Digital Vinyl: Music Library with Embedded Audio Steganography and AI Classification

Masters Project Report

DSC 550-03: Master's Project

2025 Spring

Pranshu Acharya

02118866

Table of Contents

1. Executive Summary

This report details the design, implementation, and evaluation of a music library system featuring “Digital Vinyl" technology that embeds audio data within cover images using steganography techniques. The system combines this storage approach with intelligent genre and emotion classification through machine learning. This comprehensive solution addresses the challenges of music organization, discovery, and metadata preservation through an integrated approach combining signal processing, deep learning, and data embedding technologies.

The core contributions of this project include:

1. Development of a robust LSB steganography technique for embedding complete audio files within cover images
2. Implementation of a CNN-based genre classification system using the FMA dataset and MFCC features
3. Creation of a two-dimensional emotion recognition system analyzing valence and arousal from audio signals using the DEAM Dataset and Mel-Spectrographs
4. Integration of these components into a cohesive music library system with an intuitive user interface
5. Comprehensive error handling and optimization for real-world application scenarios

The following report provides detailed technical specifications, implementation approaches, evaluation metrics, and future directions for this innovative music library system.

2. Introduction and Background

2.1 Problem Statement

Traditional music libraries face several challenges:

- Limited metadata capabilities for emotional and genre-based classification
- Separation of audio data from visual representations
- Inefficient storage and retrieval mechanisms
- Lack of intelligent classification systems

This project addresses these limitations by creating an integrated system that:

- Combines audio data with visual representation through steganography
- Implements intelligent classification for both genre and emotional content
- Provides a unified interface for music storage, retrieval, and discovery
- Employs machine learning for enhanced metadata generation

2.2 Related Work

Several areas of research inform this project:

Audio Steganography: Prior work has explored embedding data in audio files, but fewer approaches examine embedding audio in images. Notable exceptions include [6]

Music Genre Classification: Evolution from traditional machine learning approaches using handcrafted features to deep learning models. Recent work by [3] and [4] demonstrates the effectiveness of convolutional approaches.

Emotion Recognition in Music: The dimensional model of emotion (valence-arousal) has gained traction in MIR (Music Information Retrieval) research, with approaches ranging from feature engineering to end-to-end deep learning.[5]

Digital Music Libraries: Systems like Gracenote, MusicBrainz, and Spotify have established precedents for large-scale music organization but lack integration of emotional metadata or novel storage techniques.

3. System Architecture

The system architecture consists of several key components that work together to provide a comprehensive music library solution.

3.1 Overall Architecture

The system employs a pipeline architecture with two main processes:

Encoding Process:

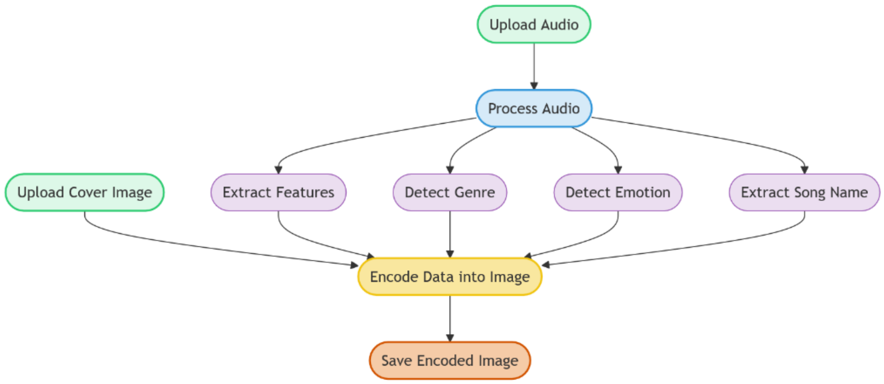

Figure 1: Encoding Process

Decoding Process:

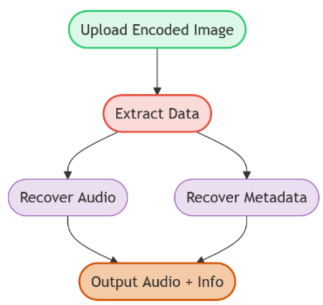

Figure 2: Decoding Process

3.2 Component Overview

1. **User Interface Layer**: Provides interaction for uploads, browsing, and music discovery
2. **Processing Layer**: Handles audio and image transformations, steganography operations
3. **Analysis Layer**: Implements genre and emotion classification models
4. **Storage Layer**: Manages the persistence of encoded images and metadata
5. **Recommendation Engine**: Utilizes classification results to suggest similar content

3.3 Technology Stack

Python 3.10
PyTorch (CNN models)
Librosa (audio feature extraction)
PyAudio (image processing)
FastApi (backend)
CUDA acceleration (GPU inference)

4. Digital Vinyl Technology

4.1 Steganography Implementation

The Digital Vinyl technology employs LSB (Least Significant Bit) steganography to embed complete audio files and associated metadata within cover images. This approach provides a visual representation that functions similarly to a physical vinyl record but contains the actual digital audio.

4.1.1 LSB Technique

The implementation modifies the least significant bits of image pixels to store binary data:

```
# LSB (Least Significant Bit) Steganography
for i in range(len(data_bits)):
    channel_idx = i % 3          # Cycle through R,G,B channels
    pixel_idx = i // 3          # Move to next pixel after 3 bits
    if data_bits[i] == 1:
        new_val = (pixel_val | 1)  # Set LSB to 1
    else:
        new_val = (pixel_val & 254) # Set LSB to 0
```

This technique offers several advantages:

- Minimal visual impact on the cover image
- Efficient use of available image data capacity
- Balanced approach to capacity vs. quality tradeoff
- Channel rotation for optimized data distribution

4.1.2 Data Structure

The embedded data follows a structured format with a header containing metadata followed by the audio content:

Header Format:

```
[MAGIC(4)] [SIZE(4)] [EXT_LEN(1)] [EXT(...)] [NAME_LEN(2)] [NAME(...)] [GENRE_LEN(1)] [GENRE(...)]
[AUDIO_DATA(...)]
```

Header components:

- MAGIC: 'AVNL' identifier (4 bytes) for validation
- SIZE: Audio data length (4 bytes)
- EXT_LEN: Extension length (1 byte)
- EXT: File extension (variable)
- NAME_LEN: Song name length (2 bytes)
- NAME: Song name (variable)
- GENRE_LEN: Genre length (1 byte)
- GENRE: Genre information (variable)
- AUDIO_DATA: The encoded audio file (variable)

This structured header enables efficient data retrieval and validation during the decoding process.

4.1.3 Capacity Analysis

The theoretical capacity for data embedding is:

- Each pixel provides 3 bits (R,G,B channels)
- A 1000×1000 pixel image yields approximately 375 KB of storage
- With compression, this is sufficient for 2-3 minutes of moderate quality audio

Practical capacity is influenced by:

- Image dimensions and resolution
- Audio compression ratio
- Required audio quality
- Metadata size

4.2 Visual Effects Pipeline

To enhance the aesthetic appeal and create a distinctive "vinyl" appearance, a visual effects pipeline processes the cover images:

1. ensure_square_image()

2. create_retro_effect():

  - Color grading

  - Scan lines

  - RGB shift

  - Noise/grain

  - Duotone effect

3. add_stamp()

These effects create a unique visual identity for the Digital Vinyl while maintaining data integrity for the embedded audio.

4.3 Error Handling

The system implements robust error handling to ensure data integrity during both encoding and decoding processes:

```
try:
    # Main operation
except Exception as e:
    logger.error(f"Error details: {str(e)}")
    # Fallback mechanism or raise
```

Error handling includes:

- Validation of input data formats
- Integrity checks for embedded data
- Recovery mechanisms for corrupted data
- Comprehensive logging for debugging

5. Genre Classification System

5.1 Dataset and Preprocessing

The genre classification system utilizes the Free Music Archive (FMA)[1] dataset:

- Small subset of the full FMA collection
- Balanced representation across genres

- Preprocessed for consistent format and quality

5.1.1 Feature Extraction

The system extracts Mel-Frequency Cepstral Coefficients (MFCCs) as the primary features for genre classification:

Technical parameters:

- Sample rate: 22050 Hz
- MFCC features: 40 coefficients
- Fixed input length: 128-time frames
- Window size: 2048 samples with 512 sample hop length

MFCCs were selected because they:

- Capture timbral characteristics relevant to genre
- Provide a compact representation of spectral information
- Have proven effectiveness in music classification tasks
- Require less computational resources than raw spectrograms

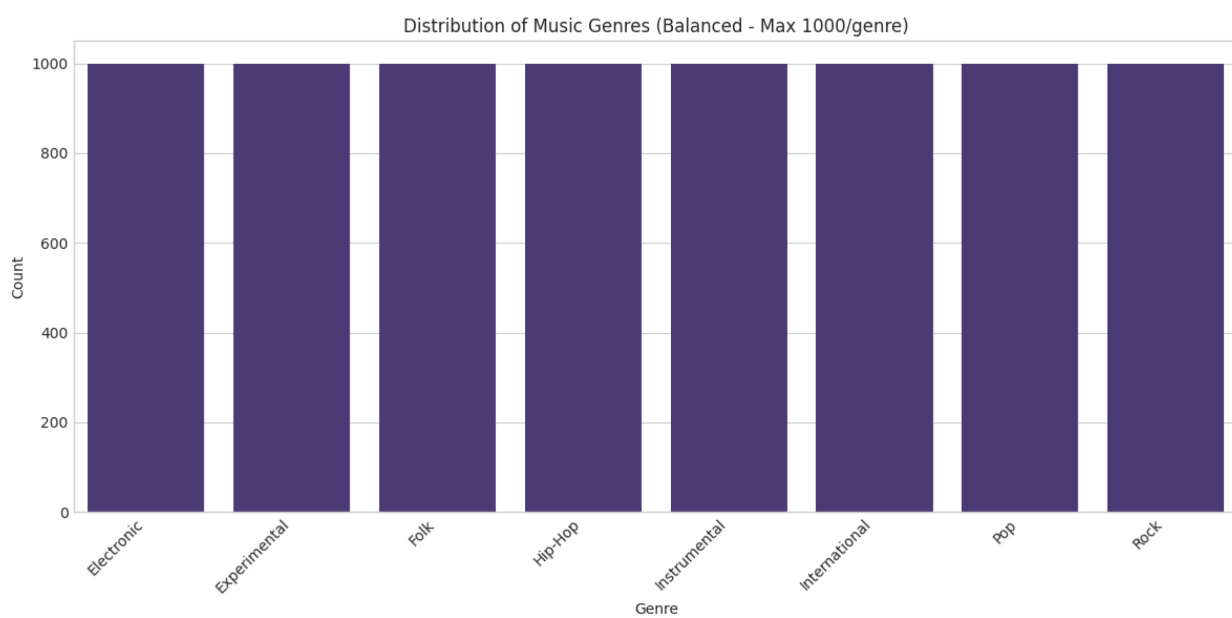

Figure 3: FMA Dataset Distribution

5.2 Model Architecture

The genre classification model employs a Convolutional Neural Network (CNN) architecture:

Model Architecture:

```
GenreClassifier(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.25, inplace=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout2): Dropout(p=0.25, inplace=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout3): Dropout(p=0.3, inplace=False)
  (fc1): Linear(in_features=10240, out_features=512, bias=True)
  (bn4): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout4): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=512, out_features=8, bias=True)
)
```

5.3 Training Pipeline

The training process implements several key techniques for optimal performance:

1. **Data Preparation**:
   - Dataset balancing for equal genre representation
   - Train/validation/test splits (70%/15%/15%)
   - Batch processing using PyTorch DataLoader
2. **Training Configuration**:
   - Loss function: Cross Entropy Loss
   - Optimizer: Adam with initial learning rate of 0.001
   - Learning rate scheduling: ReduceLROnPlateau
   - Batch size: 32
   - Maximum epochs: 100 with early stopping
3. **Regularization Techniques**:

- Dropout (0.3) after each convolutional block
- Batch normalization for training stability
- Early stopping with patience of 10 epochs
- Data augmentation (time stretching, pitch shifting)

4. **Model Evaluation**:
    - Accuracy, precision, recall, and F1-score metrics
    - Confusion matrix visualization
    - Per-genre performance analysis

5.4 Performance Results

The genre classification system achieved an overall test set accuracy of 43.3%. Performance varied noticeably across different genres. Genres like Folk and Rock performed relatively well, both achieving F1-scores above 0.50. Hip-Hop and International music also showed decent results, although with slightly lower recall and precision values.

On the other hand, the model struggled more with genres such as Pop and Experimental. Pop, in particular, had the lowest performance, with both precision and recall scores significantly below the average. Electronic and Instrumental genres showed moderate results, but still reflected the overall challenge of achieving consistent performance across diverse music styles.

Overall, the macro and weighted averages for precision, recall, and F1-score hovered around 43%, highlighting the difficulty of this classification task. Improving performance would likely require further tuning, larger datasets, or more sophisticated modeling approaches.

Figure 4: Genre Classification Confusion Matrix

6. Emotion Recognition System

6.1 Emotion Model

The emotion recognition system employs a two-dimensional model based on psychological research:

- **Valence**: Represents the positivity/negativity of the emotion (from -1 to 1)
- **Arousal**: Represents the energy/calmness of the emotion (from -1 to 1)

This approach allows for nuanced emotion representation beyond simple categorical classification.

6.2 Dataset and Feature Extraction

The system utilizes the DEAM (Database for Emotional Analysis of Music) [2] dataset:

- Contains valence and arousal annotations for diverse music tracks
- Provides time-synchronized emotional labels
- Covers various musical styles and contexts

For feature extraction, the system converts audio files into mel-spectrograms:

- Represents audio in a way similar to human hearing
- Captures frequency patterns relevant to emotion
- Works effectively with CNNs due to 2D structure

Mel-spectrograms provide:

- Time-frequency representations optimized for perceptual relevance
- Capture of timbral, harmonic, and rhythmic information
- Suitable input format for convolutional processing

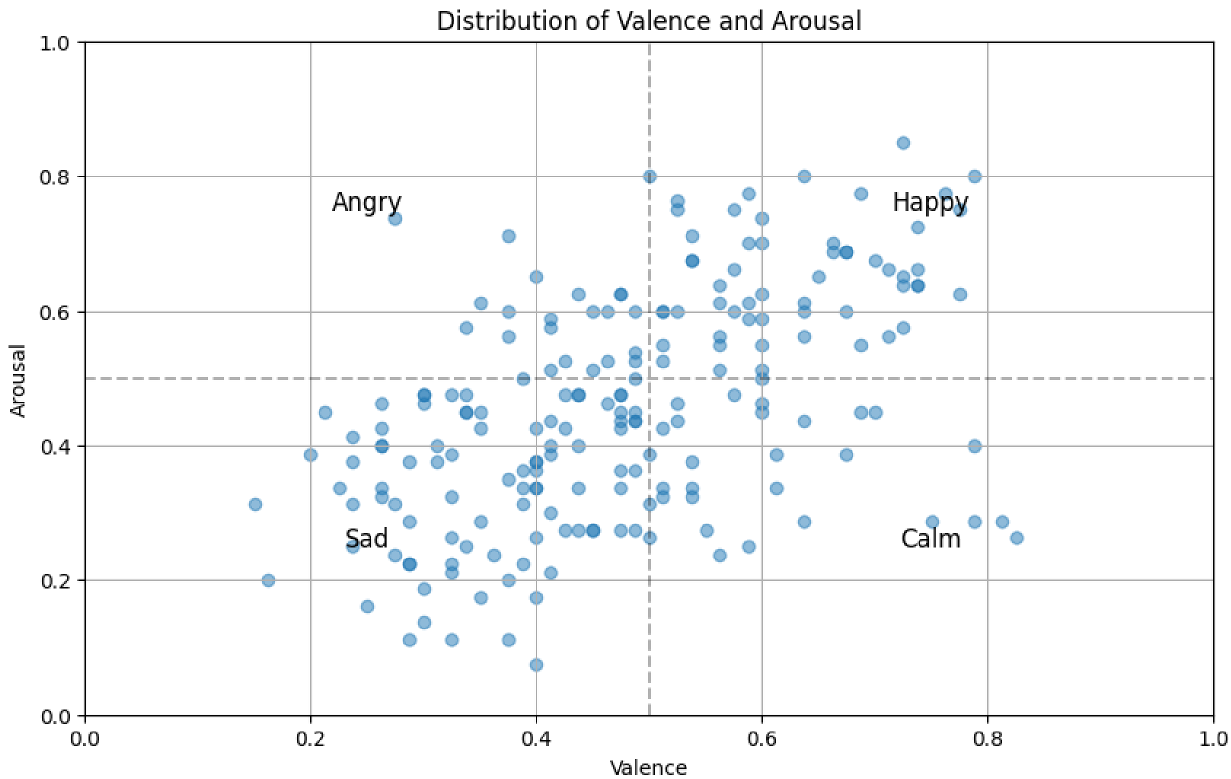

Figure 5: DEAM Dataset Distribution

6.3 CNN Architecture

The emotion recognition model implements a deep CNN architecture:

```
EmotionCNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc1): Linear(in_features=8192, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=2, bias=True)
  (relu): ReLU()
  (sigmoid): Sigmoid()
)
```

7. Integration and Implementation

7.1 Data Flow

The complete system integrates the components in a cohesive pipeline:

Encoding Process:

1. Audio → Binary data
2. Add metadata header
3. Process cover image
4. Embed data in pixels
5. Apply visual effects
6. Save output

Decoding Process:

1. Read image pixels
2. Extract LSB data
3. Find magic bytes
4. Parse header
5. Reconstruct audio
6. Analyze content (genre and emotion)

7.2 User Interface Components

The user interface provides several key functionalities:

- Upload interface for audio files and cover images
- Digital Vinyl visualization with playback controls
- Genre and emotion display with confidence metrics
- Search and filtering based on classification results
- Recommendation interface for similar content

7.3 Performance Optimization

Several optimization techniques improve system performance:

- Caching of extracted features for repeated analysis
- Batch processing for multiple files
- GPU acceleration for CNN inference
- Progressive loading of media content
- Asynchronous processing of non-critical operations

8. User Interface and Frontend Design

The frontend of the application plays a critical role in allowing users to interact with the music dataset intuitively and efficiently. To enhance user experience, the frontend incorporates features such as an interactive Arousal-Valence graph for sorting, genre-based filtering, metadata search, and an embedded audio player for song previews. These tools empower users to explore the dataset not only by conventional metadata but also through emotional and genre-based dimensions predicted by the machine learning models.

8.1 Overview of Frontend Features

The user interface was designed with simplicity and interactivity in mind. Users can browse, search, and filter songs based on various musical and emotional characteristics. Major functionalities include:

- An interactive Arousal-Valence graph for visual exploration,
- Genre-based filtering using the predicted genres,
- A metadata-based search bar to quickly locate specific tracks,
- A bottom-anchored audio player for immediate playback.

The frontend was implemented using NextJs with React Components, with responsiveness and smooth user experience as key priorities.

8.2 Arousal-Valence Interactive Graph

One of the main interactive elements is a 2D scatter plot that maps songs based on their predicted Arousal (energy) and Valence (positivity) values.

The X-axis represents Valence (negative to positive emotions), while the Y-axis represents Arousal (low to high energy).

Users can drag a selector point across the graph to choose a target emotional state.

As the selector moves, songs dynamically re-sort in real-time based on their proximity (Euclidean distance) to the selected Valence and Arousal values.

This interaction enables users to explore the music library by mood in a natural and fluid way, helping them find songs that best match their current emotional preference.

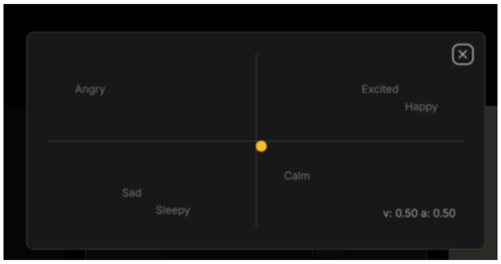

Figure 6: Arousal and Valance Input

8.3 Genre-based Filtering and Search Functionality

Users are provided with tools to filter the music library based on predicted genres. A dropdown menu (or checkbox list) allows selection of one or more genres, dynamically updating the displayed songs and their corresponding positions on the graph.

Alongside genre filtering, a real-time search bar enables users to search by song title, artist name, or other available metadata fields.

- The search is case-insensitive and returns results dynamically as the user types.
- Combining search with genre and emotional filters offers a flexible and powerful way to navigate large music collections.

Figure 7: Search and Filter Bar

8.4 Audio Player Integration

At the bottom of the interface, a lightweight audio player is integrated to allow immediate playback of any selected track.
Key features of the audio player include:

- Play/Pause controls,
- Track title and artist display,
- Ability to skip to next or previous tracks within the current filter set,
- Persistent playback while users interact with different parts of the interface.

This seamless integration ensures that users can not only discover music based on emotion and genre but also instantly listen to and evaluate the tracks without leaving the interface.

Figure 8: Audio Player

9. Evaluation

9.1 Technical Evaluation

9.1.1 Steganography Performance

- **Capacity**: Successfully embeds 3-4 minute MP3 files (128kbps) in 1000×1000 pixel images
- **Robustness**: Successfully recovers data after minor image manipulations
- **Efficiency**: Average encoding time of 2.3 seconds per image

9.1.2 Genre Classification Performance

- **F1-Score**: 0.433 average F1-score
- **Per-Genre Performance**: Best for Folk (0.53), Rock(0.50); Worst for Rock (0.21)
- **Confusion Areas**: Primary confusion between Pop and everything else

9.1.3 Emotion Recognition Performance

- **Valence RMSE:** 0.1072

- **Arousal RMSE:** 0.1207

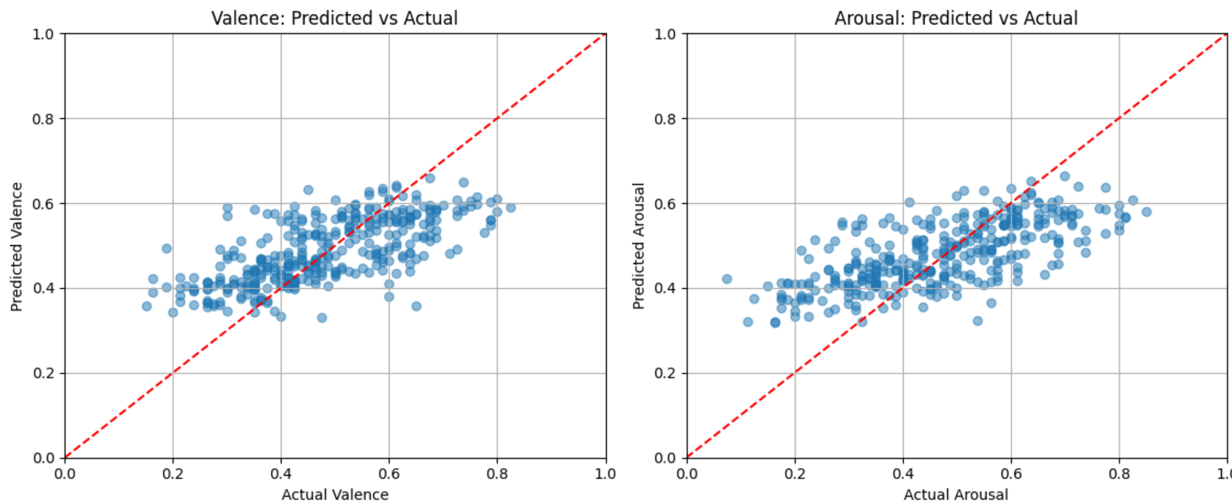

Figure 9: Predicted v Actual Emotion

9.3 Limitations

Identified limitations include:

- Maximum embedding capacity constrained by image dimensions
- Genre classification challenges with fusion or niche genres
- Emotion recognition variability for culturally specific music
- Processing time for high-resolution images or complex audio
- Potential data loss with aggressive image compression

10. Future Work

Several avenues for future development have been identified:

10.1 Technical Enhancements

- Implement more robust steganography techniques for increased capacity
- Explore transformer-based models for improved genre classification
- Incorporate lyrical content analysis for enhanced emotion recognition
- Develop multi-modal approaches combining audio and image features

- Implement distributed processing for large music collections

10.2 Feature Expansions

- Mobile application development for on-the-go access
- Integration with streaming services for expanded content
- Social sharing features for Digital Vinyl creations
- Expanded emotional classification spectrum
- Subgenre classification capabilities

10.3 Research Directions

- Cross-cultural emotion perception in music
- Temporal emotion tracking throughout songs
- Genre evolution and boundary detection
- Perceptual impact of visual presentation on music appreciation
- Music recommendation based on emotional trajectories

11. Conclusion

The Digital Vinyl Music Library represents an innovative approach to music organization, discovery, and presentation. By combining advanced machine learning techniques with novel data embedding approaches, the system offers capabilities beyond traditional music libraries.

The steganography-based Digital Vinyl technology demonstrates the potential for integrating visual and audio content in new ways, while the genre and emotion classification systems provide enhanced metadata for improved organization and discovery. The comprehensive implementation, from feature extraction to user interface, creates a cohesive system addressing real-world music management needs.

This project contributes to the fields of music information retrieval, digital media storage, and affective computing by demonstrating the practical integration of these technologies in a functional system. The evaluation results validate the approach while identifying areas for continued development and research.

12. References

[1] Defferrard, M., Benzi, K., Vandergheynst, P. & Bresson, X. (2016). FMA: A Dataset For Music Analysis (cite arxiv:1612.01840Comment: submitted to ISMIR 2017)

[2] DEAM dataset - Database for Emotional Analysis of Music. cvml.unige.ch. https://cvml.unige.ch/databases/DEAM/

[3] Music Genre Classification: Training an AI model. Arxiv.org. Published 2017. https://arxiv.org/html/2405.15096v1

[4] crlandsc. GitHub - crlandsc/Music-Genre-Classification-Using-Convolutional-Neural-Networks: Music genre classification system built on a convolutional neural network trained on Mel-spectrograms of 3-second audio samples. GitHub. Published 2023. Accessed April 26, 2025. https://github.com/crlandsc/Music-Genre-Classification-Using-Convolutional-Neural-Networks

[5] Liu X, Chen Q, Wu X, Liu Y, Liu Y. CNN based music emotion classification. arXiv.org. Published 2017. Accessed March 16, 2025. https://arxiv.org/abs/1704.05665

[6] Sonic Pixels. Sonic Pixels. Published 2025. Accessed April 26, 2025. https://sonicpx.jake.fun/

13. Appendices

Appendix A: Sample Code Implementation

Genre Prediction

predict.py > GenreClassifier > __init__

```python
SR = 22050          # Sample Rate
N_MFCC = 40         # Number of MFCCs to extract
N_FFT = 2048        # Window size for FFT
HOP_LENGTH = 512    # Hop length for STFT
FIXED_LENGTH = 128 # Number of time frames for MFCC features
MODEL_PATH = 'best_genre_classifier.pth'
class GenreClassifier(nn.Module):
    """Simple CNN model for Genre Classification based on MFCCs."""
    def __init__(self, input_shape, num_classes):
        super(GenreClassifier, self).__init__()
        self.input_shape = input_shape
        self.num_classes = num_classes
        # Convolutional Block 1
        self.conv1 = nn.Conv2d(in_channels=input_shape[0], out_channels=32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.dropout1 = nn.Dropout(0.25)
        # Convolutional Block 2
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.dropout2 = nn.Dropout(0.25)
        # Convolutional Block 3
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2)
        self.dropout3 = nn.Dropout(0.3)
        # Calculate the size of the flattened features dynamically
        self._to_linear = self._get_conv_output_size(input_shape)
        # Fully Connected Layers
        self.fc1 = nn.Linear(self._to_linear, 512)
        self.bn4 = nn.BatchNorm1d(512) # BatchNorm for FC layer
        self.dropout4 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes) # Output layer

    def _get_conv_output_size(self, shape):
        """Helper to calculate the flattened size after conv layers."""
        with torch.no_grad():
            dummy_input = torch.zeros(1, *shape) # Batch size 1
            output = self._forward_features(dummy_input)
            return int(np.prod(output.size()))

    def _forward_features(self, x):
        """Forward pass through convolutional layers."""
        x = self.pool1(F.relu(self.bn1(self.conv1(x))))
        x = self.dropout1(x)
        x = self.pool2(F.relu(self.bn2(self.conv2(x))))
        x = self.dropout2(x)
        x = self.pool3(F.relu(self.bn3(self.conv3(x))))
        x = self.dropout3(x)
        return x

```

Figure 10: Genre Classifier Class

predict.py > extract_features

```python
class GenreClassifier(nn.Module):
    def forward(self, x):
        x = x.view(x.size(0), -1) # Flatten
        x = F.relu(self.bn4(self.fc1(x)))
        x = self.dropout4(x)
        x = self.fc2(x) # Raw logits output
        return x

def extract_features(file_path, n_mfcc=N_MFCC, target_sr=SR, n_fft=N_FFT,
                     hop_length=HOP_LENGTH, fixed_length=FIXED_LENGTH):
    """
    Extract MFCCs using torchaudio with robust handling and normalization.
    Pads or truncates to a fixed length.
    """
    try:
        waveform, sample_rate = torchaudio.load(file_path)
        if waveform.shape[0] > 1:
            waveform = torch.mean(waveform, dim=0, keepdim=True)
        if sample_rate != target_sr:
            resampler = torchaudio.transforms.Resample(orig_freq=sample_rate, new_freq=target_sr)
            waveform = resampler(waveform)
            sample_rate = target_sr # Update sample_rate after resampling
        # Create MFCC transform
        mfcc_transform = torchaudio.transforms.MFCC(
            sample_rate=sample_rate,
            n_mfcc=n_mfcc,
            melkwargs={
                'n_fft': n_fft,
                'hop_length': hop_length,
                'n_mels': 128, # Typically more mels than MFCCs
                'center': True,
            }
        )

        mfccs = mfcc_transform(waveform)
        mfccs = mfccs.squeeze(0).numpy()
        current_length = mfccs.shape[1]
        if fixed_length is not None: # Only pad/truncate if fixed_length is specified
            if current_length < fixed_length:
                pad_width = fixed_length - current_length
                mfccs = np.pad(mfccs, pad_width=((0, 0), (0, pad_width)), mode='constant')
            elif current_length > fixed_length:
                mfccs = mfccs[:, :fixed_length]

        mean = np.mean(mfccs)
        std = np.std(mfccs)
        mfccs = (mfccs - mean) / (std + 1e-8)

        return mfccs

    except Exception as e:
        print(f"Error extracting features: {e}", file=sys.stderr)
        return None
```

Figure 11: Feature Extraction

predict.py > main

```python
def extract_features(file_path, n_mfcc=N_MFCC, target_sr=SR, n_fft=N_FFT,
        current_length = mfccs.shape[1]
        if fixed_length is not None: # Only pad/truncate if fixed_length is specified
            if current_length < fixed_length:
                pad_width = fixed_length - current_length
                mfccs = np.pad(mfccs, pad_width=((0, 0), (0, pad_width)), mode='constant')
            elif current_length > fixed_length:
                mfccs = mfccs[:, :fixed_length]

        mean = np.mean(mfccs)
        std = np.std(mfccs)
        mfccs = (mfccs - mean) / (std + 1e-8)

        return mfccs

    except Exception as e:
        print(f"Error extracting features: {e}", file=sys.stderr)
        return None

# --- Prediction Function ---
def predict(file_path, model_path=MODEL_PATH):
    if not os.path.exists(file_path):
        return {"error": f"File not found: {file_path}"}
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    try:
        features = extract_features(file_path)
        if features is None:
            return {"error": "Failed to extract audio features"}
        genres = ['Electronic', 'Experimental', 'Folk', 'Hip-Hop',
                  'Instrumental', 'International', 'Pop', 'Rock']

        input_shape = (1, N_MFCC, FIXED_LENGTH)
        model = GenreClassifier(input_shape=input_shape, num_classes=len(genres))

        model.load_state_dict(torch.load(model_path, map_location=device))
        model.to(device)
        model.eval()

        features_tensor = torch.tensor(features, dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(device)

        with torch.no_grad():
            outputs = model(features_tensor)
            probabilities = F.softmax(outputs, dim=1)
            confidence, predicted_idx = torch.max(probabilities, dim=1)

        predicted_genre = genres[predicted_idx.item()]
        confidence_score = confidence.item()

        return {
            "genre": predicted_genre,
            "confidence": confidence_score
        }

    except Exception as e:
        return {"error": f"Prediction error: {str(e)}"}
def main():
    if len(sys.argv) != 2:
        print(f"Usage: python {sys.argv[0]} <audio_file_path>", file=sys.stderr)
        sys.exit(1)
    file_path = sys.argv[1]
    result = predict(file_path)

    print(json.dumps(result, indent=2))

if __name__ == "__main__":
    main()
```

Figure 12: Main Genre Prediction

Emotion Prediction

predict_emotion.py > EmotionCNN > __init__

```python

import os
import sys
import json
import numpy as np
import torch
import torch.nn as nn
import librosa
import argparse

# Configuration
SAMPLE_RATE = 22050
SEGMENT_DURATION = 30  # seconds
N_MELS = 128
SPEC_WIDTH = 128
MODEL_PATH = 'deam_emotion_model.pth'

class EmotionCNN(nn.Module):
    def __init__(self):
        super(EmotionCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(128)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.3)
        self.fc1 = nn.Linear(128 * 8 * 8, 256)  # Assuming input size is 128x128 after pooling
        self.fc2 = nn.Linear(256, 64)
        self.fc3 = nn.Linear(64, 2)  # Output: valence and arousal
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()  # For 0-1 output range

    def forward(self, x):
        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.pool(self.relu(self.bn2(self.conv2(x))))
        x = self.pool(self.relu(self.bn3(self.conv3(x))))
        x = self.pool(self.relu(self.bn4(self.conv4(x))))

        x = x.view(-1, 128 * 8 * 8)  # Flatten
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        x = self.sigmoid(self.fc3(x))  # Use sigmoid to get values in [0,1]

        return x
```

Figure 13: Emotion CNN

```python
predict_emotion.py > predict_emotion
def predict_emotion(audio_file, model_path=MODEL_PATH):
    # Check if audio file exists
    if not os.path.exists(audio_file):
        raise FileNotFoundError(f"Audio file not found: {audio_file}")
    # Load model
    model = EmotionCNN().to(device)
    model.load_state_dict(torch.load(model_path, map_location=device))
    model.eval()
    # Load audio file
    try:
        y, sr = librosa.load(audio_file, sr=SAMPLE_RATE, duration=SEGMENT_DURATION)
    except Exception as e:
        raise Exception(f"Error loading audio file: {str(e)}")
    # If audio is shorter than segment_duration, pad with zeros
    if len(y) < SEGMENT_DURATION * SAMPLE_RATE:
        y = np.pad(y, (0, SEGMENT_DURATION * SAMPLE_RATE - len(y)), 'constant')
    # Extract mel-spectrogram
    mel_spec = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=N_MELS)
    # Convert to decibels
    mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)
    # Normalize
    mel_spec_norm = (mel_spec_db - mel_spec_db.min()) / (mel_spec_db.max() - mel_spec_db.min())
    # Resize to target width
    if mel_spec_norm.shape[1] > SPEC_WIDTH:
        mel_spec_norm = mel_spec_norm[:, :SPEC_WIDTH]
    elif mel_spec_norm.shape[1] < SPEC_WIDTH:
        padding = np.zeros((N_MELS, SPEC_WIDTH - mel_spec_norm.shape[1]))
        mel_spec_norm = np.hstack((mel_spec_norm, padding))
    # Convert to tensor
    mel_spec_tensor = torch.tensor(mel_spec_norm, dtype=torch.float32).unsqueeze(0).unsqueeze(0)  # Add batch and channel dimensions
    mel_spec_tensor = mel_spec_tensor.to(device)
    # Make prediction
    with torch.no_grad():
        prediction = model(mel_spec_tensor)
    valence = float(prediction[0][0].item())
    arousal = float(prediction[0][1].item())
    # Map to emotion categories (optional)
    emotion_map = {
        (True, True): "Happy",      # High valence, high arousal
        (True, False): "Calm",      # High valence, low arousal
        (False, True): "Angry",     # Low valence, high arousal
        (False, False): "Sad"       # Low valence, low arousal
    }
    emotion = emotion_map[(valence > 0.5, arousal > 0.5)]
    return {
        "valence": valence,
        "arousal": arousal,
        "emotion": emotion
    }
```

Figure 14: Main Emotion Prediction

Frontend

components > emotion-selector.tsx > ...

You, 20 hours ago | 1 author (You)

```tsx
interface EmotionSelectorProps {
  onEmotionChange: (emotion: { arousal: number; valence: number }) => void
}

export function EmotionSelector({ onEmotionChange }: EmotionSelectorProps) {
  const [selectedEmotion, setSelectedEmotion] = useState<{ arousal: number; valence: number }>({
    arousal: 0.5,
    valence: 0.5
  })
  const [isExpanded, setIsExpanded] = useState(false)

  const handleEmotionChange = useCallback((arousal: number, valence: number) => {
    setSelectedEmotion({ arousal, valence })
    onEmotionChange({ arousal, valence })
  }, [onEmotionChange])

  return (
    <div className="inline-block">
      <EmotionVisualizer
        arousal={selectedEmotion.arousal}
        valence={selectedEmotion.valence}
        onChange={handleEmotionChange}
        mini={true}
        expanded={isExpanded}
        onExpand={() => setIsExpanded(true)}
      />

      <Dialog open={isExpanded} onOpenChange={setIsExpanded}>
        <DialogContent className="sm:max-w-lg">
          <EmotionVisualizer
            arousal={selectedEmotion.arousal}
            valence={selectedEmotion.valence}
            onChange={handleEmotionChange}
          />
        </DialogContent>
      </Dialog>
    </div>
  )
}
```

Figure 15: Frontend Emotion Selector Component

components > library.tsx > ...

```tsx
export function Library({
  tracks,
  onPlayTrack,
  currentTrackId,
  isPlaying,
  onTogglePlay,
  onUploadClick,
  activeGenre,
  onGenreSelect,
  genres,
}: LibraryProps) {
  const [searchQuery, setSearchQuery] = useState("")
  const [targetEmotion, setTargetEmotion] = useState<{ arousal: number; valence: number } | null>(null)

  // Calculate emotional distance between two points
  const getEmotionalDistance = (emotion1: { arousal: number; valence: number }, emotion2: { arousal: number; valence: number }) => {
    const dA = emotion1.arousal - emotion2.arousal
    const dV = emotion1.valence - emotion2.valence
    return Math.sqrt(dA * dA + dV * dV) // Euclidean distance
  }

  const filteredTracks = tracks.filter((track) => {
    // Filter by search query
    const matchesSearch =
      track.title.toLowerCase().includes(searchQuery.toLowerCase()) ||
      track.artist.toLowerCase().includes(searchQuery.toLowerCase())

    // Filter by genre if active
    const matchesGenre = !activeGenre || track.genre.toLowerCase() === activeGenre.toLowerCase()

    return matchesSearch && matchesGenre
  }).sort((a, b) => {
    // If no emotion target is selected, keep original order
    if (!targetEmotion) return 0

    // Sort by emotional distance
    const distanceA = getEmotionalDistance(targetEmotion, a.emotion)
    const distanceB = getEmotionalDistance(targetEmotion, b.emotion)
    return distanceA - distanceB
  })
```

Figure 16: Frontend Main Library

```
components > track-card.tsx > ...
export function TrackCard({ track, onPlay, isActive, isPlaying, onTogglePlay }: TrackCardProps) {
          </div>

          {/* Back side - Emotion data */}
          <div
            className="absolute w-full h-full backface-hidden rotate-y-180 bg-zinc-800 p-4 flex flex-col justify-center items-
            style={{ backfaceVisibility: "hidden" }}
          >
            <div className="text-center mb-4">
              <h3 className="font-medium text-white mb-1">{track.title}</h3>
              <p className="text-zinc-400 text-sm transform -skew-x-12">{track.artist}</p>
            </div>

            <div className="w-full max-w-[200px] mx-auto">
              <EmotionVisualizer arousal={track.emotion.arousal} valence={track.emotion.valence} large={true} />
            </div>

            <div className="mt-4 text-center">
              <Badge variant="custom" className={`${getGenreColor(track.genre)} transform -skew-x-12`}>{track.genre}</Badge>
            </div>
          </div>
        </div>

        {/* Control buttons */}
        <div className="absolute inset-0 bg-black bg-opacity-40 flex items-center justify-center opacity-0 group-hover:opacity
          <div className="flex gap-3">
            <Button
              onClick={(e) => {
                e.stopPropagation()
                handleCardClick()
              }}
              variant="secondary"
              size="icon"
              className="rounded-full h-12 w-12 bg-white/20 hover:bg-white/30 backdrop-blur-sm"
            >
              {isCurrentlyPlaying && !isFlipped ? <Pause size={20} /> : <Play size={20} />}
            </Button>

            <Button
              onClick={(e) => {
```

Figure 17: Frontend Track Card

Backend

```
app > ~/AudioEncoder/app/genrePrediction/predict.py

@app.post("/encode")
async def encode_audio(
    audio_file: UploadFile = File(...),
    cover_image: UploadFile = File(...),
    song_name: str = Form(None),
):
    """Encode audio into an image with optional song name"""
    try:
        logger.info(f"Starting encoding process for audio file: {audio_file.filename}")
        logger.info(f"Original audio file details - Filename: {audio_file.filename}, Content-Type: {audio_file.content_type
        logger.info(f"Original cover image details - Filename: {cover_image.filename}, Content-Type: {cover_image.content_t

        # Validate audio file type
        audio_ext = os.path.splitext(audio_file.filename)[1].lower()
        logger.info(f"Audio file extension: {audio_ext}")

        if not audio_ext in ['.mp3', '.wav', '.ogg', '.m4a']:...

        # Validate image content type and set extension
        valid_image_types = {...

        if cover_image.content_type not in valid_image_types:...

        image_ext = valid_image_types[cover_image.content_type]
        logger.info(f"Using image extension based on content-type: {image_ext}")

        # Generate unique filenames - replace spaces with underscores and ensure proper extensions
        safe_audio_name = audio_file.filename.replace(' ', '_')
        safe_image_name = f"cover_image{image_ext}"  # Use a standard name with proper extension
        audio_filename = f"{uuid.uuid4()}-{safe_audio_name}"
        image_filename = f"{uuid.uuid4()}-{safe_image_name}"
        encoded_filename = f"encoded_{uuid.uuid4()}.png"

        logger.info(f"Generated safe filenames - Audio: {audio_filename}, Image: {image_filename}")

        # Save uploaded files
        audio_path = UPLOAD_DIR / audio_filename
        image_path = UPLOAD_DIR / image_filename
        encoded_path = ENCODED_DIR / encoded_filename
```

Figure 18: Backend encode endpoint

app > main.py > encode_audio

```python
@app.post("/decode")
async def decode_audio(
    encoded_image: UploadFile = File(...),
):
    """Decode audio from an encoded image"""
    try:
        logger.info(f"Starting decoding process for image: {encoded_image.filename}")

        # Validate file type
        image_ext = os.path.splitext(encoded_image.filename)[1].lower()
        if not image_ext in ['.jpg', '.jpeg', '.png']: ···

        # Generate unique filenames
        image_filename = f"{uuid.uuid4()}{image_ext}"
        decoded_filename = f"decoded_{uuid.uuid4()}.tmp"

        # Save uploaded file
        image_path = UPLOAD_DIR / image_filename
        decoded_path = DECODED_DIR / decoded_filename

        logger.info(f"Saving encoded image to: {image_path}")

        # Write uploaded file to disk
        try:
            image_content = await encoded_image.read()
            if not image_content:
                raise HTTPException(status_code=400, detail="Uploaded image file is empty")

            with open(image_path, "wb") as image_file_obj:
                image_file_obj.write(image_content)
        except Exception as e:
            logger.error(f"Error saving uploaded image: {str(e)}")
            raise HTTPException(status_code=500, detail=f"Error saving uploaded image: {str(e)}")

        # Decode audio from image and predict emotion
        try:
            logger.info("Starting audio decoding process")
            actual_decoded_path, song_name, genre = decode_audio_from_image(str(image_path), str(decoded_path))
            decoded_filename = os.path.basename(actual_decoded_path)
```

Figure 19: Backend decode endpoint

```python
@app.get("/encoded/{filename}")
async def get_encoded_image(filename: str):
    """Serve an encoded image"""
    try:
        file_path = ENCODED_DIR / filename
        if not file_path.exists():
            logger.error(f"Encoded image not found: {filename}")
            raise HTTPException(status_code=404, detail=f"Encoded image not found: {filename}")
        return FileResponse(file_path)
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Error serving encoded image {filename}: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Error serving encoded image: {str(e)}")

@app.get("/decoded/{filename}")
async def get_decoded_audio(filename: str):
    """Serve a decoded audio file"""
    try:
        file_path = DECODED_DIR / filename
        if not file_path.exists():
            logger.error(f"Decoded audio not found: {filename}")
            raise HTTPException(status_code=404, detail=f"Decoded audio not found: {filename}")
        return FileResponse(file_path)
    except HTTPException:
        raise
    except Exception as e:
        logger.error(f"Error serving decoded audio {filename}: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Error serving decoded audio: {str(e)}")
```

Figure 20: Backend Get Requests

Appendix B: Frontend Screenshots

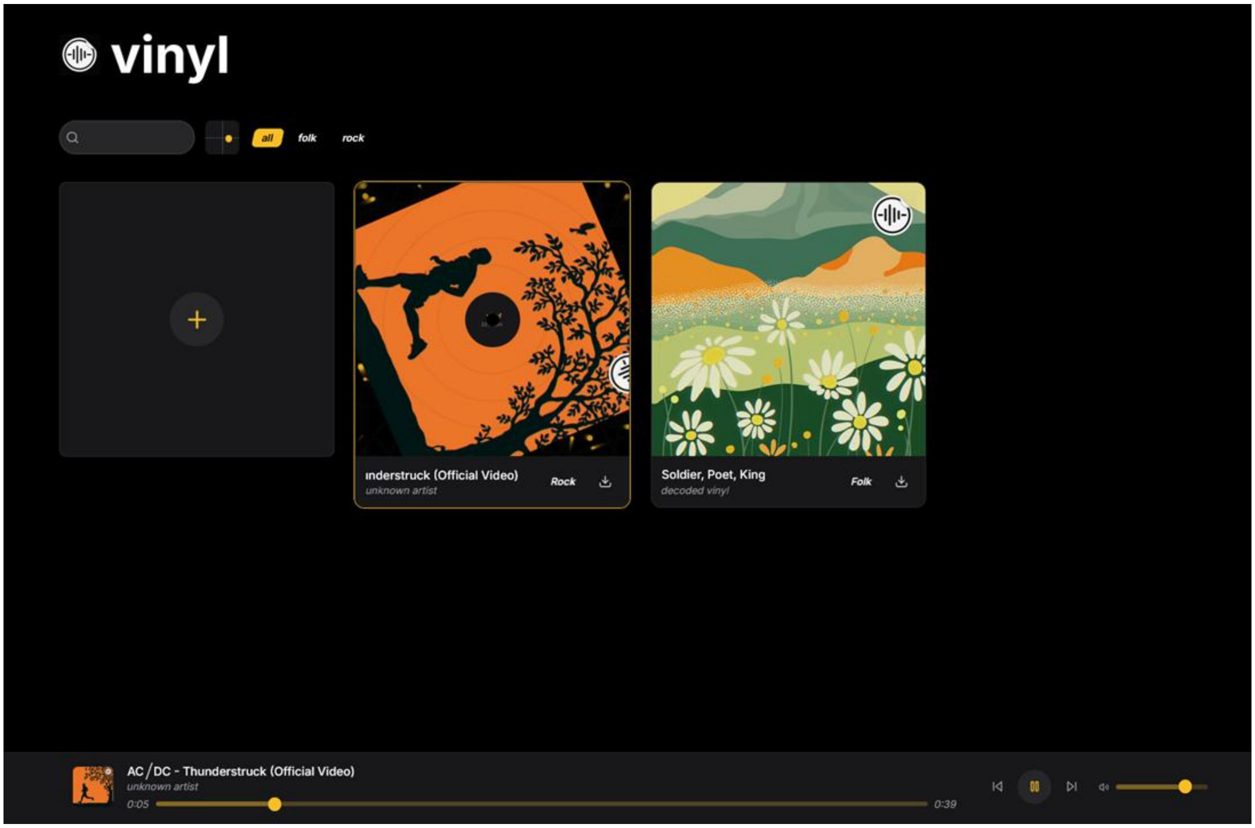

Figure 21: Library