UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Evaluation of Video Soundtracks using Machine Learning

Addressing the issues of data availability, feature extraction and classification

by

Georgios Touros

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Theodoros Giannakopoulos
Principal Researcher of Multimodal Machine Learning

Athens, December 2020

Evaluation of Video Soundtracks using Machine Learning

Georgios Touros

MSc. Thesis, MSc. Programme in Data Science

University of the Peloponnese & NCSR “Democritos”, December 2020

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Evaluation of Video Soundtracks using Machine Learning

Addressing the issues of data availability, feature extraction and classification

by

Georgios Touros

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Theodoros Giannakopoulos
Principal Researcher of Multimodal Machine Learning

Approved by the examination committee on December, 2020.

(Signature)	(Signature)	(Signature)
.....	.....	.....
Theodoros Giannakopoulos Principal Researcher	Konstantinos Limniotis External Researcher	George Papadakis Post-Doctoral Researcher

Athens, December 2020

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Declaration of Authorship

(1) I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

(2) I confirm that this thesis presented for the degree of Master of Science in Data Science, has

   (i) been composed entirely by myself

   (ii) been solely the result of my own work

   (iii) not been submitted for any other degree or professional qualification

(3) I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

(Signature)

.....

Georgios Touros

Athens, December 2020

Acknowledgments

This thesis marks the end of the journey, for my Master's degree in Data Science. Given the tremendous effort and care that was put into completing both the courses and this project, I'd say it's with enormous joy that I reach the end.

The project was initially conceived with a much more ambitious end goal; to create a full-fledged video soundtrack generation algorithm, using Deep Learning. Unfortunately, as will be explained in the chapters that follow, this wouldn't be possible given our resources and available time. We made a conscious choice when pitching this subject, as there seemed to be a high element of novelty, and at the same time, it was very close to my own interests. Now that we've reached the end of the project, this seems like a fool's hope.

Nevertheless, here we are, and I'd like to take this opportunity to thank all of the people that made this journey possible:

Firstly, I'd like to thank Mr. Giannakopoulos, who proved to be a vigilant mentor throughout this process, and who kept providing valuable insights and directions, as well as level-headed guidance, even during a time of incredible turmoil and instability around the world.

Secondly, I'd like to thank my colleagues and leaders George Charikiopoulos and Nikos Psaltopoulos, for giving me their full support and tolerance, during these years of struggle, as well as my own team members Christos, Antonis and Tassos, for their understanding and support in keeping balance at the work front.

Furthermore, I'd like to thank my buddies from the program, Ioannis Loumiotis, Nikos Nikolaou and Kostas Gygtakis, for being the best companions one could hope for, in a tough journey such as this. Their knowledge, their attitude and their

impeccable sense of humour were a source of energy and solace in the last two years. Another special thanks goes to Giannis, Alexandros, Alexandros and Marios, for swooping in like the Eagles to pick me up from the flames of Mount Doom, when needed.

Finally, and most importantly, I'd like to thank my family and friends, for their emotional support and their continuous faith in me, even when things seemed dire. Their presence alone would be a great help when struggling to maintain emotional and intellectual balance, but their persistence and active care-taking during this time is a gift I'll never be able to repay. It would be impossible for me to reach the end, without their patience, their care, their love and affection.

And now, to the task at hand...

To Maya, for her kindness and patience

Abstract

The aim of this thesis is to address the challenges of combining multimodal data to evaluate video soundtracks. To tackle tasks in the field of soundtrack generation, retrieval, or evaluation, data needs to be collected from as many relevant modalities as possible, such as audio, video, and symbolic representations of music. We propose a method of collecting relevant data from all of these modalities, and from them, we attempt to describe and extract a comprehensive multimodal feature library. We construct a database by applying our method on a small set of available data from the three relevant modalities. We implement and tune a classifier in our constructed database of features with adequate results. The classifier attempts to discriminate between real and fake examples of video soundtracks. Finally, we describe some possible improvements on the methods, and we point at some use-cases and directions for future attempts at this and adjacent tasks.

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

etc.	et cetera
i.e.	id est
a.k.a.	also known as
UGV	User Generated Videos
Hz	Hertz
AWF	Audio Waveform
MPEG-7	Moving Picture Experts Group 7
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
IFT	Inverse Fourier Transform
MIDI	Musical Instrument Digital Interface
SMF	Standard MIDI File
ASCII	American Standard Code for Information Interchange
MFCC	Mel-Frequency Cepstrum Coefficients
ZCR	Zero-Crossing Rate
RGB	Red Green Blue
ADC	Analog to Digital Conversion

DT	Decision Tree
LR	Logistic Regression
kNN	k Nearest Neighbors
SVM	Support Vector Machines
RandF	Random Forest
BTr	Bagged Trees
AdaB	Adaptive Boosting
XGB	eXtreme Gradient Boosting
NB	Naïve Bayes
PCA	Principal Component Analysis
SVD	Singular Value Decomposition
ExTrees	Extra Trees
GradBoost	Gradient Boosting

Chapter 1

Introduction

The collaboration between artist and scientist, has been an evolving relationship, ever since the first tools were used to create the first work of art; so much so that in the early days of Man, artist and technician were inseparable terms. In our modern age of Artificial Intelligence and Machine Learning, this relationship is as challenging as it ever was.

In recent years, there is a multitude of attempts to create models that touch upon the domain of artistic creativity. Either for evaluation of artistic products, or for generating works of art in their own right, this interaction between human and machine creativity has been an active field of research. Machine learning models initially would focus in artistic domains that contained singular modalities such as image, sound, music composition etc. As technology advances and more data becomes available, models tend to combine more than one content modalities to focus on more complex domains.

Such a domain is that of creating, or choosing, music that accompanies visual content, i.e. video soundtracks. This is a type of artistic task that is usually taken up by dedicated professionals. Especially in the film industry, a typical film would have a composer and a music supervisor working separately to create and select the musical content that best accentuates the themes, energy and emotion of a particular scene. Their works are inter-connected, but a close collaboration between them is not always the case. In this thesis, we attempt to address the challenges that affect

the aforementioned tasks before creating solutions for any of them. These issues include data availability, data quality, data storage, representation and encoding of the participating modalities, fusion of multimodal content and feature extraction.

Our work starts by describing our method for collecting, storing, combining and cleaning up data. We present an end-to-end pipeline from raw data to a database of features of all three participating modalities (raw audio, MIDI and video). With this pipeline, we construct a functioning database, and lay out its relational schema. We utilise an external knowledge base for songs in conjunction with text cleaning techniques to match the songs in both their forms (audio and MIDI). To detect the presence of these songs within a collection of films and television series, we use a fingerprinting technique. We present a detailed description of the challenges of this method, as we manage to collect 68 data points, and construct an equal amount of negative examples.

We then attempt to create an understanding of the fundamentals, representation and encoding strategies for each of the modalities that are relevant to the task: music from a signal processing perspective, music from the perspective of computational musicology (using symbolic representations of the compositions) and visual content, using both low-level components (such as color) and higher level aspects (such as shot and object detection). We dedicate a separate chapter for each of these modalities, and we describe the techniques that we used to extract long-term statistics for each one. Tables with all 455 available features collected are in the Appendix.

Following this, we experiment with creating a classifier that intends to discriminate between real soundtracks and fake, misfitted examples. We test various models, and we experiment with different pre-processing and dimensionality reduction techniques. We then perform hyper-parameter tuning for the most promising settings, and end up with a classifier that achieves results that are better than chance. The resulting model would not be suitable for cases where high precision matters, but at this stage could serve as an initial filtering classifier for a soundtrack retrieval system.

After evaluating the results of the classifier, we conclude our work with a summary of all the challenges and biases that arose in the process of building the data-

set. Finally, we also point towards some promising directions for future improvement of the pipeline, as well as some use-cases in adjacent tasks.

1.1 Problem Statement

The task that we will attempt to solve in this thesis is binary classification of musical choices for videos, based on representations of the participating content. In other words, the evaluation of whether the choice of a song as accompaniment for a scene in a film is good or not, using different representations of the modalities that are present in the scene. To that end, we will extract a selection of hand-crafted features from three different content modalities: audio, video and symbolic representations of music. These alternative representations and content modalities, carry information of both higher levels (such as harmonious movements in music or objects and faces contained within a video frame) and lower levels (such as statistics about the audio signal's energy or the colors in the video).

1.2 Motivation - Ethical Concerns

1.2.1 On art and technology

Human creativity has been a founding block of our civilization for centuries. From images of hands in caves in Sulawesi Indonesia that date back forty thousand years, to the latest webisode of a series in YouTube, humans have been trying to give form to their thoughts and ideas through multiple mediums of artistic creation. These mediums have been evolving, along with the technical advancements of humanity. This dialogue between technology and artistic expression, is as true today as it had been in the Bronze Age, as artists continue to use tools of variable complexity (from chisels and hammers to video cameras and synthesizers) to create their art, and in this process, scientists and inventors are motivated to keep improving, disrupting and innovating these tools [3].

The distinction between art and technology is a rather recent notion in human history, as artists, craftsmen and artisans were considered to be the same thing, and the methods of working in any particular branch of art were considered as 'technical'

[4]. Though it is beyond the scope of this thesis to define art in philosophical terms, given that such a task might even be futile or even impossible [5], if we were to follow the thought of Stephen Davies [6] we could claim that something is art

- if it shows excellence of skill and achievement in realizing significant aesthetic goals, and either doing so is its primary, identifying function or doing so makes a vital contribution to the realization of its primary, identifying function, or
- if it falls under an art genre or art form established and publicly recognized within an art tradition, or
- if it is intended by its maker/presenter to be art and its maker/presenter does what is necessary and appropriate to realizing that intention

1.2.2 On Artificial Intelligence and Art

In the context of Artificial Intelligence, it is tempting to consider how art that is produced through algorithms can in itself be considered as art, and whether or not the designer of the algorithm or Deep architecture is an artist, with generative algorithms being new mediums for artistic expression. The third part of Davies' definitions seems to allow us this conceptual leap.

Even more importantly, it seems that much of the innovation today concerning art, is being made by people who work outside the confines of fine arts or traditionally accepted centres of artistic excellence. According to Jon McCormack [7], these people work in the industries of popular culture, which he identifies as being computer graphics, film, music videos and the Internet, and in terms of 2020 we can extend to content streaming services and social media.

Achievements in computer hardware, shifting social needs and continuous research in the field of Data Science, have enabled the evolution of artistic concepts, such as generative art, computer art, computational art, interactive art and more. These concepts seem to continuously challenge the philosophical and theoretical conceptions of art, authenticity, agency and authorial responsibility [8] [9].

We will not attempt to address or challenge the opposing views on the matter

in this work, as they are beyond the scope of this thesis. Nevertheless, such ethical concerns have been important considerations during the process of formulating the task, and framing the scope of the project and the thesis. In this light, we have attempted to approach the relationship of Artificial Intelligence and Art from a utilitarian perspective; from our point of view, the algorithms should exist to assist the artist in their work, and not substitute them in any way.

1.2.3 Practical Aspects of a Soundtrack Discriminator or Generator

Despite of our aesthetic and philosophical concerns, the recent achievements in Machine Learning and Deep Neural Networks have greatly advanced this idea of generating, retrieving and classifying multimedia. Relevant algorithms are being introduced each year for different kinds of art, such as music, image and video, with varying degrees of success. In this thesis, we attempt to focus on methods that combine music and video, in order to approach the task of choosing satisfactory accompaniment music for video content.

Such a system would be quite useful in a modern context of popular art creation, as it's too often the case that music is needed as an accompaniment for video content that is produced by artists or corporations for commercial purposes or for temporary enjoyment. It would also be useful during film production, in the editing process, when the film's music composer or supervisor has not yet delivered the music for a scene, but the editors and the director need some temporary music in order to get a rhythm for their editing process. This type of temporary music, currently a usual practice in film production, sometimes leads to other implications, such as the editing of the film being so tightly linked to the temporary music, that the official music composer has no option but to create music that closely resembles it [10]. These possible use cases call for music that is either royalty-free, or even disposable, and AI would definitely reduce the cost and implications of creating or choosing such music. In the following paragraphs, we will attempt to envision some of what we had in mind while constructing our pipeline, in the hope that some of our ideas

could potentially lead to more exciting applications in the field.

Soundtrack composition systems Our initial ambition for this thesis was to create a soundtrack generation model. With our limited time and resources, this was not possible. Nevertheless, the data collection and extraction pipeline that we will propose, if applied in scale, could potentially be a suitable starting point for a soundtrack composition system.

Furthermore, the classifier that we have experimented with could be used as an evaluation method for such as system, as it could potentially discriminate the real from the fake examples. This hypothesis would need more data in order to be thoroughly tested.

Machine-aided Soundtrack Evaluation An improved version of the model that we experiment with in Chapter 6, probably one that is trained with a lot more data, could be used to create a platform for video soundtrack selection. The end-users could be video editors that need a temporary music while they wait for the music supervisor to provide them with a fitting, licensed piece of music, or music supervisors that want to validate their choice of music or let the algorithm help them decide between some options. Such a system could also serve as a platform for artists that want to expose their music to such an audience (video editors or film producers), by providing an evaluation of the goodness of fit of their music, with regards to the visual content.

Further to this, such a model could be used in order to create a soundtrack recommender system from a database of existing tracks. Such a system could be used either as a plug-in on video editing software, or as a stand-alone application.

1.3 Related Work

There is a lot of literature around video processing, audio processing and classification. To our knowledge, there are not as many papers when it comes to video soundtrack generation or classification. In the following sections we describe the most common research threads that we came across.

Soundtrack Retrieval for User Generated Videos A common thread in similar research is music retrieval for user generated videos (UGV). In [11] a system for creating automatic generation of soundtracks for outdoor videos of users is proposed. The system is built on contextual data based on the geo-location in which the video was shot. This contextual data contains geographic tags, and mood tags, collected from OpenStreetMap and Foursquare. In [12] a system for recommending soundtracks for user outdoor videos is proposed, based on geographic, visual and audio features. The visual features are based on color only, and are combined with tags of the mood of the specific area. These are then combined with music, combining the user's previous listening history with mood tags and audio features of the track. A similar approach is followed in [13]. In [14] the authors propose a process that recommends the soundtrack and edits the video simultaneously. Their approach uses a multi-task deep neural network to predict the characteristics of an ideal song for the video, and then retrieves the closest match from a database. The track is then aligned to the video using a dynamic time warping algorithm, and concatenates the video given a cost function, trained on an annotated corpus.

Soundtrack Recommendation for Video Editing Another research thread tries to create music recommenders that could be used as plug-ins in video editing software, or in order to create music videos. In [15] a method and a system is proposed for the recommendation of soundtracks by video editing software. The proposed system is based on emotional and contextual tags given by the end-user, and then retrieves and combines relevant loops of preexisting content. In [16] a music video generation system is developed, which utilizes the emotional temporal phase sequence of the multimedia content to connect music and video. It is trained on annotated data that is mapped to an arousal-valence scale that is tracking the shifts in emotion along the content of the medium. The multimedia are then matched according to the time-series of the emotional shifts, using string matching techniques. In [17], a model that synchronises the climax of a video clip and a music clip is proposed. The model is trained on annotated data for the audience perception of climax in music and video, and applies dynamic programming to synchronise the

climax in both modalities.

1.4 The Contribution of this Thesis

In our work, we attempt to define the problems that arise when constraints of different modalities are involved in tasks involving video soundtracks. Our focus will be on classification of fit for video soundtracks. We break down the task into its main components:

- Collecting the data
- Extracting features
- Selecting a suitable classifier
- Evaluating the result

We then formulate the challenges on each of these components, and try to address them.

Despite our best efforts, we could not find another work that combined visual features with features that are extracted from symbolic representations of music. Our experimentation suggests that there is potential in this approach, as high-level features about song composition are combined with low-level features from the audio tracks and visual features that are both low-level, such as color histograms, and high-level, such as object and movement detection;.

Given the complexity of the task, and the lack of available open data (challenges which are described in detail in Chapter 2), our main focus was on creating a large enough data-set that contains data from all the necessary modalities and forms (songs in raw audio form, transcriptions of these songs in midi form and video excerpts containing these songs). Our contribution is not only to gather and clean the data, but also to create an open-source scalable process to create and manage such data-sets, which could potentially be given to the research community for further expansion.

We also proceed to extract features from this data-set, and examine their suitability using a rudimentary classifier, that matches the different modalities with

each other. We experiment with tuning the model, but the lack of large-scale data in our proof-of-concept, prevents us from reaching high levels of accuracy.

1.5 Thesis structure

The rest of the thesis is organised as follows:

In Chapter 2 we describe in detail our attempt to address the data availability issue. We propose a method of acquiring, cleaning and managing the data of all three relevant modalities, and we report on the specifics and challenges of implementing this method.

In Chapter 3 we discuss issues of representation and encoding of data for audio and music, and we describe in depth the feature extraction process from the audio domain.

In Chapter 4 we describe the fundamentals of symbolic representations of musical content, we discuss issues of encoding these representations, and we explain how we extracted features from this domain.

In Chapter 5 we delve into the specifics of representing and encoding visual content, and we describe the feature extraction process that was implemented for this modality.

In Chapter 6 we experiment with the creation of a classifier that could discriminate between real and fake examples of soundtracks, given data and features that was collected in the previous chapters. We present the results of our experiments, and we highlight some promising directions.

In Chapter 7 we sum up our challenges and findings, and we discuss future work, focusing on how our own contributions could build up to a large scale data set that could better handle the task of soundtrack recommendation and potentially complement existing music generation models for soundtrack generation for visual content.

Chapter 2

Data Collection Pipeline

As we mentioned in section 1.2.3, the main challenge of creating music for videos is data availability. We could not find any open data set that includes all three necessary modalities or that was scalable enough for our purposes. We therefore proceeded with creating our own data collection pipeline, which we will describe in detail in the following sections.

2.1 Repo Structure

We have used GitHub to store our code for this project[1]. It follows a modular logic, and is split into the following modules:

- `config`: the central position of settings, paths and credentials
  - `credentials.py` is where the credentials are stored for interacting with external APIs (database, Spotify etc.).
  - `paths.py` is where all the paths are stored for data inputs and outputs, as well as for temporary folders.
  - `settings.py` is where the parameters for audio and video processing are stored. These include:
    - `CHUNK_SIZE_SECONDS`: the size of the chunks for audio recognition within video files in seconds.

[1] `https://github.com/GeorgeTouros/video-soundtrack-evaluation`

    * CHUNK_SIZE_MS: the size of the chunks for audio recognition within video files in milliseconds (for direct use within ffmpeg).
    * SAMPLE_RATE: the sample rate used for fingerprinting as well as for parsing audio files
    * CHANNELS: set 1 for mono and 2 for stereo.
    * BATCH_SIZE: the number of video files processed in each batch.
    * AUDIO_FILE_TYPE: the output type for audio files.

- db_handler: contains db_handler.py wherein exists the DatabaseHandler class, a wrapper class around the sqlalchemy[2] package. This class is used for invoking database operations, such as creating databases, tables, inserting lines, deleting schemas, tables, lines and simple queries.

- feature_extractor: contains three modules with classes that extract features for each of the modalities of interest.

  – audio_features.py, which contains the AudioFeatureExtractor class. For more details on how it works, please refer to section 3.3.

  – video_features.py, which contains the VideoFeatureExtractor class. For more details on how it works, please refer to section 5.4.

  – symbolic_features.py, which contains the SymbolicFeatureExtractor class. For more details on how it works, please refer to section 4.2.2.

- fingerprinting: contains djv.py which has the wrapper functions around the pyDejavu[3] library, for fingerprinting. For more details please refer to section 2.5.1.

- media_manipulation: a module that contains all the scripts for manipulating audio and video data.

  – audio_conversions.py: includes wrapper functions for invoking ffmpeg commands to convert audio files into the appropriate format for fingerprinting.

[2] https://www.sqlalchemy.org/
[3] https://github.com/worldveil/dejavu

  - `song_retrieval.py`: contains all the functions needed to search within a video for songs. For more information please proceed to section 2.5.
  - `video_manipulation.py`: includes wrapper functions for invoking ffmpeg commands to crop videos and mix audio with video.

- `spotify_wrapper`: includes a wrapper class around the `spotipy`[4] library. We use this class to retrieve information on song titles, during the matching of audio and MIDI files. For more information, please proceed to section 2.4.4.

- `utils`: a module containing various utilities for the all the other modules and scripts. These include:
  - `catalog_utils.py`: which contains utility functions for scanning folder directories and creating catalog entries within the database.
  - `common_utils.py`: which contains utility functions for time calculations, and other miscellaneous tasks.

2.1.1 Pipeline Overview

In figures 2.1 and 2.2 we provide a brief overview of the process that will be explained in detail in the following sections. The flowcharts also include the names of the Python scripts that are included in the root directory of the repo, and the sequence of execution is explained.

2.2 Dependencies with 3rd Party Libraries

All code is written in Python, bash and SQL, run and tested in Ubuntu Linux 20.04. There are some dependencies with 3rd party software beyond those that are mentioned in the requirements document in the repo. These are:

- `ffmpeg`[5]: A complete, cross-platform solution to record, convert and stream audio and video.

[4] `https://spotipy.readthedocs.io/en/2.16.1/`
[5] `https://ffmpeg.org/`

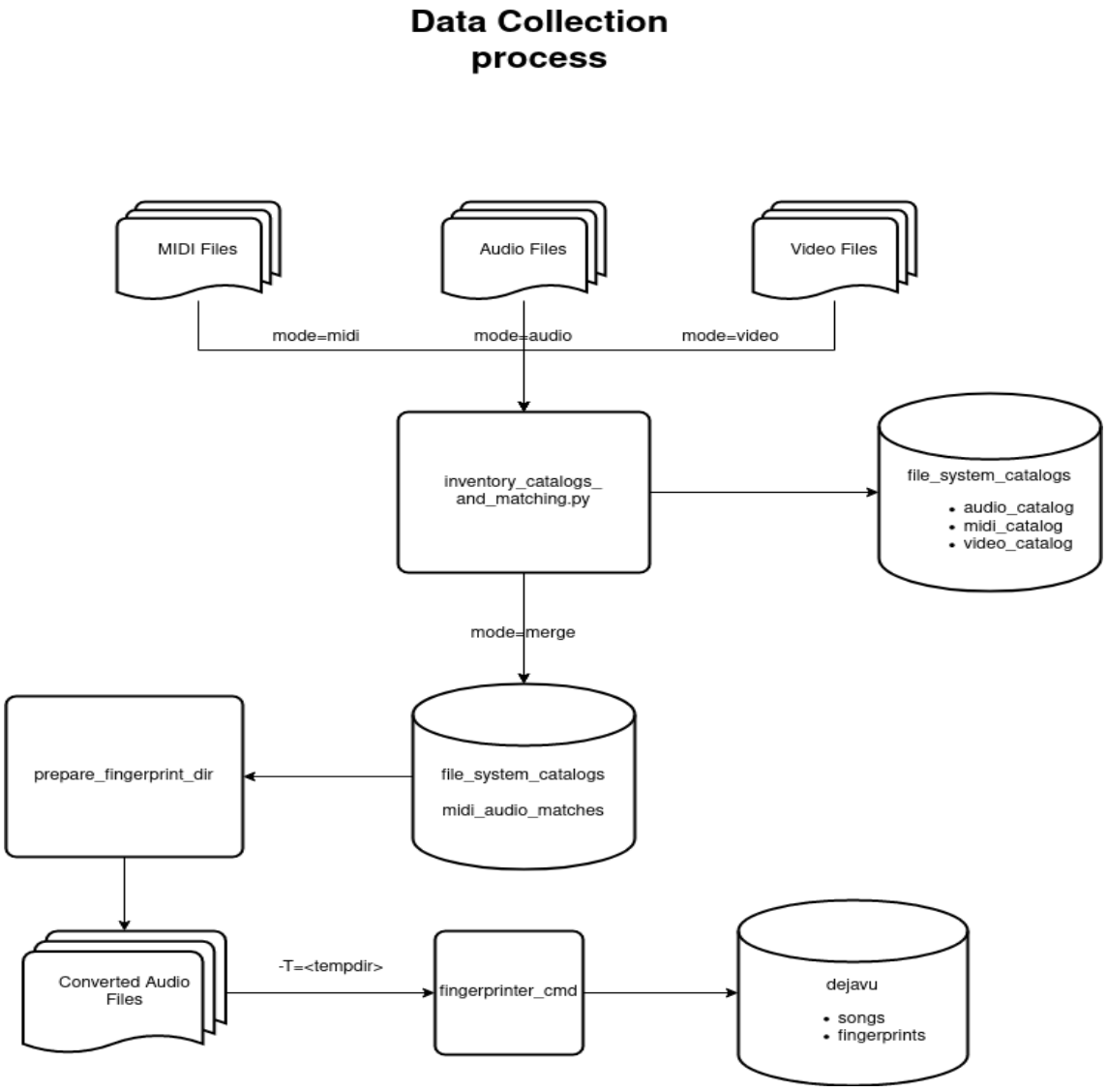


Figure 2.1: From raw files to relational tables

- `MySQL`[6]: An open-source database
- `Cuda Toolkit`[7] (optional) If available, it would speed up the extraction of visual features.
- `MuseScore 3`[8]: an open source software for visualizing MusicXML files.
- `FluidSynth` [9]: a real-time software synthesizer based on the SoundFont 2 specifications

[6] `https://www.mysql.com/`
[7] `https://developer.nvidia.com/cuda-downloads`
[8] `https://musescore.org/en`
[9] `http://www.fluidsynth.org/`

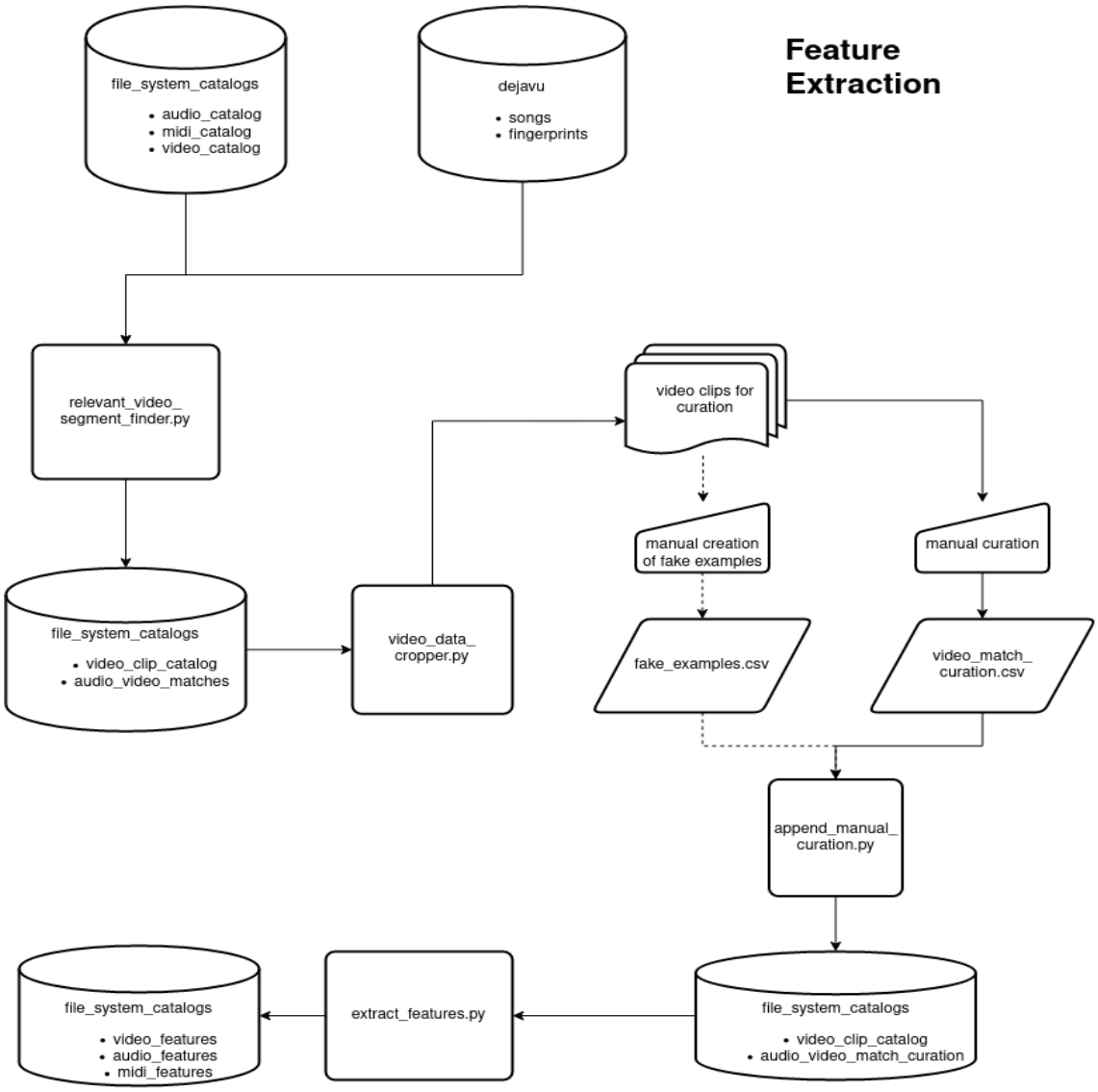


Figure 2.2: Finding songs in videos and extracting features

- `QjackCtl`[10]: a simple Qt application to control the JACK sound server daemon, specific for the Linux Audio Desktop infrastructure.
- `QSynth` [11]: a fluidsynth GUI front-end application written in C++ around the Qt framework using Qt Designer.
- `lilypond`[12]: a music engraving program, devoted to producing the highest-quality sheet music possible. We use it for some of the visualizations of sheet music in this thesis.

[10] `https://qjackctl.sourceforge.io/`
[11] `https://qsynth.sourceforge.io/`
[12] `http://lilypond.org/`

2.3 Alternative Data Sources

Our search[13] yielded the following data sets:

- The `AudioSet` Soundtrack data set from Google[14] which contains audio features from YouTube videos that contain soundtracks. The main problem is that the videos included are parsed from YouTube and too many of them consist of music playing over static images. We therefore couldn't yield good enough results by downloading the YouTube video. The data-set also lacks the track name and any symbolic representation.

- `Soundtracks data-sets for music and emotion`[15] by the University of Jyväskylä, which consists of short (approx. 15 second) excerpts from film soundtracks. The main problem is that there is no video or symbolic information.

- Plenty of MIDI data-sets in `Kaggle`[16], most of which are classical composers works and aren't relevant to the task.

- The `movie-net`[17] data-set, which is a huge collection of data around movies, including video and audio features. At the time of writing this, the underlying data-set of movies is not publicly available, therefore it wouldn't be possible to find the point in each film where a song is used. Thus, it also wouldn't be possible to synchronise video with music. Also there is no symbolic data for the songs or soundtrack titles.

- The collection of MIDI data sets found in `composing.ai`[18], which gathers several data sets of popular music in MIDI form.

[13] search keywords "film soundtrack dataset", "midi dataset" in Google, "video music midi", "soundtrack" in Kaggle

[14] `https://research.google.com/audioset/dataset/soundtrack\_music.html`

[15] `https://www.jyu.fi/hytk/fi/laitokset/mutku/en/research/projects2/past-projects/coe/materials/emotion/soundtracks/Index`

[16] `https://www.kaggle.com/search?q=music+video+midi+in\%3Adatasets`

[17] `http://movienet.site/`

[18] `https://composing.ai/dataset`

It became apparent that the task could not be carried out using one of the above data sets. To address that, we decided to create a data collection pipeline of our own, using the `composing.ai` collection of MIDI files as the basis for our further work. In the following sections we describe the method that we followed in detail.

2.4 Collection, Cleanups and Storage

In the following paragraphs we describe the available data and their sources.

2.4.1 MIDI

We downloaded the MIDI data from `composing.ai` *(last retrieved on 2020-05-02)*, which contains 124,470 files. The main problem of this data-set is its lack of structure. The files come from different data sources, using different naming conventions and directory structures, making it quite hard to determine which songs are actually available in this directory. The script `inventory_catalogs_and_matching.py` which is situated in the root directory, handles this job, when used in "midi mode". It goes through the directories and uses some rules in order to ignore irrelevant filenames. We then used some regular expressions to perform a cleanup of the names, which we passed on to the Spotify API in order to get a proper name of each MIDI filename, as well as relevant artist and URL information. The match rate is currently at 61%.

The resulting MIDI file set is useful, but comes with its own limitation for the task at hand, which has to do with the fact that it mostly consists of known pop and rock songs. While a lot of films use these as background music for montages, due to the fact that they are usually expensive to obtain, this limits the pool of films and scenes we can draw from. Films usually also have their own score, which is unique to the film and in most cases carries the bulk of the scenes and the emotional core of the film.

Furthermore, the MIDI files themselves are "dirty". They do not follow the same naming conventions for instruments or quantizations, some of them are plain wrong, and the fact that they are multi-tracks and contain multiple percussion elements (as

pop and rock songs have a strong emphasis in rhythm) adds further problems when parsing for the feature extraction process (see section 4.2.1). Nevertheless, as this is the only large enough set of MIDI files that we could find for free, we decided to work with this. The resulting MIDI dataset is comprised by 43,567 MIDI files.

2.4.2 Audio

The audio data is based on a personal collection of MP3 files. This provides some challenges, as a lot of the data might be misleading for our task at hand, as will be explained in sections 2.4.4.2 and 2.5.2.

The collection has an initial catalog of 55,800 files from different genres and of varying audio quality. They are arranged in directories according to genre, country and artist. Here we present some high-level statistics, to get a context of what we are working with:

- 32,319 files belong to a directory that has the word 'rock' in it.
- 6,696 files belong to a directory that has the word 'jazz' in it.
- 239 files belong to a directory that has the word 'classical' in it.
- 4,833 files belong to a directory that has the words 'live', 'concert' or 'bootleg' in it. Out of these, 3,385 also contain the word 'rock'.

As was the case with the MIDI files, the files are "dirty", as they do not follow strict naming conventions for the song titles, and the sampling rates and bit rates vary.

2.4.3 Video

Video data is based on our private film collection. The collection includes 106 films and 37 episodes from television series. The files are in various types: avi, mp4 and mkv. As was the case with the other two modalities, the naming conventions and image quality of the raw data are inconsistent.

2.4.4 Cataloging and Matching MIDI to Audio Data

As we hinted at in the previous sections, we utilized a rule-based method to connect our MIDI with our Audio data. The work is done in stages, using a Python script that we have named `inventory_catalogs_and_matching.py`. The script works in the following modes:

- *all:* runs the whole pipeline, creating all the catalogs for MIDI, Audio and Video files, as well as matching MIDI and Audio files.
- *midi:* creates a catalog for MIDI files, assumed to be within the path specified in `config/paths.py`. The script assumes that the data set is the one described in 2.4.1, and therefore has some hard-coded cleaning functions that skip files that are not MIDI, and also folders that have MIDI files without the song name on the title (as there are a lot of MIDI files with hashed titles). The script then proceeds to do some cleaning of these names to remove irrelevant punctuation marks etc. As a final step, these cleaner titles are fed to the Spotify API in order to get a matching song name and URL, and the data is stored in the table `midi_catalog`.
- *audio:* creates a catalog for audio files, assumed to be within the path specified in `config/paths.py`. Follows a similar cleanup process, removing directories that would introduce noise in the data-set later on (for example the 'Bootlegs' directories'), and files that do not contain music. After cleaning up the titles of the files, it feeds them to the Spotify API in order to get a matching song name and URL, and store the data in the table `audio_catalog`.
- *video:* creates a catalog of video files, assumed to be within the path specified in `config/paths.py`. The cleanup process is much milder, as the names of the videos are not of such importance at any later stages. An additional column called *searched* is defined, to be used during the video searching phase by `relevant_video_segment_finder.py`, in order to discriminate the videos that have already been parsed for song detection. This script is incremental, so that more video files can be added later on, if more are found.

- *merge:* performs an inner join of the audio and MIDI catalogs, based on the song name as given by the Spotify API (for more details on the matching process see paragraph 2.4.4.1). For those that are matched, a pair id is created, and it is stored, along with the relevant IDs for audio and MIDI files, to the table `midi_audio_matches`. In our case, the matching resulted in **3,109 audio-MIDI matches**.

The functions used to perform the file name cleanup are shown in code listing 2.1.

Listing 2.1: The file title cleanup code

```
1  import re
2  import pandas as pd
3
4
5  def cleanup_file_titles(df, file_type, allow_numbers=False):
6      """
7      get the titles of a file and return a DataFrame with a title column and a filetype column
8      :param allow_numbers: boolean that passes to the regex cleaner function
9                            and allows numbers in the output.
10     :param df: the midi track catalogue
11     :param file_type: the type of files in the catalogue. Possible values: "audio", "video", "midi"
12     :return: df with an extra column with titles and file type
13     """
14     suffix = determine_relevant_suffices(file_type)
15     titles = []
16     file_types = []
17     for filename in df['filename']:
18         # only keep relevant file types
19         suf_search = re.search(suffix, filename)
20         if suf_search:
21             # strip from suffix
22             stripped = suf_search.group(1)
23             words = reg_cleaner(stripped, allow_numbers=allow_numbers)
24             titles.append(words)
25             suf = suf_search.group(2).strip(".")
26             file_types.append(suf)
27         else:
28             titles.append('')
29             file_types.append('')
30     df['title'] = titles
31     df['file_type'] = file_types
32     df = df[df['title'] != '']
33     return df
34
```

```
35
36 def reg_cleaner(string, allow_numbers=True):
37     """
38     simple function to convert string to lowercase and remove special characters
39     :param allow_numbers: specify if you want to allow numbers or not (default yes)
40     :param string: the string you want to clean
41     :return: the clean string
42     """
43     if allow_numbers:
44         remove_special = re.sub('[^A-Za-z0-9]+', ' ', string)
45     else:
46         remove_special = re.sub('[^A-Za-z]+', ' ', string)
47     pascalcase = re.sub(r'([a-z](?=[A-Z])|[A-Z](?=[A-Z][a-z]))', r'\1 ', remove_special)
48     clean_string = pascalcase.lower()
49     removed_stopwords = remove_stopwords(clean_string)
50     removed_multi_space = remove_multi_spaces(removed_stopwords)
51     return removed_multi_space
52
53
54 def remove_stopwords(string):
55     removed_stopwords = ' '.join([word for word in string.split() if word not in FILENAME_STOPWORDS])
56     return removed_stopwords
57
58
59 def remove_multi_spaces(string):
60     multiple_spaces = re.compile('(\s\s+)')
61     stripped_string = re.sub(multiple_spaces, ' ', string)
62     return stripped_string
```

2.4.4.1 The Spotify API

In order to match the audio and MIDI files, we needed to use a knowledge base that could provide a ground truth for song information. We chose the popular music streaming platform `Spotify`[19]. The platform provides a web-based API, which we access using the relevant Python library `spotipy`[20].

In order to run the code, it is necessary to set up a free account in order to complete the user authorization in each call. An app needs to be registered at MyDashboard[21] to get the credentials necessary to make authorized calls (a client id and client secret). In order to achieve the maximum reply rate possible, we used

[19] `www.spotify.com`
[20] `https://spotipy.readthedocs.io/en/2.16.1/`
[21] `https://developer.spotify.com/documentation/web-api/`

the client credentials authorisation flow. These credentials are stored in the file `config/credentials.py`.

The class that we have created, named `Spotify`, is a rudimentary wrapper class. It exposes the functions that are useful for the matching process, namely the song searching function, that returns a song name and metadata. The basic code can be found in code listing 2.2

Listing 2.2: The Spotify API wrapper

```
1  import spotipy
2  from spotipy.oauth2 import SpotifyClientCredentials
3  from config.credentials import spotify_creds
4
5  client_credentials_manager = SpotifyClientCredentials(client_id=spotify_creds['clientID'],
6                                                        client_secret=spotify_creds['clientSecret'])
7  sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
8
9
10 class Spotify(object):
11
12     def __init__(self):
13         pass
14
15     @staticmethod
16     def ask_spotify(title):
17         """
18         for a potential title, ask spotify for a name
19         :param title: a string
20         :return: list of song attributes
21         """
22         song = sp.search(q=title, limit=1, type='track')
23         try:
24             info = song['tracks']['items'][0]
25             if info:
26                 artist = info['artists'][0]['name']
27                 song_name = info['name']
28                 song_url = info['external_urls']['spotify']
29                 data = {'name': song_name, 'artist': artist, 'URL': song_url, 'clean_title': title}
30             return data
31         except IndexError:
32             pass
```

2.4.4.2 Caveats in audio-MIDI matching

As was mentioned in the beginning of section 2.4.4, our method yields **3,109 audio-MIDI matches**. While this number seems adequate, there are some caveats. We try to describe most of them below:

- *The recall of the Spotify search API.* This depends on the level of cleanliness of the original file names, as well as how well our custom regex-based cleaner works. This is evident in the fact that in the audio files (which had much better initial file names) recall is considerably higher than in the (much "dirtier") MIDI files. More specifically, in the audio files, recall reaches 83.3%, whereas in the MIDI catalog, it's only 65.2%. This means that we lose a lot of MIDI data when we apply the matching process.

- *The precision of the Spotify search API.* This depends on the level of accuracy of the original files, i.e. whether the file name reflects the actual content of the file. Due to the size of the data-set, we weren't able to precisely compute the precision of the method. Nevertheless while performing manual curation of the matched files, in order to create fake examples for our classifier, we came across plenty of files that were wrongly matched. This has a few implications:

  - There were plenty of audio files that weren't the original versions of the songs, (most often being covers in other genres, or live versions).

  - There were cases where the MIDI and audio files had the same song title, but were referring to different songs with the same name. Since the Spotify search API yields the most popular result, one of them (or both) were falsely identified.

  - A lot of jazz songs have titles that closely resemble other popular songs, therefore there were cases where the audio was falsely identified as a more popular song by the Spotify API.

- *The quality of MIDI data.* It is often the case that MIDI data is corrupt, or otherwise inappropriate for further processing. Nevertheless this could only

become evident in the feature extraction process, which happens much later in the pipeline.

2.5 Finding Music within Videos

The ultimate goal in our data collection pipeline is finding combinations of videos and songs, in both audio and symbolic formats. Initially, we tried starting from a knowledge base like `iMDb`[22] that would contain movie soundtrack information. Nevertheless, the format of our film collection and the structure of the iMDb search API wouldn't allow for this process to run smoothly. Besides, even if we did follow that route, the database wouldn't contain the exact timestamp of the song's appearance in the film.

We therefore decided to follow a different approach, one that would allow a future expansion to videos that aren't necessarily listed in external knowledge bases; we created the script `relevant_video_segment_finder.py` which, given a database of song fingerprints (see section 2.5.1 for details), breaks the video in chunks and compares the audio against said fingerprints. The comparison is done using the same parameters as the ones that were used while building the fingerprint database, and are stored in `config/settings.py`. If a match is found for more than three consecutive video chunks with the same song, the clip is stored in the table `video_clip_catalog`. The match information between audio and video clips is stored in `audio_video_matches`.

In our experimental setup we chose the following settings, after some fine-tuning:

- *Video chunk size:* 5 seconds
- *Audio sample rate:* 16 KHz
- *Audio channels:* 1

In order to increase the recall of the method, we impose the following rule: For every three video chunks, if the first and third are matched with the same song,

[22] `www.imdb.com`

then we impose the same match to the middle one too. Given our objective, we also impose a minimum size of three chunks to each extracted video clip, in order to maintain a balance between keeping irrelevant minuscule clips and missing the opportunity to get larger clips by combining chunks together. Our implementation is demonstrated in Listing 2.3

Listing 2.3: Smoothing of Song Detection Chunks

```
def smooth_chunk_matches(chunk_match_data):
    """
    for each row, if the previous match is the same as the next one, change the label of the row and set th
    the mean between the 2 rows.
    :param chunk_match_data:
    :return: smooth song label, smooth song offset
    """
    chunk_label = chunk_match_data['song_id']
    chunk_offset = chunk_match_data['offset_seconds']
    prev_chunk_label = chunk_match_data['prev_song_id']
    prev_offset = chunk_match_data['prev_offset_seconds']
    next_chunk_label = chunk_match_data['next_song_id']
    next_offset = chunk_match_data['next_offset_seconds']
    if needs_smoothing(prev_chunk_label, next_chunk_label, chunk_label):
        return prev_chunk_label, smooth_chunk_offset(prev_offset, next_offset)
    else:
        return chunk_label, chunk_offset
```

Ideally, we would also want to maximise precision. To that end, we use the song fingerprint offsets per video chunk, which are calculated by `pyDejavu`. These offsets demonstrate which part of the song corresponds to the matched video chunk. We then calculate the mode (most frequent value) of these offsets for each clip (which is a collection of at least 3 chunks) and flag those clips that have no mode. In a perfect match scenario, the mode of these offsets would be equal to the chunk length. Our flagging process is presented in Listing 2.4.

Listing 2.4: Tagging Misaligned Song Offsets

```
def flag_possible_errors(df):
    filter_df = df.loc[df['match_id'].notna()].copy()
    grouped_df = filter_df.groupby('match_id')['offset_diff']
    modes = grouped_df.apply(get_match_area_mode)
    valid_modes = modes[modes.notna()].index.values
    sizes = grouped_df.size()
    tp = sizes[sizes >= 3].index.values
```

```
8     filter_df['too_small'] = [0 if i in tp else 1 for i in filter_df['match_id'].values]
9     filter_df['invalid_mode'] = [0 if i in valid_modes else 1
10                                 for i in filter_df['match_id'].values]
11    return filter_df
```

Overall the method yields 68 video clips.

2.5.1 On Fingerprinting

One of our main challenges during the data collection process was finding songs from our database within films. To tackle that issue, we used an audio fingerprinting technique, implemented in a Python library called `pyDejavu`.

As explained in [18], an audio fingerprint is a compact content-based signature that summarizes an audio recording. Audio fingerprinting technologies extract acoustic relevant characteristics of a piece of audio content and store them in a database. When presented with an unidentified piece of audio content, characteristics of that piece are calculated and matched against those stored in the database. Using fingerprints and matching algorithms, distorted versions of a single recording can be identified as the same music title.

The implementation of audio fingerprints in `pyDejavu` uses the spectrogram (see Section 3.2.5.1 for more) as the basis for creating the fingerprint. As described by the creator of the library Drevo [19], the algorithm finds peaks in the spectrogram, which are defined as time-frequency pairs that correspond to an amplitude value which is the greatest in a local "neighborhood" around it. Other such pairs around the peak are lower in amplitude, and thus less likely to survive noise. To find the peaks, `pyDejavu` is implementing a combination of a high pass filter and local maxima structs from the Python library `SciPy`.

The spectrogram peak frequencies along with the time difference between them are then passed through a hash function (SHA-1), representing a unique fingerprint for this song. In order to save space, the SHA-1 hash is cut down to half its size (just the first 20 characters), and then converted to binary, reducing the fingerprint's size from 320 bits down to 80 bits. After the database is filled with the fingerprints of the available songs, a new audio can be matched using the same hashing method.

An important factor in the success of the matching is hash alignment. When doing the original fingerprinting of a sample, the *absolute* offset, with regards to the beginning of the song, is stored. When the captured sound that is to be compared with the database is fingerprinted, the offset is *relative* to the start of the sample playback. If we make the assumption that the playback speed and sample rates are identical between the songs in our database and the input, then it follows that the relative offset should be the same distance apart. Under this assumption, for each match the difference between the offsets is calculated:

$$\text{difference} = \text{database offset from original track} - \text{sample offset from recording} \tag{2.1}$$

This always yields a positive integer since the database track will always be at least the length of the sample. All of the true matches will have this same difference. The system then looks over all of the matches and predicts the song ID that has the largest count of a particular difference.

We used the following settings when importing `pyDejavu`:

- Sampling Rate: 44100
- FFT Window Size: 4096
- FFT Overlap Ratio: 0.5
- Fan Value[23]: 15
- Minimum Amplitude of Peaks: 10
- Minimum Number of Cells Around an Amplitude Peak: 10

As these settings are not exposed in the library's API, we had to apply the changes locally. Prior to initialising the fingerprint database, we applied pre-processing in the original data, in order to bring the audio files in the same sample rate and number of channels as the videos (as explained in section 2.5.

[23]Degree to which a fingerprint can be paired with its neighbors.

2.5.2 Caveats and Challenges of this approach

While the proof of concept for utilizing `pyDejavu` was initially promising, we came across some significant caveats that should be taken into consideration in the future.

2.5.2.1 Execution Time

Filling in the database takes a lot of time, especially when using settings that favour high accuracy. We have identified the main bottleneck as the data input in the MySQL database. Using the settings described in 2.5.1 we calculate an average of 104,882 fingerprints per song, and total running time to parse 3,109 songs was approximately 210 hours, even though we had three instances of the script working in parallel.

When debugging the script, we identified that it takes almost 45 minutes to calculate fingerprints for a batch of 200 songs, but it takes 12 hours to perform the database insert for each batch. This translate to 3 minutes per song insert, which makes sense, if each song insert translates to 100 thousand row inserts. We tried to perform some optimisation in the INNODB settings of the MySQl server, but we didn't see considerable improvements.

This meant that each experiment would take approximately a whole week to run, making it very hard to experiment on the whole pipeline end-to-end.

2.5.2.2 Storage Size

As we explained in the previous section, each song would be represented with a hundred thousand rows on average. This leads to a crucial trade-off between storage size and accuracy. The most important parameters that affect this trade-off are summed in table 2.1.

In our implementation, the `dejavu` schema takes up 32 GB, which is larger than even the original storage size of the raw data which is 27 GB. For systems where storage and scalability play an important part, this would probably be a concern.

Parameter Name	Size	Accuracy
Sampling Rate	+	+
FFT Window Size	-	-
FFT Overlap Ratio	+	+
Fan Value	+	+

Table 2.1: `pyDejavu` parameters, storage size and detection accuracy

2.5.2.3 Playback Speed and Sample Alignment

The biggest concern that arises from this fingerprinting application is that of the alignment of samples between the audio signal that is being searched and the fingerprint database. As was explained in section 2.5.1, the assumption is made that the playback speed and sample rates are identical between the songs in our database and the input. However this is not always the case in our particular scenario.

While running the pipeline, we encountered cases where the song that was used in the video, while it existed within the database, would not be matched. Initially we hypothesized that this has to do with the quality of the original song, or that the film excerpt was too noisy. However, we noticed that the algorithm did surprisingly well in very noisy scenes, whereas it would miss quite obvious and prominent clips. Nevertheless when playing back the video, we realised that the song in the film was in fact played at a different speed to the original, which also resulted to a slightly different pitch. This goes against the assumption that was made above, and it leads to a failure to match songs in a lot of films.

Furthermore, the fact that we apply re-sampling in the audio files, before initialising the fingerprint database may also play a part in the process, even though our tests with fabricated examples didn't suggest that this was the case.

2.5.2.4 Quality of the video content

Due to the nature of the dataset, it is often the case that music is used in an inconsequential way. Such would be the cases when the song is so far in the background of the audio mix, or is so generic that virtually any song could be used in its place. Furthermore, it is possible that the song is only used in the end credits of a film or

TV episode. In these cases, when taken out of context, the visual content wouldn't really carry enough information to be easily classifiable.

This is an issue that we faced in our implementation too, but due to the size of the dataset, we decided to also keep those video clips in the dataset.

2.5.2.5 Manual Intervention

Manual interventions in such pipelines, especially when trying to build large data-sets, should preferably be avoided. Nevertheless, due to the nature of the data, we needed to intervene manually in two situations:

- *Curation of audio - video matches:* After we run the script that finds songs in videos (`relevant_video_segment_finder.py`), we go through its results to make sure that the matches are indeed correct and appropriate. We store the results in a file within the *var* folder.
- *Creation of fake examples:* Initially we considered using the segment finder script's mismatches as negative examples in the classifier that we built. Nevertheless, we found that this could result in an imbalance data-set and, further to that, these mismatches could, in fact, be potentially good alternative songs for a scene. We, therefore, chose to handcraft fake examples for the negative class, in order to make sure that the resulting data would be balanced, and that the visual content would indeed not match the song.

As is the case with manual interventions do not scale, and are not appropriate for building larger data-sets.

2.5.2.6 Bias

As is true in almost everything, most of our design choices introduce biases in the resulting data-set. Reflection on the process reveals the following sources of bias:

- *The available songs and videos.* As the raw content is based on our privately-owned collection, the limits of its diversity are the limits of our own taste. This could be problematic when trying to train a general-purpose classifier.

- *Curating the audio-video matches.* Manual curation of a data-set can lead to oversights that could increase bias in a model. This is especially true as the size of the data-set increases.

- *The notion of a song fitting a scene.* The definition of whether a song is a good fit for a video or not is a subjective matter. In our work we have implicitly assumed that the choices of the original music directors were good fits, even though some other music director could have chosen something else. Further to that, the manufactured "bad" examples are only bad because we judged them as such. This further intensifies the problem of creating a data-set that is biased to our personal taste.

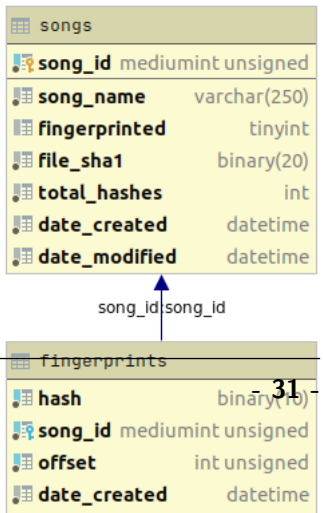
- *Cultural bias.* The data-set is comprised almost exclusively from Hollywood films and series, and the songs that were available in both MIDI and audio formats were mostly western pop and rock ranging from the 1960s to the 2000s. When training a model, this would of course introduce cultural bias to the model.

2.6 The Database structure

All the data in this process are stored in a MySQL database. There are two schemas involved. The principal schema is **file_system_catalogs** where all the work described in sections 3.3, 5.4, 4.2.2, 2.4 and 2.5 is stored. An overview of the table relations can be seen in Figure 2.3.

An additional auxiliary schema is built by the `pyDejavu` library, which is called **dejavu**, and contains the necessary data to perform fingerprint recognition queries.

All of the tables are created automatically through the scripts stored under `db_handler/sql`, which are called when necessary.



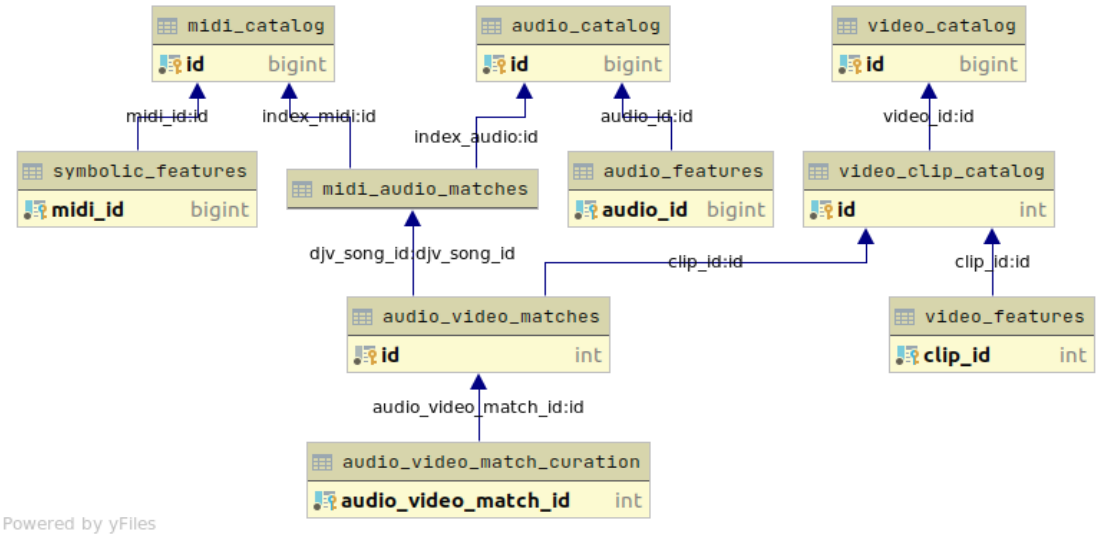


Figure 2.3: The file_system_catalogs relational schema

2.7 Concerns About Copyright

One of the main reasons that made the construction of such a pipeline necessary, was the fact that all of the raw data that are relevant to this task (music, compositions, films, videos) are subject to copyright. During the process of building this data-set, we were mindful of this, and have designed the pipeline in such a way, that copyright would be respected:

- Storing the raw data itself is not necessary. Once the feature extraction process is completed, only the comprehensive feature library is stored, along with some rudimentary metadata, so that the features can be traced back to their source.

- The raw data used is either freely available on the Web (in the case of MIDI files), or is part of our privately owned collection. At the end of the process only the database and the code is publicly shared.

- The database is expandable. All the catalog processes, feature extraction methods and matching functions work in such a way that it is possible for anyone to add more data from their own collection. Potentially this could

lead to a crowd-sourced data-base of features, without ever sharing any of the underlying raw data.

Chapter 3

Music Representation and Feature Extraction Techniques from Audio

This, and the chapters that follow, focus on the issue of data representation, i.e. how musical and visual content is represented. It is evident that in order to be used in a model, the data needs to be somehow converted to a numerical and tabular format. There are plenty of considerations that need to be made when choosing a representation strategy for audio and visual data, as well as the suitable encoding method for each type of data. In the following chapters, we will discuss the issues that need to be taken into account when choosing representation, encoding and feature extraction strategies for each of the other relevant modalities.

We will attempt to explain the difference between representations of music based on audio processing, and higher level symbolic representations. We will then focus on audio content and organise the rest of the chapter in sections: one is for representation and feature extraction strategies, drawing mainly from the works of Giannakopoulos and Pikrakis [20] and Kim et al. [1]; further to that, we will present the features that we have extracted for this thesis.

3.1 Symbolic vs. Raw Audio

Given that our task does not concern general audio signals, but music of some form, we have the choice of representing it as raw audio, or as some higher level

symbolic representation. The techniques for processing and transforming each one of these is quite different, with the former leaning more towards the realm of signal processing, and the latter being closer to the domain of knowledge representation.

This choice is particularly important as each type of representation reveals different aspects of the content. Audio features based on signal processing, might reveal more about timbre and texture of a piece (such as the energy of the signal, or frequencies present), while symbolic representations allow us to extract elements that belong to musicology, such as the flow of harmony, cadence and melodic structures.

Furthermore, if the resulting classifier was to be used as a method of evaluation, or an objective function for an architecture that generates music for videos, it should be able to handle both types of representations. Therefore, including the musical content in both modalities, is very important.

3.2 Representations based on Audio Processing

In the following sections we will examine raw audio representations. In order to understand the basic concepts, we will use the standard notation from the MPEG-7 standard [1], and briefly explain how physical audio signals are captured, represented and then transformed in various ways within the digital domain. To that end, we will mostly incorporate elements from the works of Kim et al. [1] and Giannakopoulos and Pikrakis [20].

3.2.1 Sampling and Sampling Frequency

Before we dive into the various representations, it is useful to define the process of creating a digital signal from its analog audio counterpart. Even though in nature time has a continuous flow, in the digital realm we need to manipulate samples of the real signal that have been drawn on discrete-time instances, a process which is known as **sampling** [20]. The time between two samples taken by a signal is called the **sampling period** and the inverse of the sampling period is called **sampling**

[1]MPEG - 7 is an international standard, proposed in 1997 and established in 2001, defining descriptions and description systems for searching, identifying, filtering and browsing audiovisual content.

frequency. This is usually notated as F_s and measured in Hertz (Hz). For example, if the sampling frequency is 1000 Hz, this means that the sampling period is $\frac{1}{1000} = 0.001s$ and therefore one sample is drawn every 0.001 seconds.

In this context, the sampling frequency is important; the higher it is, the closer the digital file resembles the initial analog audio signal. On the other hand, there is a lower bound for the sampling frequency, known as Nyquist rate, which is equal to twice the signal's maximum frequency. This rate ensures that the phenomenon known as aliasing is avoided, and that the resulting audio quality is adequate.

3.2.2 Short Term Audio Processing

In addition to sampling, another technique that is very important when it comes to raw audio manipulation and feature extraction is short-term processing. In this process, the audio signal is broken into short-term windows (also known as **frames**) which can be overlapping. The analysis is then done on a frame-by-frame basis, which is a practical way of dealing with the non-stationarity of the audio signal, which usually has abrupt changes over time.

The framing logic works both in the microscopic scale of the samples of the signals, but also on the more abstract level of audio events, meaning that, as a practice, it is also useful in order to extract information from small segments of an audio file. In this context, the short term analysis can be seen as a compromise between working on a per-sample basis - which would potentially entail huge memory and computational loads - and a per-file basis - which would not take into account these abrupt changes in the signal and lose too much information.

We formalize this process in the following equation:

$$x_i(n) = x(n)w(n - m_i), \qquad i = 0, \ldots, K - 1 \tag{3.1}$$

where K is the number of frames and m_i is the **shift lag**, which is the number of samples by which the window is shifted in order to yield the ith frame. The only region of samples that yields non-zero frames has indices $m_i, \ldots, m_i + W_L - 1$, where W_L is the length of the moving window (in samples). The value of the m_i depends

on the hop size (or step) of the window, notated as W_S. Specifically the value of m_i can be calculated with the formula:

$$m_i = i \cdot W_S \cdot F_s \qquad i = 0, \ldots, K-1 \tag{3.2}$$

In order to understand this, we include here an example from Giannakopoulos and Pikrakis [20]: If the window is shifted by 10 ms at each step and the sampling frequency F_s is 16 kHz, then $m_i = i \cdot 0.01 \cdot 16000 = i \cdot 160$ samples, $i = 0, \ldots, K-1$. Furthermore, if $W_L = 300$ samples, then the 5th frame ($i = 4$) starts at the sample index $160 \cdot 4 = 640$ and ends at sample index $160 \cdot 4 + 300 - 1 = 939$

With the above, we wanted to highlight the importance of the parameters W_L and W_S, i.e. the length of the moving window and the hop size. According to [1], while MPEG-7 doesn't standardize the technique itself, a number of implementation features are recommended, such as a W_L of 30 ms and a W_S of 10 ms.

The total number of short term windows K is computed by

$$K = \left\lfloor \frac{N - W_L}{W_S} \right\rfloor + 1 \tag{3.3}$$

where W_L, W_S and N are defined above and $\lfloor\ \rfloor$ is the floor operator.

3.2.3 Mid-Term Windows and Feature Extraction

In order to extract features from audio files, as was hinted in 3.2.2, we usually apply another technique, named mid-term windowing. The audio signal is initially split into mid-term segments (windows), typically one to ten seconds long (depending on the application). Subsequently, the short-term processing stage is carried out as we previously described. The product of this step is a sequence of features, which is then used for computing feature statistics, e.g. the mean zero crossing rate of the window. In the end, each mid-term segment is represented by a set of statistics. Our implicit assumption in this process is that the mid-term windows exhibit uniform behavior with respect to audio type, therefore extracting these statistics is reasonable.

The same process is extended to longer files, in order to capture salient features

and represent them in a single vector which acts as a representative of the whole music signal. In these cases, After the short-term and mid-term steps produce a vector of feature statistics per segment (e.g. 2 seconds long), these statistics are then long-term averaged, in order to provide a single vector representation of the whole signal. Through this process, temporal evolution details are sacrificed in order to obtain the most notable features of the music signal. Despite this trade-off it has been a widely accepted technique for music genre classification tasks and other related problems (for example [21], [22], [23], and more recently [24]).

3.2.4 Time domain Representations and Features

3.2.4.1 Audio Waveform

The audio waveform (AWF) is the most direct representation of the raw audio signal. Essentially, it is a time series of the signal's amplitude and it considers the minimum and maximum samples within successive non-overlapping frames. Architectures that process the raw audio signal are sometimes referred to as end-to-end architectures [25], and the main advantage is that they keep the raw material with its full initial resolution. The disadvantage is the potentially huge computational load, both in terms of processing power and memory required. An example of a waveform visualisation can be seen in Figure 3.1

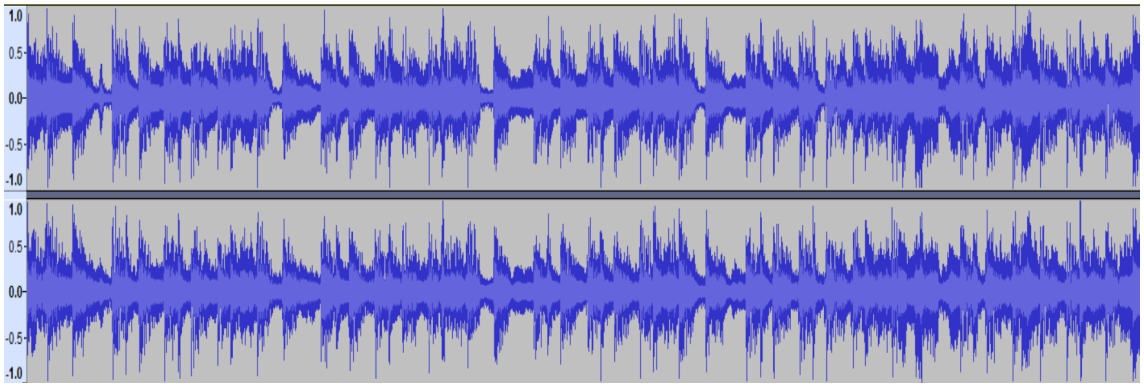


Figure 3.1: Example of a typical audio waveform

3.2.4.2 Energy

If $x_i(n), n = 1, \ldots, W_L$ is the sequence of audio samples of the ith frame, where W_L is the length of the frame, then the short-term energy is computed using Equation 3.4.

$$E(i) = \sum_{n=1}^{W_L} |x_i(n)|^2 \tag{3.4}$$

The *power* of the signal is obtained by dividing the energy by the length of the frame, which would transform Equation 3.4 to:

$$E(i) = \frac{1}{W_L} \sum_{n=1}^{W_L} |x_i(n)|^2 \tag{3.5}$$

The terms energy and power are used interchangeably in [20], and in further paragraphs we will use the term *energy* to refer to Equation 3.5

Short-term energy is a good feature to distinguish speech from music, as successive speech frames are expected to exhibit high variation, alternating rapidly between high energy states (while words are pronounced) and low energy states (during the silence between phonemes). A mid-term statistic that is usually chosen for classification tasks is the standard deviation σ^2 of the energy, or the standard deviation by mean value ratio $\frac{\sigma^2}{\mu}$

3.2.4.3 Zero-Crossing Rate

The Zero-Crossing Rate (ZCR) of an audio frame is the rate of sign-changes of the signal, from positive to negative and vice versa, divided by the frame length. We use Equation 3.6 to obtain it:

$$Z(i) = \frac{1}{2W_L} \sum_{n=1}^{W_L} |sgn[x_i(n)] - sgn[x_i(n-1)]| \tag{3.6}$$

where $sgn(\cdot)$ is the sign function:

$$sgn[x_i(n)] = \begin{cases} 1, & x_i(n) \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{3.7}$$

ZCR has usually higher values when the signal is noisy. It can, therefore, be interpreted as a crude measure of noisiness of a signal. This feature is easy to compute, and seems to provide adequate results. It is therefore popular even in recent papers that involve audio segmentation [26], [27], audio classification [28] and audio event recognition [29], [30].

In terms of mid-term statistics, it is worth noting that beside the average ZCR, the standard deviation of this feature over successive frames is higher for speech signals than for music signals, as is demonstrated in [20]. Therefore for such tasks it is preferable to the mean ZCR.

3.2.4.4 Energy Entropy

We can interpret the short-term entropy of energy as an indicator of sudden changes in an audio signal's energy, as its value is lower if there are abrupt changes in the energy of the signal. It is computed by first dividing each short-term frame in K sub-frames of fixed duration, and then calculating the energy of each sub-frame, j, and dividing it by the total energy of the short-term frame, as is demonstrated in Equations 3.8 and 3.9.

$$e_j = \frac{E_{subFrame_j}}{E_{shortFrame_i}} \tag{3.8}$$

where

$$E_{shortFrame_i} = \sum_{k=1}^{K} E_{subFrame_k} \tag{3.9}$$

Finally, the entropy $H(i)$ of the sequence e_j is computed as follows:

$$H(i) = \sum_{j=1}^{K} e_j \cdot \log_2(e_j) \tag{3.10}$$

Equation 3.8 helps us understand why energy entropy is lower if abrupt energy changes exist in the signal. If we interpret the resulting value of 3.8 as a probability, then whenever a sub-frame yields a high energy value, then one of the resulting probabilities will be high, thus reducing the entropy of e_j. This is therefore a potentially useful feature in the context of sound event detection, e.g. [31], as well as music genre detection [32].

3.2.5 Spectral Domain Representations and Features

3.2.5.1 Spectrogram

Transformed representations of audio are a way to compress the data and provide information of a higher level, in exchange for some information loss and bias. The most common transformation is the **Discrete Fourier Transform (DFT)**. According to Giannakopoulos and Pikrakis [20] the majority of important features used to analyze audio content, especially in more traditional settings, are defined in the frequency domain. In order to compute the coefficients of DFT, an efficient algorithm called **Fast Fourier Transform (FFT)** is used.

Given a discrete-time signal $x(n), \quad n = 0, \ldots, N-1, N$ samples long, its DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) \exp\left(-j\frac{2\pi}{N}kn\right), \qquad k = 0, \ldots, N-1 \tag{3.11}$$

where $j \equiv \sqrt{(-1)}$. The output of the transformation, notated here as $X(k)$ is a series of N coefficients, which are complex numbers.

The inverse DFT (IDFT) returns the original signal, given the DFT coefficients:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \exp\left(j\frac{2\pi}{N}kn\right), \qquad n = 0, \ldots, N-1 \tag{3.12}$$

or, equivalently:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\gamma_k(n), \qquad n = 0, \ldots, N-1 \tag{3.13}$$

where $\gamma_k(n) = \exp\left(j\frac{2\pi}{N}kn\right), n = 0, \ldots, N-1$. It is evident that the original signal $x(n)$ can be written as a weighted average of a family of fundamental signals, where each signal $\gamma_k(n)$ is a complex exponential and its weight is equal to the kth DFT coefficient. The magnitude of the kth DFT coefficient, $|X(k)|$, can act as a measure of intensity with which the corresponding frequency participates in the signal $x(n)$.

It is worth noting that there is a trade-off between the resolution in the time and frequency domains. The non-stationarity of the signal does not allow us to increase the length of the time frame without constraints, and simultaneously, the shorter the length, the higher the resolution in the frequency domain. This is derived from equation 3.12 if we understand that the kth exponential in discrete-time corresponds to $\omega_k = k\frac{2\pi}{N}$ and the equivalent of that in terms of the analog frequency is $f_k = k\frac{F_S}{N}$, with F_S being the sampling frequency that was used to obtain $x(n)$. The implication of this relationship is that, for a given sampling frequency, longer signals (i.e. larger values of N) lead to a more dense sampling in the frequency axis, therefore by increasing the length of the signal we produce a finer representation in the frequency domain. This trade-off will become important as we consider a short-term windowing approach later on.

Another important property of the DFT, is that for a signal of real values, the DFT coefficients appear in conjugate pairs:

$$X(k) = \overline{X}(N-k), \qquad k = 1, \ldots, N-1 \tag{3.14}$$

Therefore the magnitude of the spectrum is symmetric and, thus, we only need the first half of the DFT coefficients, i.e. those with indices $k = 0, \ldots, \lceil\frac{N-1}{2}\rceil$, where $\lceil\rceil$ is the ceiling operator. This means that in practice we only need the frequencies up to $\frac{F_S}{2}$, something that is also compatible with the Nyquist theorem, that we described in paragraph 3.2.1.

In order to make up for the trade-off between the resolution in the time and frequency domains, the Short-Time Fourier Transform (STFT) has been introduced. Its goal is to break the signal into possibly overlapping frames using a moving window technique and compute the DFT at each frame, much in the way that was discussed in paragraph 3.2.2. The length of the moving window is important, as longer windows lead to better resolution in the frequency domain but worse resolution in the time domain, given the sampling frequency. The MPEG-7 standard recommends that it is a multiple of 10 ms.

A common visual representation of the spectrum is the spectrogram, where the x axis represents time (in seconds), the y-axis represents the frequency (in kHZ) and the color represents the intensity of the signal (in a logarithmic scale, known as db). An example is shown in Figure 3.2.

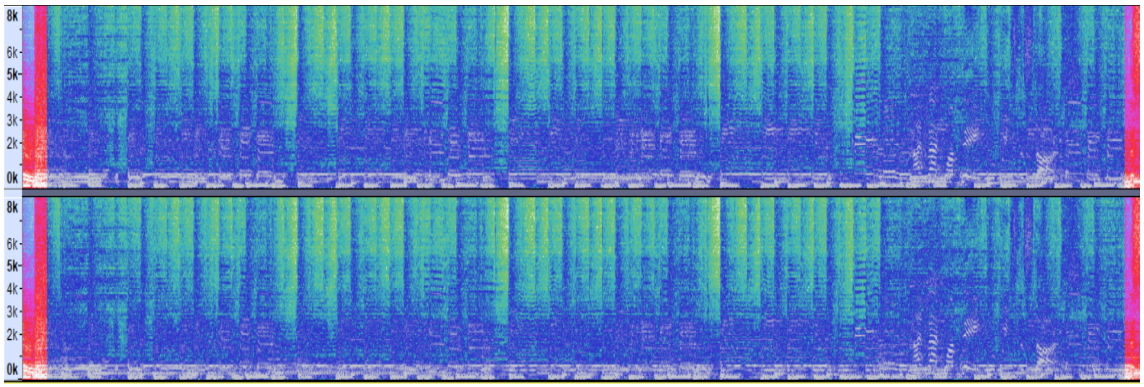


Figure 3.2: The spectrogram of the audio track of Figure 3.1

3.2.5.2 Spectral Centroid and Spread

Two simple measures for spectral position and shape are the spectral centroid and spectral spread, which correspond to the first and second central moments of the spectrum. We calculate the spectral centroid, using Equation 3.15.

$$C_i = \frac{\sum_{k=1}^{W_{f_L}} kX_i(k)}{\sum_{k=1}^{W_{f_L}} X_i(k)} \tag{3.15}$$

We calculate the spectral spread, using Equation 3.16.

$$S_i = \sqrt{\frac{\sum_{k=1}^{W_{f_L}} (k - C_i)^2 X_i(k)}{\sum_{k=1}^{W_{f_L}} X_i(k)}} \tag{3.16}$$

We interpret the spectral centroid as a measure of sonic brightness, while the spectral spread measures how concentrated is the spectrum around the spectroid.

3.2.5.3 Spectral Entropy

In a way similar to Paragraph 3.2.4.4, we calculate the entropy in the frequency domain. We first divide the spectrum of the short-term frame into L bins (sub-bands). The energy E_f of the fth sub-band, $f = 0, \ldots, L-1$ is then normalised by the total spectral energy:

$$n_f = \frac{E_f}{\sum_{f=0}^{L-1} E_f}, \qquad f = 0, \ldots, L-1 \tag{3.17}$$

The entropy of the normalised spectral energy is then computed with Equation 3.18.

$$H = -\sum_{f=0}^{L-1} n_f \cdot \log_2(n_f) \tag{3.18}$$

3.2.5.4 Spectral Flux

The spectral flux is a measure of change between two frames in the frequency domain. It is calculated as the squared difference between the normalized magnitudes of the spectra of the two successive short-term windows.

$$Fl_{i,i-1} = \sum_{k=1}^{W_{f_L}} (EN_i(k) - EN_{i-1}(k))^2 \tag{3.19}$$

where

$$EN_i(k) = \frac{X_i(k)}{\sum_{l=1}^{W_{f_l}} X_i(l)} \tag{3.20}$$

Equation 3.20 defines the kth normalised DFT coefficient of the ith frame.

3.2.5.5 Spectral Rolloff

Spectral rolloff is the Cth percentile of the magnitude distribution of the spectrum. If the mth DFT coefficient corresponds to the spectral rolloff of the ith frame, then it should stand that:

$$\sum_{k=1}^{m} X_i(k) = C \sum_{k=1}^{W_{f_L}} X_i(k) \tag{3.21}$$

where C is the adopted percentile and is defined by the user (usually around 90%).

According to Giannakopoulos and Pikrakis [20] the spectral rolloff is usually normalized by dividing it with W_{f_L} so that it takes values between 0 and 1, with 1 being the maximum frequency of the signal. It is used as a descriptor of the spectral shape of the audio signal, and can be used for classification, e.g. [33], and clustering tasks, e.g. [34], [35].

3.2.5.6 Mel-Frequency Cepstrum Coefficients

The mel-frequency cepstrum coefficients (MFCCs) has been a very popular feature vector in speech recognition [36] and music classification tasks [37], [38]. Generating these features relies on the notion of *cepstrum*, as introduced by Bogert [39], which is the result of computing the inverse Fourier transform (IFT) of the logarithm of the estimated signal spectrum. The extraction of these co-efficients is depicted in Figure 3.3

More specifically, MFCCs are based on the mel-scale. The mel is a transformation of a unit of pitch. To convert a frequency f in hertz into its equivalent in mel, the following formula is used:

$$Pitch(mel) = 1127.0148log(1 + freqf(Hz)700) \tag{3.22}$$

The mel-scale is a scale of pitches judged by listeners to be equal in distance from one another. As is described in [1], the reference point between this scale and normal frequency measurement is defined by equating a 1000 Hz tone, 40 dB above the listener's threshold, with a pitch of 1000 mels. Below about 500 Hz the mel and hertz scales coincide, while above that listeners perceive as equal ever-growing

pitch increments. In other words, the mel-scale is a way to formulate the observed behaviour of the human ear to better distinguish pitch in lower frequencies.

In order to extract MFCCs from a frame, we first compute the DFT and apply on it a series of L triangular filters that are based on the mel-scale. If $\tilde{O}_k, k = 1, \ldots, L$ is the power at the output of the kth filter, then the resulting MFCCs are given by the equation 3.23

$$c_m = \sum_{k=1}^{L} (log\tilde{O}_k) \cos\Big[m\Big(k - frac12\Big)\frac{\pi}{L}\Big], \quad m = 1, \ldots, L \tag{3.23}$$

We can therefore interpret MFCCs as the discrete cosine transform coefficients of the mel-scaled log-power spectrum.

3.2.5.7 Chromagram

A common variation on the spectrogram is the Chroma vector, which is a discretized 12-element representation of the spectral energy. We compute the chroma vector by grouping the DFT coefficients of a short-term window into 12 bins, each one representing one of the equal-tempered pitch classes of Western-type music, a.k.a. semitones, independently of the octave position.

Each bin produces the mean of log-magnitudes of the respective DFT coefficients, as shown in Equation 3.24.

$$v_k = \sum_{n \in S_k} \frac{X_i(n)}{N_k}, \qquad k \in 0 \ldots, 11 \tag{3.24}$$

where S_k is a subset of the frequencies corresponding to teh DFT coefficients and N_k is the cardinality of S_k. The chroma vectors of each short-term frame are grouped in a matrix V, which acts as a matrix representation of chroma vectors commonly known as the chromagram. An example is provided in Figure 3.4.

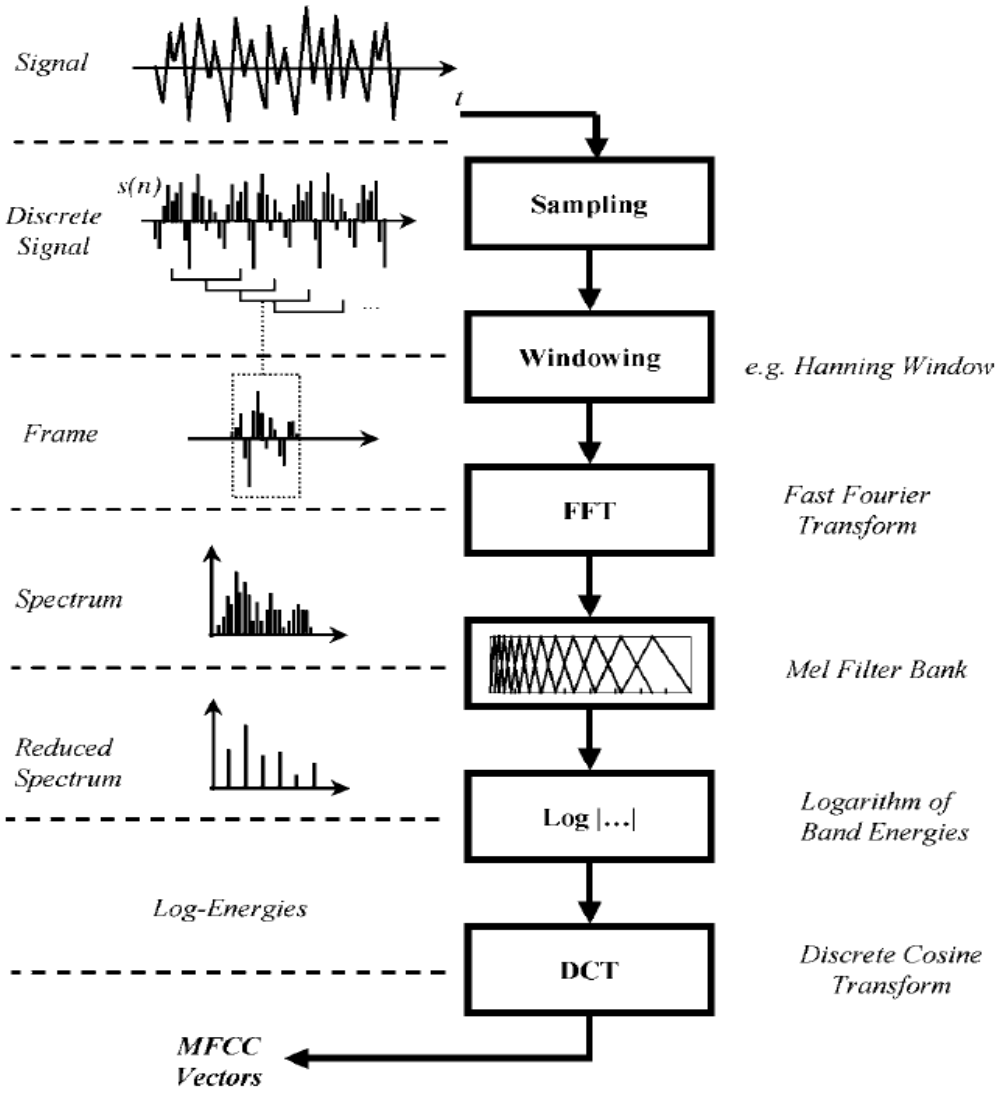


Figure 3.3: The extraction process of the MFCC features, as described by Kim et al. [1]

3.3 Extracting Audio Features

Based on the aspects described in Chapter 3, we have chosen to utilize the modules provided in the pyAudioAnalysis[2] library in Python. We have created the class `AudioFeatureExtractor` to extract a selection of features. We decided to follow the recommendations of [40] a 2 second mid-term window with a 50% overlap, combined with a short term window of 0.05 seconds and 50% overlap.

After calculating the mid-term features, we store long-term averaging statistics

[2] `https://github.com/tyiannak/pyAudioAnalysis`

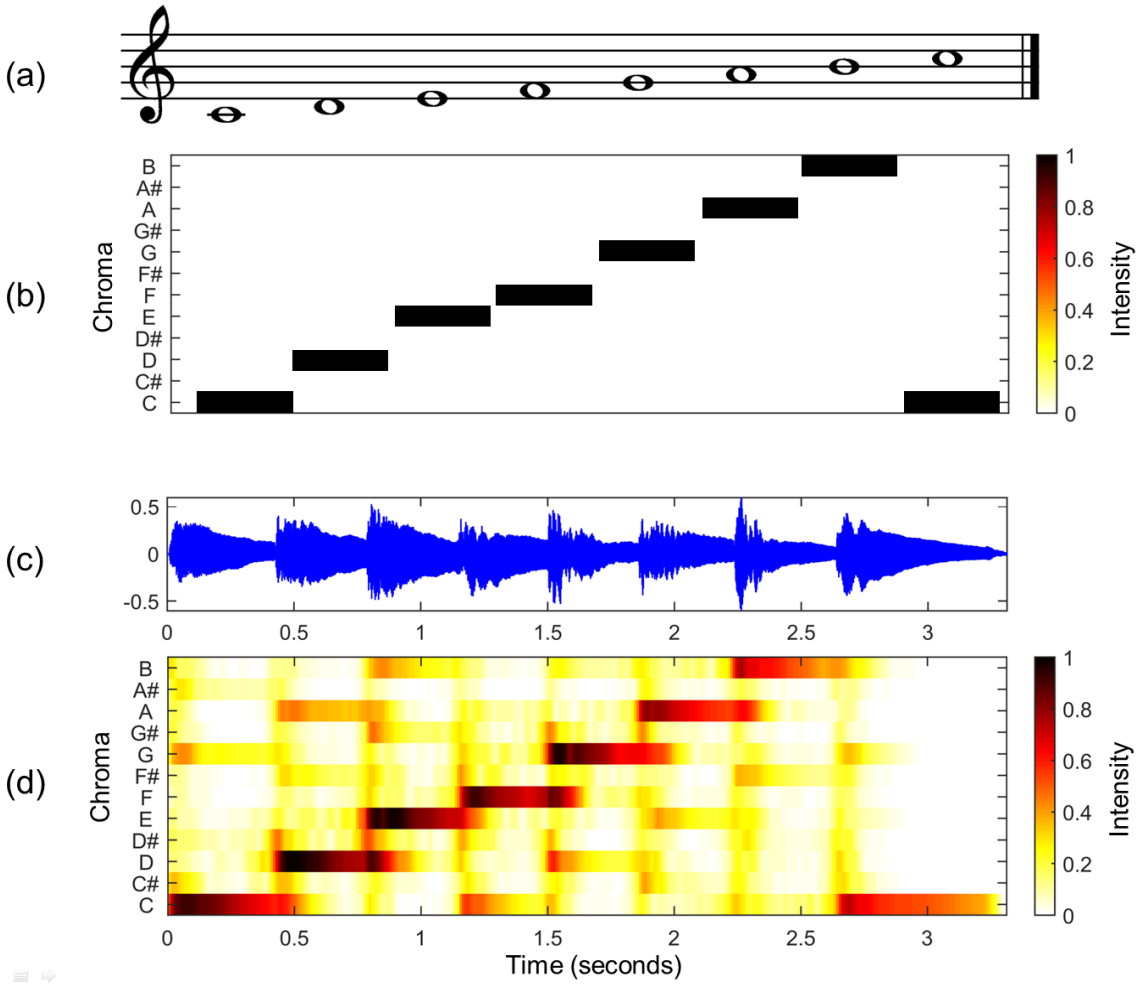


Figure 3.4: (a) Musical score of a C-major scale. (b) Chromagram obtained from the score. (c) Audio recording of the C-major scale played on a piano. (d) Chromagram obtained from the audio recording. Reproduced from Meinard.mueller, CC BY-SA 3.0 "https://creativecommons.org/licenses/by-sa/3.0", via Wikimedia Commons

for each feature, which include:

- *Mean*
- *Standard Deviation*
- *Average Delta between frames*

3.3.1 Beat Detection

In addition to what has been described in detail in the previous sections, we also include the two features that `pyAudioAnalysis` calculates about beat detection. The task of determining the rate of musical beats in time is a rather important task,

especially for the case of music information retrieval applications. As is described in [41] the implementation of this library is rather straightforward, as it adopts a local maxima detection procedure, applied on a set of short-term feature sequences. An aggregated histogram of the time distances between successive local maxima is also computed and its maximum element corresponds to the most dominant time distance between successive beats. Finally, this detected value is used to compute the BPM rate. Apart from the BPM value itself, the ratio of the maximum histogram value by the total sum of histogram values is used as a feature, corresponding to the overall "dominance" of the detected beat rate.

3.3.2 Feature Overview

The features are summarised in Table 3.1 and given in detail in Table A.1 in Appendix A.

Table 3.1: Audio Feature Overview

Feature Name	**Feature Description**
Zero Crossing Rate	The rate of sign-changes of the signal during the duration of a particular frame (for more, see section 3.2.4.3)
Energy	The sum of squares of the signal values, normalized by the respective frame length (for more, see section 3.2.4.2)
Entropy of Energy	The entropy of sub-frames' normalized energies. It can be interpreted as a measure of abrupt changes (for more, see section 3.2.4.4)
Spectral Centroid	The center of gravity of the spectrum (for more, see section 3.2.5.2)
Spectral Spread	The second central moment of the spectrum (for more, see section 3.2.5.2)
Spectral Entropy	Entropy of the normalized spectral energies for a set of sub-frames (for more, see section 3.2.5.3)
Spectral Flux	The squared difference between the normalized magnitudes of the spectra of the two successive frames (for more, see section 3.2.5.4)
Spectral Rolloff	The frequency below which 90% of the magnitude distribution of the spectrum is concentrated (for more, see section 3.2.5.5)
MFCCs	Mel Frequency Cepstral Coefficients form a cepstral representation where the frequency bands are not linear but distributed according to the mel-scale (for more, see section 3.2.5.6)
Chroma Vector	A 12-element representation of the spectral energy where the bins represent the 12 equal-tempered pitch classes of western-type music (for more, see section 3.2.5.7)
Chroma Deviation	The standard deviation of the 12 chroma coefficients (for more, see section 3.2.5.7)
Beat Detection	A local maxima detection procedure, applied on a set of short-term feature sequences. An aggregated histogram of the time distances between successive local maxima is also computed and its maximum element corresponds to the most dominant time distance between successive beats.

Chapter 4

Symbolic Representations of Music and Feature Extraction Techniques

We will now shift our focus to symbolic representations of music. These types of representations are not concerned with the audio as a signal, but instead are focused on higher-level information concerning musical concepts (such as notes, chords etc.) which we will briefly describe in the following sections. Our overview of the symbolic representations draws mainly from the work of Briot et al. [25].

4.1 Representation Strategies

4.1.1 Fundamental Symbolic Aspects

4.1.1.1 Notes

A note is a symbolic unit of musical notation, which is defined by the following characteristics:

- **Pitch**, as specified by
  - *frequency*, in Hz
  - *vertical position (height)* on a score, or

  - *pitch notation*, which combines a musical note name, such as C, C# etc. (what we previously referred to as a pitch class in paragraph 3.2.5.7) and a number, usually notated in subscript, identifying the octave, normally belonging to the [-1, 9] discrete interval.

  An example is A_4 which corresponds to A440, which has a frequency of 440 Hz and serves as a general pitch tuning standard.

- **Duration**, as specified by

  - *absolute* value, in milliseconds (ms), or
  - *relative* value, notated as a division or a multiple of a reference note duration. The reference point is usually the whole note 𝅝, and examples of its divisions are the quarter note 𝅘𝅥and the eighth note 𝅘𝅥𝅮.

- **Dynamics**, as specified by

  - *absolute* and *quantitative* value in decibels (dB), or
  - *qualitative* value, an annotation on a score about how to perform the note, belonging to the discrete set ***pppp***, ***ppp***, ***pp***, ***p***, ***mf***, ***f***, ***fp***, ***sf***, ***ff***, ***fff***, ***ffff***, ***sfz***, ***sfzp***, from pianissimo to fortissimo.

4.1.1.2 Rests

Rests are representations of intervals of silence in a musical score. They behave much like notes, only they do not have pitch or dynamics components. The duration of a rest can be specified by

- *absolute* value, in milliseconds (ms), or
- *relative* value, notated as a division or a multiple of a reference note duration. The reference point is usually the whole rest 𝄻, which corresponds to the whole note 𝅝, and examples of its divisions are the quarter rest 𝄽 and the eighth rest 𝄾.

4.1.1.3 Intervals

The interval between two notes is their relative distance in terms of pitch, as quantized in semitones (what we defined as pitch classes in 3.2.5.7. For example, the major third interval means that two notes are four semitones apart, the minor third means that two notes are three semitones apart and the perfect fifth means seven semitones apart. Intervals are the building blocks to create chords, as will be explained in Paragraph 4.1.1.4.

Intervals can also be used as a means of representation. We can consider any melody as a time-series of deviations from a starting note, as measured by intervals between successive notes, in semitones. For example the melody C4 G4 B♭4 would be C4 +7 +3. This technique was introduced by Todd [42], and its main advantage is that it is not bound by pitch range, and it's independent of a given tonality. On the other hands, as the author also points out, this second aspect could also prove to be a drawback, as, in case the original tonality is somehow erroneously set, the error would follow for the whole melodic line. Furthermore, this technique can only apply when building monophonic melodies. Due to these limitations, this representation is not used commonly in deep learning-based music generation architectures.

4.1.1.4 Chords

Chords are sets of at least three notes. While, conceptually, they are built by combining intervals (e.g. the major chord is a combination of a major triad and a perfect fifth), in terms of generative systems there are two possible representations:

- *implicit* and *extensional*, enumerating the exact notes composing it; or
- *explicit* and *intensional*, by using a chord symbol that combines the pitch class of the root note and the type of the chord (e.g. major, minor, diminished etc.)

The extensional approach is more common in deep learning architectures, as it disambiguates octave positioning and relative voicing of each note in the chord - the notes of a chord are not always placed in the same way, but can also be inverted

for more harmonious results. Nevertheless, some noteable architectures, such as the MidiNet [43], use the intensional approach.

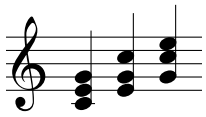


Figure 4.1: Three alternative positions (voicing inversions) for the C major chord

4.1.1.5 Rhythm

While rhythm is an indispensable aspect of music and its execution, it is quite hard to define in pragmatic terms. For a conversation on the various definitions and interpretations of rhythm, one can study the work of Fraisse [44]. This discussion is beyond the scope of this thesis, and indeed is often overlooked by deep learning music generation architectures, according to Briot et al. [25].

The basic unit of rhythm, or musical pulsation, is the *beat*. Groups of beats are called *measures* and are separated by bars. The number of beats per measure and the duration between successive beats constitute the rhythmic signature of the measure, a.k.a. *time signature* or *meter*. Usually it is expressed as the fraction of the number of beats within the measure to the beat duration, expressed as a division of the duration of a whole note. Some frequent meters are $\frac{2}{4}$, $\frac{3}{4}$ and $\frac{4}{4}$.

4.1.2 Format

As we are not dealing with signals in this type of representation, it is important to state the various different formats with which those symbolic aspects will be expressed in, in order to be parsed by a computer.

4.1.2.1 MIDI

MIDI stands for Musical Instrument Digital Interface, and it is a technical standard that describes a protocol, a digital interface and connectors for interoperability between various electronic musical instruments. Using the MIDI protocol, up to sixteen

channels of information can be carried through a single link, between multiple instruments and computers. The MIDI messages contain both real-time performance data and special control data. The protocol also defines a particular file type, the Standard MIDI File (SMF), in which MIDI data can be stored and retrieved by other systems. The protocol was first introduced in 1981 and standardised in 1983 by the MIDI Manufacturers Association who are maintaining and expanding upon it to this date. [45]

MIDI messages are comprised of 8-bit "words" that are transmitted serially at a rate of 31.25 kbit/s. The first bit of each "word" identifies whether it is a status byte or a data byte, and is followed by seven bits of information. The two most important messages for our concern (i.e. the expression and storing of music) are the following:

- *Note on* which indicates that a note is played. This contains
  - a channel number, indicating the instrument or track, specified by an integer within the set {0,1, ..., 15}
  - a MIDI note number, which indicates the pitch of the note, specified by an integer within the set {0,1, ..., 127}
  - a velocity, which indicates the dynamics of the note (as defined in paragraph 4.1.1.1, specified by an integer within the set {0, 1, ..., 127}.

  Therefore, the message "Note on, 0, 60, 50" means "On channel 1 start playing a middle C with velocity 50"

- *Note off* which indicates that a note ends. In this context, velocity means how fast the note is released. For example, the message "Note off, 0, 60, 20" means "On channel 1 start playing a middle C with velocity 20".

The note events are embedded into track chunks, which are data structures that contain a delta-time value specifying the timing information and the event itself. The delta-time value is usually a relative metrical time, which is the number of ticks from the beginning. The number of ticks per quarter note are defined in a reference in the file header.

We give an example of a midi file, converted to readable ASCII and to standard musical score, in Figures 4.2 and 4.3. The division is 480 ticks per quarter note. It

```
0, 0, Header, 1, 13, 480
1, 0, Start_track
1, 0, System_exclusive, 05, 7E, 7F, 09, 01, F7
1, 0, Text_t, "By Daniel Fogelberg\015\012Generated by NoteWorthy Composer"
1, 0, Title_t, "Same Old Lang Syne  midi by rada@revealed.net"
1, 0, Copyright_t, "All Rights Reserved"
1, 0, SMPTE_offset, 0, 0, 0, 0, 0
1, 0, Tempo, 500000
1, 0, Time_signature, 4, 2, 24, 8
1, 0, Key_signature, -1, "major"
2, 1920, Note_on_c, 9, 46, 60
2, 2320, Note_off_c, 9, 46, 64
2, 2400, Note_on_c, 9, 53, 60
2, 2880, Note_on_c, 9, 42, 10
2, 3280, Note_off_c, 9, 42, 64
2, 3840, Note_on_c, 9, 42, 10
2, 4200, Note_off_c, 9, 53, 64
2, 4240, Note_off_c, 9, 42, 64
2, 4800, Note_on_c, 9, 42, 10
2, 5200, Note_off_c, 9, 42, 64
2, 5760, Note_on_c, 9, 42, 10
2, 6160, Note_off_c, 9, 42, 64
2, 6720, Note_on_c, 9, 42, 10
2, 7120, Note_off_c, 9, 42, 64
2, 7680, Note_on_c, 9, 42, 10
2, 8080, Note_off_c, 9, 42, 64
2, 8640, Note_on_c, 9, 42, 10
2, 9040, Note_off_c, 9, 42, 64
2, 9600, Note_on_c, 9, 42, 10
2, 10000, Note_off_c, 9, 42, 64
2, 10560, Note_on_c, 9, 42, 10
```

Figure 4.2: Excerpt from beginning of the song “Same Old Lang Syne” in MIDI format

has been argued in [46] that direct encoding of MIDI messages does not efficiently preserve the notion of multiple notes being played at once through the use of multiple tracks. Due to the nature of their experiment, where they concatenated tracks end-to-end, they conclude that it would be harder for such a model to learn vertical relationships across different tracks. They opted for a piano roll format instead, which comes with its own limitations.

4.1.2.2 Piano Roll

This representation is inspired by the automated pianos of the 19th century, a continuous roll of paper with holes punched into it. Each hole represents a piece of note control information to trigger a specific note on the piano. The length of the hole corresponds to the duration of the note, while its location in the other dimension corresponds to the pitch. Its digital counterpart has pretty much the same logic, and an example is shown in Figure 4.4. The y-axis is the pitch, and the

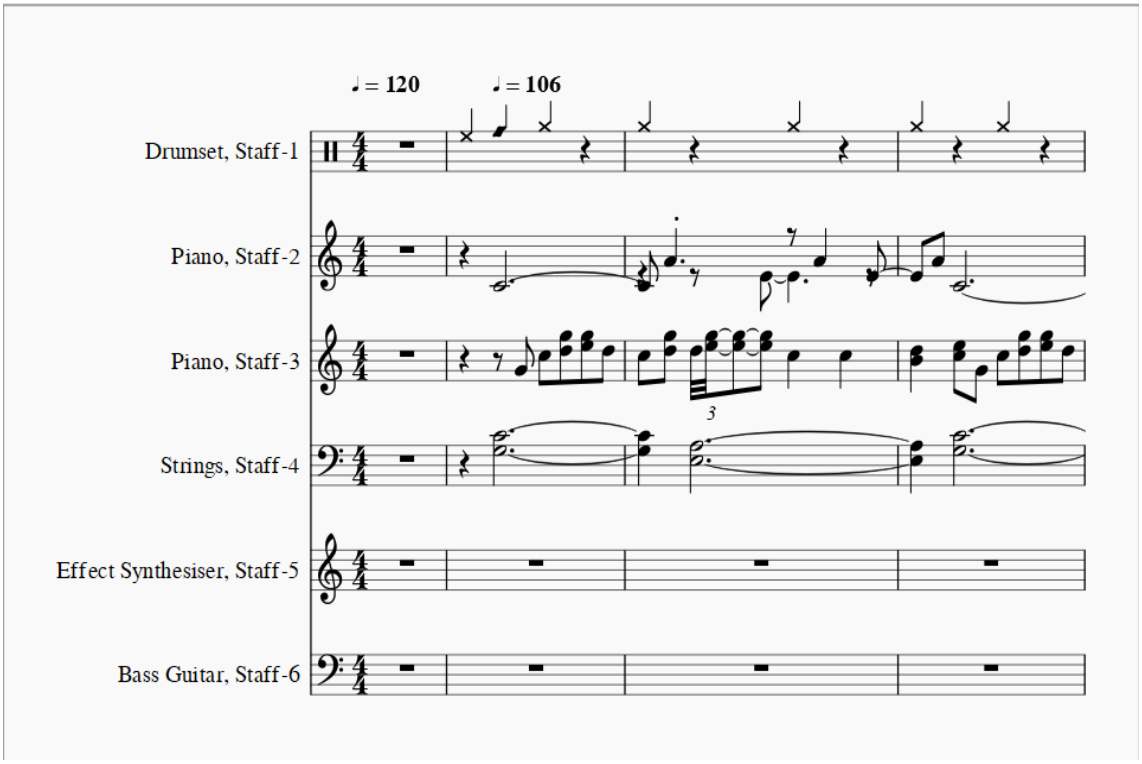


Figure 4.3: Excerpt from beginning of the song "Same Old Lang Syne" in musical score

x-axis is the time.

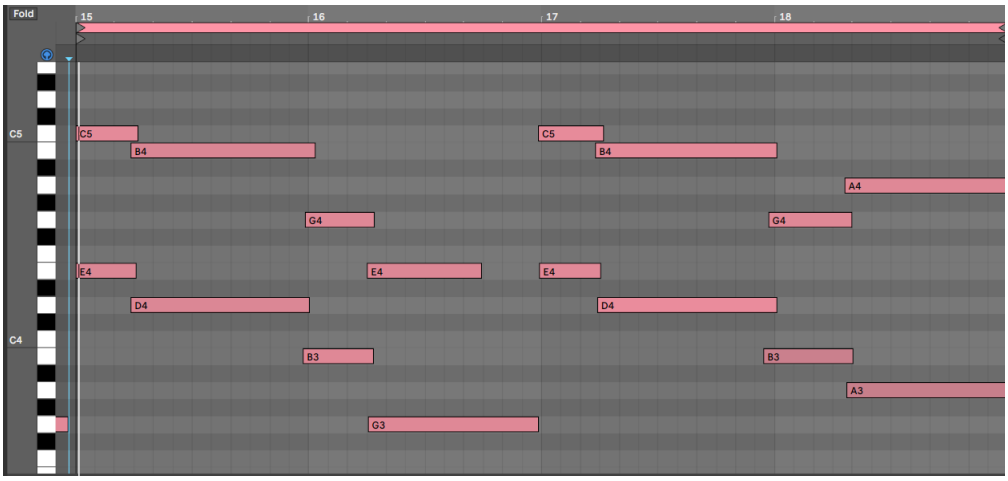


Figure 4.4: A piano-roll representation, as exported from a Digital Audio Workstation

The piano roll is quite common in Deep Learning architectures, as it makes it easier to encode polyphony across multiple instruments [25]. Nevertheless, it has a major limitation compared to MIDI, as there is no note-off information. This makes it impossible to distinguish between a long note and many small notes played repeatedly.

There are multiple solutions to this conundrum. The most straight-forward is to use a special hold symbol "_" in place of a note to specify when a previous note is held, which has been used in the DeepBach system [47]. Another is to divide the size of the time step, which is the unit of time granularity in a deep learning system - and typically is defined as the shortest note duration in the data set - by two, and mark a note ending with a special tag. This approach has been followed in [48]. Alternatively, the new note beginning can be marked instead of the ending, as in [42]. Finally it is possible to introduce a hold/replay representation as a dual representation of the sequence of notes, a method used in [49] where a replay matrix is introduced.

It seems that the approach followed by Hadjeres et al. [47] has a few advantages over the other solutions, as it is simple and there is no need to tamper with time granularity and complex notations. The authors also state that this representation technique is the principal reason for their good results. Nevertheless there is a limitation, as this can only be applied to monophonic melodies, which implies that the representation of polyphony should be split to multiple tracks in order to accommodate this.

4.1.2.3 Melody as Text

A common example of melody as a text is the ABC notation, which was designed primarily for folk and traditional tunes of Western European origin (such as English, Irish and Scottish) which can be written on one stave in standard classical notation [50]. The main idea is to use tokens instead of notes, applying the following conventions:

- A letter is used for each pitch class of a note (e.g. `A` for La)
- The pitch is encoded with the following convention: `A` corresponds to A_4 and `a` to an A one octave up. Lower octaves are reached by using commas and higher octaves are written using apostrophes; each extra comma/apostrophe lowers/raises the note by an octave.
- The duration of a note is encoded by simply putting a multiplier or a divider

after the letter. These should be used in conjunction with the default note length, marked at the top of the file. If no multiplier or divider is provided, than the default length should be used. Thus if the unit note length is 1/8, `A` is an eighth note, `A2` a quarter note, `A/2` a sixteenth note etc.

- measures are separated by "|"

An example of a song encoded in ABC notation is provided in Figure 4.5, while it's score can be seen in Figure 4.6

```
X:1
T:Speed the Plough
M:4/4
C:Trad.
K:G
|:GABc dedB|dedB dedB|c2ec B2dB|c2A2 A2BA|
  GABc dedB|dedB dedB|c2ec B2dB|A2F2 G4:|
|:g2gf gdBd|g2f2 e2d2|c2ec B2dB|c2A2 A2df|
  g2gf g2Bd|g2f2 e2d2|c2ec B2dB|A2F2 G4:|
```

Figure 4.5: The traditional tune Speed the Plough in ABC Notation

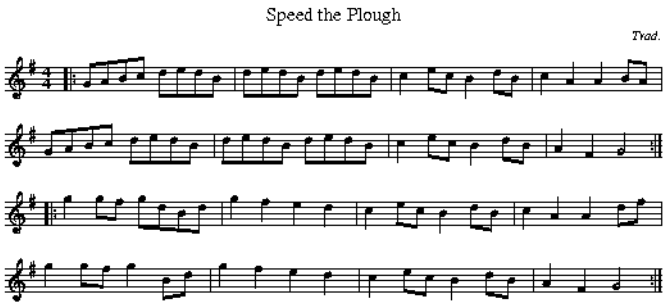


Figure 4.6: The traditional tune Speed the Plough as standard score

The main drawback of this system is that it is very hard to encode polyphonic melodies. According to [50] multi-voice music is under active review, and is intended that the syntax will be finalised in abc 2.2.

4.1.2.4 Chords as Text

While the usual way to represent chords extensively is through simultaneous notes in a (vertical) vector, an alternative, named Chord2Vec, has been proposed by Madjiheurem et al. [51] which inverts the representation to a horizontal sequence of

constituent notes. The chords are then separated by a special symbol, similar to sentence markers in a natural language processing context. This representation has been used to develop a new architecture named RNN Encoder-Decoder.

4.1.2.5 Markup Language

Another type of text-based representation for music is based on markup languages like XML. The open standard MusicXML [52] is a markup language designed to support interchange between musical notation, performance, analysis, and retrieval applications, and is intended for common western musical notation from the seventeenth century onward, including both classical and popular music. This type of representation is mostly aiming towards the interoperability of musical software, and it is therefore quite rich and verbose.

According to Briot et al. [25] this verbosity makes it inappropriate for machine learning tasks as it would create too much overhead and bias. Nevertheless, according to Cuthbert et al. [53], the biggest obstacle to moving beyond MIDI has been the lack of feature extraction software that can read these formats and use their added information. They, therefore, have created a feature extraction toolkit for symbolic music representations named *music21* in [54]. Indeed Giraldo and Ramirez [55] have proposed an algorithm that generates expressive jazz performances from inexpressive music scores, using scores in MusicXML format.

4.1.2.6 Lead Sheet

The lead sheet is a very common format of representation for popular music (jazz, pop, blues etc.). A typical lead sheet is usually one or two pages long and includes the score of the song's melody and the corresponding chord progression in an intensional (or explicit) notation. Additionally, some further metadata, such as lyrics, performer, composer, style and tempo are usually present. Variations of the lead sheet have been utilised in some deep architectures, such as the MidiNet [43] and the work on Blues generation by Eck and Schmidhuber [48], and a notable collection of lead sheets for jazz music has been assembled by the Flow Machines project [56].

4.1.3 Temporal Scope and Granularity

In order to transcribe music in any symbolic representation, it is very important to make some considerations on the aspect of time. The most fundamental one is the temporal scope, i.e. the way the data will be interpreted by the model with respect to time:

- *Global*, wherein the temporal scope is the whole musical piece. The model processes the input and produces the output in a single step.

- *Time Step*, wherein the temporal scope is a local time slice of the musical piece, corresponding to a specific temporal moment. The granularity of the processing by the model is a time step and generation is iterative, as this is the most common choice for recurrent neural networks. The time step is usually set to the shortest note duration, but it may be larger.

- *Note Step*, wherein there is no fixed time step, and the granularity of the model is the note. This approach was proposed by Mozer [57] and was also later used by Walder [58]

It is worth noting that in the case of a global temporal scope, the musical content would have a fixed length, while in the other two choices the output has an arbitrary length, because generation is iterative.

When either a global or time step temporal scope is used, the granularity of the time step must be defined. Usually, the time step is set to a relative duration, which typically is the smallest duration of a note in the corpus. This ensures that all notes will be represented at their proper duration, with a whole number of time steps. On the other hand, this also increases the number of time steps that need to be processed, regardless of the duration of actual notes. Another strategy is setting the time step to a fixed absolute duration (e.g. 15 milliseconds). This allows for more expressiveness when capturing human performances [59]. As we identify in section 4.2.1, the feature extraction library that we used has some difficulty parsing the time step granularity.

4.1.4 Encoding Strategies

Having described in detail the formatting issues of symbolic representations in section 4.1.2, it is important to also describe the possible encoding strategies and challenges when mapping the representation into a set of inputs.

Possible types of a variable in this context include continuous (e.g. the frequency pitch in Hertz), discrete (e.g. the pitch of a note as a MIDI note number) and Boolean variables (e.g. a note end indication). The most straightforward way to encode any of these, is to directly encode them as a scalar in the real number domain. This strategy is called *value encoding*.

The above strategy can't work in the case of categorical variables (e.g. the instrument that is used in the track). In that case, we usually encode these variables as vectors, with a cardinality equal to the set of possible values. The value of the corresponding element in the vector is set to 1, and all other values to 0. This strategy is called *one-hot encoding*. This strategy is often used to encode discrete integer variables, such as MIDI note numbers.

A challenge of one-hot encoding is handling polyphony. Ideally, one would prefer that a line in the input would correspond to a single time step. Nevertheless, if one-hot-encoding was employed, it would only be possible to encode one note at a time. In the case of polyphony, one would have to consider:

- *many-hot encoding* where all elements of the vector corresponding to the notes of the time step are set to 1.
- *multi-one-hot encoding* where different tracks are considered and a one-hot encoding is used for each different track
- *multi-many-hot encoding* which is a multi-voice representation with simultaneous notes for at least one or all of the voices.

In some contexts, it might even be preferable to use binning techniques to transform a continuous variable into the discrete domain. Such techniques involve dividing the domain in smaller intervals (bins) and replacing each bin (and its values) by

a value representative, which is often the central value. The same technique can be used to reduce the cardinality of the discrete domain as well.

According to Briot et al. [25] one-hot encoding is the most common strategy for symbolic representation, and value encoding is used very rarely. The advantage of value encoding is its compact representation at the cost of sensibility because of numerical operations. The advantage of one-hot encoding is its robustness (due to the fact that it's in the discrete domain), at the cost of high cardinality and therefore a potentially higher number of inputs. It is also common that one-hot encoding is also used in the output of generative architectures, as it makes it easier to use a softmax activation function, corresponding to a classification task between the possible values of the categorical value.

4.2 Symbolic Feature Extraction

Having described in detail the symbolic representation and encoding strategies in the previous sections, we now shift our focus toward the extraction of features. While the encoding techniques are mostly useful for generative tasks, in the context of classification it is most common to extract even higher-level features. We turn to Cuthbert et al. [54] and the `music21` package[1]

4.2.1 Caveats on Symbolic Feature Extraction with `music21`

In our feature extraction process, we use the `music21` Python package to parse and translate MIDI files to MusicXML. This parsing process often results in errors, mostly in regards to temporal synchronisation and part instrumentation. It is often the case that the MIDI files that exist freely in the Web contain corrupt sections, or do not have the proper numbering system for instrument selection. Given the fact that we are using `music21` in conjunction with `MuseScore 3`[2] for MusicXML compiling, we realised that in order to resolve this error, we had to make considerable changes to the way that the library is handling multi-track midi files. We

[1] http://web.mit.edu/music21/
[2] https://musescore.org/en

communicated with the people from Cuthbert Lab who maintain this library, but the refactor could not be completed at the time of writing this.

In the following sections, the feature extraction process is described and designed in order to work around this issue. We have decided to do our best to minimize these shortcomings from our end, without altering any code from the library itself, and let the models decide which features carried meaningful information.

Another issue with the library is that it is not optimised for speed, making the extraction process take a lot of time. We have calculated that it took around 6 minutes per MIDI file in our machine, which made the feature extraction process quite expensive in terms of computational time.

4.2.2 Extracting Symbolic Features

While the basis for our extraction process is the `music21` library for computational musicology, we have created a wrapper around it, in order to expose all the necessary classes, functions and parameters that we needed for our own implementation. We named this class `SymbolicFeatureExtractor`.

4.2.2.1 Parsing the MIDI data

As we found problems with the implementation of MIDI parsing with `music21`, we decided to assist the process by doing some pre-processing using the `mido`[3] library. We have identified that the main library had an issue with non-tuned percussion instruments, as they were interpreted as regular tuned instruments, and their messages counted as pitches, essentially destroying all the melodic interval features. We therefore removed them from each midi file, by parsing the track names with `mido`.

[3] `https://mido.readthedocs.io/en/latest/`

4.2.2.2 Combining Feature Extraction classes

The `music21` library has two main groups of feature extractors. One is based on `jSymbolic`[4], a Java-based program for computational musicology. The other group is native feature extractors, developed in Python for `music21`.

The categories of features of `jSymbolic` are described in the following paragraphs, drawing from the original publications of McKay et al. [60]. The native group of feature extractors contains a small collection of handcrafted features that mostly fall into the same categories as `jSymbolic`. Some of them, like *Quality Feature* improve upon `jSymbolic`'s implementation of the same concept. In these cases, we accept the native class by default, and skip the `jSymbolic` version of the feature.

For an overview of all the feature categories, please consult Table 4.1. The full list of features is given in Table A.2 in Appendix A.

Table 4.1: Symbolic Feature Overview

Feature Name	**Feature Description**
Pitch Statistics	Pitch variety and tonality of a piece (for more, see paragraph 4.2.2.2)
Melodies and Horizontal Intervals	Melodic variation of a piece (for more, see paragraph 4.2.2.2)
Chords and Vertical Intervals	Harmonic movement and chord building (for more, see paragraph 4.2.2.2)
Rhythm	Rhythmic patterns, measures and rhythmic changes (for more, see paragraph 4.2.2.2)
Instrumentation	Choice and prevalence of instruments, not included in our implementation (for more, see paragraph 4.2.2.2)
Texture	Voice independence (for more, see paragraph 4.2.2.2)
Miscellaneous Features	Features based on corpus metadata or otherwise not falling within the other categories (for more, see paragraph 4.2.2.2)

[4] `http://jmir.sourceforge.net/`

Pitch Statistics These features describe the amount of co-occurrence between various pitches, in terms of both absolute pitches and pitch classes. Through these, they also flag the amount of tonality in a piece (how much it follows rules of tonal music), the range of pitches in the piece and how much variety in pitch there is.

Melodies and Horizontal Intervals These features examine the kinds of melodic intervals that are present, as well as the amount of melodic variation of the piece. Furthermore the melodic contours are measured, as well as common phrases.

Chords and Vertical Intervals These features concentrate on vertical intervals, the types of chords they represent, as well as the presence and velocity of harmonic movement.

Rhythm These features are calculated based on the time intervals between note attacks and the duration of individual notes, in order to identify rhythmic patterns and variations. It contains features about rhythm measure, both on a musical piece level and on a bar-by-bar level, for more complex pieces.

Instrumentation These features are, in theory, relevant to the types of instruments that are present, and the emphasis that is given to each instrument. In our implementation we found these features to be weaker, as they are dependent on accurately parsing track meta-data from the MIDI files, which we have already identified as problematic. We have therefore chosen to exclude them from the feature extraction process

Texture These features compute the level of polyphony within the piece. They also measure the amount of interaction between those voices, identifying voice independence or equality.

Miscellaneous Features The native classes also include some features based on the metadata of the musical piece, such as *Composer Popularity* and *Language Feature*, as well as a few features specific to classical music, such as *Landini Cadence*.

Chapter 5

Video representation and Feature Extraction Techniques

In this chapter we focus on the third and final modality of interest for the task at hand, which is video. We provide the necessary definitions concerning video as a sequence of images, and we describe the fundamental concepts that are necessary to extract features for our task. We start from more basic features, such as color statistics, and proceed to more complex ones, based on spectral and filtered representations of the signal, and finally we incorporate some deep features based on pre-trained models for object and face detection.

5.1 Video Processing Fundamentals

5.1.1 Video Signals and Images

A video signal is defined in [2] as a one-dimensional analog or digital signal varying over time, whose spatiotemporal contents represent a sequence of images (or frames) according to a predefined scanning convention. Scanning is a method that converts optical images into electrical signals. An analog video signal refers to a 1D electrical signal $f(t)$ obtained by sampling the original optical signal $f(x, y, t)$ in the vertical and temporal dimensions, whereas a digital video signal is also sampled along the horizontal axis of each frame, as depicted in Figure 5.1. According to [61], digital

video is obtained either by sampling an analog video signal $f(t)$ or by directly sampling the three-dimensional space–time intensity distribution that is incident on a sensor. In either case, what results is a time sequence of two-dimensional spatial intensity arrays or equivalently a three-dimensional space–time array.

Getting into details on the scanning process, and the distinctions between analog and digital signals is beyond the scope of this Thesis, but we will briefly present some fundamentals, in order to establish a common language for the features that will be explained in the following sections. For our convenience we will assume that we are dealing only with digital signals, either directly from a digital camera, or through sampling a raster scan and performing analog-to-digital conversion (ADC) (to be explained in paragraphs 5.1.1.1) and 5.1.1.3. We will also assume batch processing, instead of streams, i.e. the video signal is known in advance in its entirety.

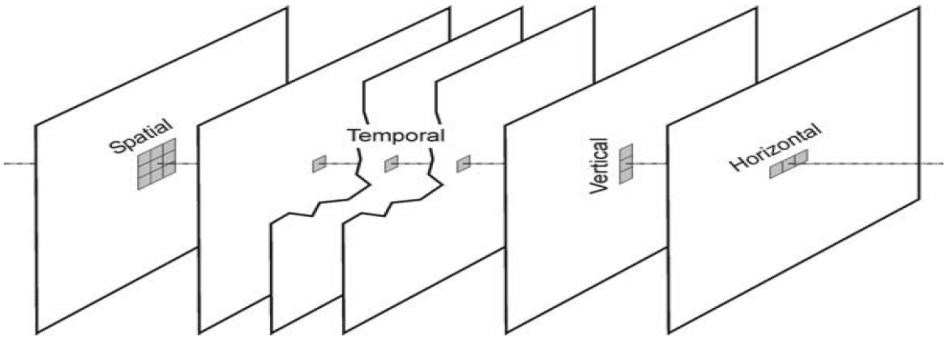


Figure 5.1: Sampling in the horizontal, vertical and temporal dimensions [2]

5.1.1.1 Scanning, Frame & Refresh Rate

Scanning is the most basic method of conversion of optical images to electrical signals, and is common within all video systems. The camera's electronic sensor is moving across the image, in a pattern known as *raster* and converts the differences in brightness to differences in voltage. A complete scan of the image is called a *frame*.

There are two main ways to perform scanning, progressive and interlaced scanning, with their main difference being the way that the sensor returns to the edge of the frame, i.e. the *retrace* of the sensing spot. The nuances between those two

methods are beyond the scope of analysis, but it suffices to say that progressive scan is a more natural way of scanning that is most common in recent video systems, whereas interlaced scanning was introduced to accommodate limitations of early television [2].

The rate at which the image is scanned is called *frame rate* and is measured in frames per second (fps). In order for a series of images to be perceived as continuous by the human eye, the frame rate needs to be higher than what is called critical flicker frequency. In order to further optimize this "illusion" for the human eye, the period during which the reproduced image is absent from the display needs to be minimized. To that end, the image is flashed at a rate that is higher than the one necessary. This rate is called *flash*, or *refresh* rate. Typically for progressive scan systems this is equal to the frame rate, whereas in interlaced scan systems it is double the frame rate.

5.1.1.2 Aspect Ratio

Aspect Ratio is the ratio of frame width to height. In digital videos, this is usually defined in terms of pixels (e.g. 640 x 480 or 1920 x 1200).

5.1.1.3 Digital Videos and Analog-to-Digital Conversion

When the video originates from a digital camera, there is no need for a conversion from analog to digital, as it is performed at the imager. If, however, the video is captured by an analog camera (such is the case in most films) an Analog-to-Digital Conversion (ADC) step is necessary. For the scope of this thesis, it suffices to say that it includes an antialiasing (low-pass) filter, a sampling step (which samples pixel values along a horizontal line), a quantizing step and an encoding step.

As explained by Marques [2] a digital video signal can be characterized by

- The frame rate ($f_{s,t}$)
- The line number ($f_{s,y}$)
- The number of samples per line $f_{s,x}$

From the above quantities, we can find

- The temporal sampling interval or frame interval $\Delta_t = \frac{1}{f_{s,t}}$
- The vertical sampling interval $\Delta_y = \frac{PH}{f_{s,y}}$, where PH is the picture height
- The horizontal sampling interval $\Delta_x = \frac{PW}{f_{s,x}}$ where PW is the picture width

Another important factor is the number of bits used to represent a pixel value, N_bl. For each color component, 8 bits are needed. Therefore, a regular RGB (Red Green Blue, see section 5.1.2) representation requires $N_b = 24$.

Based on the above, the data rate of a digital video, R, can be determined as

$$R = f_{s,t} \cdot f_{s,y} \cdot f_{s,x} \cdot N_b \tag{5.1}$$

Therefore a typical video using RGB (24 bits per pixel), with an aspect ratio of 1920x1080 (FullHD) and a frame rate of 50 fps, will result to a data rate of 2.5 GB per second. In order to decrease that, further compression techniques are applied.

5.1.2 Color

It is beyond the scope of this Thesis to touch upon the complexities and challenges of color encoding during the early days of video. There are three principal quantities that play a role in the encoding of video:

- *Intensity*: the amount of energy that flows from the light source, measured in watts.
- *Luminance*: the amount of information perceived by an observer of a light source, measured in lumen.
- *Brightness*: the subjective perception of luminous intensity.

For the purposes of data acquisition and feature extraction, it suffices to understand the two common ways of representing color (Color Models):

- *The RGB color model*, which is based on a Cartesian coordinate system, whose axes represent the three primary colors of light (Red, Green and Blue), normalised to the [0,1] range. When images are stored using the RGB model three values per sample are needed, one for each of the three colors. The combinations can be seen in Figure 5.2.

- *The HSV color model*, which is based on the human perception of color. Its foundation is the psychophysics of color, and is obtained by looking at the RGB color cube along its main diagonal. The name of the model stems from Hue-Saturation-Value, with Hue being the color tone, Saturation being the purity of the color, and Value being the intensity of light reflected from objects. If viewed as a cylinder, such as the one in Figure 5.3 hue represents the angle on the cylinder's circumference, saturation is the distance from the central axis of the cylinder, and value is the position on the vertical axis. The advantage of this way of representing color is that it is closer to the human perception.

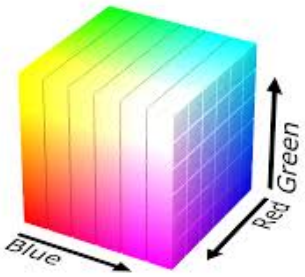


Figure 5.2: The RGB color model.
Courtesy of Wikimedia Commons

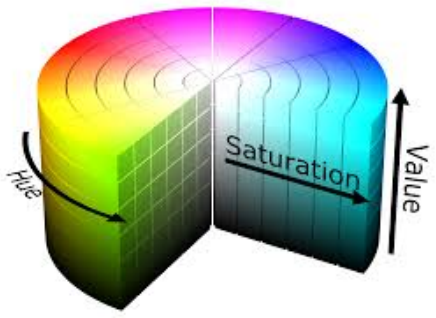


Figure 5.3: The HSV color model.
Courtesy of Wikimedia Commons

In video encoding, another way of representing is used; *Y'CrCb* needs three values per sample: one value (Y') represents *luma*, i.e. the gamma-corrected luminance of the pixel, one value (Cr) is the difference between the red component and a reference value, and the third value (Cb) is the difference between the blue component and a reference value. The advantage of *Y'CrCb* over RGB, according to [2] is that it is less sensitive to changes in color than in luminance, therefore is more suitable for sub-sampling and compression, without sacrificing much in terms of quality.

5.2 Flow Features and Shot Detection

The pattern of apparent motion of the objects in an image between two consecutive frames, is named Optical Flow. This perception of motion can either stem from the movement of the object itself within the 2D frame, or from the movement of the camera.

In order to understand the mathematical definition of optical flow [62], let us consider a pixel $I(x, y, t)$ in the first frame, which moves by distance (dx, dy) in the next frame, which is taken after dt time. If the pixel is exactly the same, and its intensity has not changed, we can claim that

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{5.2}$$

If we then take the Taylor series approximation of the right-hand side, remove the common terms and divide by dt, we get the following equation, which is called Optical Flow equation:

$$f_x u + f_y v + f_t = 0 \tag{5.3}$$

where

$$f_x = \frac{\partial f}{\partial x}; f_y = \frac{\partial f}{\partial y}; u = \frac{dx}{dt}; v = \frac{dy}{dt} \tag{5.4}$$

To extract the features, the implementation of `OpenCV` that was used, is based on the Lucas-Kanade method of solving the above equation. The method, proposed in [63], is using a 3x3 area around the point, and makes the assumption that neighbouring pixels will have the same motion. The equation, after applying a least square method, becomes:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix} \tag{5.5}$$

In Figure 5.4 we demonstrate the flow features extracted from the frame of a film in our data-set. Further to the Optical Flow features, we also utilise a threshold on transformations of the frames, to detect large scale changes between frames. These

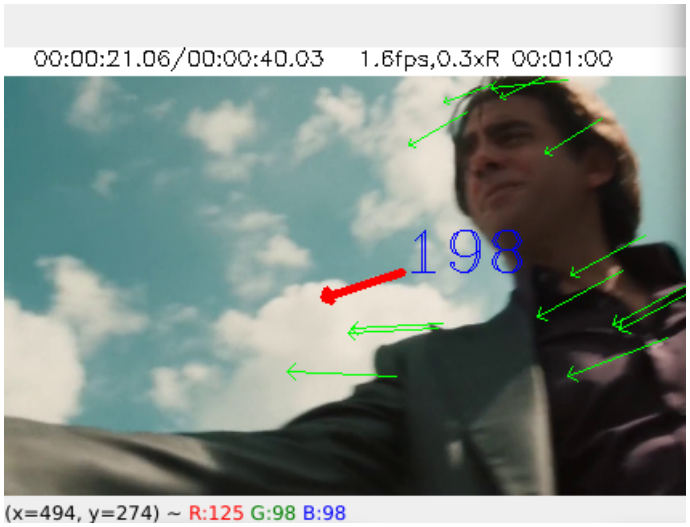


Figure 5.4: The flow features extracted from a frame, using `OpenCV`

transformations include gray-scale versions of the frames, optical flow and color value changes. These changes are interpreted as changes of the shot, and kept as *Shot Detection* features. The thresholds are applied in the order shown in Listing 5.1.

Listing 5.1: Shot Detection function

```
1  def shot_change(mean_value_of_angles,
2                  grayscale_difference,
3                  color_diff,
4                  current_shot_duration):
5      count = 0
6      if ((mean_value_of_angles > 0.08)):
7          count += 1
8      if (grayscale_difference > 0.65):
9          count += 1
10     if (color_diff[-1] > 0.02):
11         count += 1
12
13     if (count>=2) and (current_shot_duration > 1.1):
14         return True
15     else:
16         return False
```

5.3 Object Detection

Another subset of the features that have been included in our feature library, are related to the detection of objects. We briefly describe the methods that have been

applied in the following sections.

5.3.1 Face Detection

A pre-trained Machine Learning model called *Haar Cascade Classifier* is used to detect faces. It is based on a method proposed in [64] and extracts Haar features by subtracting the sum of pixels for areas of the image, as shown in Figure 5.5. The classification algorithm is based on Adaptive Boosting (*AdaBoost*), which calculates the weighted sum of many weak classifiers.

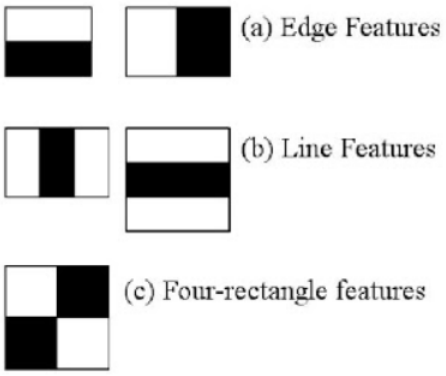


Figure 5.5: Haar Cascade Features

The "cascading" part of the model's name is owed to the fact that features are extracted in groups and the classification is applied in sequence, only if the first classifier fails.

5.3.2 Single Shot MultiBox Detector

In order to detect objects within frames, we made use of a pre-trained Deep Learning model. Based on the method described in [65] a pre-trained Convolutional Neural Network is used to detect objects of 92 classes. The implementation that we used is derived from a ResNet-50 model (instead of the VGG applied in the paper), which is trained on the COCO data-set[1] . The code for applying the model, as well as an

[1]A popular data-set for object detection, containing photos of multiple objects from multiple angles. More information can be found in the official website, at `https://cocodataset.org/`

explanation on the specifics of the model, are found in PyTorch Hub[2]. An example of the model's prediction for a particular frame containing an airplane, is shown in Figure 5.6.

Figure 5.6: Single Shot MultiBox Detector for object detection using PyTorch

5.4 Extracting Video Features

Based on the aspects described in this chapter , we have used the modules found in the `multimodal_movie_analysis`[3] library in Python, which in turn is based in openCV[4] and pyTorch[5], to extract a selection of features. The library is an extension of the work that was done for [66]. In order to use this in our own context, we have created the class `VideoFeatureExtractor` which wraps around this module to extract features. The features are summarised in Table 5.1 and given in detail in Table A.3 in Appendix A.

[2] `https://pytorch.org/hub/nvidia_deeplearningexamples_ssd/`
[3] `https://github.com/tyiannak/multimodal_movie_analysis`
[4] `https://github.com/opencv/opencv`
[5] `https://pytorch.org/`

Table 5.1: Video Feature Overview

Feature Name	**Feature Description**
Color Histograms	Histograms for both RGB and HSV color schemes(for more, see section 5.1.2)
Flow Features	Angle changes, movements of objects and shot detection (for more, see section 5.2)
Object Detection and Face Detection	Application of SSD and Haar Cascade classifiers (for more, see section 5.3)

Chapter 6

A Classification Experiment

In the previous chapters, we discussed in depth the nature of each of the three modalities that are relevant to our task. We reviewed the properties of each one, and explained our feature extraction process. Further to that, we described in detail our proposed method for a data collection pipeline, as well as our specific implementation and its results. Having collected the data, we will now put them to the test, by experimenting with building a classifiers.

In the following sections, we will first describe the data-set we worked with, our methodology for cross validation and train-test splits, and a baseline classifier. We will then experiment with a few dimensionality reduction strategies and different data pre-processing techniques. Finally, we will perform hyper-parameter tuning to the most promising settings, and experiment with a voting ensemble.

All experimentation is done using Python and `sklearn`.

6.1 Exploring the data-set

6.1.1 Dimensions

The data-set is comprised of 136 data points in total. These points are split in two classes "match" and "no-match", depending on whether they are real examples of video soundtracks that come from the process described in section 2.5 or hand-picked mismatched examples (as described in paragraph 2.5.2.5). The goal

Table 6.1: Statistics for clips per film and the percentage clips that belong to the positive class per film

	Positive Rate per Film	# of Clips per Film	# of Positives per Film
mean	0.523232	4.121212	2.060606
std	0.182333	3.846732	1.951592
median	0.500000	3.000000	2.000000

of the classifier is to discriminate the matching soundtracks from the mismatched, constructed examples.

Overall, the data-set is balanced, as there are 68 matches and 68 fake examples. In Figure 6.1 we demonstrate the distribution of clips per video, and how the positive class is allocated within each video. We notice that, while the unweighted mean in the data-set is exactly 50%, if we group the clips by originating film, the baseline is slightly increased. Nevertheless, Table 6.1 demonstrates that the median number of clips per film is 3, and that the mean rate of positive clips per film is 52%. Therefore, the method of cross-validation by leaving one film out each time seems to make sense, and our baseline threshold against pure chance (within the sample of data that is available) is 52%.

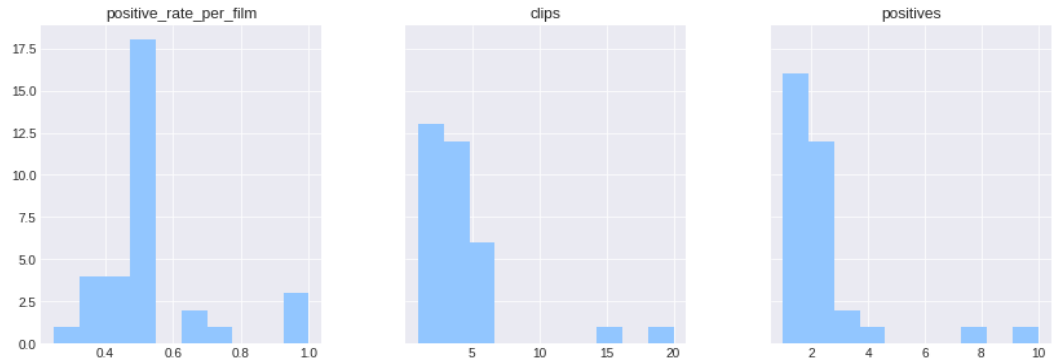


Figure 6.1: The distribution of classes and clips per film

The feature extraction process that was described in sections 3.3, 4.2.2 and 5.4 results in 455 attributes per data point. More specifically there are:

- 138 features from the audio content
- 73 features from the symbolic content (MIDI)

- 244 features from the visual content

6.1.2 Attribute Completeness and Scaling

In order to be able to apply further statistical tests, we first test the completeness of the features. It seems that there is no serious issue with data completeness, as there are no columns with more than 10% empty values, and only three features that had 5% of empty values. They all come from the symbolic modality:

- AverageTimeBetweenAttacksForEachVoiceFeature
- AverageVariabilityOfTimeBetweenAttacksForEachVoiceFeature
- InitialTimeSignatureFeature

The completeness issue is mild enough that we will only use a simple imputation technique, using the mean value of the column. We wouldn't expect to greatly affect the classification results by using more complex imputation techniques.

The maximum value in the data-set is 8105.1424, while the minimum value is -28.57613. The mean value in the data-set is 10.0. This shows us that there are big outliers and potential scaling issues in our dataset. We address this issue for the test that ensue, by applying a simple scaling method[1] which brings each feature in the [0,1] range:

$$X_{scaled} = (X - X_{min})/(X_{max} - X_{min}) \tag{6.1}$$

There are 10 columns whose values are all zero. They come from mostly from the set of object detection features, therefore we expect that to be volatile to the diversity of the visual content. As expected, there are also three columns from the symbolic domain, and these mostly concern MIDI file metadata, which is probably connected to the issues we identified with how `music21` parses these. The full list is as follows:

- delta chroma_8_mean

[1] sklearn.preprocessing.MinMaxScaler

- QuintupleMeterFeature
- ComposerPopularity
- LanguageFeature
- outdoor_freq
- sports_freq
- outdoor_mean_confidence
- sports_mean_confidence
- outdoor_mean_area_ratio
- sports_mean_area_ratio

6.1.3 Attribute Relevance

We run a Spearman correlation test, which is a non-parametric test for statistical dependence between two variable, between all the features and the class. We sort that list based on the absolute value of the correlation, and collect the 20 features which rank the highest. We collect these in Table 6.2. It is worth noting that although the visual features are much more than those from the other two modalities combined, none of them belong in the top 5% of features with the highest monotonic correlation with the target variable. While this is not a definitive measure of feature importance, this observation could perhaps lead to further improvements in the data extraction process in the future.

We also performed a mutual information test between each (scaled) feature and the target variable, using a method[2] that relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances. This test yielded 162 features with non-zero mutual information with the target. Out of these, only nine are from the visual modality, and they all come from the object detection group: accessory_freq, sports_freq, appliance_freq, outdoor_mean_confidence,

[2] sklearn.feature_selection.mutual_info_classif

Table 6.2: Features with the highest absolute value of Spearman's correlation coefficient with the target variable.

Feature Name	Absolute Spearman Correlation
chroma_7_std	0.268402
delta chroma_7_std	0.248361
RelativeStrengthOfTopPitchClassesFeature	0.247611
chroma_7_mean	0.234875
delta energy_entropy_mean	0.228958
delta chroma_std_std	0.200790
delta chroma_10_std	0.194607
VariabilityOfNoteDurationFeature	0.185802
RelativeStrengthOfTopPitchesFeature	0.184303
mfcc_3_mean	0.182805
chroma_std_mean	0.182245
delta chroma_3_mean	0.172250
chroma_9_mean	0.170820
chroma_3_mean	0.165761
MinorTriadSimultaneityPrevalence	0.163337
beat_conf	0.161266
delta chroma_9_std	0.160145
MelodicTritonesFeature	0.159452
chroma_std_std	0.154717
DiminishedTriadSimultaneityPrevalence	0.152955

kitchen_mean_confidence, furniture_mean_confidence, electronic_mean_confidence, indoor_mean_area_ratio, appliance_mean_confidency.

This is a bit worrying, as these features are very specific to the videos that were used, and we would expect them to be volatile to the types of video clips that were extracted during our collection process. If in the future a larger and more diverse data-set is collected, we would expect the lower-level visual features (such as color histograms) or flow features to also become more important.

6.2 Train-Test Split Methodology

Given the length of our data-set, we would need some form of cross validation in order to get the best possible results from this small set of data. At the same time, we want to avoid the danger of overfitting, by including clips extracted from

the same video source in train-test splits. We therefore decided to create splits that always include all clips that come from the same film in either the training set, or the test set.

Ideally we would hold out videos from a few movies, in order to create a final held-out test set. As the video clips are originated from only 33 films, there were only 33 possible folds that we could create, so holding some of these films for a final test set was not possible. Knowing that this is not the best solution, we decided to instead use the relevant method[3] in `sklearn`, create the folds and store the predicted class, the probability of the prediction and the real value. We then computed the metrics of interest using all the validation set predictions. Scaling, imputation and decomposition techniques were fitted to the training set of each fold.

An issue that we identified with this method is that whenever the held-out film has a lot of relevant clips, the results are significantly affected. We would expect this volatility to decrease as the data-set size increases.

In order to perform hyper-parameter tuning, we took the most promising settings and compared them against each other, using the Leave-One-Group-Out method on the whole data.

6.3 Models tested

We tested a few classification algorithms, from various model families. Given the length of the data-set, neural network algorithms were deemed inappropriate. The default settings of each algorithm are described below:

- **Decision Tree** (DT): maximum depth = 4, criterion = gini
- **Logistic Regression** (LR): solver = liblinear
- **k Nearest Neighbors** (kNN): number of neighbors = 3, weights = uniform
- **Support Vector Machines** (SVM): kernel = RBF, C = 1, gamma = auto
- **Random Forest** (RandF): number of trees = 100, maximum depth = 1

[3] sklearn.model_selection.LeaveOneGroupOut

- **Bagged Trees** (BTr): base estimator = Decision Tree with maximum depth = 1 and criterion = gini
- **AdaBoost** (AdaB): base estimator = Decision Tree with maximum depth = 1 and criterion = gini
- **XGBoost** (XGB): all parameters were left to their automatic default values.
- **Naïve Bayes** (NB): no hyper-parameters were tuned
- **Extra Trees** (ExTrees): no hyper-parameters were tuned
- **Gradient Boosting Ensemble** (GradBoost) no hyper-parameters were tuned

6.4 Baseline Results

Before attempting any optimisation, we test the algorithms in their default settings. Imputation and scaling are done using the simple imputation and scaling techniques described in section 6.1.2. The Naïve Bayes classifier is the winner with 57% mean average and 66% f1 measure. The results are summed up in Table 6.3 and Figure 6.2.

Table 6.3: Baseline Results

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.5	0.43	0.53	0.41	0.46	0.42	0.56	0.49	0.53	0.51	**0.57**
precision	0	0.5	0.43	0.53	0.43	0.47	0.43	0.56	0.49	0.52	0.51	**0.65**
	1	0.5	0.43	0.53	0.38	0.46	0.41	**0.56**	0.48	0.54	0.53	0.55
recall	0	0.6	0.43	0.54	0.56	0.53	0.49	0.56	0.60	0.66	**0.72**	0.32
	1	0.4	0.44	0.51	0.26	0.40	0.35	0.56	0.37	0.40	0.31	**0.82**
f1 - measure	0	0.55	0.43	0.54	0.49	0.50	0.46	0.56	0.54	0.58	**0.60**	0.43
	1	0.44	0.44	0.52	0.31	0.43	0.38	0.56	0.42	0.46	0.39	**0.66**

6.5 Experiments with scaling

In the following sections, we conduct a series of experiments around feature scaling, and compare the results to the baseline MinMax scaling technique.

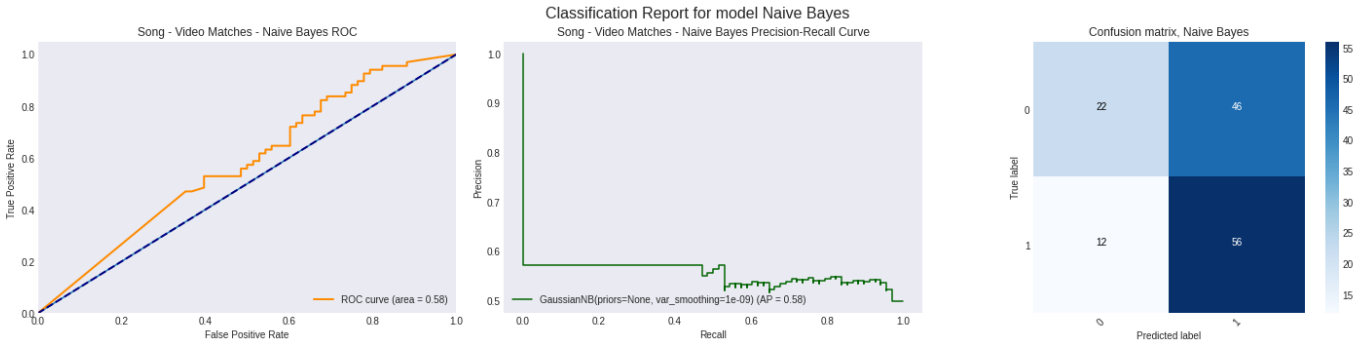


Figure 6.2: Baseline Results: ROC curve, Precision Recall Curve, Confusion Matrix

6.5.1 Standardisation

In this setting we used the Standard Scaler[4] of `Scikit`, which removes the mean and divides with the standard deviation of each feature. We notice that this deteriorates the performance of most classifiers, giving the first place to AdaBoost, with 56% accuracy and f1-measure. We summarise the results in Table 6.4 and Figure 6.3.

Table 6.4: Scaling Setting 1: Using the StandardScaler

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.5	0.46	0.49	0.47	0.46	0.42	**0.56**	0.49	0.53	0.51	0.55
precision	0	0.5	0.46	0.49	0.48	0.47	0.43	0.56	0.49	0.52	0.51	**0.62**
	1	0.5	0.46	0.49	0.46	0.46	0.41	**0.56**	0.48	0.54	0.53	0.53
recall	0	0.6	0.49	0.47	0.63	0.53	0.49	0.56	0.60	0.66	**0.72**	0.26
	1	0.4	0.44	0.51	0.31	0.40	0.35	0.56	0.37	0.40	0.31	**0.84**
f1 - measure	0	0.55	0.47	0.48	0.54	0.50	0.46	0.56	0.54	0.58	**0.60**	0.43
	1	0.44	0.45	0.50	0.37	0.43	0.38	0.56	0.42	0.46	0.39	**0.65**

6.5.2 Normalization Using the L2 norm

In this setting, we use `sklearn`'s implementation [5] of a per-sample normalisation. Each row is rescaled independently so that its L2 norm is equal to one. The results this time significantly reduce the performance of all classifiers. The results are given in Table 6.5 and Figure 6.4.

[4] sklearn.preprocessing.StandardScaler()
[5] sklearn.preprocessing.Normalizer()

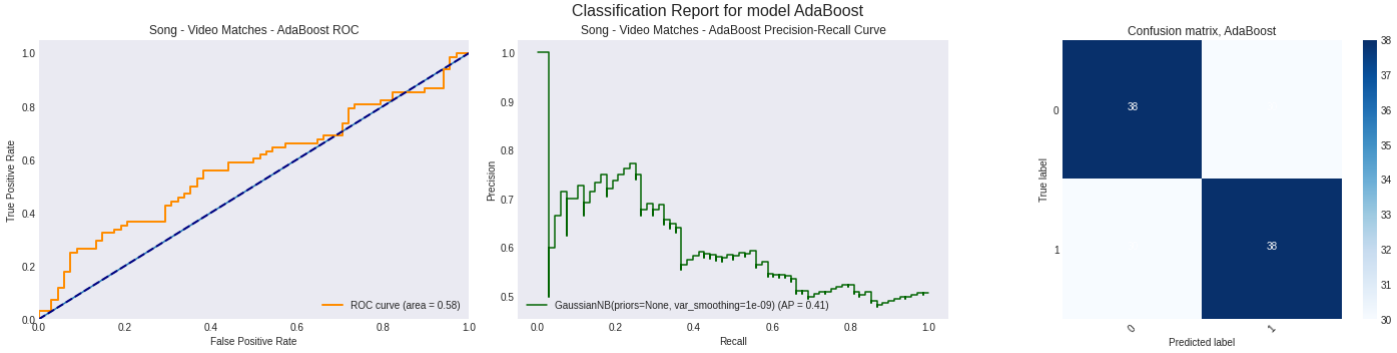


Figure 6.3: Scaling Setting 1: ROC curve, Precision Recall Curve, Confusion Matrix

Table 6.5: Scaling Setting 2: Applying Normalization

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.48	0.51	0.48	0.43	0.45	0.48	0.51	0.47	0.51	0.48	**0.54**
precision	0	0.48	0.51	0.48	0.44	0.44	0.48	0.51	0.47	0.51	0.48	**0.53**
	1	0.47	0.53	0.48	0.40	0.45	0.48	0.51	0.47	0.52	0.48	**0.64**
recall	0	0.56	0.75	0.43	0.57	0.41	0.47	0.49	0.49	0.54	0.47	**0.88**
	1	0.40	0.28	**0.53**	0.28	0.49	0.49	**0.53**	0.46	0.49	0.49	0.21
f1 - measure	0	0.52	0.61	0.45	0.50	0.43	0.47	0.50	0.48	0.53	0.47	**0.66**
	1	0.43	0.37	0.50	0.33	0.47	0.48	**0.52**	0.46	0.50	0.48	0.31

6.6 Experiments with Dimensionality Reduction

From our experiments with scaling in section 6.5, we decide to keep the MinMax scaling method for future experiments, unless standardisation is explicitly required by any of the methods that we put to the test (e.g. PCA). In the following sections we experiment with different strategies of feature selection and dimensionality reduction.

For the custom feature selection techniques, we create a custom transformation class, building upon the base classes for transforming data in `sklearn`. We depict this class in Listing 6.1.

Listing 6.1: Our custom feature selection class

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.base import BaseEstimator, TransformerMixin
4
5 class MyDimRed(BaseEstimator, TransformerMixin):
6   def __init__(self, names, mode= 'mild'):
```

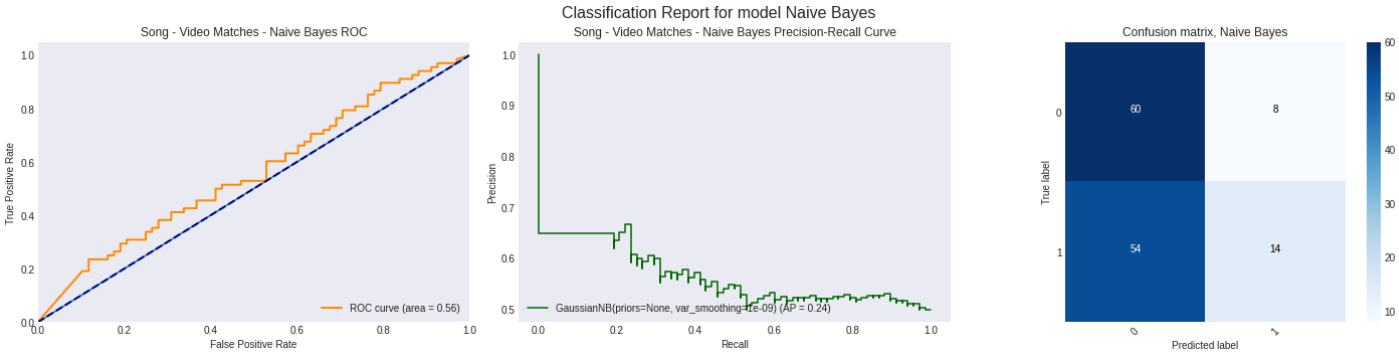


Figure 6.4: Scaling Setting 2: ROC curve, Precision Recall Curve, Confusion Matrix

```
7     self.names = names
8     self.mode = mode
9     self.keep = []
10    self.singles = []
11    self.drops = []
12    self.empty = []
13
14  def fit(self, X, y = None):
15    X = pd.DataFrame(X, columns=self.names)
16    self.empty_cols = [col for col in X.columns if (X[col] == 0).all()]
17    corr = X.corr(method='pearson')
18    corr_a = corr.abs()
19    s = corr_a.unstack().sort_values(kind="quicksort")
20    pairs = set(s[s > 0.7].index)
21    clean_pairs = set([tuple(sorted(i)) for i in pairs if i[0] != i[1]])
22    # only keep one for each pair of highly correlated features.
23    for pair in sorted(clean_pairs):
24        if pair[1] not in self.drops:
25            self.drops.append(pair[1])
26        if pair[0] not in self.drops and pair[0] not in self.singles:
27            self.singles.append(pair[0])
28    self.keep = [col for col in X.columns if col in self.singles or col not in self.drops]
29    return self
30
31  def transform(self, X, y = None, names=[]):
32    X = pd.DataFrame(X, columns=self.names)
33    if self.mode == 'extreme':
34      cols_to_keep = [col for col in self.singles if col not in self.empty_cols]
35    elif self.mode == 'mild':
36      cols_to_keep = [col for col in self.keep if col not in self.empty_cols]
37    else:
38      raise ValueError
39    X = X[cols_to_keep]
40    X = X.to_numpy()
41    return X
```

6.6.1 Correlation-based feature selection

We remove from the data-set columns that only contain the value zero. We perform a Pearson correlation test for cross-correlation between features, and identify 1057 pairs with at least 80% correlation. For this setting, we keep the first element of each pair. We also keep the columns that have no correlation with other columns. Naïve Bayes is still the winner, though the results are still worse on average to the baseline. The results are presented in Table 6.6 and Figure 6.5.

Table 6.6: Dimensionality Reduction Setting 1: Correlation-based feature selection

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		**0.56**	0.43	0.55	0.40	0.43	0.35	0.43	0.42	0.44	0.54	**0.56**
precision	0	0.58	0.43	0.56	0.39	0.45	0.36	0.44	0.44	0.45	0.53	**0.68**
	1	0.55	0.43	0.54	0.42	0.42	0.34	0.40	0.39	0.42	**0.58**	0.54
recall	0	0.43	0.43	0.46	0.32	0.54	0.40	0.56	0.54	0.56	**0.76**	0.22
	1	0.69	0.44	0.65	0.49	0.32	0.31	0.29	0.29	0.32	0.32	**0.90**
f1 - measure	0	0.49	0.43	0.50	0.35	0.49	0.38	0.49	0.48	0.50	**0.63**	0.33
	1	0.61	0.44	0.59	0.45	0.36	0.32	0.34	0.34	0.37	0.42	**0.67**

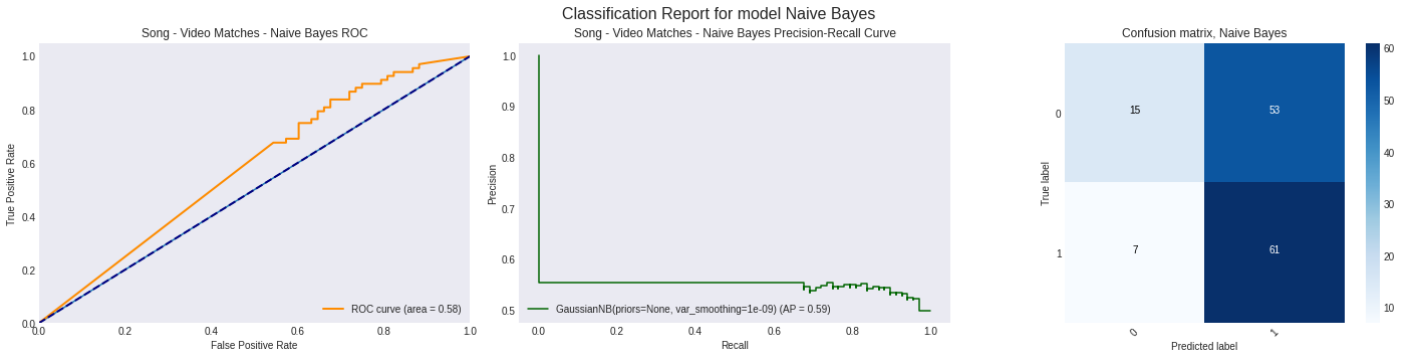


Figure 6.5: Dimensionality Reduction Setting 1: ROC curve, Precision Recall Curve, Confusion Matrix

6.6.2 Correlation-based Extreme Dimensionality Reduction

In our second experiment we only keep the features that participate in high correlation pairs. This time kNN is the better algorithm, with 55% average accuracy, and 55% average f1 measure. So far the baseline experiment still yields the best outcome. The results are presented in 6.7 and Figure 6.6

Table 6.7: Dimensionality Reduction Setting 2: Correlation-based Extreme Dimensionality Reduction

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.50	0.43	**0.55**	0.41	0.45	0.41	0.37	0.51	0.53	0.51	0.51
precision	0	0.50	0.42	**0.57**	0.42	0.45	0.42	0.39	0.51	0.52	0.51	0.51
	1	0.50	0.43	**0.54**	0.41	0.44	0.40	0.33	0.51	**0.54**	0.52	0.50
recall	0	0.65	0.41	0.44	0.44	0.51	0.46	0.47	0.62	0.68	**0.69**	0.26
	1	0.35	0.44	0.66	0.38	0.38	0.37	0.26	0.40	0.38	0.34	**0.75**
f1 - measure	0	0.56	0.42	0.50	0.43	0.48	0.44	0.43	0.56	**0.59**	**0.59**	0.35
	1	0.41	0.43	**0.60**	0.39	0.41	0.38	0.30	0.45	0.45	0.41	**0.60**

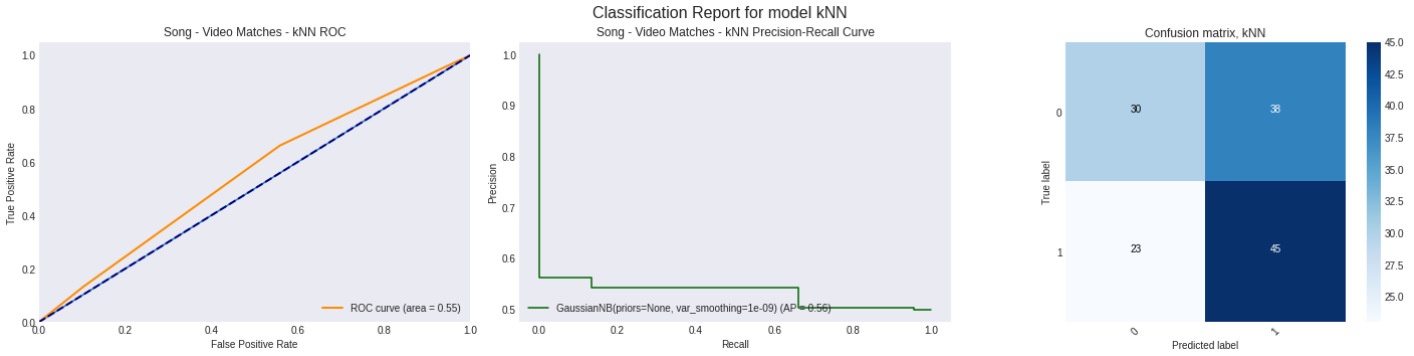


Figure 6.6: Dimensionality Reduction Setting 2: ROC curve, Precision Recall Curve, Confusion Matrix

6.6.3 Mutual Information-based Dimensionality Reduction

In the third setting we select the top 200 features based on mutual information with the target value. The calculation is fitted only on the training set, for every fold. We notice that this deteriorates the performance of almost all the models, and the winner is once again Naïve Bayes. The results are presented in 6.8 and Figure 6.7

Table 6.8: Dimensionality Reduction Setting 3: Mutual Information

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.42	0.49	0.53	0.38	0.41	0.40	0.41	0.49	0.51	0.50	**0.58**
precision	0	0.43	0.49	0.53	0.36	0.42	0.42	0.41	0.49	0.51	0.50	**0.67**
	1	0.41	0.48	0.53	0.39	0.41	0.39	0.41	0.48	0.52	0.50	**0.55**
recall	0	0.49	0.54	0.49	0.31	0.44	0.47	0.43	0.63	0.65	**0.76**	0.32
	1	0.35	0.43	0.57	0.44	0.38	0.34	0.40	0.34	0.38	0.24	**0.84**
f1 - measure	0	0.46	0.51	0.51	0.33	0.43	0.44	0.42	0.55	0.57	**0.60**	0.44
	1	0.38	0.45	0.55	0.41	0.39	0.36	0.40	0.40	0.44	0.32	**0.67**

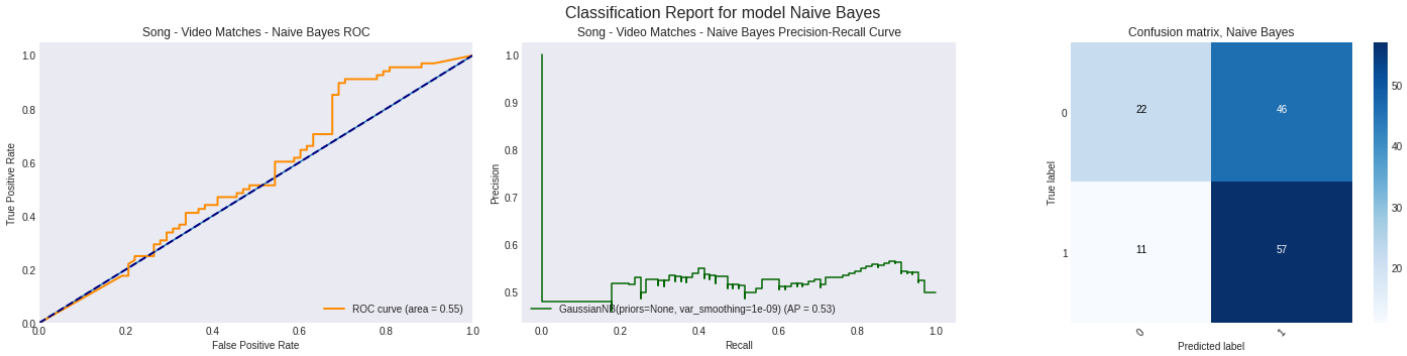


Figure 6.7: Dimensionality Reduction Setting 3: ROC curve, Precision Recall Curve, Confusion Matrix

6.6.4 Dimensionality Reduction with PCA

In a final dimensionality reduction experiment, we perform Principal Component Analysis (PCA) on the training set of each fold. We use the implementation in `scikit`, with the solver 'arpack', which truncates the Singular Value Decomposition keeping components based on the following rule:

$$n_{components} = \min(n_{samples}, n_{features}) - 1 \tag{6.2}$$

In order to apply PCA, we swap the MinMax scaling function with the Standard-Scaler of section 6.5.1. This time, Extra Trees is the winning model with 54% accuracy and f1-measure. This is consistent with the results in section 6.5.1, where the Decision Tree had an improved performance. While on average this setting is better than the baseline, it is less robust, as the standard deviation of the accuracy is much larger. The results are summarised in Table 6.9 and Figure 6.8

Table 6.9: Dimensionality Reduction Setting 4: PCA with arpack solver

		DT	LR	kNN	SVM	RandF	BTr	AdaBoost	XGBoost	GradB	ExTr	NB
accuracy		0.46	0.46	0.49	0.51	0.40	0.46	0.49	0.46	0.51	**0.54**	0.50
precision	0	0.46	0.46	0.49	0.51	0.41	0.46	0.49	0.46	0.52	**0.56**	0.00
	1	0.46	0.46	0.49	0.55	0.38	0.45	0.48	0.46	0.51	0.54	**0.50**
recall	0	0.46	0.49	0.47	**0.87**	0.46	0.51	0.54	0.44	0.49	0.43	0.00
	1	0.46	0.44	0.51	0.16	0.34	0.40	0.43	0.49	0.54	0.66	**1.00**
f1 - measure	0	0.46	0.47	0.48	**0.64**	0.43	0.49	0.51	0.45	0.50	0.48	0.00
	1	0.46	0.45	0.50	0.25	0.36	0.42	0.45	0.47	0.53	0.59	**0.67**

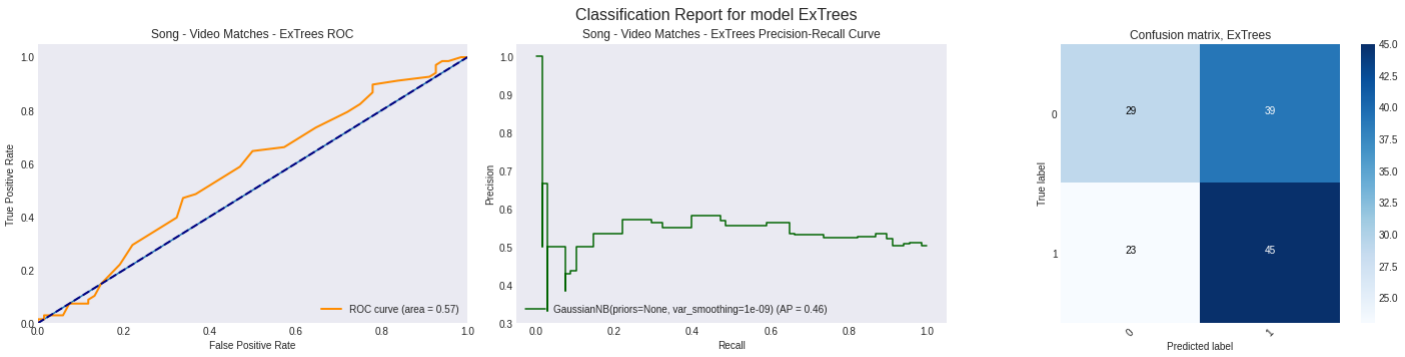


Figure 6.8: Dimensionality Reduction Setting 4: ROC curve, Precision Recall Curve, Confusion Matrix

6.7 Model Optimisations

After all the experimentation in previous sections, we have three candidate pipelines. These include a scaling stage, a dimensionality reduction stage, and an estimator stage. All three candidate pipelines belong to different families of classifiers. The parameters checked, as well as the results are demonstrated in Table 6.10:

Table 6.10: Estimator Tuning

Pipeline	Parameter	Values	Accuracy	Macro f1 score
Standard Scaling, PCA, Extra Trees	Number of Estimators	50	0.49	0.48
		100	0.54	0.54
		150	0.60	0.58
		200	0.58	0.56
MinMax Scaling, Mutual Information Dimensionality Reduction, Naïve Bayes	Number of Features Kept	50	0.43	0.42
		100	0.48	0.45
		150	0.54	0.49
		200	0.51	0.47
MinMax Scaling, Custom Dimensionality Reduction, kNN	Number of Neighbors	1	0.53	0.53
		2	0.51	0.49
		3	0.53	0.53
		4	0.48	0.46

The winning pipeline is Standard Scaling, PCA and Extra Trees, with an accuracy of 60% and macro-average f1-measure of 58%. The complete classification report is given in Table 6.11 and Figure 6.9. Both the area under the ROC curve

and the area under the Precision-Recall curve indicate that the classifier, although not ideal, is better than random choice. By examining the confusion matrix, we notice that the recall in the positive class is high, and the precision is adequate. The trade-off between precision and recall, as shown by the shape of the Precision-Recall Curve, seems to indicate that the operation point chosen is the best balance between the two.

Table 6.11: Classification report for winning estimator

	0	1	Macro Avg
Precison	0.64	0.57	0.61
Recall	0.43	0.76	0.60
F1 Measure	0.51	0.65	0.58
Accuracy			0.60

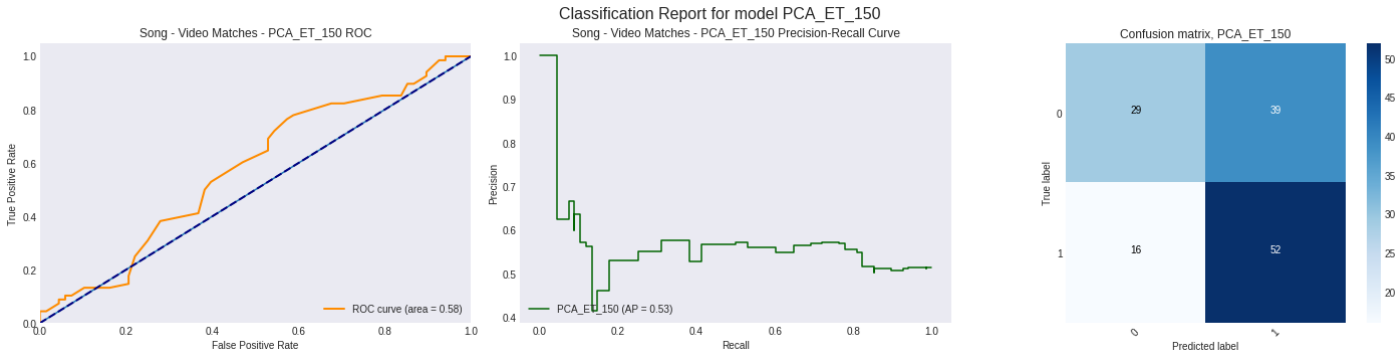


Figure 6.9: Winning Estimator: ROC curve, Precision Recall Curve, Confusion Matrix

The exact parameters of the winning algorithm are presented in Listing 6.2.

Listing 6.2: Parameters of the Selected Estimation Pipeline

```
from sklearn import preprocessing
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.pipeline import Pipeline

pipe_PCA_ET_150 = Pipeline([
              # PCA with Extra Trees, using min samples leaf = 1 etc.
              ('impute', SimpleImputer(missing_values=np.NaN, strategy='mean')),
              ('scaling', preprocessing.StandardScaler(copy=True, with_mean=True, with_std=True)),
              ('reduce_dim', PCA(copy=True, random_state=seed, svd_solver='arpack')),
              ('classify', ExtraTreesClassifier(random_state=seed, n_jobs=-1, n_estimators=150))
          ])
```

6.8 Results Discussion

We attempted to approach our methodology in a way that would reduce bias, while taking advantage of the limited data set as much as possible. This latter constraint stopped us from using a held out set of films in order to estimate how well the models generalise. Given that we only had samples from 33 films, removing any of them from the set would have a big impact in the learning process. The limitations of the data-set also stopped us from testing deep architectures.

Furthermore, the mediocre results of algorithms, such as SVM, that usually perform well, indicate that probably more data is needed in order for the classifier to learn the relationships between so many variables. This makes sense, given the original dimensions that we are working with. It is also worth noting that the mutual information and Spearman correlation tests performed in section 6.1.3 scored the visual features quite low. Perhaps this is an indication that the model doesn't have enough data yet, in order to determine the association between visual features and music. To determine if this is a valid assumption, we would need to perform some tests by isolating pairs of modalities, but time would not allow for this to happen.

Despite those challenges, a fine-tuned pipeline including a scaling, a dimensionality reduction and a classification step achieved results that were better than chance. We believe that this is a promising indication that more data would allow a future researcher to build a much more robust model. This being a proof of concept, we consider the results adequate enough to conclude our own research. While, overall, the model would not be suitable for cases where high precision matters, at this stage it could serve as an initial filtering classifier for a soundtrack retrieval system.

Chapter 7

Extensions and Future Work

We have now reached the end of our experiments on the domain of video soundtracks and machine learning. In the previous chapters, we proposed a data collection and feature extraction pipeline, for data of all relevant modalities (audio, MIDI and video). We also created a classification model using the data that we collected, and performed hyper-parameter tuning in order to get results that were statistically better than chance, even though the data-set that we collected was quite small.

In this final chapter, we will propose some ideas for improving most aspects of our work, by addressing the caveats and challenges described in sections 4.2.1, 2.4.4.2 and 2.5.2. Further to that, we will also attempt to point towards some potential extensions, use-cases and areas of further research.

7.1 Improvements in the Data Collection Pipeline

Due to the nature of the task, most of our work revolved around building up the data-set. Despite our best efforts, the issues that we identified in sections 2.4.4.2 and 2.5.2 made it especially hard to gather a large enough data-set in terms of data points. Furthermore, as we described in section 2.5.2.5, we had to manually intervene in the pipeline in two cases, which is a major opportunity for improvement. In the following sections, we propose some ideas to address these issues.

7.1.1 Expanding the Collection

As we mentioned in section 2.7, the way that the pipeline is built, allows for bootstrapping more data from the community, without sharing any of the raw underlying content. While the robustness of the process in other computer set-ups (especially set-ups that are not Ubuntu Linux) has not been tested, we believe that if this project was to be given to the wider Data Science community, the crowd-source element would make it possible to amass much larger volumes of data.

7.1.2 Data Quality

Another issue that we identified in both sections 2.4.4.2 and 2.5.2.4 was that often the actual content of a file was not the same as what the filename would suggest. This problem is very important, as slight errors in labelling can lead to errors in matching, and essentially create severe issues in later stages of the pipeline. Each of the file types had its own types of data corruption, which we will attempt to address in the following paragraphs.

7.1.2.1 Audio Files

The most common issue with the audio files was the existence of live performances or covers of songs in the data-set. Even though we tried to impose some filtering in the initial parsing and cataloging of the data, when we were reviewing the final catalog we came across plenty of mislabeled files. We propose some possible solutions:

- Filter out folders using a wider range of relevant keywords. This could be a much faster and effective alternative to clean up the data at a very early stage.
- Use the metadata stored within the file, using `ffmpeg`. A lot of files, especially when stored in MP3 form, have quite comprehensive metadata, such as album name, song name, artist, composer etc. This metadata could be compared against the name of the file and, when available, used to improve the query that is made to the external knowledge base (Spotify).

- Try other knowledge bases, other than Spotify, to retrieve a proper ID for the file.

7.1.2.2 MIDI Files

MIDI files were a particularly problematic data source in this pipeline. Most usually, this type of file exists for mobile phone ringtones or karaoke applications. The files that are freely available on the Web and belong to pop, rock or jazz genres, are often of very low quality. The choice of instruments is rarely correct, and it is not uncommon that the notes are incorrect or the file is corrupt. Furthermore, the naming conventions are inconsistent, and the existence of metadata, like artist name or song name, within the file, is very rare. Some possible alternative approaches include:

- Use the "Lakh MIDI Dataset" by Raffel [67]. This is an open-source data set that contains plenty of clean, tested MIDI files, that have been matched to songs that exist in the "Million Song Database". In our own implementation, while the data existed within the files that we had access to, we failed to find the database that contains all the essential clean names until much later in the process. The hash-based naming convention that the file system has, made it impossible to use the files without the clean name lookup tables.

- Attempt to parse the files with a library like `mido` in order to identify the instrument names and the playback duration of the file, and storing those in the appropriate table. This would allow running some diagnostics on the health of the file, and throw away corrupt files at an early stage.

- Improve the text cleaning rules that are applied on the names of the files, before performing a query in an external knowledge base.

- Try different knowledge bases, other than Spotify, to retrieve a proper ID for the file.

7.1.2.3 Audio-MIDI Matching Methods

When it came to matching audio with MIDI files, we used a simple inner join on the song name retrieved from our external knowledge base (Spotify). This method is as good as the identification methods for each modality, therefore we expect our proposals in sections 7.1.2.1 and 7.1.2.2 to greatly improve the quality of the matching. Further to that, it is possible to increase the precision of the matching, by joining on more than one fields (such as artist or album).

7.1.2.4 Video Files

The video files were mostly consistent, in terms of quality. The biggest improvement would be to use a more sophisticated cleanup method to get better file names, and compare that to an external knowledge base, such as `iMDB` in order to get more metadata, or lists of soundtracks.

7.1.3 Scalability

7.1.3.1 Reducing Manual Interventions

We explained in section 2.5.2.5 that there were two tasks that called for manual intervention in this process. If this pipeline was to scale, some additional module should be devised in order to automate, or accelerate this process.

In the task of validating the song detection in the videos, the *invalid_mode* flag in the `audio_video_matches` table could be used to filter out invalid matches. We found that the precision of the matching process is close to 100%, while recall was close to 80%, so in a larger scale, this should not be a problem.

In the task of creating samples for the negative class, the quickest and easiest solution could be to use the the same *invalid_mode* flag and assume that all these examples are mismatches. This comes at the risk of misguiding the classifier, in the cases where the audio and visual content is actually compatible.

7.1.3.2 Storage volume

As we identified in section 2.5.2.2 the fingerprint table can become massive, depending on the settings of the fingerprinting algorithm. There is a trade-off between saving storage size and achieving a high matching accuracy. A future work should determine the best balance point for these settings, but the time needed to complete an end to end experiment made it impossible to conduct within the time constraints of delivering a master thesis.

7.1.3.3 Efficiency and Speed

There are a few points in the pipeline that could benefit from some speeding up. The most time-consuming element during our own experiments, as we described in section 2.5.2.1 was the bottleneck of inserting hundreds of thousands of lines per song in the fingerprint table. A big part of this bottleneck has to do with rebuilding the index on the primary key, after each batch insert, as well as the integrity checks on the foreign key, that follow each insert. In an attempt to increase the speed, we tried to remove these constraints and build them once in the end of the fingerprint creation process, but this didn't seem to have significant positive results in efficiency. Perhaps some tuning in the InnoDB settings of MySQL server is also necessary in order to optimise the table for inserts.

Further to that, running the part of the pipeline that concerns song detection in videos also took a lot of time. The process should be easy to modify so that it can take run in a distributed way, which would probably be an easy win to reduce the time between experiments.

Another time consuming task was feature extraction, especially for the symbolic features. We have identified a few problems with the underlying library that we used, but it should not be too hard to modify our extractor class so that it runs more than one feature extraction at the same time.

7.1.4 Reducing Bias

In section 2.5.2.6 our reflection on the process revealed some design choices that introduce bias to the classifier, in a number of ways. We attempt to mitigate some of these, in the following paragraphs.

7.1.4.1 Data Diversity

The lack of diversity in our data manifests itself in a number of ways. Some of the factors that could be addressed are:

- Better handling of UTF-8 characters in our script, so that audio and midi files that have non-ASCII characters in their title can be fed to the knowledge-base query in a robust way.
- Use an external knowledge-base that has song names in their original language.
- Crowd-source the process, so that people of more diverse aesthetic and cultural background bring their own taste in film and music into the raw data-set.

7.1.4.2 Fake Example Creation

The manual creation of fake examples should be dropped in favor of some automated or semi-automated process. Some kind of bootstrapping method could be applied, but the specifics of such a method could be the object of another study.

7.1.5 Song Detection in Videos

We consider the length of the resulting data-set to be the biggest shortcoming in this work. In the following sections we propose some possible solutions that could improve on the song detection aspect of the pipeline.

7.1.5.1 Using External Knowledge Bases

Increasing the precision and recall of our method is important, but before that, we should first be able to calculate these measures. This is not currently possible, as

we do not have an accurate idea of how many songs that exist in our database also exist in each film. In order to find this out, one could use an external knowledge base, such as `iMDB`, in order to get the names of the songs that are present in each film. Furthermore, this would also help in the video clip curation process, as it could serve as a ground truth for a process to accept or reject what the song detection script yields as result.

In our case this would require some extra effort, as the names of the files that contained the films were not clean enough for the API of `iMDB` to return accurate results. Unfortunately there was not enough time for us to make the necessary changes, at the time of writing this.

7.1.5.2 Fingerprint Alignment

An important issue that we have identified in section 2.5.2.3 is that a common practice in films is alter the playback speed (and therefore the pitch) of a song, so that it can better fit the rhythm of the scene. This is detrimental for the success of the fingerprint algorithm, as it is based on the spectrogram of the sound, therefore the dependency of frequency and time. When these are altered, the peaks of the spectrogram are in different frequencies and at different timestamps.

Solving this problem could be the object of a future study, as it would probably mean that other, more robust algorithms, in comparison to the implementation of `pyDejavu`, should be tested. Other proposed algorithms exist, such as [68] which uses auto-correlation, [69] that uses wavelets, and more recently [70].

7.1.6 More Modalities

Another potential field of extending this method, would be to add additional modalities for the participating content, such as text, in the form of lyrics or subtitles. It would be interesting to research whether there are semantic or topic-related similarities between the lyrics of a song and the lines of dialogue that are uttered during the scene.

Furthermore, another potentially interesting direction would be to add metadata

about the scene, perhaps using the results of a classifier that labels scene content as features. Some examples of this are [71] and [72].

7.2 Improvements in Model Selection

Given the dimensionality of the data-set, we would argue that the acquisition of more and better data is much more important than building more sophisticated algorithms. Nevertheless, in the following sections, we present some potentially interesting experiments could perhaps yield better results.

7.2.1 Voting Ensembles

Our experiments led to three classification settings that included different scaling and dimensionality reduction techniques, as well as classifiers from different families. We performed some tuning of their basic hyper-parameters in order to choose the best possible model. Unfortunately time didn't allow for us to try combining all three into a Voting ensemble. Theoretically, this could improve the overall accuracy and robustness of the model, if each of the participating classifiers is behaving differently, in different subsets of the data.

7.2.2 Deep Architectures

While testing models, we ruled out Deep architectures, due to the size of the data that was available for training and testing. Neural Networks are commonly used for large data-sets, in order to avoid the danger of over-fitting. If future research accomplishes to solve the data availability issue, the large number of features that are extracted would make this a suitable setting for experimentation with Neural Networks.

7.2.3 Classifier by modality combination

One of our biggest concerns around the classifier's performance, is what exactly it is learning. The Mutual Information tests that we applied in section 6.1 show that the

visual content features have significantly lower mutual information with the target variable than the features of the other modalities. We have hypothesized that this is due to the lack of diversity in the visual content, and the small size of the data-set. If more data were collected, this hypothesis should be put to the test in future work.

7.3 Conclusion

In this thesis, we got involved with the task of video soundtrack evaluation. We proposed an end-to-end data collection and feature extraction pipeline, which takes raw video, audio and MIDI files as its input, matches them using text processing and audio fingerprinting, and creates a multimodal feature library of features. To our knowledge, this is the first attempt to combine these three modalities. We managed to create a small proof of concept using some limited data, and built a classifier with the task of discriminating between real and fake soundtracks. The results of this classifier were adequate, as the resulting accuracy is better than random choice, though we believe that the performance could be improved by assembling a larger data-set. Overall, the model would not be suitable for cases where high precision matters, but at this stage could serve as an initial filtering classifier for a soundtrack retrieval system. Finally, we explored some solutions to the various challenges and problems of both the data collection process and of the resulting classifier, and presented some interesting directions for future research.

Appendix A

The Complete Feature Library

Table A.1: Audio Feature Overview

Feat. ID	**Feat. Class**	**Feat. Name**	**Section**
1	Time Domain	zcr_mean	3.2.4.3
2	Time Domain	energy_mean	3.2.4.2
3	Time Domain	energy_entropy_mean	3.2.4.4
4	Spectral Domain	spectral_centroid_mean	3.2.5.2
5	Spectral Domain	spectral_spread_mean	3.2.5.2
6	Spectral Domain	spectral_entropy_mean	3.2.5.3
7	Spectral Domain	spectral_flux_mean	3.2.5.4
8	Spectral Domain	spectral_rolloff_mean	3.2.5.5
9	Cepstral Domain	mfcc_1_mean	3.2.5.6
10	Cepstral Domain	mfcc_2_mean	3.2.5.6
11	Cepstral Domain	mfcc_3_mean	3.2.5.6
12	Cepstral Domain	mfcc_4_mean	3.2.5.6
13	Cepstral Domain	mfcc_5_mean	3.2.5.6
14	Cepstral Domain	mfcc_6_mean	3.2.5.6
15	Cepstral Domain	mfcc_7_mean	3.2.5.6
16	Cepstral Domain	mfcc_8_mean	3.2.5.6

Feat. ID	Feat. Class	Feat. Name	Section
17	Cepstral Domain	mfcc_9_mean	3.2.5.6
18	Cepstral Domain	mfcc_10_mean	3.2.5.6
19	Cepstral Domain	mfcc_11_mean	3.2.5.6
20	Cepstral Domain	mfcc_12_mean	3.2.5.6
21	Cepstral Domain	mfcc_13_mean	3.2.5.6
22	Chroma Coefficients	chroma_1_mean	3.2.5.7
23	Chroma Coefficients	chroma_2_mean	3.2.5.7
24	Chroma Coefficients	chroma_3_mean	3.2.5.7
25	Chroma Coefficients	chroma_4_mean	3.2.5.7
26	Chroma Coefficients	chroma_5_mean	3.2.5.7
27	Chroma Coefficients	chroma_6_mean	3.2.5.7
28	Chroma Coefficients	chroma_7_mean	3.2.5.7
29	Chroma Coefficients	chroma_8_mean	3.2.5.7
30	Chroma Coefficients	chroma_9_mean	3.2.5.7
31	Chroma Coefficients	chroma_10_mean	3.2.5.7
32	Chroma Coefficients	chroma_11_mean	3.2.5.7
33	Chroma Coefficients	chroma_12_mean	3.2.5.7
34	Chroma Coefficients	chroma_std_mean	3.2.5.7
35	Time Domain	delta zcr_mean	3.2.4.3
36	Time Domain	delta energy_mean	3.2.4.2
37	Time Domain	delta energy_entropy_mean	3.2.4.4
38	Spectral Domain	delta spectral_centroid_mean	3.2.5.2
39	Spectral Domain	delta spectral_spread_mean	3.2.5.2
40	Spectral Domain	delta spectral_entropy_mean	3.2.5.3
41	Spectral Domain	delta spectral_flux_mean	3.2.5.4
42	Spectral Domain	delta spectral_rolloff_mean	3.2.5.5
43	Cepstral Domain	delta mfcc_1_mean	3.2.5.6
44	Cepstral Domain	delta mfcc_2_mean	3.2.5.6

Feat. ID	Feat. Class	Feat. Name	Section
45	Cepstral Domain	delta mfcc_3_mean	3.2.5.6
46	Cepstral Domain	delta mfcc_4_mean	3.2.5.6
47	Cepstral Domain	delta mfcc_5_mean	3.2.5.6
48	Cepstral Domain	delta mfcc_6_mean	3.2.5.6
49	Cepstral Domain	delta mfcc_7_mean	3.2.5.6
50	Cepstral Domain	delta mfcc_8_mean	3.2.5.6
51	Cepstral Domain	delta mfcc_9_mean	3.2.5.6
52	Cepstral Domain	delta mfcc_10_mean	3.2.5.6
53	Cepstral Domain	delta mfcc_11_mean	3.2.5.6
54	Cepstral Domain	delta mfcc_12_mean	3.2.5.6
55	Cepstral Domain	delta mfcc_13_mean	3.2.5.6
56	Chroma Coefficients	delta chroma_1_mean	3.2.5.7
57	Chroma Coefficients	delta chroma_2_mean	3.2.5.7
58	Chroma Coefficients	delta chroma_3_mean	3.2.5.7
59	Chroma Coefficients	delta chroma_4_mean	3.2.5.7
60	Chroma Coefficients	delta chroma_5_mean	3.2.5.7
61	Chroma Coefficients	delta chroma_6_mean	3.2.5.7
62	Chroma Coefficients	delta chroma_7_mean	3.2.5.7
63	Chroma Coefficients	delta chroma_8_mean	3.2.5.7
64	Chroma Coefficients	delta chroma_9_mean	3.2.5.7
65	Chroma Coefficients	delta chroma_10_mean	3.2.5.7
66	Chroma Coefficients	delta chroma_11_mean	3.2.5.7
67	Chroma Coefficients	delta chroma_12_mean	3.2.5.7
68	Chroma Coefficients	delta chroma_std_mean	3.2.5.7
69	Time Domain	zcr_std	3.2.4.3
70	Time Domain	energy_std	3.2.4.2
71	Time Domain	energy_entropy_std	3.2.4.4
72	Spectral Domain	spectral_centroid_std	3.2.5.2

Feat. ID	Feat. Class	Feat. Name	Section
73	Spectral Domain	spectral_spread_std	3.2.5.2
74	Spectral Domain	spectral_entropy_std	3.2.5.3
74	Spectral Domain	spectral_flux_std	3.2.5.4
75	Spectral Domain	spectral_rolloff_std	3.2.5.5
76	Cepstral Domain	mfcc_1_std	3.2.5.6
77	Cepstral Domain	mfcc_2_std	3.2.5.6
78	Cepstral Domain	mfcc_3_std	3.2.5.6
79	Cepstral Domain	mfcc_4_std	3.2.5.6
80	Cepstral Domain	mfcc_5_std	3.2.5.6
81	Cepstral Domain	mfcc_6_std	3.2.5.6
82	Cepstral Domain	mfcc_7_std	3.2.5.6
83	Cepstral Domain	mfcc_8_std	3.2.5.6
83	Cepstral Domain	mfcc_9_std	3.2.5.6
84	Cepstral Domain	mfcc_10_std	3.2.5.6
85	Cepstral Domain	mfcc_11_std	3.2.5.6
86	Cepstral Domain	mfcc_12_std	3.2.5.6
87	Cepstral Domain	mfcc_13_std	3.2.5.6
88	Chroma Coefficients	chroma_1_std	3.2.5.7
89	Chroma Coefficients	chroma_2_std	3.2.5.7
90	Chroma Coefficients	chroma_3_std	3.2.5.7
91	Chroma Coefficients	chroma_4_std	3.2.5.7
92	Chroma Coefficients	chroma_5_std	3.2.5.7
93	Chroma Coefficients	chroma_6_std	3.2.5.7
94	Chroma Coefficients	chroma_7_std	3.2.5.7
95	Chroma Coefficients	chroma_8_std	3.2.5.7
96	Chroma Coefficients	chroma_9_std	3.2.5.7
97	Chroma Coefficients	chroma_10_std	3.2.5.7
98	Chroma Coefficients	chroma_11_std	3.2.5.7

Feat. ID	Feat. Class	Feat. Name	Section
99	Chroma Coefficients	chroma_12_std	3.2.5.7
100	Chroma Coefficients	chroma_std_std	3.2.5.7
101	Time Domain	delta zcr_std	3.2.4.3
102	Time Domain	delta energy_std	3.2.4.2
103	Time Domain	delta energy_entropy_std	3.2.4.4
104	Spectral Domain	delta spectral_centroid_std	3.2.5.2
105	Spectral Domain	delta spectral_spread_std	3.2.5.2
106	Spectral Domain	delta spectral_entropy_std	3.2.5.3
107	Spectral Domain	delta spectral_flux_std	3.2.5.4
108	Spectral Domain	delta spectral_rolloff_std	3.2.5.5
109	Cepstral Domain	delta mfcc_1_std	3.2.5.6
110	Cepstral Domain	delta mfcc_2_std	3.2.5.6
111	Cepstral Domain	delta mfcc_3_std	3.2.5.6
112	Cepstral Domain	delta mfcc_4_std	3.2.5.6
113	Cepstral Domain	delta mfcc_5_std	3.2.5.6
114	Cepstral Domain	delta mfcc_6_std	3.2.5.6
115	Cepstral Domain	delta mfcc_7_std	3.2.5.6
116	Cepstral Domain	delta mfcc_8_std	3.2.5.6
117	Cepstral Domain	delta mfcc_9_std	3.2.5.6
118	Cepstral Domain	delta mfcc_10_std	3.2.5.6
119	Cepstral Domain	delta mfcc_11_std	3.2.5.6
120	Cepstral Domain	delta mfcc_12_std	3.2.5.6
121	Cepstral Domain	delta mfcc_13_std	3.2.5.6
122	Chroma Coefficients	delta chroma_1_std	3.2.5.7
123	Chroma Coefficients	delta chroma_2_std	3.2.5.7
124	Chroma Coefficients	delta chroma_3_std	3.2.5.7
125	Chroma Coefficients	delta chroma_4_std	3.2.5.7
126	Chroma Coefficients	delta chroma_5_std	3.2.5.7

Feat. ID	Feat. Class	Feat. Name	Section
127	Chroma Coefficients	delta chroma_6_std	3.2.5.7
128	Chroma Coefficients	delta chroma_7_std	3.2.5.7
129	Chroma Coefficients	delta chroma_8_std	3.2.5.7
130	Chroma Coefficients	delta chroma_9_std	3.2.5.7
131	Chroma Coefficients	delta chroma_10_std	3.2.5.7
132	Chroma Coefficients	delta chroma_11_std	3.2.5.7
133	Chroma Coefficients	delta chroma_12_std	3.2.5.7
134	Chroma Coefficients	delta chroma_std_std	3.2.5.7
135	Tempo	beat	3.3.1
136	Tempo	beat_conf	3.3.1

Table A.2: Symbolic Feature Overview

Feat. ID	Feat. Class	Feat. Name	Feature Category
1	jSymbolic	AverageMelodicIntervalFeature	Melodies and Horizontal Intervals
2	jSymbolic	MostCommonMelodicIntervalFeature	Melodies and Horizontal Intervals
3	jSymbolic	DistanceBetweenMostCommonMelodicIntervalsFeature	Melodies and Horizontal Intervals
4	jSymbolic	MostCommonMelodicIntervalPrevalenceFeature	Melodies and Horizontal Intervals
5	jSymbolic	RelativeStrengthOfMostCommonIntervalsFeature	Melodies and Horizontal Intervals

Feat. ID	Feat. Class	Feat. Name	Feature Category
6	jSymbolic	NumberOfCommonMelodicIntervalsFeature	Melodies and Horizontal Intervals
7	jSymbolic	AmountOfArpeggiationFeature	Melodies and Horizontal Intervals
8	jSymbolic	RepeatedNotesFeature	Melodies and Horizontal Intervals
9	jSymbolic	ChromaticMotionFeature	Melodies and Horizontal Intervals
10	jSymbolic	StepwiseMotionFeature	Melodies and Horizontal Intervals
11	jSymbolic	MelodicThirdsFeature	Melodies and Horizontal Intervals
12	jSymbolic	MelodicFifthsFeature	Melodies and Horizontal Intervals
13	jSymbolic	MelodicTritonesFeature	Melodies and Horizontal Intervals
14	jSymbolic	MelodicOctavesFeature	Melodies and Horizontal Intervals
15	jSymbolic	DirectionOfMotionFeature	Melodies and Horizontal Intervals

Feat. ID	Feat. Class	Feat. Name	Feature Category
16	jSymbolic	DurationOfMelodicArcsFeature	Melodies and Horizontal Intervals
17	jSymbolic	SizeOfMelodicArcsFeature	Melodies and Horizontal Intervals
18	jSymbolic	MostCommonPitchPrevalenceFeature	Pitch Statistics
19	jSymbolic	MostCommonPitchClassPrevalenceFeature	Pitch Statistics
20	jSymbolic	RelativeStrengthOfTopPitchesFeature	Pitch Statistics
21	jSymbolic	RelativeStrengthOfTopPitchClassesFeature	Pitch Statistics
22	jSymbolic	IntervalBetweenStrongestPitchesFeature	Pitch Statistics
23	jSymbolic	IntervalBetweenStrongestPitchClassesFeature	Pitch Statistics
24	jSymbolic	NumberOfCommonPitchesFeature	Pitch Statistics
25	jSymbolic	PitchVarietyFeature	Pitch Statistics
26	jSymbolic	PitchClassVarietyFeature	Pitch Statistics
27	jSymbolic	RangeFeature	Pitch Statistics
28	jSymbolic	MostCommonPitchFeature	Pitch Statistics

Feat. ID	Feat. Class	Feat. Name	Feature Category
29	jSymbolic	PrimaryRegisterFeature	Pitch Statistics
30	jSymbolic	ImportanceOfBassRegisterFeature	Pitch Statistics
31	jSymbolic	ImportanceOfMiddleRegisterFeature	Pitch Statistics
32	jSymbolic	ImportanceOfHighRegisterFeature	Pitch Statistics
33	jSymbolic	MostCommonPitchClassFeature	Pitch Statistics
34	jSymbolic	NoteDensityFeature	Rhythm
35	jSymbolic	AverageNoteDurationFeature	Rhythm
36	jSymbolic	VariabilityOfNoteDurationFeature	Rhythm
37	jSymbolic	MaximumNoteDurationFeature	Rhythm
38	jSymbolic	MinimumNoteDurationFeature	Rhythm
39	jSymbolic	StaccatoIncidenceFeature	Rhythm
40	jSymbolic	AverageTimeBetweenAttacksFeature	Rhythm
41	jSymbolic	VariabilityOfTimeBetweenAttacksFeature	Rhythm
42	jSymbolic	AverageTimeBetweenAttacksForEachVoiceFeature	Rhythm
43	jSymbolic	AverageVariabilityOfTimeBetweenAttacksForEachVoiceFeature	Rhythm
44	jSymbolic	InitialTempoFeature	Rhythm
45	jSymbolic	InitialTimeSignatureFeature	Rhythm
46	jSymbolic	CompoundOrSimpleMeterFeature	Rhythm
47	jSymbolic	TripleMeterFeature	Rhythm
48	jSymbolic	QuintupleMeterFeature	Rhythm
49	jSymbolic	ChangesOfMeterFeature	Rhythm
50	jSymbolic	DurationFeature	Rhythm
51	jSymbolic	MaximumNumberOfIndependentVoicesFeature	Texture

Feat. ID	Feat. Class	Feat. Name	Feature Category
52	jSymbolic	AverageNumberOfIndependentVoicesFeature	Texture
53	jSymbolic	VariabilityOfNumberOfIndependentVoicesFeature	Texture
54	native	QualityFeature	Texture
55	native	TonalCertainty	Texture
56	native	UniqueNoteQuarterLengths	Texture
57	native	MostCommonNoteQuarterLength	Texture
58	native	MostCommonNoteQuarterLengthPrevalence	Texture
59	native	RangeOfNoteQuarterLengths	Texture
60	native	UniquePitchClassSetSimultaneities	Chords and Vertical Intervals
61	native	UniqueSetClassSimultaneities	Chords and Vertical Intervals
62	native	MostCommonPitchClassSetSimultaneityPrevalence	Chords and Vertical Intervals
63	native	MostCommonSetClassSimultaneityPrevalence	Chords and Vertical Intervals
64	native	MajorTriadSimultaneityPrevalence	Chords and Vertical Intervals
65	native	MinorTriadSimultaneityPrevalence	Chords and Vertical Intervals
66	native	DominantSeventhSimultaneityPrevalence	Chords and Vertical Intervals

Feat. ID	Feat. Class	Feat. Name	Feature Category
67	native	DiminishedTriadSimultaneityPrevalence	Chords and Vertical Intervals
68	native	TriadSimultaneityPrevalence	Chords and Vertical Intervals
69	native	DiminishedSeventhSimultaneityPrevalence	Chords and Vertical Intervals
70	native	IncorrectlySpelledTriadPrevalence	
71	native	ComposerPopularity	Miscellaneous Features
72	native	LandiniCadence	Miscellaneous Features
73	native	LanguageFeature	Miscellaneous Features

Table A.3: Video Feature Overview

Feat. ID	Feat. Class	Feat. Name	Section
1	Color Features	mean_hist_r0	5.1.2
2	Color Features	mean_hist_r1	5.1.2
3	Color Features	mean_hist_r2	5.1.2
4	Color Features	mean_hist_r3	5.1.2
5	Color Features	mean_hist_r4	5.1.2
6	Color Features	mean_hist_r5	5.1.2
7	Color Features	mean_hist_r6	5.1.2
8	Color Features	mean_hist_r7	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
9	Color Features	mean_hist_g0	5.1.2
10	Color Features	mean_hist_g1	5.1.2
11	Color Features	mean_hist_g2	5.1.2
12	Color Features	mean_hist_g3	5.1.2
13	Color Features	mean_hist_g4	5.1.2
14	Color Features	mean_hist_g5	5.1.2
15	Color Features	mean_hist_g6	5.1.2
16	Color Features	mean_hist_g7	5.1.2
17	Color Features	mean_hist_b0	5.1.2
18	Color Features	mean_hist_b1	5.1.2
19	Color Features	mean_hist_b2	5.1.2
20	Color Features	mean_hist_b3	5.1.2
21	Color Features	mean_hist_b4	5.1.2
22	Color Features	mean_hist_b5	5.1.2
23	Color Features	mean_hist_b6	5.1.2
24	Color Features	mean_hist_b7	5.1.2
25	Color Features	mean_hist_v0	5.1.2
26	Color Features	mean_hist_v1	5.1.2
27	Color Features	mean_hist_v2	5.1.2
28	Color Features	mean_hist_v3	5.1.2
29	Color Features	mean_hist_v4	5.1.2
30	Color Features	mean_hist_v5	5.1.2
31	Color Features	mean_hist_v6	5.1.2
32	Color Features	mean_hist_v7	5.1.2
33	Color Features	mean_hist_rgb0	5.1.2
34	Color Features	mean_hist_rgb1	5.1.2
35	Color Features	mean_hist_rgb2	5.1.2
36	Color Features	mean_hist_rgb3	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
37	Color Features	mean_hist_rgb4	5.1.2
38	Color Features	mean_hist_s0	5.1.2
39	Color Features	mean_hist_s1	5.1.2
40	Color Features	mean_hist_s2	5.1.2
41	Color Features	mean_hist_s3	5.1.2
42	Color Features	mean_hist_s4	5.1.2
43	Color Features	mean_hist_s5	5.1.2
44	Color Features	mean_hist_s6	5.1.2
45	Color Features	mean_hist_s7	5.1.2
46	Flow Features	mean_frame_value_diff	5.2
47	Flow Features	mean_frontal_faces_num	5.3
48	Flow Features	mean_fronatl_faces_ratio	5.3
49	Flow Features	mean_tilt_pan_confidences	5.2
50	Flow Features	mean_mag_mean	5.2
51	Flow Features	mean_mag_std	5.2
52	Flow Features	mean_shot_durations	5.2
53	Color Features	std_hist_r0	5.1.2
54	Color Features	std_hist_r1	5.1.2
55	Color Features	std_hist_r2	5.1.2
56	Color Features	std_hist_r3	5.1.2
57	Color Features	std_hist_r4	5.1.2
58	Color Features	std_hist_r5	5.1.2
59	Color Features	std_hist_r6	5.1.2
60	Color Features	std_hist_r7	5.1.2
61	Color Features	std_hist_g0	5.1.2
62	Color Features	std_hist_g1	5.1.2
63	Color Features	std_hist_g2	5.1.2
64	Color Features	std_hist_g3	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
65	Color Features	std_hist_g4	5.1.2
66	Color Features	std_hist_g5	5.1.2
67	Color Features	std_hist_g6	5.1.2
68	Color Features	std_hist_g7	5.1.2
69	Color Features	std_hist_b0	5.1.2
70	Color Features	std_hist_b1	5.1.2
71	Color Features	std_hist_b2	5.1.2
72	Color Features	std_hist_b3	5.1.2
73	Color Features	std_hist_b4	5.1.2
74	Color Features	std_hist_b5	5.1.2
75	Color Features	std_hist_b6	5.1.2
76	Color Features	std_hist_b7	5.1.2
77	Color Features	std_hist_v0	5.1.2
78	Color Features	std_hist_v1	5.1.2
79	Color Features	std_hist_v2	5.1.2
80	Color Features	std_hist_v3	5.1.2
81	Color Features	std_hist_v4	5.1.2
82	Color Features	std_hist_v5	5.1.2
83	Color Features	std_hist_v6	5.1.2
84	Color Features	std_hist_v7	5.1.2
85	Color Features	std_hist_rgb0	5.1.2
86	Color Features	std_hist_rgb1	5.1.2
87	Color Features	std_hist_rgb2	5.1.2
88	Color Features	std_hist_rgb3	5.1.2
89	Color Features	std_hist_rgb4	5.1.2
90	Color Features	std_hist_s0	5.1.2
91	Color Features	std_hist_s1	5.1.2
92	Color Features	std_hist_s2	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
93	Color Features	std_hist_s3	5.1.2
94	Color Features	std_hist_s4	5.1.2
95	Color Features	std_hist_s5	5.1.2
96	Color Features	std_hist_s6	5.1.2
97	Color Features	std_hist_s7	5.1.2
98	Flow Features	std_frame_value_diff	
99	Flow Features	std_frontal_faces_num	5.3
100	Flow Features	std_fronatl_faces_ratio	5.3
101	Flow Features	std_tilt_pan_confidences	
102	Flow Features	std_mag_mean	
103	Flow Features	std_mag_std	
104	Flow Features	std_shot_durations	
105	Color Features	stdmean_hist_r0	5.1.2
106	Color Features	stdmean_hist_r1	5.1.2
107	Color Features	stdmean_hist_r2	5.1.2
108	Color Features	stdmean_hist_r3	5.1.2
109	Color Features	stdmean_hist_r4	5.1.2
110	Color Features	stdmean_hist_r5	5.1.2
111	Color Features	stdmean_hist_r6	5.1.2
112	Color Features	stdmean_hist_r7	5.1.2
113	Color Features	stdmean_hist_g0	5.1.2
114	Color Features	stdmean_hist_g1	5.1.2
115	Color Features	stdmean_hist_g2	5.1.2
116	Color Features	stdmean_hist_g3	5.1.2
117	Color Features	stdmean_hist_g4	5.1.2
118	Color Features	stdmean_hist_g5	5.1.2
119	Color Features	stdmean_hist_g6	5.1.2
120	Color Features	stdmean_hist_g7	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
121	Color Features	stdmean_hist_b0	5.1.2
122	Color Features	stdmean_hist_b1	5.1.2
123	Color Features	stdmean_hist_b2	5.1.2
124	Color Features	stdmean_hist_b3	5.1.2
125	Color Features	stdmean_hist_b4	5.1.2
126	Color Features	stdmean_hist_b5	5.1.2
127	Color Features	stdmean_hist_b6	5.1.2
128	Color Features	stdmean_hist_b7	5.1.2
129	Color Features	stdmean_hist_v0	5.1.2
130	Color Features	stdmean_hist_v1	5.1.2
131	Color Features	stdmean_hist_v2	5.1.2
132	Color Features	stdmean_hist_v3	5.1.2
133	Color Features	stdmean_hist_v4	5.1.2
134	Color Features	stdmean_hist_v5	5.1.2
135	Color Features	stdmean_hist_v6	5.1.2
136	Color Features	stdmean_hist_v7	5.1.2
137	Color Features	stdmean_hist_rgb0	5.1.2
138	Color Features	stdmean_hist_rgb1	5.1.2
139	Color Features	stdmean_hist_rgb2	5.1.2
140	Color Features	stdmean_hist_rgb3	5.1.2
141	Color Features	stdmean_hist_rgb4	5.1.2
142	Color Features	stdmean_hist_s0	5.1.2
143	Color Features	stdmean_hist_s1	5.1.2
144	Color Features	stdmean_hist_s2	5.1.2
145	Color Features	stdmean_hist_s3	5.1.2
146	Color Features	stdmean_hist_s4	5.1.2
147	Color Features	stdmean_hist_s5	5.1.2
148	Color Features	stdmean_hist_s6	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
149	Color Features	stdmean_hist_s7	5.1.2
150	Flow Features	stdmean_frame_value_diff	5.2
151	Flow Features	stdmean_frontal_faces_num	5.3
152	Flow Features	stdmean_fronatl_faces_ratio	5.3
153	Flow Features	stdmean_tilt_pan_confidences	5.2
154	Flow Features	stdmean_mag_mean	5.2
155	Flow Features	stdmean_mag_std	5.2
156	Flow Features	stdmean_shot_durations	5.2
157	Color Features	mean10top_hist_r0	5.1.2
158	Color Features	mean10top_hist_r1	5.1.2
159	Color Features	mean10top_hist_r2	5.1.2
160	Color Features	mean10top_hist_r3	5.1.2
161	Color Features	mean10top_hist_r4	5.1.2
162	Color Features	mean10top_hist_r5	5.1.2
163	Color Features	mean10top_hist_r6	5.1.2
164	Color Features	mean10top_hist_r7	5.1.2
165	Color Features	mean10top_hist_g0	5.1.2
166	Color Features	mean10top_hist_g1	5.1.2
167	Color Features	mean10top_hist_g2	5.1.2
168	Color Features	mean10top_hist_g3	5.1.2
169	Color Features	mean10top_hist_g4	5.1.2
170	Color Features	mean10top_hist_g5	5.1.2
171	Color Features	mean10top_hist_g6	5.1.2
172	Color Features	mean10top_hist_g7	5.1.2
173	Color Features	mean10top_hist_b0	5.1.2
174	Color Features	mean10top_hist_b1	5.1.2
175	Color Features	mean10top_hist_b2	5.1.2
176	Color Features	mean10top_hist_b3	5.1.2

Feat. ID	Feat. Class	Feat. Name	Section
177	Color Features	mean10top_hist_b4	5.1.2
178	Color Features	mean10top_hist_b5	5.1.2
179	Color Features	mean10top_hist_b6	5.1.2
180	Color Features	mean10top_hist_b7	5.1.2
181	Color Features	mean10top_hist_v0	5.1.2
182	Color Features	mean10top_hist_v1	5.1.2
183	Color Features	mean10top_hist_v2	5.1.2
184	Color Features	mean10top_hist_v3	5.1.2
185	Color Features	mean10top_hist_v4	5.1.2
186	Color Features	mean10top_hist_v5	5.1.2
187	Color Features	mean10top_hist_v6	5.1.2
188	Color Features	mean10top_hist_v7	5.1.2
189	Color Features	mean10top_hist_rgb0	5.1.2
190	Color Features	mean10top_hist_rgb1	5.1.2
191	Color Features	mean10top_hist_rgb2	5.1.2
192	Color Features	mean10top_hist_rgb3	5.1.2
193	Color Features	mean10top_hist_rgb4	5.1.2
194	Color Features	mean10top_hist_s0	5.1.2
195	Color Features	mean10top_hist_s1	5.1.2
196	Color Features	mean10top_hist_s2	5.1.2
197	Color Features	mean10top_hist_s3	5.1.2
198	Color Features	mean10top_hist_s4	5.1.2
199	Color Features	mean10top_hist_s5	5.1.2
200	Color Features	mean10top_hist_s6	5.1.2
201	Color Features	mean10top_hist_s7	5.1.2
202	Flow Features	mean10top_frame_value_diff	
203	Flow Features	mean10top_frontal_faces_num	5.3
204	Flow Features	mean10top_fronatl_faces_ratio	5.3

Feat. ID	Feat. Class	Feat. Name	Section
205	Flow Features	mean10top_tilt_pan_confidences	5.2
206	Flow Features	mean10top_mag_mean	5.2
207	Flow Features	mean10top_mag_std	5.2
208	Flow Features	mean10top_shot_durations	5.2
209	Object Detection	person_freq	5.3
210	Object Detection	vehicle_freq	5.3
211	Object Detection	outdoor_freq	5.3
212	Object Detection	animal_freq	5.3
213	Object Detection	accessory_freq	5.3
214	Object Detection	sports_freq	5.3
215	Object Detection	kitchen_freq	5.3
216	Object Detection	food_freq	5.3
217	Object Detection	furniture_freq	5.3
218	Object Detection	electronic_freq	5.3
219	Object Detection	appliance_freq	5.3
220	Object Detection	indoor_freq	5.3
221	Object Detection	person_mean_confidence	5.3
222	Object Detection	vehicle_mean_confidence	5.3
223	Object Detection	outdoor_mean_confidence	5.3
224	Object Detection	animal_mean_confidence	5.3
225	Object Detection	accessory_mean_confidence	5.3
226	Object Detection	sports_mean_confidence	5.3
227	Object Detection	kitchen_mean_confidence	5.3
228	Object Detection	food_mean_confidence	5.3
229	Object Detection	furniture_mean_confidence	5.3
230	Object Detection	electronic_mean_confidence	5.3
231	Object Detection	appliance_mean_confidence	5.3
232	Object Detection	indoor_mean_confidence	5.3

Feat. ID	Feat. Class	Feat. Name	Section
233	Object Detection	person_mean_area_ratio	5.3
234	Object Detection	vehicle_mean_area_ratio	5.3
235	Object Detection	outdoor_mean_area_ratio	5.3
236	Object Detection	animal_mean_area_ratio	5.3
237	Object Detection	accessory_mean_area_ratio	5.3
238	Object Detection	sports_mean_area_ratio	5.3
239	Object Detection	kitchen_mean_area_ratio	5.3
240	Object Detection	food_mean_area_ratio	5.3
241	Object Detection	furniture_mean_area_ratio	5.3
242	Object Detection	electronic_mean_area_ratio	5.3
243	Object Detection	appliance_mean_area_ratio	5.3
244	Object Detection	indoor_mean_area_ratio	5.3

References

[1] Hyoung-Gook Kim, Nicolas Moreau, and Thomas Sikora. *MPEG-7 audio and beyond: Audio content indexing and retrieval*. John Wiley & Sons, 2006.

[2] Oge Marques. *Practical image and video processing using MATLAB*. John Wiley & Sons, 2011.

[3] Marcus Du Sautoy. *The Creativity Code: Art and Innovation in the Age of AI*. Harvard University Press, 2020.

[4] Tim Ingold. Beyond art and technology: the anthropology of skill. *Anthropological perspectives on technology*, pages 17–31, 2001.

[5] Ludwig Wittgenstein. Philosophical investigations, trans. *GEM Anscombe*, 261: 49, 1953.

[6] Stephen Davies. Defining art and artworlds. *The Journal of Aesthetics and Art Criticism*, 73(4):375–384, 2015.

[7] Jon McCormack. Art and the mirror of nature. *Digital Creativity*, 14(1):3–22, 2003.

[8] Margaret A Boden and Ernest A Edmonds. What is generative art? *Digital Creativity*, 20(1-2):21–46, 2009.

[9] Margaret A Boden. *AI: Its nature and future*. Oxford University Press, 2016.

[10] Ronald H Sadoff. The role of the music editor and the'temp track'as blueprint for the score, source music, and scource music of films. *Popular Music*, pages 165–183, 2006.

[11] Yi Yu, Zhijie Shen, and Roger Zimmermann. Automatic music soundtrack generation for outdoor videos from contextual sensor information. In *Proceedings of the 20th ACM international conference on Multimedia*, pages 1377–1378, 2012.

[12] Rajiv Ratn Shah, Yi Yu, and Roger Zimmermann. Advisor: Personalized video soundtrack recommendation by late fusion with heuristic rankings. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 607–616, 2014.

[13] Rajiv Ratn Shah, Yi Yu, and Roger Zimmermann. User preference-aware music video generation based on modeling scene moods. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 156–159, 2014.

[14] Jen-Chun Lin, Wen-Li Wei, James Yang, Hsin-Min Wang, and Hong-Yuan Mark Liao. Automatic music video generation based on simultaneous soundtrack recommendation and video editing. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 519–527, 2017.

[15] Tilman Herberger and Titus Tost. System and method of automatically creating an emotional controlled soundtrack, July 13 2010. US Patent 7,754,959.

[16] Jen-Chun Lin, Wen-Li Wei, and Hsin-Min Wang. Emv-matchmaker: emotional temporal course modeling and matching for automatic music video generation. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 899–902, 2015.

[17] Haruki Sato, Tatsunori Hirai, Tomoyasu Nakano, Masataka Goto, and Shigeo Morishima. A soundtrack generation system to synchronize the climax of a video clip with music. In *2016 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2016.

[18] Pedro Cano, Eloi Batlle, Ton Kalker, and Jaap Haitsma. A review of audio fingerprinting. *Journal of VLSI signal processing systems for signal, image and video technology*, 41(3):271–284, 2005.

[19] Will Drevo, Nov 2013. URL `https://willdrevo.com/fingerprinting-and-audio-recognition-with-python/`.

[20] Theodoros Giannakopoulos and Aggelos Pikrakis. *Introduction to Audio Analysis: a MATLAB® approach*. Academic Press, 2014.

[21] Nicolas Scaringella, Giorgio Zoia, and Daniel Mlynek. Automatic genre classification of music content: a survey. *IEEE Signal Processing Magazine*, 23(2): 133–141, 2006.

[22] Anders Meng, Peter Ahrendt, and Jan Larsen. Improving music genre classification by short time feature integration. In *Proceedings.(ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, volume 5, pages v–497. IEEE, 2005.

[23] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302, 2002.

[24] Eya Mezghani, Maha Charfeddine, Chokri Ben Amar, and Henri Nicolas. Multifeature speech/music discrimination based on mid-term level statistics and supervised classifiers. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–8. IEEE, 2016.

[25] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. Deep learning techniques for music generation–a survey. *arXiv preprint arXiv:1709.01620*, 2017.

[26] Ke Chen, Weilin Zhang, Shlomo Dubnov, Gus Xia, and Wei Li. The effect of explicit structure encoding of deep neural networks for symbolic music generation. In *2019 International Workshop on Multilayer Music Representation and Processing (MMRP)*, pages 77–84. IEEE, 2019.

[27] Gen-Fang Chen and Ya-Dong Wu. Segmentation of singing, speech and instruments in kunqu audio based on zero-crossing rate. In *2019 12th International Symposium on Computational Intelligence and Design (ISCID)*, volume 1, pages 270–273. IEEE, 2019.

[28] Feng Rong. Audio classification method based on machine learning. In *2016 International conference on intelligent transportation, big data & smart city (ICITBS)*, pages 81–84. IEEE, 2016.

[29] Anastasios Vafeiadis, Konstantinos Votis, Dimitrios Giakoumis, Dimitrios Tzovaras, Liming Chen, and Raouf Hamzaoui. Audio-based event recognition system for smart homes. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–8. IEEE, 2017.

[30] Theodoros Giannakopoulos and Georgios Siantikos. A ros framework for audio-based activity recognition. In *Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, pages 1–4, 2016.

[31] Noor Almaadeed, Muhammad Asim, Somaya Al-Maadeed, Ahmed Bouridane, and Azeddine Beghdadi. Automatic detection and classification of audio events for road surveillance applications. *Sensors*, 18(6):1858, 2018.

[32] Tina Raissi, Alessandro Tibo, and Paolo Bientinesi. Extended pipeline for content-based feature engineering in music genre recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2661–2665. IEEE, 2018.

[33] Chandanpreet Kaur and Ravi Kumar. Study and analysis of feature based automatic music genre classification using gaussian mixture model. In *2017 International Conference on Inventive Computing and Informatics (ICICI)*, pages 465–468. IEEE, 2017.

[34] Aisha Gemala Jondya and Bambang Heru Iswanto. Indonesian's traditional music clustering based on audio features. *Procedia computer science*, 116:174–181, 2017.

[35] Cj Carr and Zack Zukowski. Curating generative raw audio music with dome. In *IUI Workshops*, 2019.

[36] Chadawan Ittichaichareon, Siwat Suksri, and Thaweesak Yingthawornsuk. Speech recognition using mfcc. In *International Conference on Computer Graphics, Simulation and Modeling*, pages 135–138, 2012.

[37] Jesper Højvang Jensen, Mads Græsbøll Christensen, Manohar N Murthi, and Søren Holdt Jensen. Evaluation of mfcc estimation techniques for music similarity. In *2006 14th European Signal Processing Conference*, pages 1–5. IEEE, 2006.

[38] Arijit Ghosal, Rudrasis Chakraborty, Bibhas Chandra Dhara, and Sanjoy Kumar Saha. Music classification based on mfcc variants and amplitude variation pattern: a hierarchical approach. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 5(1):131–150, 2012.

[39] Bruce P Bogert. The quefrency alanysis of time series for echoes; cepstrum, pseudo-autocovariance, cross-cepstrum and saphe cracking. *Time series analysis*, pages 209–243, 1963.

[40] Dabbabi Karim, Cherif Adnen, and Hajji Salah. An optimization of audio classification and segmentation using gasom algorithm. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 9(4):143–157, 2018.

[41] Theodoros Giannakopoulos. pyaudioanalysis: An open-source python library for audio signal analysis. *PLOS ONE*, 10(12):1–17, 12 2015. doi: 10.1371/journal.pone.0144610. URL `https://doi.org/10.1371/journal.pone.0144610`.

[42] Peter M Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[43] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847*, 2017.

[44] Paul Fraisse. Rhythm and tempo. *The psychology of music*, 1:149–180, 1982.

[45] Official midi specifications, 1996. URL `https://www.midi.org/specifications/item/the-midi-1-0-specification`. Accessed: 2020-11-07.

[46] Allen Huang and Raymond Wu. Deep learning for music. *arXiv preprint arXiv:1606.04930*, 2016.

[47] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation. In *International Conference on Machine Learning*, pages 1362–1371. PMLR, 2017.

[48] Douglas Eck and Juergen Schmidhuber. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, 103:48, 2002.

[49] Huanru Henry Mao, Taylor Shin, and Garrison Cottrell. Deepj: Style-specific music generation. In *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, pages 377–382. IEEE, 2018.

[50] Chris Walshaw. About abc notation, 2009. URL `http://abcnotation.com/about`. Accessed: 2020-11-07.

[51] Sephora Madjiheurem, Lizhen Qu, and Christian Walder. Chord2vec: Learning musical chord embeddings. In *Proceedings of the constructive machine learning workshop at 30th conference on neural information processing systems (NIPS2016), Barcelona, Spain*, 2016.

[52] Michael Good. Musicxml for notation and analysis. *The virtual score: representation, retrieval, restoration*, 12:113–124, 2001.

[53] Michael Scott Cuthbert, Chris Ariza, Jose Cabal-Ugaz, Beth Hadley, and Neena Parikh. Hidden beyond midi's reach: Feature extraction and machine learning with rich symbolic formats in music21. In *Proc. of the NIPS 2011 Workshop on Music and Machine Learning*, 2011.

[54] Michael Scott Cuthbert, Christopher Ariza, and Lisa Friedland. Feature extraction and machine learning on symbolic music using the music21 toolkit. In *Ismir*, pages 387–392, 2011.

[55] Sergio Giraldo and Rafael Ramirez. A machine learning approach to ornamentation modeling and synthesis in jazz guitar. *Journal of Mathematics and Music*, 10(2):107–126, 2016.

[56] François Pachet, Jeff Suzda, and Dani Martinez. A comprehensive online database of machine-readable lead-sheets for jazz standards. In *ISMIR*, pages 275–280, 2013.

[57] Michael C Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 6(2-3):247–280, 1994.

[58] Christian Walder. Modelling symbolic music: Beyond the piano roll. In *Asian Conference on Machine Learning*, pages 174–189, 2016.

[59] Ian Simon and Sageev Oore. Performance rnn: Generating music with expressive timing and dynamics. *Magenta Blog: https://magenta. tensorflow. org/performancernn*, 2017.

[60] Cory McKay, Tristano Tenaglia, and Ichiro Fujinaga. jsymbolic2: Extracting features from symbolic music representations. In *Late-Breaking Demo Session of the 17th International Society for Music Information Retrieval Conference*, 2016.

[61] Alan C Bovik. *Handbook of image and video processing*. Academic press, 2010.

[62] Anshuman Agarwal, Shivam Gupta, and Dushyant Kumar Singh. Review of optical flow technique for moving object detection. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 409–413. IEEE, 2016.

[63] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.

[64] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[65] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[66] Konstantinos Bougiatiotis and Theodore Giannakopoulos. Multimodal content representation and similarity ranking of movies. *arXiv preprint arXiv:1702.04815*, 2017.

[67] Colin Raffel. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching.* PhD thesis, Columbia University, 2016.

[68] Jaap Haitsma and Ton Kalker. Speed-change resistant audio fingerprinting using auto-correlation. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, volume 4, pages IV–728. IEEE, 2003.

[69] Shumeet Baluja and Michele Covell. Waveprint: Efficient wavelet-based audio fingerprinting. *Pattern recognition*, 41(11):3467–3480, 2008.

[70] Shanshan Yao, Baoning Niu, and Jianquan Liu. Enhancing sampling and counting method for audio retrieval with time-stretch resistance. In *2018 IEEE Fourth International Conference on Multimedia Big Data (BigMM)*, pages 1–5. IEEE, 2018.

[71] Theodoros Giannakopoulos, Alexandros Makris, Dimitrios Kosmopoulos, Stavros Perantonis, and Sergios Theodoridis. Audio-visual fusion for detecting violent scenes in videos. In *Hellenic conference on artificial intelligence*, pages 91–100. Springer, 2010.

[72] Ba Tu Truong, Svetha Venkatesh, and Chitra Dorai. Scene extraction in motion pictures. *IEEE Transactions on Circuits and Systems for Video Technology*, 13 (1):5–15, 2003.