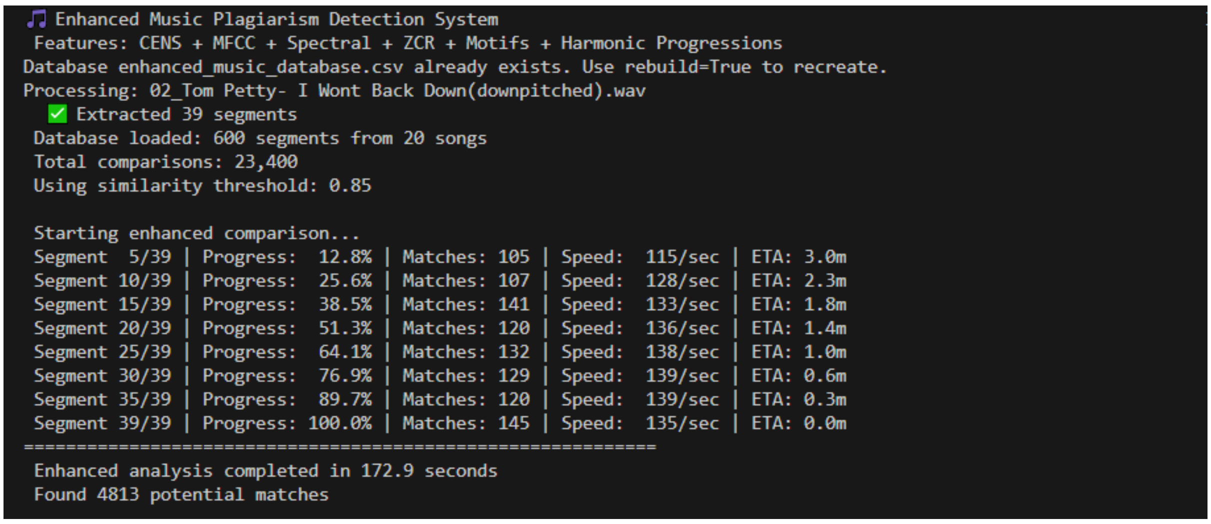
Figure 5.4.3: System Screenshot for Building Database Sucessfully

Figure 5.4.3 shows the database-building process of the Enhanced Music Plagiarism Detection System. After scanning the folder ./Database_audios, the system successfully processed 20 audio files, segmenting each into 30 beat-based segments on average. This resulted in a total of 600 segments being extracted and analyzed. For each segment, features such as CENS, MFCCs, spectral descriptors, zero-crossing rate, motifs, and harmonic progressions were computed and stored. The processed data, along with metadata for efficient retrieval, was compiled into the file enhanced_music_database.csv which shows in figure 5.4.4 that serves as the backbone of the feature database used for subsequent similarity comparison with query audio.



song_name	segment_id	segment_length	start_beat	end_beat	duration	feature_path	motifs_path	harmonic_path	audio_path
01_Schld_You won't be	01_Schld_You wo	8	0	8	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg0.wav
01_Schld_You won't be	01_Schld_You wo	8	5	13	2.507755	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg1.wav
01_Schld_You won't be	01_Schld_You wo	8	10	18	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg2.wav
01_Schld_You won't be	01_Schld_You wo	8	15	23	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg3.wav
01_Schld_You won't be	01_Schld_You wo	8	20	28	2.53093	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg4.wav
01_Schld_You won't be	01_Schld_You wo	8	25	33	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg5.wav
01_Schld_You won't be	01_Schld_You wo	8	30	38	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg6.wav
01_Schld_You won't be	01_Schld_You wo	8	35	43	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg7.wav
01_Schld_You won't be	01_Schld_You wo	8	40	48	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg8.wav
01_Schld_You won't be	01_Schld_You wo	8	45	53	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg9.wav
01_Schld_You won't be	01_Schld_You wo	8	50	58	2.507755	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg10.wav
01_Schld_You won't be	01_Schld_You wo	8	55	63	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg11.wav
01_Schld_You won't be	01_Schld_You wo	8	60	68	2.693515	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg12.wav
01_Schld_You won't be	01_Schld_You wo	8	65	73	2.902449	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg13.wav
01_Schld_You won't be	01_Schld_You wo	8	70	78	2.809615	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg14.wav
01_Schld_You won't be	01_Schld_You wo	8	75	83	2.832834	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg15.wav
01_Schld_You won't be	01_Schld_You wo	8	80	88	2.856054	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg16.wav
01_Schld_You won't be	01_Schld_You wo	8	85	93	2.739955	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg17.wav
01_Schld_You won't be	01_Schld_You wo	8	90	98	2.786395	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg18.wav
01_Schld_You won't be	01_Schld_You wo	8	95	103	2.786395	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg19.wav
01_Schld_You won't be	01_Schld_You wo	8	100	108	2.832834	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg20.wav
01_Schld_You won't be	01_Schld_You wo	8	105	113	2.80966	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg21.wav
01_Schld_You won't be	01_Schld_You wo	8	110	118	2.786395	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg22.wav
01_Schld_You won't be	01_Schld_You wo	8	115	123	2.600635	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg23.wav
01_Schld_You won't be	01_Schld_You wo	8	120	128	2.530975	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg24.wav
01_Schld_You won't be	01_Schld_You wo	8	125	133	2.786395	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg25.wav
01_Schld_You won't be	01_Schld_You wo	8	130	138	2.763175	./features\01_Sc	./features\01	./features\01_S	./segments\01_Schld_You won't be there for me_len8_seg26.wav

Figure 5.4.4: System Screenshot for enhanced_music_database.csv

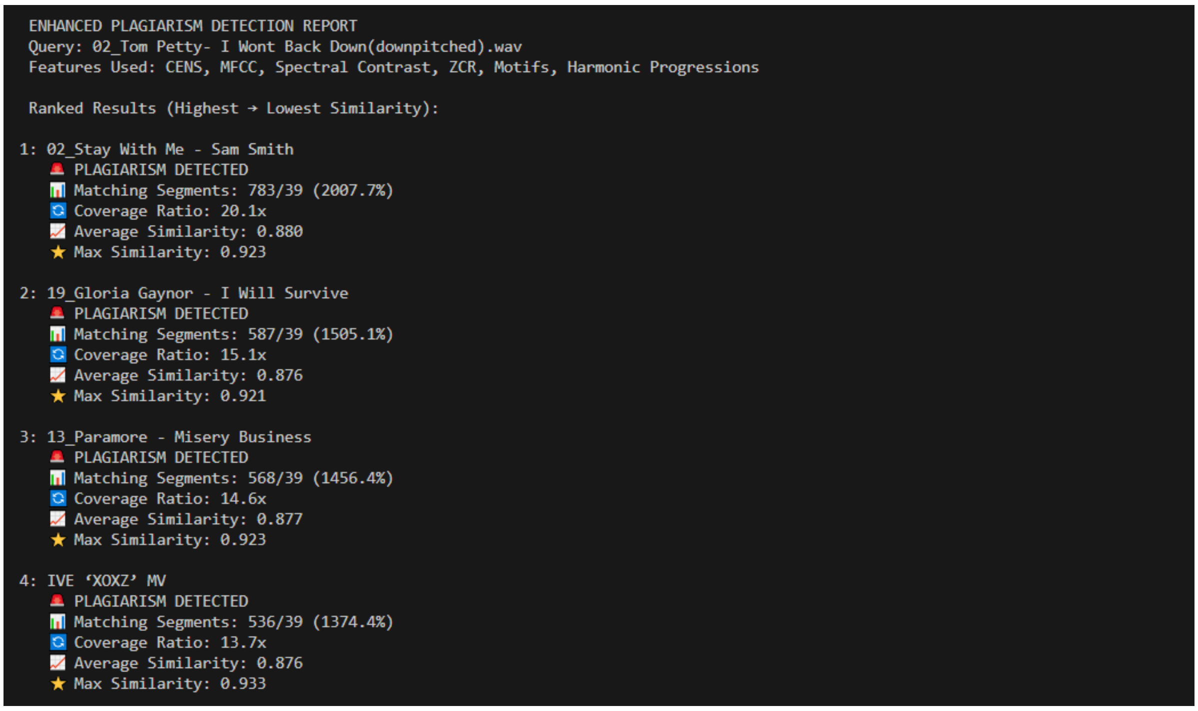


```
🎵 Enhanced Music Plagiarism Detection System
 Features: CENS + MFCC + Spectral + ZCR + Motifs + Harmonic Progressions
Database enhanced_music_database.csv already exists. Use rebuild=True to recreate.
Processing: 02_Tom Petty- I Wont Back Down(downpitched).wav
   ✅ Extracted 39 segments
 Database loaded: 600 segments from 20 songs
 Total comparisons: 23,400
 Using similarity threshold: 0.85

 Starting enhanced comparison...
 Segment  5/39 | Progress:  12.8% | Matches: 105 | Speed:  115/sec | ETA: 3.0m
 Segment 10/39 | Progress:  25.6% | Matches: 107 | Speed:  128/sec | ETA: 2.3m
 Segment 15/39 | Progress:  38.5% | Matches: 141 | Speed:  133/sec | ETA: 1.8m
 Segment 20/39 | Progress:  51.3% | Matches: 120 | Speed:  136/sec | ETA: 1.4m
 Segment 25/39 | Progress:  64.1% | Matches: 132 | Speed:  138/sec | ETA: 1.0m
 Segment 30/39 | Progress:  76.9% | Matches: 129 | Speed:  139/sec | ETA: 0.6m
 Segment 35/39 | Progress:  89.7% | Matches: 120 | Speed:  139/sec | ETA: 0.3m
 Segment 39/39 | Progress: 100.0% | Matches: 145 | Speed:  135/sec | ETA: 0.0m
============================================================
 Enhanced analysis completed in 172.9 seconds
 Found 4813 potential matches
```

Figure 5.4.5: System Screenshot for Processing Query Audio and Comparing Segments

Figure 5.4.5 presents the system log during the enhanced comparison phase of the Music Plagiarism Detection System. The process begins by checking the availability of the feature database, which in this case is already stored as enhanced_music_database.csv containing 600 segments extracted from 20 songs. The query file, *"02_Tom Petty – I Won't Back Down(downpitched).wav"*, is then processed, yielding 39 segments ready for comparison. Using a similarity threshold of 0.85, the system initiates pairwise comparisons between query and database segments, resulting in a total of 23,400 comparisons.

The progress log highlights key details of the computation: the number of segments processed, the percentage of completion, the number of matches detected, the processing speed in segments per second, and the estimated time to completion. For example, by the 20th query segment, the system had already achieved 51.3% completion with 120 matches identified, running at an average speed of 136 comparisons per second. These outputs demonstrate the efficiency of the system in handling large-scale comparisons while maintaining accurate tracking of progress and results in real time.



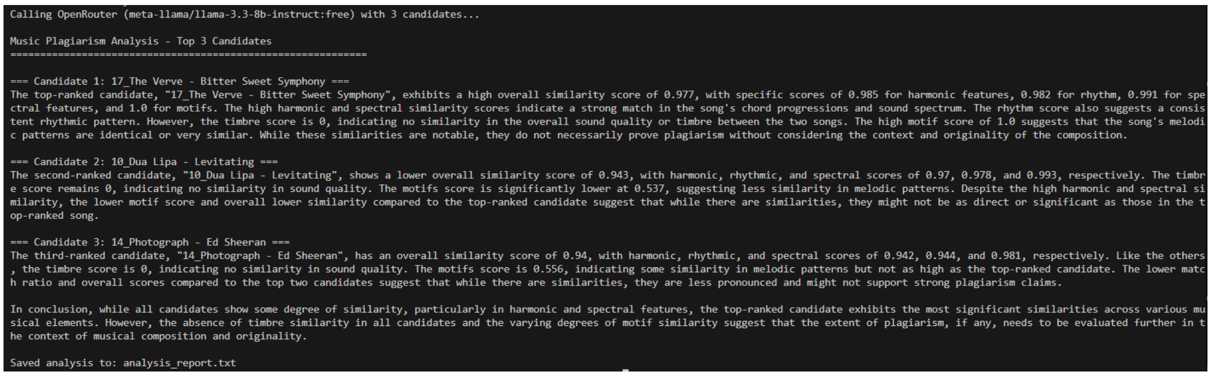
```
 ENHANCED PLAGIARISM DETECTION REPORT
 Query: 02_Tom Petty- I Wont Back Down(downpitched).wav
 Features Used: CENS, MFCC, Spectral Contrast, ZCR, Motifs, Harmonic Progressions

 Ranked Results (Highest → Lowest Similarity):

1: 02_Stay With Me - Sam Smith
   🚨 PLAGIARISM DETECTED
   📊 Matching Segments: 783/39 (2007.7%)
   🔄 Coverage Ratio: 20.1x
   📈 Average Similarity: 0.880
   ⭐ Max Similarity: 0.923

2: 19_Gloria Gaynor - I Will Survive
   🚨 PLAGIARISM DETECTED
   📊 Matching Segments: 587/39 (1505.1%)
   🔄 Coverage Ratio: 15.1x
   📈 Average Similarity: 0.876
   ⭐ Max Similarity: 0.921

3: 13_Paramore - Misery Business
   🚨 PLAGIARISM DETECTED
   📊 Matching Segments: 568/39 (1456.4%)
   🔄 Coverage Ratio: 14.6x
   📈 Average Similarity: 0.877
   ⭐ Max Similarity: 0.923

4: IVE 'XOXZ' MV
   🚨 PLAGIARISM DETECTED
   📊 Matching Segments: 536/39 (1374.4%)
   🔄 Coverage Ratio: 13.7x
   📈 Average Similarity: 0.876
   ⭐ Max Similarity: 0.933
```

Figure 5.4.6: Screenshot of System's Plagiarism Report Ranked Result

Figure 5.4.6 presents the Enhanced Plagiarism Detection Report generated after analyzing the query file *"02_Tom Petty – I Won't Back Down (downpitched).wav."* The report ranks database entries in descending order of similarity, using indicators such as the number of matching segments, coverage ratio, average similarity, and maximum similarity score. For example, the top-ranked result, *"Stay With Me – Sam Smith"*, achieved an average similarity of 0.880 and a maximum similarity of 0.923, supported by 783 matching segments with a coverage ratio more than twenty times the query length. Similarly, other high-ranking results, such as *"I Will Survive – Gloria Gaynor"* and *"Misery Business – Paramore"*, show substantial overlaps, highlighting the system's ability to capture structural resemblances across diverse tracks.

The evaluation of system correctness relies on a structured indexing approach established during the categorization of audio files. If the query and database entries share the same index, this indicates a ground truth plagiarism pair. The system's decision can therefore be validated by checking whether such indexed pairs are ranked among the top results. This method allows for systematic verification of the system's performance, providing clear evidence of whether its similarity judgments align with pre-established categories.

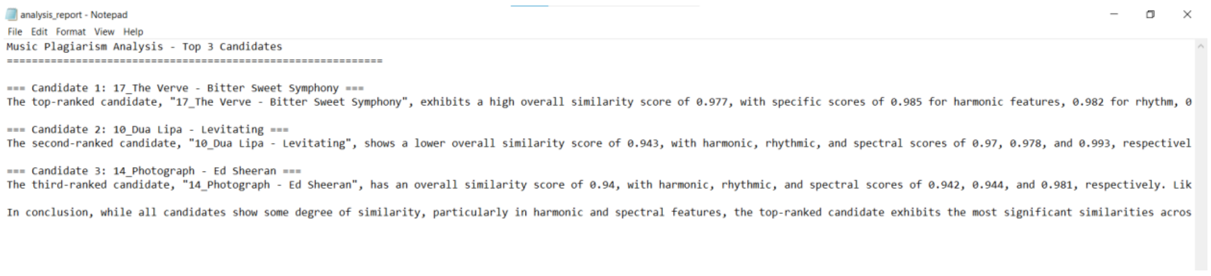
It is also important to emphasize that the system is not designed to make final plagiarism claims, but rather to act as an aid for musical experts. To ensure balanced interpretation, the focus is placed on the top three ranked matches, which are subjected to closer expert analysis. This approach enables the system to provide meaningful guidance, offering detailed similarity evidence while leaving the ultimate judgment to human expertise.



```
Calling OpenRouter (meta-llama/llama-3.3-8b-instruct:free) with 3 candidates...

Music Plagiarism Analysis - Top 3 Candidates
============================================================

=== Candidate 1: 17_The Verve - Bitter Sweet Symphony ===
The top-ranked candidate, "17_The Verve - Bitter Sweet Symphony", exhibits a high overall similarity score of 0.977, with specific scores of 0.985 for harmonic features, 0.982 for rhythm, 0.991 for spe
ctral features, and 1.0 for motifs. The high harmonic and spectral similarity scores indicate a strong match in the song's chord progressions and sound spectrum. The rhythm score also suggests a consis
tent rhythmic pattern. However, the timbre score is 0, indicating no similarity in the overall sound quality or timbre between the two songs. The high motif score of 1.0 suggests that the song's melodi
c patterns are identical or very similar. While these similarities are notable, they do not necessarily prove plagiarism without considering the context and originality of the composition.

=== Candidate 2: 10_Dua Lipa - Levitating ===
The second-ranked candidate, "10_Dua Lipa - Levitating", shows a lower overall similarity score of 0.943, with harmonic, rhythmic, and spectral scores of 0.97, 0.978, and 0.993, respectively. The timbr
e score remains 0, indicating no similarity in sound quality. The motifs score is significantly lower at 0.537, suggesting less similarity in melodic patterns. Despite the high harmonic and spectral si
milarity, the lower motif score and overall lower similarity compared to the top-ranked candidate suggest that while there are similarities, they might not be as direct or significant as those in the t
op-ranked song.

=== Candidate 3: 14_Photograph - Ed Sheeran ===
The third-ranked candidate, "14_Photograph - Ed Sheeran", has an overall similarity score of 0.94, with harmonic, rhythmic, and spectral scores of 0.942, 0.944, and 0.981, respectively. Like the others
, the timbre score is 0, indicating no similarity in sound quality. The motifs score is 0.556, indicating some similarity in melodic patterns but not as high as the top-ranked candidate. The lower matc
h ratio and overall scores compared to the top two candidates suggest that while there are similarities, they are less pronounced and might not support strong plagiarism claims.

In conclusion, while all candidates show some degree of similarity, particularly in harmonic and spectral features, the top-ranked candidate exhibits the most significant similarities across various mu
sical elements. However, the absence of timbre similarity in all candidates and the varying degrees of motif similarity suggest that the extent of plagiarism, if any, needs to be evaluated further in t
he context of musical composition and originality.

Saved analysis to: analysis_report.txt
```

Figure 5.4.7: Screenshot of System's Plagiarism Textual Analysis

Figure 5.4.7 displays the system's plagiarism textual analysis feature, where numerical similarity results are transformed into expert-level narrative explanations. This component is powered by the Meta-Llama/LLaMA-3.3-8B-Instruct:Free model accessed through the OpenRouter API. The analysis provides human-readable insights into harmonic, rhythmic, and melodic overlaps, making the detection results more interpretable and useful for expert review.



```
Music Plagiarism Analysis - Top 3 Candidates
============================================================

=== Candidate 1: 17_The Verve - Bitter Sweet Symphony ===
The top-ranked candidate, "17_The Verve - Bitter Sweet Symphony", exhibits a high overall similarity score of 0.977, with specific scores of 0.985 for harmonic features, 0.982 for rhythm, 0

=== Candidate 2: 10_Dua Lipa - Levitating ===
The second-ranked candidate, "10_Dua Lipa - Levitating", shows a lower overall similarity score of 0.943, with harmonic, rhythmic, and spectral scores of 0.97, 0.978, and 0.993, respectivel

=== Candidate 3: 14_Photograph - Ed Sheeran ===
The third-ranked candidate, "14_Photograph - Ed Sheeran", has an overall similarity score of 0.94, with harmonic, rhythmic, and spectral scores of 0.942, 0.944, and 0.981, respectively. Lik

In conclusion, while all candidates show some degree of similarity, particularly in harmonic and spectral features, the top-ranked candidate exhibits the most significant similarities acros
```

Figure 5.4.8: Screenshot of analysis_report.txt

Figure 5.4.8 shows the plagiarism analysis report saved locally as analysis_report.txt. This file contains the system-generated textual explanations of detected similarities, ensuring that the results are not only viewable within the application but also preserved for future reference. By storing the report in a portable text format, experts can revisit, share, and cross-check the analysis independently of the system interface, supporting transparency and reproducibility in the plagiarism detection process.

5.5 Implementation Issue and Challenges

During the development and testing of the system, several implementation challenges were encountered. One of the primary issues was the accuracy of beat tracking, particularly in noisy audios where tempo detection often failed. To address this, a fallback strategy was implemented whereby fixed time-based segmentation was applied with the condition where beat tracking was unsuccessful.

Another challenge involved computational efficiency, as comparing every query segment against all database segments resulted in high processing time for large datasets. This was mitigated by introducing a maximum segment limit per song and optimizing feature storage using compressed .npz files instead of raw arrays.

The system also faced difficulties in handling feature extraction errors. Certain features such as constant-Q chroma (CQT) occasionally failed due to short or low-quality segments. To ensure robustness, the system incorporated error-handling mechanisms and fallback features, defaulting to CENS chroma or zero-valued placeholders when necessary.

An additional challenge was memory management during large-scale processing. Extracting multiple features simultaneously increased memory usage, especially when storing motifs and harmonic progressions. This was addressed by separating numerical features (stored in compressed format) from symbolic features (stored as text files), thereby reducing storage overhead and improving scalability.

5.6 Concluding Remark

In conclusion, the system was successfully implemented according to the planned methodology and design. The hardware and software environments were configured to support efficient audio preprocessing, feature extraction, and similarity comparison. Through systematic parameter settings and organized directory structures, the system was able to process both database and query audio files consistently. The implementation further demonstrated the operational flow of the EnhancedPlagiarismDetector class, from database construction and query analysis to plagiarism detection and report generation. Although several challenges were encountered, such as beat tracking inaccuracies, feature extraction errors, and computational overhead, these were mitigated through fallback mechanisms, optimized feature storage, and error-handling strategies. The implementation stage therefore validates the feasibility of the proposed approach and establishes a functioning plagiarism detection system capable of analyzing and reporting musical similarities with robustness and reliability.

Chapter 6
System Design

6.1 System Testing and Performance Metrics

The evaluation of the Enhanced Music Plagiarism Detection System focuses on its ability to correctly retrieve and rank potential plagiarism pairs from the database. Unlike binary classification systems, this project is designed as a retrieval and ranking framework, where outputs are ordered lists of similarity scores rather than categorical labels. Therefore, the evaluation relies on ranking-oriented performance metrics, which better capture the system's practical effectiveness.

6.1.1 Top-k Accuracy (Hit Rate)

Top-k accuracy measures the percentage of queries in which the correct plagiarism pair is retrieved within the top k results. In this project, Top-3 Accuracy is used as the primary benchmark, as the system is designed to assist experts by presenting the top three candidates for review. Out of 14 plagiarized queries tested (11 correct detections and 3 missed cases), the system successfully ranked the correct pair within the top three in 11 cases.

$$Top - 3\ Accuracy = \frac{Correct\ Detections\ (TP)}{Total\ Plagiarized\ Pairs} = \frac{11}{14} = 78.6\%$$

This shows that in nearly four out of five cases, the true plagiarism pair is highlighted among the top three results, making the system effective as a decision-support tool.

6.1.2 Mean Reciprocal Rank (MRR)

The Mean Reciprocal Rank (MRR) was employed to evaluate how highly the system ranked the correct plagiarized audio for each query. This metric provides a more detailed perspective than Top-k accuracy by rewarding the system more when the correct pair appears at the top of the ranking and penalizing lower placements or missed detections. The MRR is defined as:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$$

where N represents the number of queries and $rank_i$ denotes the position of the correct pair in the ranking for the i-th query, with a value of 0 assigned if the pair is not retrieved.

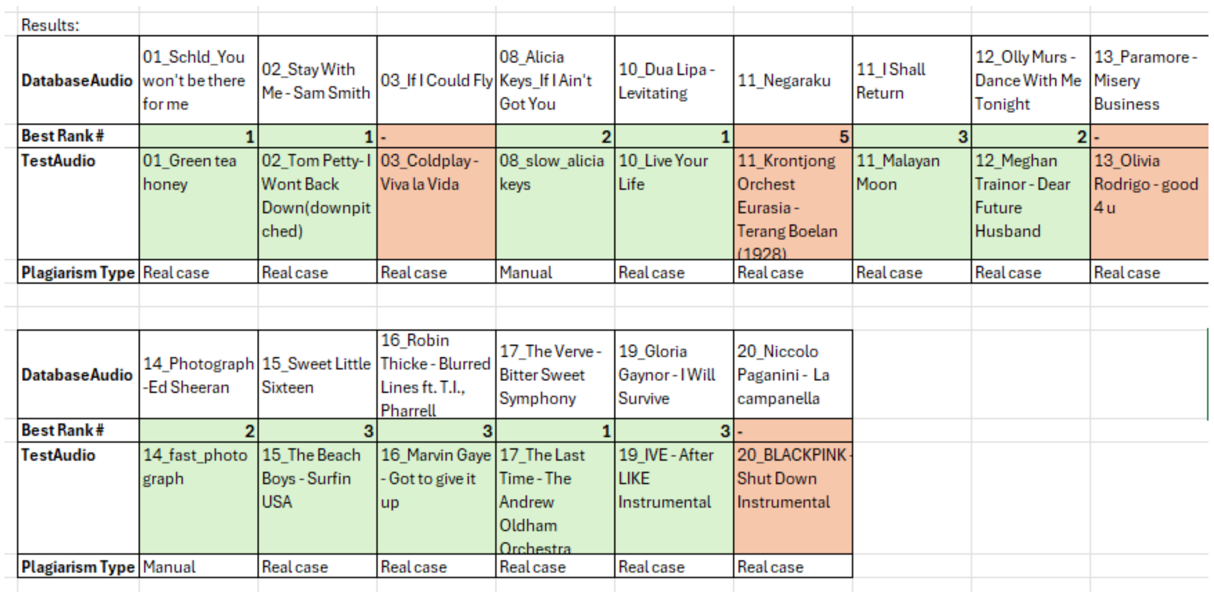
In this project, 14 plagiarized queries were tested. The correct pairs were ranked 1st in four cases, 2nd in two cases, and 3rd in four cases, while three pairs did not appear within the top three results. Based on these results, the total reciprocal rank score was 6.332, leading to an average MRR value of 0.45. This indicates that, on average, the system ranked the true plagiarism pair closer to the second position across all queries.

An MRR of 0.45 reflects a reasonable level of ranking efficiency, as the system frequently surfaces the correct pair within the top three, though not always at the highest position. This metric complements the Top-3 Accuracy by highlighting not only whether the system identified the correct pair but also how effectively it prioritized it within the ranking. Together, these results demonstrate that while the system reliably retrieves true plagiarism pairs, improvements in ranking consistency could further enhance its utility for expert decision-making.

6.2 Testing Setup and Results

The testing phase was conducted to evaluate the effectiveness of the system in identifying and ranking potential plagiarism pairs. A total of 20 audio recordings were used to construct the database, with each audio categorized and indexed to establish ground truth plagiarism pairs. Among these, 14 query audios had corresponding plagiarized versions in the database, while the remaining non-paired audios were included to test the system's ability to differentiate unrelated songs. Some audios were modified using pitch shifting and tempo changes to test the robustness of the system against common transformations used to disguise plagiarism.

The implementation was developed and executed in Visual Studio Code (VS Code), using Python as the primary programming language. The system relied on libraries such as Librosa for feature extraction, NumPy and Pandas for data handling, and supporting modules for similarity computation and performance evaluation. For textual analysis, the system integrated with the Meta-Llama/LLaMA-3.3-8B-Instruct:Free model via the OpenRouter API, which generated narrative reports to complement numerical results with expert-level interpretations.



Results:

DatabaseAudio	01_Schld_You won't be there for me	02_Stay With Me - Sam Smith	03_If I Could Fly	08_Alicia Keys_If I Ain't Got You	10_Dua Lipa - Levitating	11_Negaraku	11_I Shall Return	12_Olly Murs - Dance With Me Tonight	13_Paramore - Misery Business
Best Rank #	1	1	-	2	1	5	3	2	-
TestAudio	01_Green tea honey	02_Tom Petty- I Wont Back Down(downpitched)	03_Coldplay - Viva la Vida	08_slow_alicia keys	10_Live Your Life	11_Krontjong Orchest Eurasia - Terang Boelan (1928)	11_Malayan Moon	12_Meghan Trainor - Dear Future Husband	13_Olivia Rodrigo - good 4 u
Plagiarism Type	Real case	Real case	Real case	Manual	Real case	Real case	Real case	Real case	Real case

DatabaseAudio	14_Photograph -Ed Sheeran	15_Sweet Little Sixteen	16_Robin Thicke - Blurred Lines ft. T.I., Pharrell	17_The Verve - Bitter Sweet Symphony	19_Gloria Gaynor - I Will Survive	20_Niccolo Paganini - La campanella
Best Rank #	2	3	3	1	3	-
TestAudio	14_fast_photograph	15_The Beach Boys - Surfin USA	16_Marvin Gaye - Got to give it up	17_The Last Time - The Andrew Oldham Orchestra	19_IVE - After LIKE Instrumental	20_BLACKPINK - Shut Down Instrumental
Plagiarism Type	Manual	Real case	Real case	Real case	Real case	Real case

Figure 6.2: Screenshot of System Results Table

A detailed results table was prepared to document the outcome of each query database audio pair comparison. The table is organized into four main columns:

- **Database Audio** – the reference tracks stored in the system's database, serving as the baseline for comparison.
- **Best Rank** – the highest position achieved by the corresponding plagiarism pair in the similarity ranking (e.g., ranked 1st, 2nd, 3rd, or not retrieved).
- **Test Audio** – the query audio input provided to the system, which may be an original track, a pitch-shifted variant, or a tempo-modified version to evaluate robustness.
- **Plagiarism Type** – an indicator column specifying whether the test audio corresponds to a real case (naturally plagiarized audio from the dataset) or an artificially created variant (generated through transformations such as pitch shifting or time stretching).

To enhance interpretability, the table uses green cells to indicate successfully detected plagiarism within the Top-3 ranks, while red cells highlight failures, where the system missed or misranked the plagiarized pair. This visual distinction provides an immediate overview of detection successes and shortcomings.

The summarized results revealed that out of 14 plagiarized queries, 11 were successfully retrieved within the top three rankings, corresponding to a Top-3 Accuracy of 78.6%. The rank distribution (4 at first place, 2 at second, and 4 at third) contributed to an overall MRR of 0.45, validating the system's effectiveness while also highlighting areas for improvement.

6.3 Project Challenges

Throughout the development of the music plagiarism detection system, several challenges were encountered that influenced both design decisions and implementation outcomes. One major challenge was ensuring robustness against tempo and pitch variations, as conventional similarity measures were highly sensitive to differences in musical key or tempo. This required the integration of key-invariant and beat-based approaches to achieve consistent comparisons. Another significant difficulty was balancing accuracy and computational efficiency. The extraction of multiple features from large audio databases introduced high memory and processing demands, which were mitigated through segmentation limits, compressed storage, and optimized workflows. Additionally, beat tracking inaccuracies in noisy or irregular recordings posed difficulties in segmentation, necessitating fallback methods to maintain system reliability. Finally, the integration of LLM for textual analysis presented network and dependency challenges, requiring exception handling to ensure uninterrupted operation. Despite these obstacles, the challenges ultimately guided the refinement of the system, leading to a more resilient and scalable solution.

6.4 Objectives Evaluation

The main objective of this project was to develop a sophisticated system for identifying musical signatures and detecting plagiarism in audio recordings. Based on the implementation and testing outcomes, the system has successfully addressed this overarching goal by integrating a multi-feature extraction pipeline and a robust similarity comparison framework. The EnhancedPlagiarismDetector class enables the analysis of audio signals through chroma, MFCCs, spectral, rhythmic, tonal, and symbolic features such as motifs and harmonic progressions, thereby capturing diverse aspects of musical signatures.

With respect to the first sub-objective of improving the precision of similarity measurements, the system incorporates weighted similarity scoring and key-invariant cosine similarity for chroma features, along with Jaccard-based set comparisons for motifs and harmonic progressions. These mechanisms enhance robustness against pitch shifts and tempo variations, ensuring that the system can detect both exact and approximate overlaps. The decision rules, which combine multiple similarity thresholds and segment coverage ratios, further reduce the likelihood of false positives and false negatives, thereby meeting the requirement for higher precision.

In relation to the second sub-objective of enhancing the efficiency of audio extraction and comparison, the system employs beat-based segmentation with overlap control and compressed storage of extracted features in .npz format. By limiting the maximum number of segments per song and separating numerical and symbolic features, the system reduces both processing time and memory usage. This configuration enables the system to process large audio databases more effectively, without significantly compromising accuracy.

It should also be noted that, as outlined in the project scope, the system does not perform vocal-instrumental separation. Instead, similarity analysis is conducted on the full audio signal, simplifying the processing pipeline while still providing reliable similarity detection. This design choice aligns with the stated objective of focusing on overall musical similarity rather than component-level analysis.

6.5 Concluding Remark

In summary, the current system demonstrates that the identified objectives have been largely achieved. Precision in similarity measurement has been strengthened through multi-feature analysis and invariant comparison methods, while efficiency has been improved through optimized segmentation and storage strategies. The system thus fulfills its role as a reliable and practical tool for detecting potential plagiarism in audio recordings, contributing towards safeguarding creative works in the music industry.

Chapter 7
Conclusion and Recommendations

7.1 Conclusion

This project was motivated by the growing concern of intellectual property safeguarding in the music industry, where plagiarism detection has become increasingly vital due to vast availability of digital audio data and the ease of unauthorized replication. The main objective was to develop a sophisticated system capable of identifying musical signatures and detecting plagiarism in audio recordings with improved accuracy and efficiency.

The completed system successfully integrates a multi-feature extraction pipeline, encompassing chroma, MFCC, spectral, tonal, rhythmic, and symbolic features such as motifs and harmonic progressions. These diverse features allow the system to capture multiple dimensions of musical similarity, going beyond traditional fingerprinting methods that focus solely on spectral peaks. By incorporating key-invariant similarity measures, beat-based segmentation, and weighted scoring, the system demonstrates robustness towards pitch shifts, tempo variations, and subtle overlaps between musical pieces. Furthermore, efficiency was achieved through optimized segmentation strategies, compressed feature storage, and scalable database management, enabling the system to handle larger datasets without significant computational delays.

Although several challenges were encountered, including beat tracking inaccuracies, high computational load, and external dependency issues, the solutions implemented such as fallback segmentation methods, segment limits, error handling, and API integration strategies contributed to a more stable and reliable system. Overall, the project has met its intended objectives and produced a functional plagiarism detection framework that provides both quantitative similarity scores and interpretive analysis, thereby aiding music experts in making more informed claims about potential plagiarism cases.

7.2 Recommendations

While the current system has demonstrated its effectiveness, there are opportunities for further enhancement. One recommendation is to explore deep learning-based feature extraction methods, such as convolutional or recurrent neural networks, which could capture higher-level representations of music and potentially improve detection accuracy in complex cases. Additionally, incorporating GPU acceleration and parallel processing would further reduce computation time, especially for large-scale audio databases. Future work may also consider extending the system to handle polyphonic vocal-instrumental separation, allowing more fine-grained analysis of melodic and harmonic structures. Another potential improvement is the integration of a user-friendly graphical interface, which would make the system more user-friendly for non-technical individuals, such as music educators, legal professionals, and artists. Finally, expanding the database with a broader variety of musical genres and cultural styles would enhance the system's generalizability and applicability across diverse contexts.

In summary, the project provides a significant step toward addressing the challenges of music plagiarism detection. By combining methodological rigor with practical implementation, the system contributes to safeguarding creative works and maintaining fairness in the music industry. With further refinements and extensions, it holds strong potential to evolve into a comprehensive tool for protecting artistic integrity in the digital age.

REFERENCES

[1] Ambadaradmin, “The measure of plagiarism in music or song - am badar,” Am Badar, May 02, 2024. https://ambadar.com/insights/copyright/the-measure-of plagiarism-in-music-or-song/

[2] Y. Zheng, J. Liu, and S. Zhang, “An infringement detection system for videos based on audio fingerprint technology,” 2020 International Conference on Culture-oriented Science & Technology (ICCST), Oct. 2020, doi: 10.1109/iccst50977.2020.00066.

[3] N. Borkar, S. Patre, R. S. Khalsa, R. Kawale, and P. Chakurkar, “Music Plagiarism Detection using Audio Fingerprinting and Segment Matching,” 2021 Smart Technologies, Communication and Robotics (STCR), Oct. 2021, doi: 10.1109/stcr51658.2021.9587927.

[4] V. Khatri, L. Dillingham, Z. Chen, and Dept. of Electrical and Computer Engineering University of Rochester Rochester, NY 14627, USA, “SONG RECOGNITION USING AUDIO FINGERPRINTING,” SONG RECOGNITION USING AUDIO FINGERPRINTING, [Online]. Available: https://hajim.rochester.edu/ece/sites/zduan/teaching/ece472/projects/2019/AudioFingerprinting.pdf

[5] M. Müller and S. Ewert, “Chroma Toolbox: MATLAB implementations for extracting variants of Chroma-based audio features,” *International Symposium/Conference on Music Information Retrieval*, pp. 215–220, Oct. 2011, doi: 10.5281/zenodo.1416032.

[6] M. Muller and F. Kurth, "Enhancing Similarity Matrices for Music Audio Analysis," 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, Toulouse, France, 2006, pp. V-V, doi: 10.1109/ICASSP.2006.1661199.

[7] Shah,Ayush & Kattel, Manasi & Nepal, Araju & Shrestha, D.. *Chroma Feature Extraction*. 2019. [Online]. Available: https://www.researchgate.net/publication/330796993_Chroma_Feature_Extraction

[8] B. Logan, “Mel frequency cepstral coefficients for music modeling,” in *Proc. Int. Symp. Music Inf. Retrieval (ISMIR)*, 2000.

[9] G. Tzanetakis and P. Cook, “Musical genre classification of audio signals,” *IEEE Trans. Speech Audio Process.*, vol. 10, no. 5, pp. 293–302, Jul. 2002.

[10] P. Grosche and M. Müller, “Extracting predominant local pulse information from music recordings,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 19, no. 6, pp. 1688–1701, Aug. 2011.

[11] R.Typke, F.Wiering, and R.Veltkamp, “A Survey of Music Information Retrieval Systems.” presented at the ISMIR 2005, 6th International Conference on Music Information Retrieval, London, UK, 11-15 September 2005,

[12] F. Pachet, P. Roy, and S. Ab, “Plagiarism risk detector and interface” U.S. Patent US11289059B2, May 23, 2019. https://patents.google.com/patent/US11289059B2/en?oq=US+11%2c289%2c059+B2 +

[13] M. Müller, F. Kurth, and M. Clausen, “Audio matching via Chroma-Based statistical features.,” *International Symposium/Conference on Music Information Retrieval*, pp. 288–295, Jan. 2005, [Online]. Available: http://ismir2005.ismir.net/proceedings/1019.pdf

[14] Wei, Z., Yao, Z., Su, Q., Lian, X. et al., "A Deviation-Based Centroid Displacement Method for Combustion Parameters Acquisition," SAE Technical Paper 2024-01-2839, 2024, https://doi.org/10.4271/2024-01-2839.

[15] A.Ayzenberg et al. (2023). Chordal Embeddings Based on Topology of the Tonal Space. 10.1007/978-3-031-29956-8_2

APPENDIX

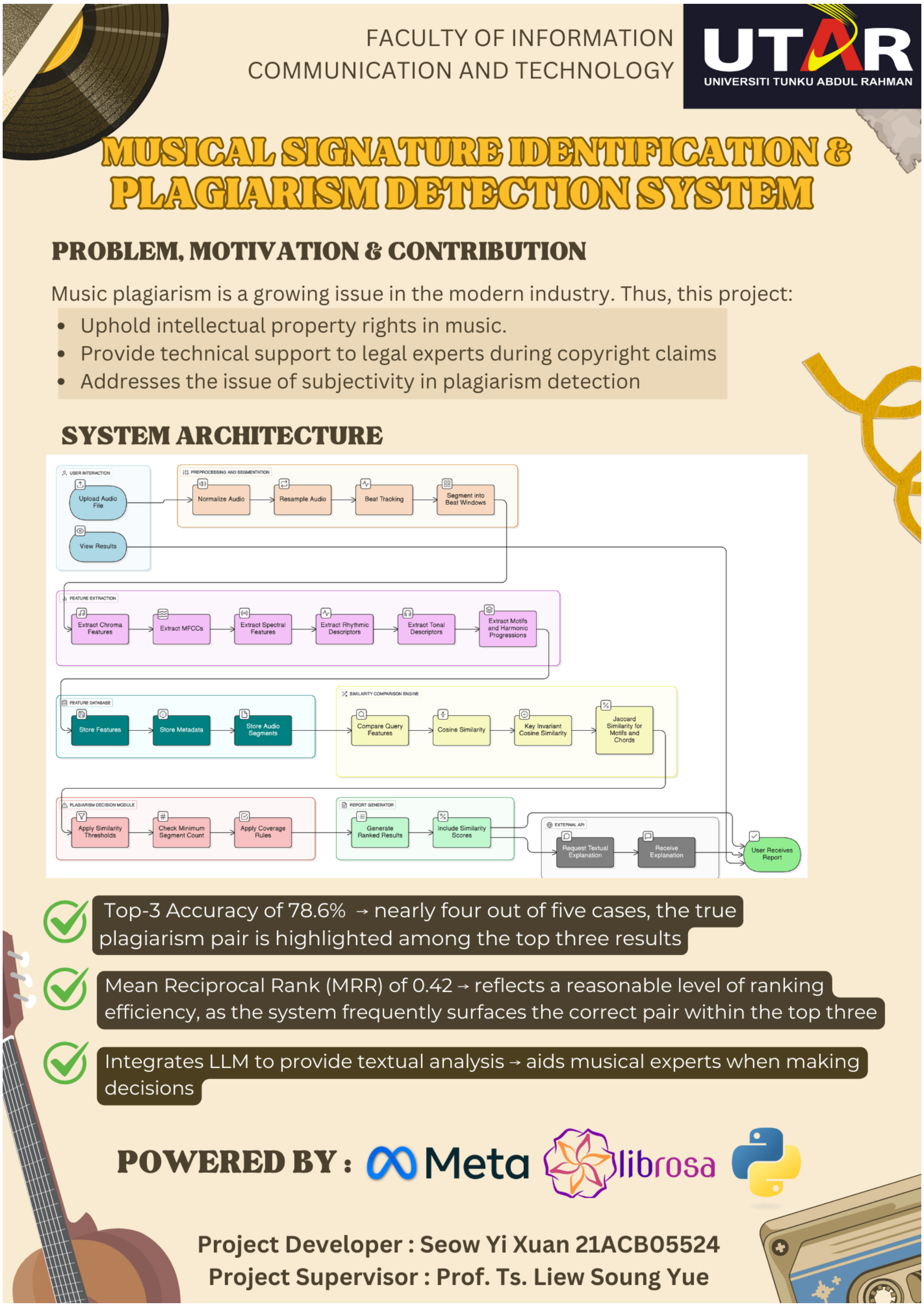
```
class EnhancedPlagiarismDetector:
    def __init__(self, target_bmp=120, sr=22050, database_folder="./Database_audios"):
        self.target_bpm = target_bmp
        self.sr = sr
        self.database_folder = database_folder

        # Create directories
        self.output_dirs = {
            'segments': './segments',   #store segments as wav
            'features': './features',   #store features as text file

            'query_segments': './query_segments',
            'query_features': './query_features'
        }

        for dir_path in self.output_dirs.values():
            os.makedirs(dir_path, exist_ok=True)


    #  Motif Extraction (pitch n-grams)
    def extract_motifs(self, y, sr, n=4):
        try:
            chroma = librosa.feature.chroma_cens(y=y, sr=sr)  # extract CENS
            pitches = np.argmax(chroma, axis=0)              # get the dominant pitch for the frame

            note_names = ["C","C#","D","D#","E","F","F#","G","G#","A","A#","B"]
            sequence = [note_names[p] for p in pitches]         # convert to note names

            # Create n-grams with different sizes for robustness
            all_ngrams = set()

            for ngram_size in [3, 4, 5]:  # Multiple n-gram sizes
                # generate motifs (tuples of notes_names)
                ngrams = {tuple(sequence[i:i+ngram_size]) for i in range(len(sequence)-
ngram_size+1)}
                all_ngrams.update(ngrams)
            return all_ngrams

        except Exception as e:
            print(f"⚠ Motif extraction failed: {e}")
            return set()

    def compare_motifs(self, motifs1, motifs2):
        """Compare motif similarity using Jaccard similarity"""
        if not motifs1 or not motifs2:
            return 0.0
        overlap = len(motifs1 & motifs2)
        union = len(motifs1 | motifs2)
        return overlap / union if union > 0 else 0.0


    def extract_harmonic_progression(self, y, sr, n=4):
        """Extract harmonic progressions as chord n-grams"""
        try:
            chroma = librosa.feature.chroma_cens(y=y, sr=sr)

            # Enhanced chord templates (major and minor triads)
            maj_templates = np.array([
                [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0],  # C major
                [0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],  # C# major
                [0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0],  # D major
                [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0],  # D# major
                [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1],  # E major
                [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0],  # F major
                [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0],  # F# major
                [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1],  # G major
                [1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0],  # G# major
                [0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],  # A major
                [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0],  # A# major
                [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1],  # B major
            ]).T

            min_templates = np.array([
                [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],  # C minor
                [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],  # C# minor
                [0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0],  # D minor
                [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0],  # D# minor
                [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1],  # E minor
                [1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],  # F minor
                [0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],  # F# minor
                [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # G minor
                [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1],  # G# minor
                [1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],  # A minor
                [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0],  # A# minor
                [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1],  # B minor
            ]).T

            chord_templates = np.concatenate([maj_templates, min_templates], axis=1)
            chord_labels = [
                f"{note}" for note in ["C","C#","D","D#","E","F","F#","G","G#","A","A#","B"]
            ] + [
                f"{note}m" for note in ["C","C#","D","D#","E","F","F#","G","G#","A","A#","B"]
            ]

            progression = []
            for i in range(chroma.shape[1]):
                frame = chroma[:, i]
                if np.sum(frame) > 0:  # Avoid silent frames
                    frame_norm = frame / np.sum(frame)  # Normalize
                    sims = chord_templates.T @ frame_norm
                    best_idx = np.argmax(sims)
                    progression.append(chord_labels[best_idx])

            # Smooth progression (remove rapid changes)
            if progression:
                smoothed = [progression[0]]
                for chord in progression[1:]:
                    if chord != smoothed[-1]:
                        smoothed.append(chord)

                # Create n-grams with different sizes
                all_ngrams = set()
                for ngram_size in [3, 4, 5]:  # Multiple n-gram sizes
                    if len(smoothed) >= ngram_size:
                        ngrams = {tuple(smoothed[i:i+ngram_size]) for i in range(len(smoothed)-
ngram_size+1)}
                        all_ngrams.update(ngrams)

                return smoothed, all_ngrams

            return [], set()

        except Exception as e:
            print(f"⚠ Harmonic extraction failed: {e}")
            return [], set()

    def compare_harmonic_progressions(self, ngrams1, ngrams2):
        """Compare harmonic progression similarity using Jaccard similarity"""
        if not ngrams1 or not ngrams2:
            return 0.0
        overlap = len(ngrams1 & ngrams2)
        union = len(ngrams1 | ngrams2)
        return overlap / union if union > 0 else 0.0


    def extract_multi_features(self, y, sr):
        """Extract multiple features including new motif and harmonic features"""
        features = {}

        try:
            # Original features
            features['cens'] = librosa.feature.chroma_cens(y=y, sr=sr)
            features['cqt'] = librosa.feature.chroma_cqt(y=y, sr=sr)
            # MFCC stack: static + delta + delta2 for timbre dynamics
            mfcc_static = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
            mfcc_delta = librosa.feature.delta(mfcc_static)
            mfcc_delta2 = librosa.feature.delta(mfcc_static, order=2)
            features['mfcc'] = {
                'static': mfcc_static,
                'delta': mfcc_delta,
                'delta2': mfcc_delta2,
            }

            features['spectral_contrast'] = librosa.feature.spectral_contrast(y=y, sr=sr)
            features['zcr'] = librosa.feature.zero_crossing_rate(y)

            # Background and rhythm features (HPSS → percussive/harmonic)
            # spilt the audio into harmonic (piano) and percussive (drums) components
            try:
                y_harm, y_perc = librosa.effects.hpss(y)
            except Exception:
                y_harm, y_perc = y, y

            # Rhythmic patterns
            onset_env = librosa.onset.onset_strength(y=y_perc, sr=sr) # detect where drum
happends
            tempogram = librosa.feature.tempogram(onset_envelope=onset_env, sr=sr) #tempo
changes
            onset_ac = librosa.autocorrelate(onset_env, max_size=min(512, onset_env.shape[0]))
#repetitve patterns
            features['tempogram'] = tempogram
            features['onset_ac'] = onset_ac

            # Tonal background relations
            features['tonnetz'] = librosa.feature.tonnetz(y=y_harm, sr=sr)

            # Spectral texture stats
            features['centroid'] = librosa.feature.spectral_centroid(y=y, sr=sr) #brightness
            features['rolloff'] = librosa.feature.spectral_rolloff(y=y, sr=sr)   #sharpness
            features['bandwidth'] = librosa.feature.spectral_bandwidth(y=y, sr=sr) #spread
            features['flatness'] = librosa.feature.spectral_flatness(y=y) #tonal/noisy

            # Motif features
            motifs = self.extract_motifs(y, sr)
            features['motifs'] = motifs

            # Harmonic progression features
            progression, harmonic_ngrams = self.extract_harmonic_progression(y, sr)
            features['harmonic_progression'] = progression
            features['harmonic_ngrams'] = harmonic_ngrams

        except Exception as e:
            print(f"Warning: Feature extraction error: {e}")
            # Fallback to basic CENS only
            features['cens'] = librosa.feature.chroma_cens(y=y, sr=sr)
            try:
                features['cqt'] = librosa.feature.chroma_cqt(y=y, sr=sr)
            except Exception:
                features['cqt'] = features['cens']
            features['mfcc'] = {}
            features['motifs'] = set()
            features['harmonic_ngrams'] = set()
            # Minimal placeholders for added features
            features['tempogram'] = np.zeros((12, 1))
            features['onset_ac'] = np.zeros((1,))
            features['tonnetz'] = np.zeros((6, 1))
            features['centroid'] = np.zeros((1, 1))
            features['rolloff'] = np.zeros((1, 1))
            features['bandwidth'] = np.zeros((1, 1))
            features['flatness'] = np.zeros((1, 1))

        return features

    def process_audio_optimized(self, audio_path, is_query=False, max_segments=50):
        """Optimized audio processing with segment limits"""
        print(f"Processing: {os.path.basename(audio_path)}")

        try:
            y, sr = librosa.load(audio_path, sr=self.sr, mono=True)
            y = librosa.util.normalize(y)
        except Exception as e:
            print(f"❌ Failed to load {audio_path}: {e}")
            return []

        # Beat tracking
        try:
            tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)
            beat_times = librosa.frames_to_time(beat_frames, sr=sr)
        except Exception as e:
            print(f"Warning: Beat tracking failed, using time-based segmentation")
            beat_times = np.arange(0, len(y)/sr, 0.5)

        base_name = os.path.splitext(os.path.basename(audio_path))[0]
        segments_data = []

        # Segment configuration
        if is_query:
            segment_lengths = [16] # 16 beat segments for queries
            overlap_ratios = [0.5] # 50% overlap for queries
        else:
            segment_lengths = [8, 16, 24] # 8, 16, 24 beat segments for database
            overlap_ratios = [0.3] # 30% overlap for database

        segment_count = 0

        for seg_len in segment_lengths:
            # determine how far the window slides forward
            hop_size = max(1, int(seg_len * (1 - overlap_ratios[0])))

            # looping through beats to create segments
            for i in range(0, len(beat_times) - seg_len + 1, hop_size):
                if segment_count >= max_segments:
                    break

                # extract segment
                start_sample = int(beat_times[i] * sr)
                end_sample = int(beat_times[min(i + seg_len, len(beat_times) - 1)] * sr)

                segment = y[start_sample:end_sample]

                # Skip too short segments (less than 1 sec)
                if len(segment) < sr * 1.0:
                    continue

                try:
                    features = self.extract_multi_features(segment, sr)

                    segment_id = f"{base_name}_len{seg_len}_seg{segment_count}"

                    # Save features (handle sets separately)
                    if is_query:
                        feature_path = os.path.join(self.output_dirs['query_features'],
    f"{segment_id}.npz")
                        audio_path_seg = os.path.join(self.output_dirs['query_segments'],
    f"{segment_id}.wav")
                    else:
                        feature_path = os.path.join(self.output_dirs['features'], f"{segment_id}.npz")
                        audio_path_seg = os.path.join(self.output_dirs['segments'],
    f"{segment_id}.wav")

                    # Save numerical features as npz (numpy compressed)
                    numerical_features = {}
                    for k, v in features.items():
                        if k == 'motifs' or k == 'harmonic_ngrams':
                            continue
                        if k == 'mfcc' and isinstance(v, dict):
                            if 'static' in v and v['static'] is not None:
                                numerical_features['mfcc_static'] = v['static']
                            if 'delta' in v and v['delta'] is not None:
                                numerical_features['mfcc_delta'] = v['delta']
                            if 'delta2' in v and v['delta2'] is not None:
                                numerical_features['mfcc_delta2'] = v['delta2']
                            continue
                        if isinstance(v, (set, list)):
                            continue
                        numerical_features[k] = v
                    np.savez_compressed(feature_path, **numerical_features)

                    # Save set-based features as text
                    motifs_path = feature_path.replace('.npz', '_motifs.txt')
                    harmonic_path = feature_path.replace('.npz', '_harmonic.txt')

                    with open(motifs_path, 'w') as f:
                        for motif in features.get('motifs', set()):
                            f.write(','.join(motif) + '\n')

                    with open(harmonic_path, 'w') as f:
                        for ngram in features.get('harmonic_ngrams', set()):
                            f.write(','.join(ngram) + '\n')

                    sf.write(audio_path_seg, segment, sr)

                    segments_data.append({
                        'song_name': base_name,
                        'segment_id': segment_id,
                        'segment_length': seg_len,
                        'start_beat': i,
                        'end_beat': i + seg_len,
                        'duration': len(segment) / sr,
                        'feature_path': feature_path,
                        'motifs_path': motifs_path,
                        'harmonic_path': harmonic_path,
                        'audio_path': audio_path_seg
                    })

                    segment_count += 1

                except Exception as e:
                    continue

            if segment_count >= max_segments:
                break

        print(f"✅ Extracted {len(segments_data)} segments")
        return segments_data

        def load_set_features(self, file_path):
        """Load set-based features from text file"""
        features = set()
```

```
        try:
            if os.path.exists(file_path):
                with open(file_path, 'r') as f:
                    for line in f:
                        line = line.strip()
                        if line:
                            features.add(tuple(line.split(',')))
        except Exception as e:
            pass
        return features

    def compute_similarity_enhanced(self, features1, features2, paths1, paths2):

        # Original feature weights
        weights = {
            'cens': 0.18,              # Harmonic content (CENS) - instant harmony detection
            'cqt': 0.18,               # Harmonic content (CQT chroma)
            'mfcc': 0.12,              # Timbre with dynamics
            'spectral_contrast': 0.05,   # Frequency distribution
            'zcr': 0.03,               # Signal noisiness
            'tonnetz': 0.08,           # Tonal relations background
            'tempogram': 0.12,          # Rhythmic pattern
            'onset_ac': 0.06,           # Beat periodicity
            'centroid': 0.03,           # Spectral brightness
            'rolloff': 0.03,
            'bandwidth': 0.03,
            'flatness': 0.02,
            'motifs': 0.04,            # Melodic patterns - pattern
            'harmonic_ngrams': 0.03      # Chord level - Chord progressions
        }

        similarities = []
        total_weight = 0
        per_feature_scores = {}     # for storing top-3 max feature similarities

        # normalise vectors - to compare patterns insread of magnitudes
        def _unit_norm(vec):
            if hasattr(vec, 'ndim') and vec.ndim > 1:
                vec = vec.reshape(-1)
            vec = np.asarray(vec, dtype=float)
            norm = np.linalg.norm(vec) + 1e-12
            return vec / norm

        # compares chroma vectors - tries all 12 possible shifts of 1 chroma and picks the best
match
        def _max_key_invariant_cosine(chroma_vec1, chroma_vec2):
            """Cosine similarity under optimal circular shift (key invariance) for 12-d chroma."""
            v1 = _unit_norm(chroma_vec1)
            v2 = _unit_norm(chroma_vec2)
            if v1.shape[0] != 12 or v2.shape[0] != 12:
                return float(np.dot(v1, v2))
            best = -1.0
            for shift in range(12):
                v2s = np.roll(v2, shift)
                sim = float(np.dot(v1, v2s))
                if sim > best:
                    best = sim
            return best

        # Handle numerical features
        for feature_name, weight in weights.items():

            # handle motif and harmonic features later
            if feature_name in ['motifs', 'harmonic_ngrams']:
                continue  # Handle separately

            # handle cens, cqt, mfcc, rhythm/texture, spectral_contrast, zcr
            if feature_name in features1 and feature_name in features2:
                try:
                    if feature_name in ['cens', 'cqt']:
                        f1 = features1[feature_name]
                        f2 = features2[feature_name]
                        f1_chroma = np.mean(f1, axis=1) if hasattr(f1, 'ndim') and f1.ndim > 1 else f1
                        f2_chroma = np.mean(f2, axis=1) if hasattr(f2, 'ndim') and f2.ndim > 1 else f2
                        if len(f1_chroma) == 0 or len(f2_chroma) == 0:
                            continue
                        sim = _max_key_invariant_cosine(f1_chroma, f2_chroma)
                    elif feature_name == 'mfcc':
                        # Support both in-memory dict and saved arrays (mfcc_static, mfcc_delta,
mfcc_delta2)
                        def agg_mfcc_from_source(src):
                            if isinstance(src.get('mfcc', None), dict):
                                mfcc_dict = src['mfcc']
                            else:
                                mfcc_dict = {}
                                if 'mfcc_static' in src:
                                    mfcc_dict['static'] = src['mfcc_static']
                                if 'mfcc_delta' in src:
                                    mfcc_dict['delta'] = src['mfcc_delta']
                                if 'mfcc_delta2' in src:
                                    mfcc_dict['delta2'] = src['mfcc_delta2']
                            parts = []
                            for key in ['static', 'delta', 'delta2']:
                                if key in mfcc_dict and mfcc_dict[key] is not None and
hasattr(mfcc_dict[key], 'ndim'):
                                    m = np.mean(mfcc_dict[key], axis=1)
                                    s = np.std(mfcc_dict[key], axis=1)
                                    parts.extend([m, s])
                            if not parts:
                                return None
                            return np.concatenate(parts, axis=0)
                        f1_vec = agg_mfcc_from_source(features1)
                        f2_vec = agg_mfcc_from_source(features2)
                        if f1_vec is None or f2_vec is None:
                            continue
                        d = min(len(f1_vec), len(f2_vec))
                        f1_vec = _unit_norm(f1_vec[:d])
                        f2_vec = _unit_norm(f2_vec[:d])
                        sim = float(np.dot(f1_vec, f2_vec))
                    elif feature_name == 'tempogram':
                        f1 = features1[feature_name]
                        f2 = features2[feature_name]
                        f1_prof = np.mean(f1, axis=1) if hasattr(f1, 'ndim') and f1.ndim > 1 else f1
                        f2_prof = np.mean(f2, axis=1) if hasattr(f2, 'ndim') and f2.ndim > 1 else f2
                        f1_prof = _unit_norm(f1_prof)
                        f2_prof = _unit_norm(f2_prof)
                        sim = float(np.dot(f1_prof, f2_prof))
                    elif feature_name == 'onset_ac':
                        f1 = _unit_norm(features1[feature_name])
                        f2 = _unit_norm(features2[feature_name])
                        d = min(len(f1), len(f2))
                        sim = float(np.dot(f1[:d], f2[:d]))
                    elif feature_name == 'tonnetz':
                        f1 = features1[feature_name]
                        f2 = features2[feature_name]
                        f1_flat = np.mean(f1, axis=1) if hasattr(f1, 'ndim') and f1.ndim > 1 else f1
                        f2_flat = np.mean(f2, axis=1) if hasattr(f2, 'ndim') and f2.ndim > 1 else f2
                        f1_flat = _unit_norm(f1_flat)
                        f2_flat = _unit_norm(f2_flat)
                        sim = float(np.dot(f1_flat, f2_flat))
                    else:
                        f1 = features1[feature_name]
                        f2 = features2[feature_name]
                        f1_flat = np.mean(f1, axis=1) if hasattr(f1, 'ndim') and f1.ndim > 1 else f1
                        f2_flat = np.mean(f2, axis=1) if hasattr(f2, 'ndim') and f2.ndim > 1 else f2
                        min_len = min(len(f1_flat), len(f2_flat)) if hasattr(f1_flat, '__len__') and
hasattr(f2_flat, '__len__') else 0
                        if min_len == 0:
                            continue
                        f1_flat = np.array(f1_flat[:min_len], dtype=float)
                        f2_flat = np.array(f2_flat[:min_len], dtype=float)
                        if np.any(np.isnan(f1_flat)) or np.any(np.isnan(f2_flat)):
                            continue
                        if np.all(f1_flat == 0) or np.all(f2_flat == 0):
                            continue
                        f1_norm = _unit_norm(f1_flat)
                        f2_norm = _unit_norm(f2_flat)
                        sim = float(np.dot(f1_norm, f2_norm))

                    if not np.isnan(sim) and not np.isinf(sim):
                        per_feature_scores[feature_name] = sim  #store into dict
                        similarities.append(sim * weight)
                        total_weight += weight
                except Exception as e:
                    continue

            # Handle motif features
            try:
                motifs1 = self.load_set_features(paths1['motifs_path'])
                motifs2 = self.load_set_features(paths2['motifs_path'])
                motif_sim = self.compare_motifs(motifs1, motifs2)

                if motif_sim > 0:
                    per_feature_scores['motifs'] = motif_sim
                    similarities.append(motif_sim * weights['motifs'])
                    total_weight += weights['motifs']
            except Exception as e:
                pass

            # Handle harmonic features
            try:
                harmonic1 = self.load_set_features(paths1['harmonic_path'])
                harmonic2 = self.load_set_features(paths2['harmonic_path'])
                harmonic_sim = self.compare_harmonic_progressions(harmonic1, harmonic2)

                if harmonic_sim > 0:
                    per_feature_scores['harmonic_ngrams'] = harmonic_sim
                    similarities.append(harmonic_sim * weights['harmonic_ngrams'])
                    total_weight += weights['harmonic_ngrams']
            except Exception as e:
                pass

            if total_weight == 0:
                return 0.0, {}

            overall_similarity = sum(similarities) / total_weight
            return overall_similarity, per_feature_scores

        def build_database_optimized(self, rebuild=True, max_segments_per_song=30):
            """Build database with segment limits per song"""
            database_csv = "enhanced_music_database.csv"

            if os.path.exists(database_csv) and not rebuild:
                print(f"Database {database_csv} already exists. Use rebuild=True to recreate.")
                return database_csv
            print(f"Building enhanced database from {self.database_folder}...")

            audio_files = []
            for root, dirs, files in os.walk(self.database_folder):
                for file in files:
                    if file.lower().endswith(('.wav', '.mp3', '.m4a')):
                    audio_files.append(os.path.join(root, file))

            print(f"Found {len(audio_files)} audio files")

            all_segments = []

            for i, file_path in enumerate(audio_files, 1):
                print(f"[{i}/{len(audio_files)}] Processing {os.path.basename(file_path)}...")
                try:
                    segments = self.process_audio_optimized(
                    file_path,
                    is_query=False,
                    max_segments=max_segments_per_song   # trade-off between accuracy and
efficiency
                    )
                    all_segments.extend(segments)
                except Exception as e:
                    print(f"  ❌ Failed: {e}")
                    continue

            df = pd.DataFrame(all_segments)
            df.to_csv(database_csv, index=False)

            print(f"✅ Enhanced database built!")
            print(f"  📊 Total segments: {len(all_segments)}")
            print(f"  🎵 Average segments per song: {len(all_segments)/len(audio_files):.1f}")
            print(f"  💾 Saved to: {database_csv}")

            return database_csv

        def detect_plagiarism_enhanced(self, query_path, database_csv=None,
                                   similarity_threshold=0.85,
                                   plagiarism_threshold=0.2,
                                   min_matching_segments=3):

            if database_csv is None:
                database_csv = self.build_database_optimized()

            # Process query with limits
            query_segments = self.process_audio_optimized(query_path, is_query=True,
max_segments=50)

            if not query_segments:
                print("❌ Failed to extract query segments")
                return pd.DataFrame(), {}

            # Load database
            db_df = pd.read_csv(database_csv)
            print(f" Database loaded: {len(db_df)} segments from {db_df['song_name'].nunique()}
songs")

            total_comparisons = len(query_segments) * len(db_df)
            print(f" Total comparisons: {total_comparisons:,}")
            print(f" Using similarity threshold: {similarity_threshold}")

            all_matches = []
            start_time = time.time()
            comparisons_done = 0

            print(f"\n Starting enhanced comparison...")

            query_song_name = os.path.splitext(os.path.basename(query_path))[0]

            for q_idx, query_seg in enumerate(query_segments):
                try:
                    if not os.path.exists(query_seg['feature_path']):
                    continue

                    query_features = dict(np.load(query_seg['feature_path']))
                    query_paths = {
                    'motifs_path': query_seg['motifs_path'],
                    'harmonic_path': query_seg['harmonic_path']
                    }
                    segment_matches = 0

                    for _, db_row in db_df.iterrows():
                    comparisons_done += 1

                    # Skip comparing with same song
                    if db_row['song_name'] == query_song_name:
                        continue

                    try:
                        if not os.path.exists(db_row['feature_path']):
                            continue

                        db_features = dict(np.load(db_row['feature_path']))
                        db_paths = {
                            'motifs_path': db_row['motifs_path'],
                            'harmonic_path': db_row['harmonic_path']
                        }
                        similarity , _ = self.compute_similarity_enhanced(
                            query_features, db_features, query_paths, db_paths
                        )

                        if similarity >= similarity_threshold:
                            segment_matches += 1
                            all_matches.append({
                                'query_segment_id': query_seg['segment_id'],
                                'query_start_beat': query_seg['start_beat'],
                                'query_end_beat': query_seg['end_beat'],
                                'db_segment_id': db_row['segment_id'],
                                'db_song': db_row['song_name'],
                                'similarity': similarity,
                                'query_duration': query_seg['duration'],
                                'db_duration': db_row['duration'],
                                'query_feature_path': query_seg['feature_path'],
                                'db_feature_path': db_row['feature_path']
                            })

                    except Exception as e:
                        continue

                # Progress reporting
                if (q_idx + 1) % 5 == 0 or q_idx + 1 == len(query_segments):
                    elapsed = time.time() - start_time
                    progress = (q_idx + 1) / len(query_segments) * 100
                    speed = comparisons_done / elapsed if elapsed > 0 else 0
                    eta = (total_comparisons - comparisons_done) / speed if speed > 0 else 0

                    print(f" Segment {q_idx+1:2d}/{len(query_segments)} | "
                          f"Progress: {progress:5.1f}% | "
                          f"Matches: {segment_matches:2d} | "
                          f"Speed: {speed:4.0f}/sec | "
                          f"ETA: {eta/60:.1f}m")

            except Exception as e:
                continue

        total_time = time.time() - start_time
        print("=" * 60)
        print(f" Enhanced analysis completed in {total_time:.1f} seconds")
        print(f" Found {len(all_matches)} potential matches")

        # Analyze results
        matches_df = pd.DataFrame(all_matches)

        if matches_df.empty:
            print(" No matches above threshold")
            return matches_df, {}
        # Group by database song and analyze
        plagiarism_results = {}

        for db_song in matches_df['db_song'].unique():
            song_matches = matches_df[matches_df['db_song'] == db_song]

            total_query_segments = len(query_segments)
            matching_segments = len(song_matches)
            match_ratio = matching_segments / total_query_segments
            avg_similarity = song_matches['similarity'].mean()
            max_similarity = song_matches['similarity'].max()
            best_index = song_matches['similarity'].idxmax()
            best_match_row = song_matches.loc[best_index]

            # Enhanced plagiarism detection logic
            is_plagiarism = (
                (matching_segments >= min_matching_segments and avg_similarity >= 0.8) or
                (match_ratio >= plagiarism_threshold and avg_similarity >= 0.75)
            )

            # package the results
            plagiarism_results[db_song] = {
                'is_plagiarism': is_plagiarism,
                'matching_segments': matching_segments,
                'total_query_segments': total_query_segments,
                'match_ratio': match_ratio,
                'avg_similarity': avg_similarity,
                'max_similarity': max_similarity,
                'best_idx' : best_index,
                'best_row' : best_match_row
            }

        return matches_df, plagiarism_results

    def generate_report(self, matches_df, plagiarism_results, query_name):
        print(f"\n ENHANCED PLAGIARISM DETECTION REPORT")
        print(f" Query: {os.path.basename(query_name)}")
        print(f" Features Used: CENS, MFCC, Spectral Contrast, ZCR, Motifs, Harmonic
Progressions")

        if not plagiarism_results:
            print(" No potential plagiarism detected.")
            return

        def ranking_score(item):
            song, results = item
            # Primary : Match ratio (how much of query matches)
            # Secondary: Max similairty (how strong for a single match)
            # Tertiary : Average similarity (quality of matches)
            return (
                results['match_ratio'],
                results['max_similarity'],
                results['avg_similarity'],
            )

        sorted_results = sorted(plagiarism_results.items(), key=ranking_score, reverse=True)
        top_3 = sorted_results[:3]

        plagiarism_detected = False

        print(f"\n Ranked Results (Highest → Lowest Similarity):")
        for rank, (song, results) in enumerate(sorted_results, start=1):
            status = "🚨 PLAGIARISM DETECTED" if results['is_plagiarism'] else "✅ No
significant similarity"

            if results['is_plagiarism']:
                plagiarism_detected = True

                print(f"\n{rank}: {song}")
                print(f"   {status}")
                print(f"   📊 Matching Segments:
{results['matching_segments']}/{results['total_query_segments']}
({results['match_ratio']:.1%})")
                print(f"   🔄 Coverage Ratio: {results['match_ratio']:.1f}x")
                print(f"   📈 Average Similarity: {results['avg_similarity']:.3f}")
                print(f"   ⭐ Max Similarity: {results['max_similarity']:.3f}")
        return top_3



    def _call_model(self, model, api_key, prompt_obj):
        """
        Call OpenRouter API with the chosen model.
        model: "meta-llama/llama-3.3-8b-instruct:free"
        """
        url = "https://openrouter.ai/api/v1/chat/completions"
        headers = {
            "Authorization": f"Bearer {api_key}",
            "HTTP-Referer": "http://localhost:8000",  # required by OpenRouter, set to your
app/site
            "X-Title": "Music Plagiarism Detector",   # optional, descriptive title
            "Content-Type": "application/json"
        }
        body = {
            "model": model,
            "messages": [
                {
```

```
                "role": "system",
                "content": (
                    "You are a music plagiarism analysis assistant for experts. "
                    "Provide clear, detailed, and evidence-driven textual analysis "
                    "of musical segment similarities and differences. Do not assume "
                    "the top-ranked result is plagiarized. Instead, explain how melody, "
                    "rhythm, harmony, and timbre features contribute to similarity or "
                    "distinction, so users can form their own judgments."
                )
            },
            {"role": "user", "content": json.dumps(prompt_obj)}
        ],
        "temperature": 0.2
    }

    try:
        resp = requests.post(url, headers=headers, data=json.dumps(body), timeout=60)
        resp.raise_for_status()
        data = resp.json()
        return data["choices"][0]["message"]["content"].strip()
    except Exception as e:
        return f"[OpenRouter call failed] {e}"

def _create_compact_prompt(self, candidates):
    """Create ultra-compact prompt with only core similarity scores"""
    compact_candidates = []

    for candidate in candidates:
        scores = candidate['raw_scores']

        # Extract only the most essential scores
        compact_data = {
            "rank": candidate['rank'],
            "song": candidate['song'],
            "overall": {
                "max_similarity": round(candidate['overall']['max_similarity'], 3),
                "avg_similarity": round(candidate['overall']['avg_similarity'], 3),
                "match_ratio": round(candidate['overall']['match_ratio'], 3)
            },
            "scores": {
                "harmonic": round(scores.get('cens', 0), 3),       # harmonic similarity
            "rhythm": round(scores.get('tempogram', 0), 3),   # rhythmic pattern
            "timbre": round(scores.get('mfcc', 0), 3),        # timbre
            "spectral": round(scores.get('spectral_contrast', 0), 3),
            "motifs": round(scores.get('motifs', 0), 3),      # add motifs
            "harmonic_ngrams": round(scores.get('harmonic_ngrams', 0), 3) # chord
progression
            }
        }
```

```
            compact_candidates.append(compact_data)

        return {
            "task": "music_plagiarism_analysis",
            "candidates": compact_candidates,
            "guidance": "Analyze plagiarism evidence. For each candidate, explain how
harmonic, rhythmic, timbral, and spectral similarities support/weaken plagiarism claims.
Format: === Candidate X: [song] === [analysis]"
        }

    def analyze_top_matches(self, matches_df, top_3, api_key, model="meta-llama/llama-3.3-
8b-instruct:free", save_path="analysis_report.txt"):

        if matches_df.empty or not top_3:
            print("No matches to analyze.")
            return ""

        candidates = []
        for rank, (song, stats) in enumerate(top_3, start=1):
            # Filter matches for this song
            song_matches = matches_df[matches_df['db_song'] == song]
            if song_matches.empty:
                continue

            # Get best match row (highest similarity segment pair)
            best_match_idx = song_matches['similarity'].idxmax()
            best_match_row = song_matches.loc[best_match_idx]

            # Load features
            q = dict(np.load(best_match_row['query_feature_path']))
            d = dict(np.load(best_match_row['db_feature_path']))
            q_paths = {
                'motifs_path': best_match_row['query_feature_path'].replace('.npz', '_motifs.txt'),
                'harmonic_path': best_match_row['query_feature_path'].replace('.npz',
'_harmonic.txt'),
            }
            d_paths = {
                'motifs_path': best_match_row['db_feature_path'].replace('.npz', '_motifs.txt'),
                'harmonic_path': best_match_row['db_feature_path'].replace('.npz', '_harmonic.txt'),
            }

            # Recompute per-feature scores for this specific pair
            _, per_feature_scores = self.compute_similarity_enhanced(q, d, q_paths, d_paths)

            candidate_data = {
                "rank": rank,
                "song": song,
                "query_segment_id": str(best_match_row['query_segment_id']),
                "db_segment_id": str(best_match_row['db_segment_id']),
                "overall": {
                    "max_similarity": float(best_match_row['similarity']),
                    "avg_similarity": float(stats['avg_similarity']),
                    "match_ratio": float(stats['match_ratio'])
                },
                "raw_scores": per_feature_scores
            }
            candidates.append(candidate_data)

        # Create compact prompt
        compact_prompt = self._create_compact_prompt(candidates)

        print(f"Calling OpenRouter ({model}) with {len(candidates)} candidates...")

        model_text = self._call_model(model, api_key, compact_prompt)

        if model_text.startswith("[OpenRouter call failed]"):
            print("⚠ OpenRouter call failed. ")
        else:
            final_text = (
                f"Music Plagiarism Analysis - Top {len(candidates)} Candidates\n"
                + "="*60 + "\n\n"
                + model_text
            )

        print("\n" + final_text + "\n")

        try:
            with open(save_path, 'w', encoding='utf-8') as f:
                f.write(final_text)
            print(f"Saved analysis to: {save_path}")
        except Exception as e:
            print(f"Failed to write analysis: {e}")

        return final_text
```

POSTER



FACULTY OF INFORMATION COMMUNICATION AND TECHNOLOGY

UTAR UNIVERSITI TUNKU ABDUL RAHMAN

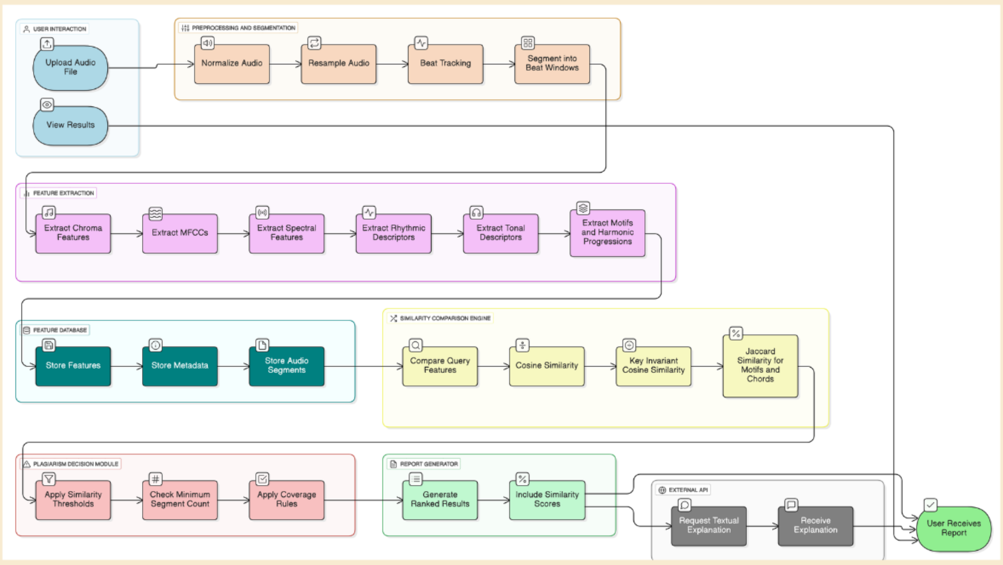
MUSICAL SIGNATURE IDENTIFICATION & PLAGIARISM DETECTION SYSTEM

PROBLEM, MOTIVATION & CONTRIBUTION

Music plagiarism is a growing issue in the modern industry. Thus, this project:

- Uphold intellectual property rights in music.
- Provide technical support to legal experts during copyright claims
- Addresses the issue of subjectivity in plagiarism detection

SYSTEM ARCHITECTURE


USER INTERACTION
Upload Audio File
View Results
PREPROCESSING AND SEGMENTATION
Normalize Audio
Resample Audio
Beat Tracking
Segment into Beat Windows
FEATURE EXTRACTION
Extract Chroma Features
Extract MFCCs
Extract Spectral Features
Extract Rhythmic Descriptors
Extract Tonal Descriptors
Extract Motifs and Harmonic Progressions
FEATURE DATABASE
Store Features
Store Metadata
Store Audio Segments
SIMILARITY COMPARISON ENGINE
Compare Query Features
Cosine Similarity
Key Invariant Cosine Similarity
Jaccard Similarity for Motifs and Chords
PLAGIARISM DECISION MODULE
Apply Similarity Thresholds
Check Minimum Segment Count
Apply Coverage Rules
REPORT GENERATOR
Generate Ranked Results
Include Similarity Scores
EXTERNAL API
Request Textual Explanation
Receive Explanation
User Receives Report

✓ Top-3 Accuracy of 78.6% → nearly four out of five cases, the true plagiarism pair is highlighted among the top three results

✓ Mean Reciprocal Rank (MRR) of 0.42 → reflects a reasonable level of ranking efficiency, as the system frequently surfaces the correct pair within the top three

✓ Integrates LLM to provide textual analysis → aids musical experts when making decisions

POWERED BY : Meta librosa

Project Developer : Seow Yi Xuan 21ACB05524
Project Supervisor : Prof. Ts. Liew Soung Yue