Faculty of Mathematics and Computer Sciences

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Master's Thesis

Music Similarity Analysis Using the Big Data Framework Spark

Submitted in partial fulfillment of the requirements for the degree

Master of Science (M.Sc.)
Computer Science

presented by:	Johannes Schoder
born:	3rd September 1994, Gera
ID:	169197
course of studies:	M.Sc. Computer Science
supervisors:	Prof. Dr. Martin Bücker, Ralf Seidler
date of approval:	17th May 2019
date of submission:	23rd September 2019

Zusammenfassung

Ein parametrisierbares Empfehlungssystem, basierend auf dem Big Data Framwork Spark, wird präsentiert. Dieses berücksichtigt verschiedene klangliche Eigenschaften der Musik und erstellt Musikempfehlungen basierend auf den persönlichen Vorlieben eines Nutzers. Das implementierte Empfehlungssystem ist voll skalierbar. Mehr Lieder können dem Datensatz hinzugefügt werden, mehr Rechner können in das Computercluster eingebunden werden und die Möglichkeit andere Audiofeatures und aktuellere Ähnlichkeitsmaße hizuzufügen und zu verwenden, ist ebenfalls gegeben. Des Weiteren behandelt die Arbeit die parallele Berechnung der benötigten Audiofeatures auf einem Computercluster. Die Features werden von dem auf Spark basierenden Empfehlungssystem verarbeitet und Empfehlungen für einen Datensatz, bestehend aus ca. 114000 Liedern, können unter Berücksichtigung von acht verschiedenen Arten von Audiofeatures und Abstandsmaßen innerhalb von zwölf Sekunden auf einem Computercluster mit 16 Knoten berechnet werden.

Abstract

A parameterizable recommender system based on the Big Data processing framework Spark is introduced, which takes multiple tonal properties of music into account and is capable of recommending music based on a user's personal preferences. The implemented system is fully scalable; more songs can be added to the dataset, the cluster size can be increased, and the possibility to add different kinds of audio features and more state-of-the-art similarity measurements is given. This thesis also deals with the extraction of the required audio features in parallel on a computer cluster. The extracted features are then processed by the Spark based recommender system, and song recommendations for a dataset consisting of approximately 114000 songs are retrieved in less than 12 seconds on a 16 node Spark cluster, while combining eight different audio feature types and similarity measurements.

Contents

Abbreviations

BH	Beat Histogram
BPM	Beats Per Minute
BRP	Bucketed Random Projection
DAG	Directed Acyclic Graph
DCT	Discrete Cosine Transformation
DF	Spark DataFrame
DFT	Discrete Fourier Transform
DTW	Dynamic Time Warping
ESA	Explicit Semantic Analysis
FFT	Fast Fourier Transformation
FMA	Free Music Archive
GMM	Gaussian Mixture Model
HDFS	Hadoop Distributed File System
HPCP	Harmonic Pitch Class Profiles
HT	Hyperthreading
JS divergence	Jensen Shannon divergence
JVM	Java Virtual Machine
KL divergence	Kullback-Leibler divergence
LSH	Locality-Sensitive Hashing
MFCC	Mel Frequency Cepstral Coefficients
MIDI	Musical Instrument Digital Interface
MIR	Music Information Retrieval
MP	Mutual Proximity
MSD	Million Song Dataset
RDD	Resilient Distributed Dataset
RH	Rhythm Histogram
RP	Rhythm Pattern
SKL	Symmetric Kullback-Leibler divergence
SQL	Structured Query Language
TF-IDF	Term Frequency - Inverse Document Frequency
UDF	User Defined Function
YARN	Yet Another Resource Negotiator

List of Figures

List of Tables

List of Code Snippets

1. Introduction

The idea originated from Dr. T. Bosse from the Chair for Advanced Computing at the Friedrich Schiller University in Jena. When proposing the idea for a master's thesis with the topic of "Music similarity measurement using genre-specific features" by using different guitar play styles in modern-day metal music, he jokingly said that he would also like to know how metal music compares to construction building noise. The idea is actually not so groundless, considering that most people would agree on the fact that metal music is often described as noise by people not used to listening to genres like death and black metal. While refining the original idea of the theme for this master's thesis and during the first tests, it became apparent, that while there is a lot of research in the area of music similarity for single aspects of music like melody, timbre, or rhythm and even for a few fixed combinations thereof, there was no attempt made yet, to build a parameterizable system combining various of these features in a Big Data environment. With music streaming services like Spotify, Amazon Music, Deezer or Tidal and music sharing websites like SoundCloud, access to millions of songs is given. To explore this humongous amount of data, the need for music recommender systems rises. SoundCloud Go+, the streaming service of SoundCloud alone gives access to more than 150 million songs [1]. Obviously, the streaming platforms are aware of these challenges. When using services like "[...] Spotify Radio, iTunes Radio, Google Play Access All Areas and Xbox Music. Recommendations are typically made using (undisclosed) content-based retrieval techniques, collaborative filtering data or a combination thereof." [2, p. 9]
But music similarity is not well defined. This is one of the first problems while dealing with this topic. It is a rather subjective value that can differ from listener to listener. Two tracks could be considered as "similar" when they are equal in tempo, loudness, melody, instrumentation, key, rhythm mood, lyrics, or a combination of more than a few of these features.

1.1 Objectives

The target of this thesis is to propose a transparent music similarity recommendation system based on various weighted aspects of the music instead of a fixed combination. Applying different weights to different features allows similarity retrieval methods to search for different kinds of similarities, empowering the user to decide which aspects are most important to the user and returning song recommendations based on the user's preferences. E.g., weighting the tempo and beat of a song more than melodic similarity allows the creation of playlists for workout and sport, while melodic/ timbre, etc. similarities allow searching for similar songs from musical subgenres.
The usage of a Big Data framework such as Spark allows the creation of a parameterized similarity definition. Various aspects of the music could easily be merged and taken into consideration when calculating the musical distance between two different pieces. This offers a more diverse music recommendation system than already existing ones.
To do this, a lot of different features are required and have to be extracted from the audio data first. Content (e.g., audio features) and context (e.g., listener behavior) data can then be fed into a Big Data framework to speed up operations. For this thesis, however, the focus lies on content-based data only.
Context-based collaborative-filtering techniques, which take the listening behavior of other users into consideration, in combination with Big Data frameworks are already well researched. But this thesis is meant to propose a user-centered recommendation engine, relying on musical properties of the songs only. By solely relying on the musical features of the songs, no biasing due to the popularity of artists is to be expected.

1.2 Outline

The thesis is structured into four main issues, pictured in Figure 1.1. These different problems are resolved throughout the chapters of the thesis.
First of all, a lot of music data is required. In Chapter 2, different scientific datasets and sources for audio files are evaluated. It also explains the basics of music information retrieval (MIR) and gives a short overview of different similarity measurements based on different audio features and aspects of the music. In the last section of this chapter an introduction to Big Data frameworks is given and the choice of Spark as the Big Data processing framework is explained.
In Chapter 3, multiple algorithms and approaches for the computation of similarity between timbral, melodic and rhythmic features are evaluated and selected.
Chapter 4 explains the implementation of the feature extraction process in parallel on a

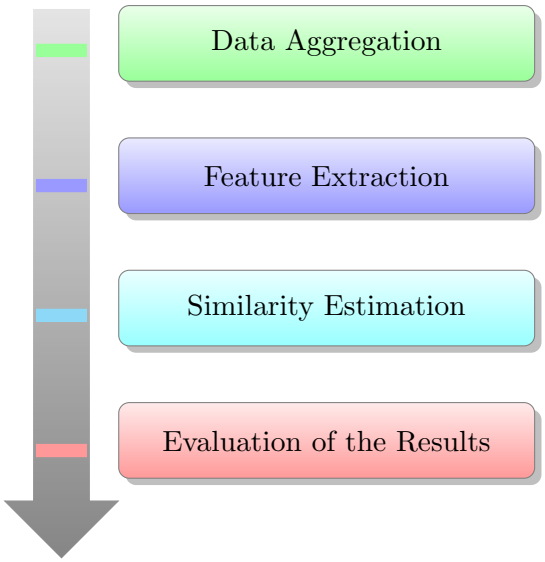


Figure 1.1: Structure of the thesis

cluster and the implementation of the recommender system with Spark.
In Chapter 5 the resulting song recommendations are proposed and evaluated, and lastly Chapter 6 summarizes all results and provides an outlook for possible enhancements.

2. Music Information Retrieval and Big Data

The field of music information retrieval is a large research area combining studies in computer science like signal processing and machine learning with psychology and academic music study. To get started, a brief overview is given in the next section providing the most important information about publicly available datasets, MIR toolkits, and different approaches to music similarity using various audio features. An overview over Big Data frameworks is included as well. More in-depth information about selected metrics is given in Chapter 3.

2.1 Terminology

To clarify the usage of a few terms throughout this thesis (especially later in Section 4.3), the following list provides an overview of the terms used.

- song request
- distance
- similarities

The term ”song request” describes the song title passed to the recommendation engine to estimate the similarities.
The terms ”similarities” and ”distances” are used synonymously in this thesis because all the similarity estimations are based on distances between feature vectors of different feature types (x and y), following the equation

$$\text{sim}(x, y) = \frac{1}{d(x, y)}. \tag{2.1}$$

The smaller the distance $d(x, y)$ between the audio features of two songs x and y is, the greater the similarity $\text{sim}(x, y)$ between these songs gets.

2.2 Audio Features

This section provides a short overview of commonly used audio features in MIR, including:

- Discrete Fourier Transform
- Mel Frequency Cepstral Coefficients
- Chroma features
- Pitch curve
- Onsets
- Beats

These audio features are the starting point for the later following calculation of the distances between songs.

2.2.1 Fourier Transformation

Most of the algorithms for audio data analysis start with switching from the time domain to the frequency domain by performing a discrete Fourier transform (DFT) as described in the equation

$$X_l = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2 \cdot \pi \cdot i}{N} \cdot l \cdot n}, \qquad l = 0, 1, ..., N-1 \tag{2.2}$$

and then computing the power spectrum

$$|X_l| = \sqrt{\mathrm{Re}(X_l)^2 + \mathrm{Im}(X_l)^2}, \qquad l = 0, 1, ..., N-1. \tag{2.3}$$

The value N resembles the frame/window size, x_n is the n^{th} input amplitude in the frame ranging from 0 to $N-1$, and l is an integer also ranging from 0 to $N-1$ (as many frequency values are computed per frame as discrete-time values are in the window). Sampling a song with length t in seconds by a sample rate f_s results in

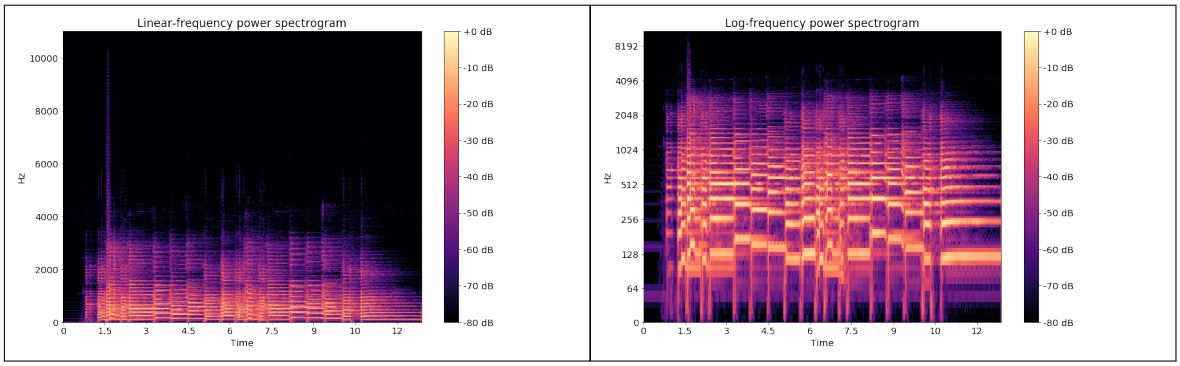
$$K = f_s \cdot t \tag{2.4}$$

data points x in an audio file. Considering a sample rate $f_s = 44,1kHz$ (usual CD sample-rate) and the length of a song of about $t = 180s$, the time domain contains $K = 7938000$ data points usually with 16-bit resolution for mono-channel audio, following Equation (2.4).
Calculating a DFT with a window size of $N = 1024$ samples and a hop size of 512 samples, the full resulting spectrogram would contain $N_{fv} = 11627$ frames with 1024

amplitude values per frame for a 3 minute example song sampled with 44.1kHz, according to [2, p. 56]:

$$N_{fv} = 1.5 \cdot (\frac{44100 \text{ samples/s}}{1024 \text{ samples/frame}}) \cdot t \tag{2.5}$$

The hop size determines how many discrete-time values are skipped between the computation of each DFT frame. In the example with a hop size of 512 and a window size of 1024 the various frames overlap by 50%, resulting in the factor 1.5 in Equation (2.5). As an example, figure 2.1(a) shows the resulting spectrogram (spectrum of frequencies over time) of the first bars of the song Layla by Eric Clapton recorded on an electric guitar.



(a) Spectrogram (b) Log-scaled spectrogram

Figure 2.1: Example spectrograms linear (a) and log-scaled (b)

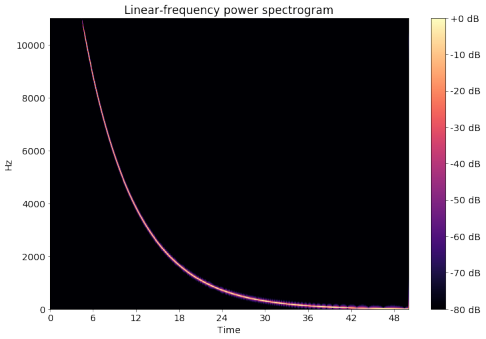
Since the human ear perceives sound in a non-linear fashion, a logarithmic (see Figure 2.1(b)) or mel scale is more suitable to represent different pitches. For example, the note A4 is perceived at a frequency of 440Hz, the A note of next octave (A5) is at 880Hz and the next one is at 1600Hz and so on. The mel scale was introduced to resemble the non-linear human perception of frequency [2, pp. 53f]. The conversion between a frequency f in Hz and m in mel is given by

$$m = 1127 \cdot \ln(1 + \frac{f}{700}). \tag{2.6}$$

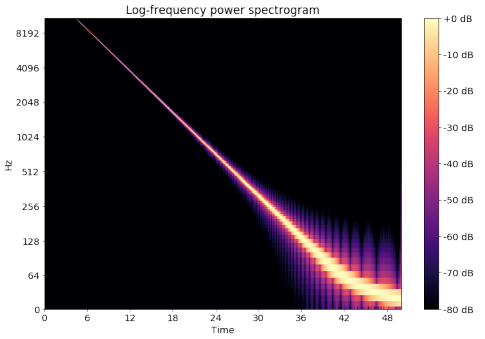
The high dimensionality of the spectrogram is a problem for machine learning applications and music similarity tasks, as computation based on vectors with such a high dimensionality on larger datasets would require excessive computational power, e.g., for real-time applications. To further reduce the dimensionality of the feature vector resulting from the DFT, a possible approach in MIR would be to calculate the so-called Mel Frequency Cepstral Coefficients (MFCCs) [2, pp. 55ff].

2.2.2 Mel Frequency Cepstral Coefficients

Of all features presented in this chapter, the MFCC is the hardest one to grasp because of its abstract nature and hardly visible relatedness to musical aspects of audio files like pitch or rhythm. This section gives a brief overview of the computation of the MFCC as stated in [2, pp. 55ff]. Figure 2.2 shows the magnitude spectrum of a logarithmic frequency sweep signal as an example for better understanding.



(a) Spectrogram with linear frequency axis



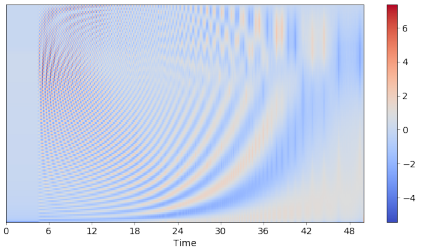
(b) Spectrogram with logarithmic frequency axis

Figure 2.2: Example spectrograms of a logarithmic frequency sweep

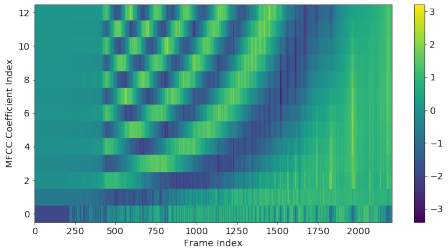
First of all the magnitude spectrum is transformed to the mel scale following Equation (2.6) by assigning each frequency value to a mel band. Doing this, dimensionality reduction can be achieved by assigning multiple frequency values to one of typically 12 to 40 mel bands. The resulting vectors are then fed into a discrete cosine transformation (DCT) resulting in the MFCCs for each frame:

$$X_k = \sum_{q=0}^{Q-1} x_q \cos\left[\frac{\pi}{Q}(q+\frac{1}{2})k\right], \qquad k = 0, 1, ..., Q-1 \tag{2.7}$$

where Q denotes the amount of mel bands.



(a) MFCC high resolution



(b) MFCC 13 bands scaled

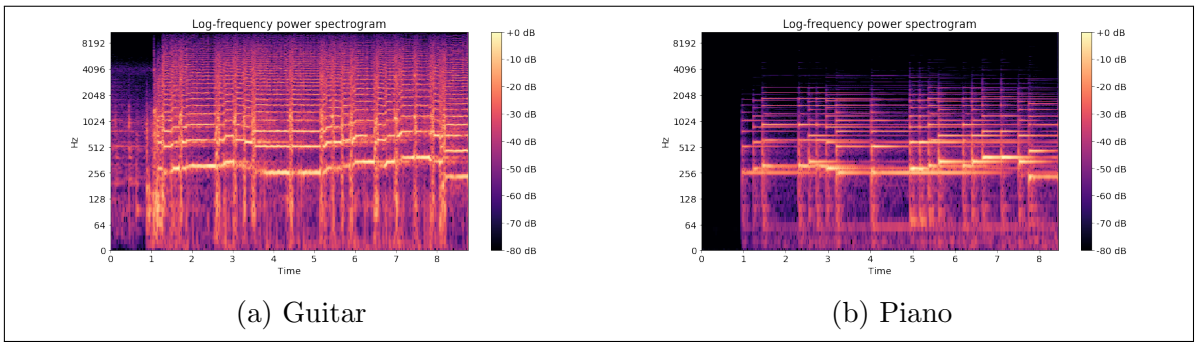
Figure 2.3: MFCCs of a logarithmic frequency sweep

Figure 2.3(a) shows the resulting MFCCs with a high resolution of 1024 mel bands.

This is not what would be done in a usual application, because this is nearly as high-dimensional as the original spectrogram. In comparison, Figure 2.3(b) shows the MFCC reduced to 13 mel bands. To better visualize the MFCCs, all values are typically scaled to have a standard deviation of 1 and a mean value of 0 per band in the plots. To describe a tone, three moments can be used according to [3, pp. 15f]:

- tonal intensity perceived as loudness
- tonal quality perceived as the pitch
- timbre or tonal color

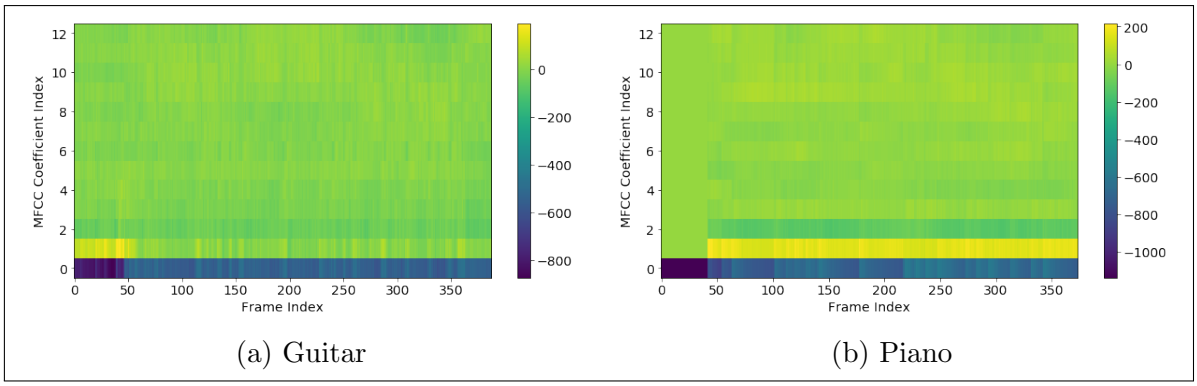
MFCCs were found to be suited to represent the timbral attributes of music [2, pp. 55 ff]. Looking at an example melody line played on an electric distorted guitar and a piano, distinct differences can be seen in Figure 2.4.



(a) Guitar (b) Piano

Figure 2.4: Spectrogram of a guitar (a) and piano (b) sample

Due to the physical properties of a string, every note played consists of the main frequency (the actually played note) and harmonic overtones because of the way a string, e.g. in a piano, vibrates and the wooden body resonates.

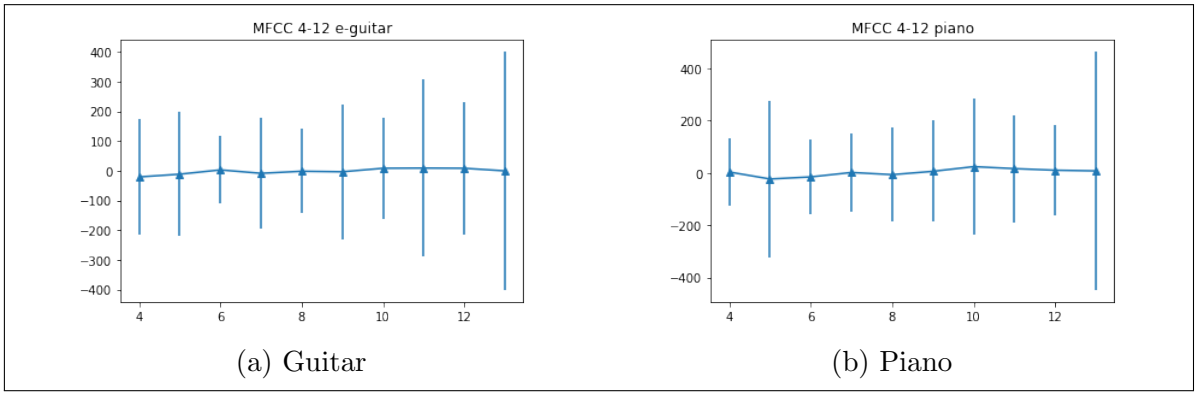


(a) Guitar (b) Piano

Figure 2.5: MFCCs of a guitar (a) and piano (b) sample

Typically the harmonics of a piano consist of the main key, the same key a few octaves higher and major thirds and fifths of the octave. Depending on the instrument, these harmonics decay faster or slower or do not appear at all. An electrically amplified guitar

amplifies these overtones as well, which is visible in Figure 2.4(b). These differences in timbre are also visible when looking at the MFCCs in Figure 2.5. This time the MFCC plots are pictured without the previously mentioned scaling. Additionally, the mean value and standard deviation of the MFCCs indexed from 4 to 13 are pictured in Figure 2.6. This calculation of statistical summaries of the MFCC features reduces the dimensionality of the MFCC features and is later explained in more detail in Section 3.1. Although both times the exact same melody is played in the same tempo, the MFCC features vary due to the different timbral properties of the instruments.

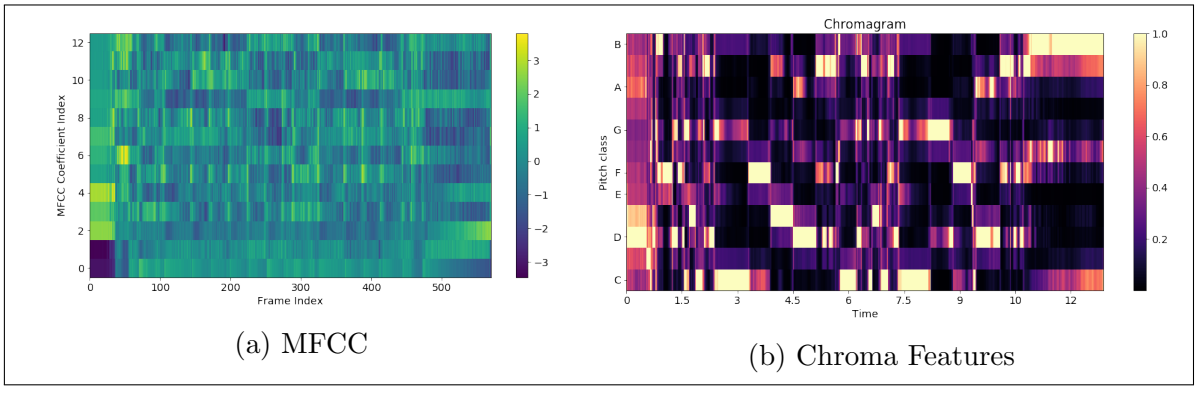


(a) Guitar (b) Piano

Figure 2.6: MFCCs mean and standard deviation of a guitar (a) and piano (b) sample

2.2.3 Other Audio Features

As another, better comprehensible, and higher-level set of features, the chromagram represents the melodic and harmonic properties of a song. The chroma plot shows the distribution of the different pitches mapped to the various semi-tones in one octave (see Figure 2.7(b)).



(a) MFCC (b) Chroma Features

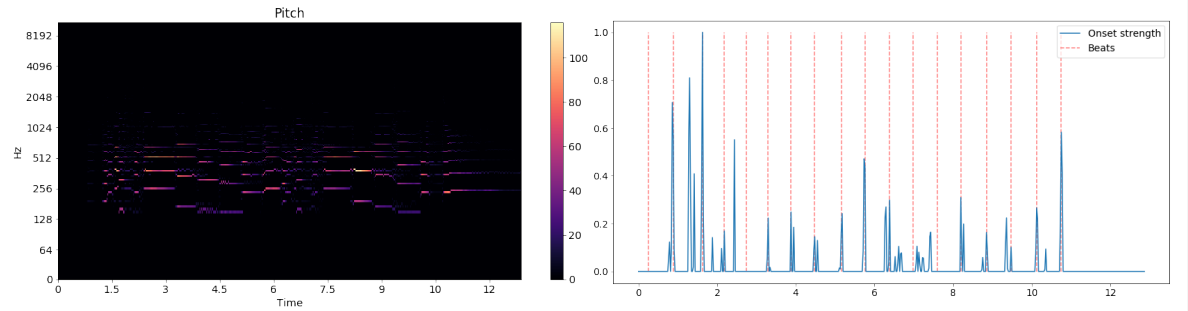
Figure 2.7: Melodic and timbral features of the song Layla by Eric Clapton

The mapping can be done with the help of binning strategies on the spectral representation or with special non-uniform filter banks [3, p. 153]. The chroma values of

each time frame are then normalized to one by the strongest dimension. So if all values are close to one, it is most likely that there will be only noise or silence at that frame in the recording, as depicted in the first few frames (0 to around 0.75 seconds) in Figure 2.7(b). The chromagram has one significant downside because it is reduced to one octave and thus can not represent the melody of a song to its full extent. The chromagram and the extraction of melody information from chroma features is further evaluated in Section 3.2.

Figure 2.8(a) shows the pitch curve of the recording. None but the most dominant frequencies are shown. Pitches below a certain threshold are filtered out. In contrast to the chromagram the pitch curve provides information over the whole spectrum and is not limited to one octave. These pitch curves can be used to estimate and transcribe musical notes from audio data as presented in Section 2.3.3.

The low-level rhythmic features of a song include the estimation of the overall tempo, beats, and onset events. The plot in Figure 2.8(b) depicts the onsets (blue) and estimated beats (red dotted lines) in the first few seconds from the guitar recording of the song Layla by Eric Clapton. The onsets resemble, e.g., detected note events and note changes. The onset detection is described in [3, pp. 412 ff] and most of the toolkits presented in the next section include methods for onset detection.



(a) Pitch

(b) Rhythm / beat

Figure 2.8: Rhythm features of the song Layla by Eric Clapton

2.3 MIR Toolkits

This section provides a short overview of available toolkits for MIR, note extraction, and similarity estimation between songs. Some of the toolkits are used in Chapter 4 for the extraction and pre-processing of the audio features.

2.3.1 Low-Level Audio Feature Extraction

To extract audio features like the ones presented in Section 2.2 (MFCCs, chromagram, beats, onsets) a wide variety of toolkits is publicly available and a few are presented in [4]. The YAAFE toolkit [5] is capable to extract a lot of different audio features like energy, MFCC, or loudness directly into the Hierarchical Data Format (*.h5) making it ideal for Big Data frameworks to use. It can be used with C++, Python [6], or MATLAB [7].
The Essentia toolkit [8] is fairly similar to YAAFE, extending it by the calculation of rhythm descriptors, bpm, etc. It can also be used with C++ and Python.
The Librosa Toolkit provides similar functionality [9] as Essentia. It is user-friendly, well-documented, and can be used from within a Jupyter-Notebook [10], allowing rapid prototyping and testing of different algorithms. Most of the plots in this chapter were created using librosa. Code snippets for the extraction of low-level features with Essentia and librosa are given in Section 4.2 as well as a performance analysis of both.

2.3.2 Music Similarity

The MIR Toolkit [11] is a toolbox for MATLAB. A port to GNU Octave [12] is also available [13]. The Code Snippet 2.1 is all it takes to compute a similarity matrix based on MFCC features, but the calculation is rather slow.

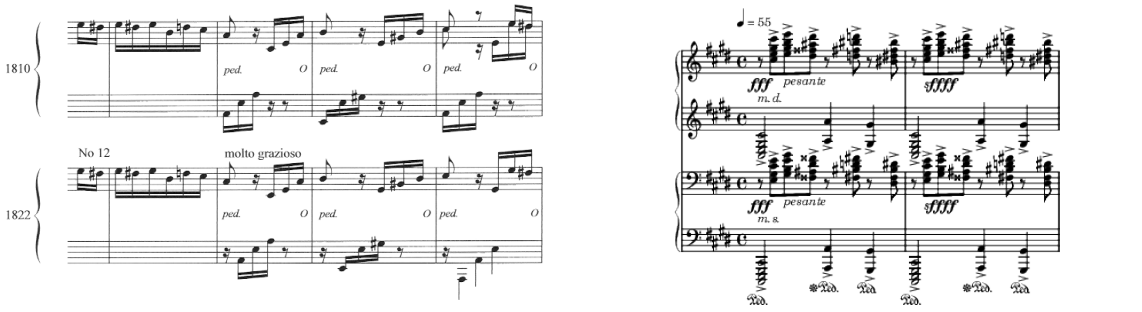
```
mydata = cell(1, numfiles);
for k = 1:numfiles
        myfilename = sprintf('%d.wav', k);
        mydata{k} = mirmfcc(myfilename);
        close all force
endfor
simmat = zeros(numfiles,numfiles);
for k = 1:numfiles
        for l = 1:numfiles
                simmat(k, l) = mirgetdata( ...
                mirdist(mydata{k}, ...
                mydata{l}));
        endfor
endfor
```

Code Snippet 2.1: MATLAB code for estimating similarities based on MFCCs

An other easy way to test state-of-the-art music similarity algorithms is to use the open-source toolkit Musly [14]. It is based on statistical models of MFCC features and calculates the similarities between songs very quickly, supporting OpenMP acceleration.

It automatically extracts the features it requires from the audio files. To compare the extracted features and calculate the distances, it implements the method introduced by Mandel-Ellis [15] and a timbre based improved version of the Mandel-Ellis algorithm using a Jensen-Shannon-like divergence [16]. More details and a re-implementation of some of the features from this toolkit are presented in Section 3.1.

2.3.3 Melody / Pitch Extraction

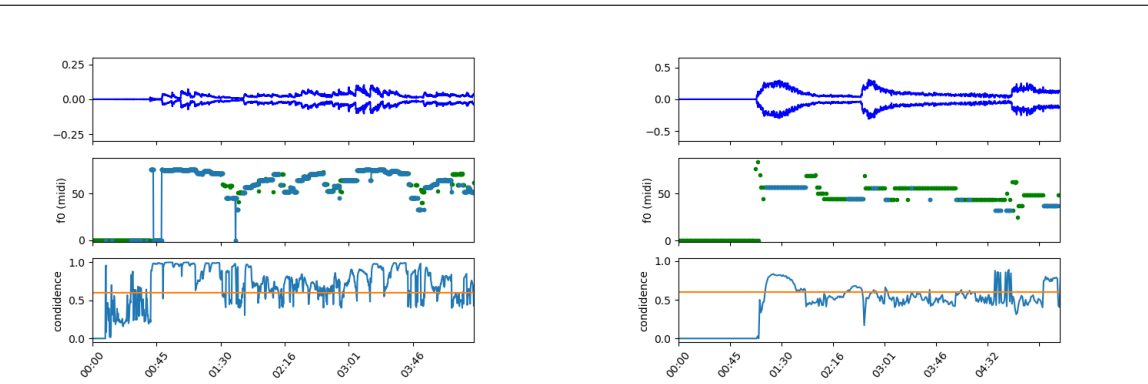


(a) Für Elise [17]

(b) Rachmaninoff prelude in C# minor [18]

Figure 2.9: Original scores, Rachmaninoff (a) and Beethoven (b)

To test the various pitch extraction toolkits, one piece by Rachmaninoff and one composed by Beethoven was used. Figure 2.9(a) shows the first five bars of Beethoven's Bagatelle in A Minor ("Für Elise"). Two bars of Rachmaninoff's Prelude in C# minor can be found in Figure 2.9(b).



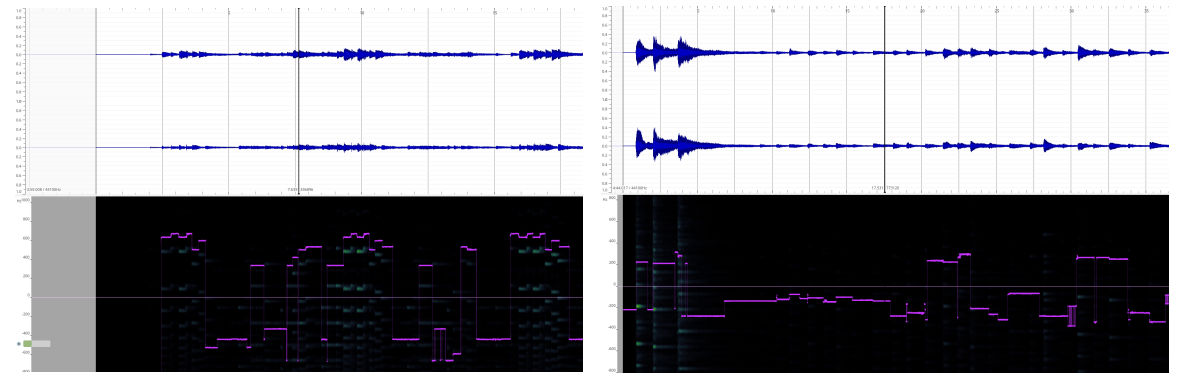
(a) Für Elise Aubio pitch

(b) Rachmaninoff prelude Aubio pitch

Figure 2.10: Pitch extraction with Aubio

The first toolkit tested is called "aubio" [19]. The result can be seen in Figure 2.10(a) and Figure 2.10(b). The upper subplot shows the waveform of the first few seconds of each piece. The second subplot figures the estimated pitch with green dots. If the

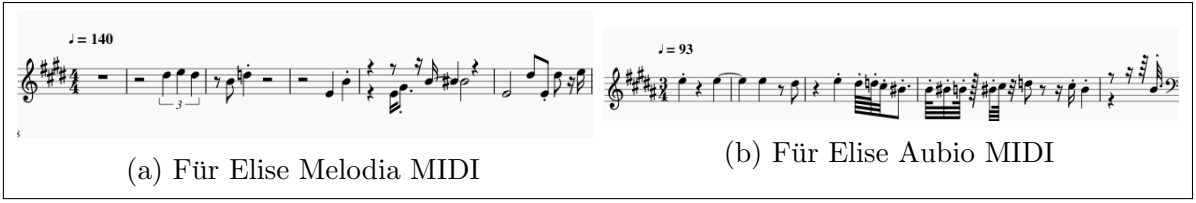
pitch is zero, then no pitch can be estimated, most likely because the associated frame contains silence. The blue dots resemble the estimated pitches where the confidence (shown as the blue graphs in the third subplot) is above a certain threshold (orange lines in the third subplots). The other melody extraction tool is called ”Melodia” [20], which is available as a VAMP plugin and can be used together with the ”Sonic Visualiser” [21]. The results are shown in Figure 2.11(a) and 2.11(b). The purple line resembles the estimated pitch; however, there are unwanted jumps between different octaves of the harmonics.



(a) Für Elise Melodia pitch (b) Rachmaninoff Melodia pitch

Figure 2.11: Pitch extraction with Melodia

Music related information, e.g, about note length, tempo, etc., would typically be stored digitally into standard MIDI (Musical Instrument Digital Interface) files [3, p. 180]. Unfortunately, the conversion from the extracted pitches to MIDI notes does not work flawlessly. It is apparent in Figure 2.12 that the transcription does not work accurately enough, even for a classical music piece with only one instrument. Figure 2.12(a) shows the output of a Python script using the Melodia VAMP plugin to calculate a MIDI file containing the main melody line, and Figure 2.12(b) shows the transcribed MIDI notes from Aubio. The detected melody lines are jumping between different octaves, and finding the right threshold for the separation between silence and detected notes turns out to be problematic as well.



(a) Für Elise Melodia MIDI (b) Für Elise Aubio MIDI

Figure 2.12: MIDI transcription Für Elise

2.4 Music Similarity Measurements

This section gives an introduction to the possibilities of the estimation of similarities based on the proposed audio features. Selected metrics and similarity measurements are selected and will be later evaluated in Chapter 3.

2.4.1 Timbre Based

The proposed approach by Dominik Schnitzer [22], creator of the Musly toolkit introduced earlier, is to take MFCCs as low-level features and then compute statistical features like mean, standard deviation, and covariances of the different MFCCs to reduce dimensionality before computing similarities. Another example for the computation of approximate nearest neighbors was published in the paper titled "Large-scale music similarity search with spatial trees" by Brian McFee and Gert Lanckriet [23].
A selection of different timbre based similarity measurements is evaluated later in Section 3.1.

2.4.2 Pitch Based

One proposed approach by Matija Marolt in 2006 is to take mid-level melodic representations of audio files like the chromagram instead of high-level features like sheet music or low-level features like Gaussian mixture models of MFCCs, to compute the similarity between songs [24]. A more detailed analysis of this topic is given in Section 3.2.

2.4.3 Note Based

For comparing musical pieces by their symbolic representation (notes, tablatures, etc.), different text retrieval methods can be used. MIDI files as a digital representation of notes are a good starting point. For example Xia (et. al) uses a variation of the Levenshtein distance measurement to compute similarities between MIDI files [25]. The problem with notation based algorithms is that there are not many datasets available containing audio and MIDI information. As shown in Section 2.3.3, the automatic transcription of notes from raw audio does not work flawlessly. There is still ongoing research to automatically annotate musical notes with the help of neural networks (for example [26]). In Section 3.3 an attempt to extract note information as text features from chromagrams and calculating the similarity by using the Levenshtein distance is shown and evaluated.

2.4.4 Rhythm Based

Rhythm based music similarity algorithms use timing information of various events as a baseline. For example low-level features like the onset and beat data from the plot in Figure 2.8(b) could be used as a starting point for rhythmic similarity retrieval. As an example, Foote (et. al) introduced a feature called the ”beat spectrum” for the computation of rhythmic similarities [27]. Other more recent or advanced approaches make use of the rhythm histogram, beat histogram, and rhythm patterns later evaluated in Section 3.3.

2.4.5 Metadata Based / Collaborative Filtering

Most of the research that combines the field of music information retrieval with Big Data frameworks relies on data based on the listening behavior of many users of, e.g., music streaming platforms. In 2012 the Million Song Dataset (MSD) Challenge was brought to the MIR community. Researchers were challenged to give a list of song recommendations based on a large set of user data, the Million Song Dataset (see Section 2.5.1 for more details on the dataset). As an example, if user X listens a lot to artist A and B and user Y listens mostly to artist A and C, then user X could probably like artist C as well. These kinds of collective listening behavior based recommendations are called ”collaborative filtering” and are pretty common in large music streaming services, although not necessarily representing direct musical similarity. [2, pp. 192f]
Recommendation systems based on collaborative filtering tend to propose commonly well-known artists rather than not so well-known ones, possibly biasing the resulting recommendation. On the other hand, these kinds of similarity algorithms can work very fast and efficient in a Big Data environment. The usage of annotations and metadata information like genre and artist based recommendations are common as well. The recommendation of songs based on lyrics and also hybrid recommendation systems that combine lyrics, metadata, and collaborative filtering are also possible.
However, all of these recommendation strategies are not directly based on musical features and are, therefore, not evaluated further throughout this thesis. But they are a possible addition for a hybrid recommendation engine for future research. An example using user-based collaborative filtering is the paper ”Design and Implementation of Music Recommendation System Based on Hadoop” [28].

2.4.6 Genre Specific Features

The impact of the choice of parameters for similarity measurements on different music subgenres was evaluated by Gulati (et al.) for Indian art music (Carnatic and Hindustani

music) [29]. They state: "We evaluate all possible combinations of the choices made at each step of the melodic similarity computation [...]. We consider 5 different sampling rates of the melody representation, 8 different normalization scenarios, 2 possibilities of uniform time-scaling and 7 variants of the distance measures. In total, we evaluate 560 different variants" [29, p. 3]. This evaluation showed that the choice of features and parameters for music similarity measurement is a critical point. "Sampling rates do not have a significant impact for Hindustani music, but can significantly degrade the performance for Carnatic music." [29, p. 3]. So using different kind of feature sets and parameters for the recommendation of songs from different genres could be an option but would go beyond the frame of this thesis.
As another idea, e.g., in Rock, Pop and Metal music, the analysis of different guitar playing techniques would be beneficial. Guitar tablature extraction [30] toolkits could be used to extract information, whether the guitar in a song is, for instance, mostly picked or strummed or if there are hammer-on, pull-off, side bending, or tapping techniques used. In classical music, the play style of the string section of an orchestra could be taken into consideration (staccato, pizzicato, etc.). These kinds of information could be used as a baseline for song recommendations. However, there is no MIR toolkit available for the estimation of play styles, so this idea would have to be evaluated in future research.

2.4.7 Selection

In this thesis, music similarity measurements based on three different types of features are evaluated. The first is based on MFCCs to represent timbral features of the songs and therefore offering a set of features to make recommendations that are similar in tone color and should be able to give recommendations inside the boundaries of different genres. The second is based on chroma features and note information to provide a measurement of melodic similarity. Utilizing these features targets the detection of cover versions. The third set of features is based on the rhythmic properties of a song. This should enable the recommendation of songs with the same tempo and rhythmic structure, and possibly also enable the recommendation of songs within the same genre. The usage of MIDI files is not considered further due to the rather poor performance of automatic score extraction tools for songs with multiple instruments and melody lines, and the lack of datasets containing MIDI and audio files. Also, the melodic component of the songs is already represented by the chroma features, although that limits the representation to one or at most very few octaves.
Collaborative filtering is left out because it does not necessarily represent the musical features and properties but instead the personal taste of other people. Additionally, it is left out because no fitting dataset with the required information and matching music

files was found (see Section 2.5).
Lastly, genre-specific features are also not an option because this field is not very well-researched yet, and the development of algorithms and the extraction of features would go beyond the scope of this thesis.

2.5 Data Aggregation

To evaluate the music similarity algorithms and metrics, a lot of music data is needed. This section provides an overview of publicly available sources for audio data. A selection from which the audio features are extracted is given in Section 4.2.1.

2.5.1 Datasets

Free Music Archive

The largest dataset is the Free Music Archive (FMA) consisting of 106733 different songs totaling an amount of nearly one terabyte of music data from a variety of different music genres [31] (see Figure 2.13(a)). There is also a lot of metadata like genre tags available for most of the songs.

Private Music Collection

The private music collection used in this work consists mainly of metal music. The music was legally purchased; all rights belong to the respective owners. Therefore this dataset can not be published alongside this thesis. But the private music collection is fully cataloged, and the according PDF file is in the appendices. The distribution of different songs per genre for this dataset is listed in Figure 2.13(b).
Additionally, a private recording dataset was used, consisting of ambient recordings and self-produced music. Most of these files are available on SoundCloud [32].
Because music recommendations are always related to personal taste and the perception of the quality of the results may differ, the inclusion of the private music collection is necessary to enable a subjective evaluation of the results from the developed recommendation engine.

1517-Artists and Musicnet

Other sources of music are the Musicnet dataset [33] and the 1517-Artists dataset [34]. The Musicnet dataset includes 330 pieces of classical music with musical note values and positions as annotations and the 1517-Artists dataset contains 3180 songs of multiple genres (see Figure 2.13(c)).

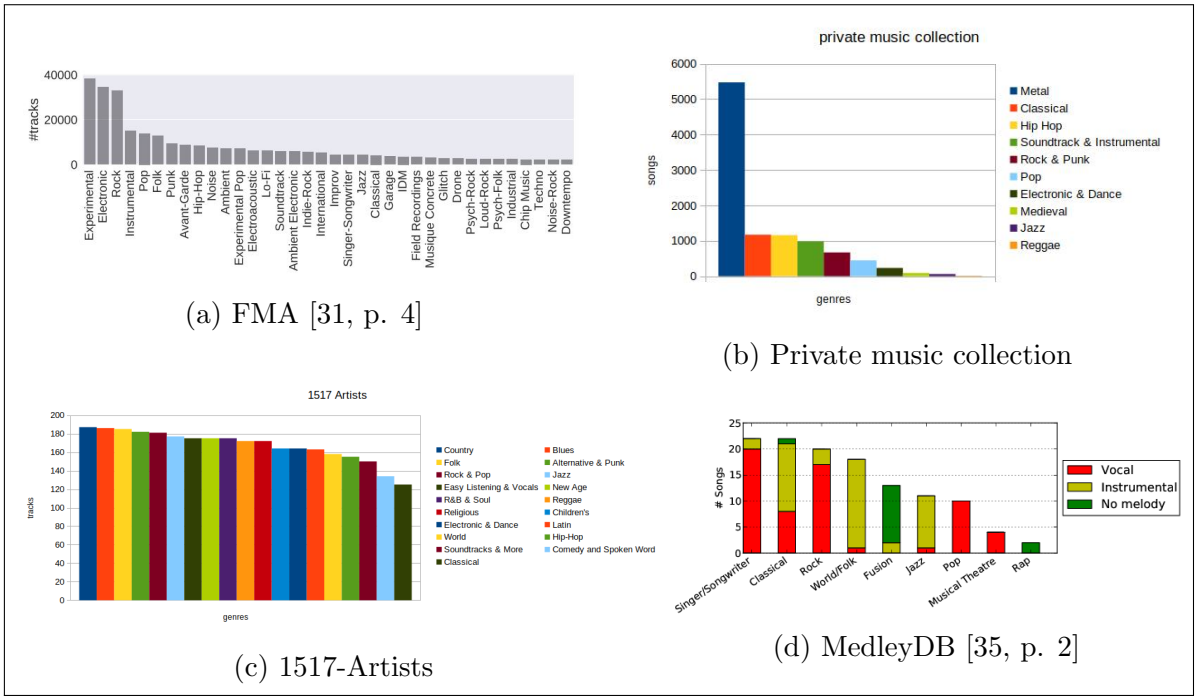
MedleyDB

For a melody or pitch based similarity analysis, multitrack datasets could provide useful data, because the pitch estimation can be done instrument by instrument. There are, e.g., the MedleyDB [35] (see Figure 2.13(d)) and MedleyDB2 [36] datasets, as well as the Open Multitrack Dataset [37] currently consisting of 593 multitracks in which the MedleyDB dataset is already included, leaving 481 other tracks for analysis.

Covers80

For cover song detection analysis, the covers80 dataset is available [38] containing eighty original songs predominantly from the musical genres rock and pop and 84 cover versions. These cover versions tend to differ significantly from the original in musical style, rhythm, and timbre.

Overview and Other Sources



(a) FMA [31, p. 4]

(b) Private music collection

(c) 1517-Artists

(d) MedleyDB [35, p. 2]

Figure 2.13: Genre distribution of songs in various datasets

The music sources and amounts of songs used for the task at hand are listed in Table 2.1.

dataset	#songs	features
FMA	106.733	-
private	8.484	-
1517-Artists	3.180	-
Maestro	1.184	MIDI (piano sheet music)
musicnet	330	note annotation
Open Multitrack Testbed	593(481)	multitracks
covers80	164	80 originals + 84 covers
MedleyDB	122	multitracks
MedleyDB2	74	multitracks

Table 2.1: Number of songs in different music datasets

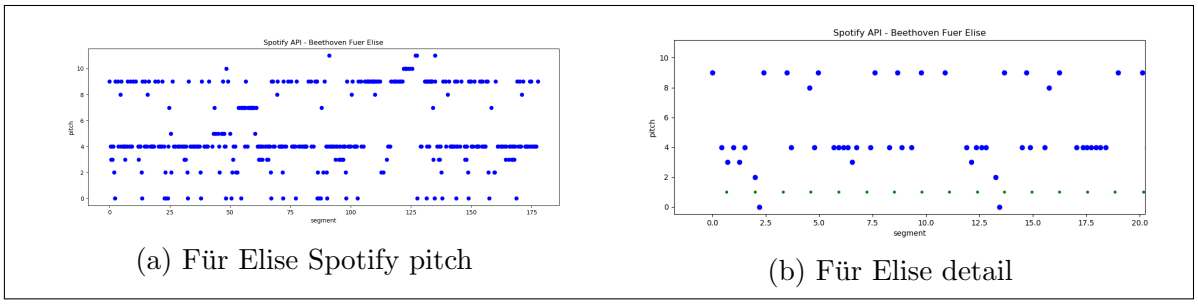
2.5.2 Alternatives

Spotify API

Another way of getting music samples, audio features, and metadata could be by using the Spotify API [39]. The downside of using the Spotify API is that no packed and ready to use test dataset containing the relevant features is available. Therefore, for scientific purposes, a test dataset would have to be created first. Using a small Python library named Spotipy [40], the available information can be accessed very easily.
Appendix 7.2 lists a small script, that is able to download all audio features and analysis data from selected songs in a playlist that contain a preview URL to a 30-second audio snippet. The audio features and analysis data is saved as a JSON file containing information about:

- acousticness
- danceability
- instrumentalness
- liveness
- loudness
- speechiness
- valence
- predicted key
- tempo
- pitch
- tempo
- timbre information
- beats and bars

In Figure 2.14(a) the returned chroma features (using the script in Appendix 7.2) of the piano piece ”Für Elise” by Beethoven are shown and Figure 2.14(b) shows the beginning of the piece in more detail, including green dots that resemble estimated bar markings. The blue dots represent the note values of one octave. That means they can resemble a value between zero and eleven with 0 representing the key C and 11 representing a B. The Spotify API actually returns a chroma feature value for every single one of the semi-tones per segment, with one segment being a section of samples that are relatively uniform in timbre and harmony. But in the plots, only the most dominant key per segment is shown to visualize the main melody line.



(a) Für Elise Spotify pitch

(b) Für Elise detail

Figure 2.14: Extracted pitches, Spotify API (Spotipy)

Together with the 30-second audio sample from which more features like MFCCs could be extracted, Spotipy could provide all the information needed to build a large dataset for MIR. However, the terms and conditions explicitly prohibit crawling the Spotify service. As stated by the Spotify ”Terms and Conditions of Use”, section 9 (User guidelines):
”The following is not permitted for any reason whatsoever:
[...]
12. “crawling” the Spotify Service or otherwise using any automated means (including bots, scrapers, and spiders) to view, access, or collect information from Spotify or the Spotify Service” [41]
Therefore a larger dataset based on the Spotify API can not be created without the risk of legal infringements. One could argue that there was a difference between data mining and data crawling and for small datasets these restrictions may not apply. Spotify states that by creating an algorithmically generated playlist similar to the ”Discover Weekly” playlists one may encounter legal problems if using such features commercially [42]. However it does not prohibit the usage for non-commercial cases.
Upon an initial request, the Spotify API developer team did not respond and therefore in this thesis the Spotify API will not be used to create a test dataset. Without further reaching out to Spotify, using the Spotify API to create a test dataset is not an option.

Million Song Dataset

Another outstanding and very large dataset is the Million Song Dataset (MSD) [43]. It contains a large set of metadata per track as well as a lot of supplementary datasets, like the tagtraum genre annotation (Figure 2.15) [44] and the Last.fm dataset [45]. In addition to that, the Echo Nest API dataset contains a lot of additional audio features like pitch, loudness, energy, and danceability to name just a few [46]. Another addition is the SecondHandSongs Dataset [47], containing a list of cover songs in the Million Song Dataset.

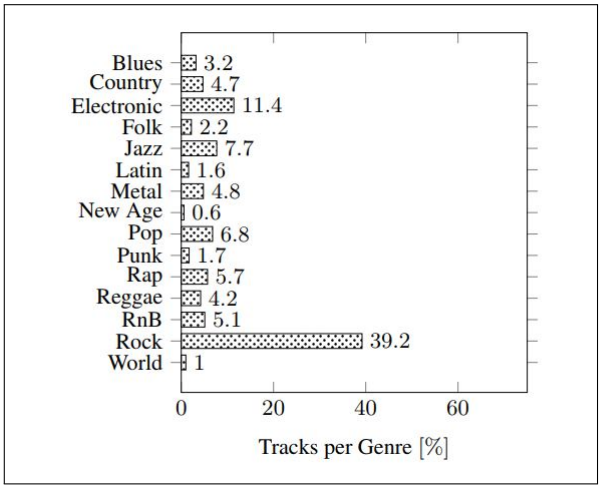


Figure 2.15: Million Song Dataset genre distribution [44, p. 6]

Due to the fact that the Spotify API [39] also works with audio features from the Echo Nest [48], the MSD could be used in a Big Data environment to simulate the work with Spotify data, without the need of mining the actual data. The MSD was already used with Big Data frameworks for music similarity retrieval based on metadata and user information (see [23]). Although the MSD does not contain any audio files in the first place, 30-second samples could be gathered through simple scripts from 7digital.com when the dataset was made publicly available. Unfortunately, 7digital does not offer the download of the 30-second sample files any longer, which makes this dataset impractical for this thesis, because missing audio features like MFCCs can not be computed from the audio files itself.

2.6 Big Data

After evaluating different data sources presenting various methods to extract and process different audio features, the following section describes the data analysis with Big Data processing frameworks like Apache Spark [49] and Hadoop [50]. Most of the basic information on Hadoop and Spark in the next few sections are taken from the

book ”Data Analytics with Spark using Python” by Jeffrey Aven, which gives a very comprehensible and practical introduction to the field of Big Data processing with PySpark [51].

2.6.1 Hadoop

With the ever-growing availability of huge amounts of high-dimensional data, the need for toolkits and efficient algorithms to handle these grew over the past years. One key to handle Big Data problems is the use of parallelism.
Search engine providers like Google and Yahoo firstly ran into the problem of using ”internet-scale” data in the early 2000s when faced with the problem of storing and processing the ever-growing amount of indexes from documents on the internet. In 2003, Google presented their white paper called ”The Google File System” [52]. MapReduce is a programming paradigm introduced by Google as an answer to the problem of internet-scale data and dates back to 2004 when the paper ”MapReduce: Simplified Data Processing on Large Clusters” was published [53].
Doug Cutting and Mike Cafarella worked on a web crawler project called ”Nutch” during that time. Inspired by the two papers Cutting incorporated the storage and processing principles from Google, leading to what we know as Hadoop today. Hadoop joined the Apache Software Foundation in 2006. The MapReduce programming paradigm for data processing is the core concept used by Hadoop. [51, p. 6]
Hadoop is a scalable solution capable of running on large computer clusters. It does not necessarily require a supercomputing environment and is able to run on clusters of lower-cost commodity hardware. The data is stored redundantly on multiple nodes with a configurable replication factor defining how many copies of each data chunk are stored redundantly on other nodes. This enables an error management where faulty operations can simply be restarted.
Hadoop is based on the idea of data locality. In contrast to the usual approach, where the data is requested from its location and transferred to a remote processing system or host, Hadoop brings the computation to the data instead. This minimizes the problem of data transfer times over the network at compute time when working with very large-scale data / Big Data. One prerequisite is that the operations on the data are independent of each other. Hadoop follows this approach called ”shared nothing”, where data is processed locally in parallel on many nodes at the same time by splitting the data into independent, small subsets without the need for communication with other nodes. Additionally, Hadoop is a schemaless (schema-on-read) system which means that it is able to store and process unstructured, semi-structured (JSON, XML), or well structured data (relational database). [51, p. 7]
To make all this possible, Hadoop relies on its core components YARN (Yet Another

Resource Negotiator) as the processing and resource scheduling subsystem and the Hadoop Distributed File System (HDFS) as Hadoop's data storage subsystem.

MapReduce

Figure 2.16 shows the basic scheme of a MapReduce program.

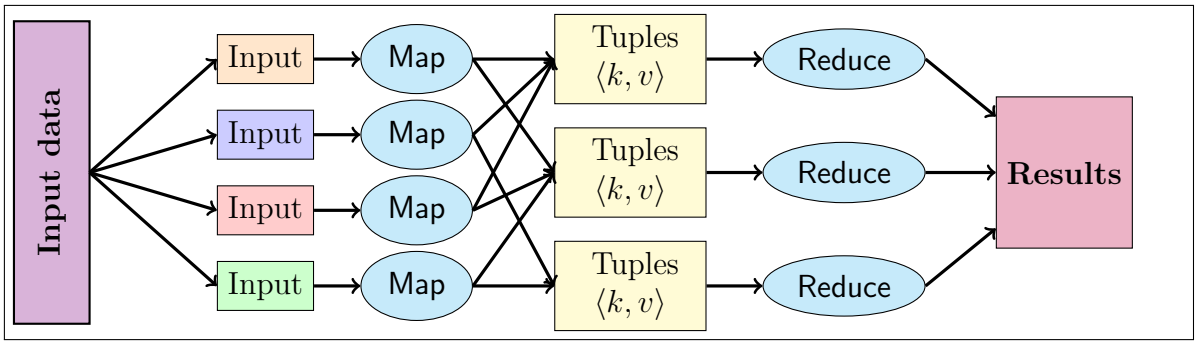


Figure 2.16: MapReduce algorithm [54]

In the first stage, the input data is split into chunks and distributed over the nodes of a cluster. This is usually managed by a distributed file system like the HDFS. One master node stores the addresses of all data chunks.
The data is then fed into the mappers which operate on the input data and finally transforms the input into key-value tuples.
In an intermediate step the key-value pairs are usually grouped by their keys before being fed into the reducers. The reducers apply another method to all tuples with the same key.
The amount of key-value pairs at the output from all mappers divided by the number of input files is called "replication rate" (r). The highest count of values for one key being fed into a reducer can be denoted as q (reducer size). Usually, there is a trade-off between a high replication rate r and small reducer size q (highly parallel with more network traffic) or small r and larger q (less network traffic but worse parallelism due to an overall smaller reducer count).

2.6.2 Spark

Hadoop as a Big Data processing framework has a few downsides compared to other, newer options like Spark. The Spark project was started in 2009 and was created as a part of the Mesos research project. It was developed as an alternative to the implementation of MapReduce in Hadoop. Spark is written in the programming language Scala [55] and runs in Java Virtual Machines (JVM) but also provides native support

for programming interfaces in Python, Java and R. One major advantage compared to Hadoop is the efficient way of caching intermediate data to the main memory instead of writing it onto the hard drive. While Hadoop has to read all data from the disk and writes all results back to the disk, Spark can efficiently take advantage of the RAM available in the different nodes, making it suitable for interactive queries and iterative machine learning operations. To be able to offer these kinds of in-memory operations Spark uses a structure called "Resilient Distributed Dataset" (RDD). [51, p. 13]
Figure 2.17 shows the simplified architecture of a compute cluster running Spark.

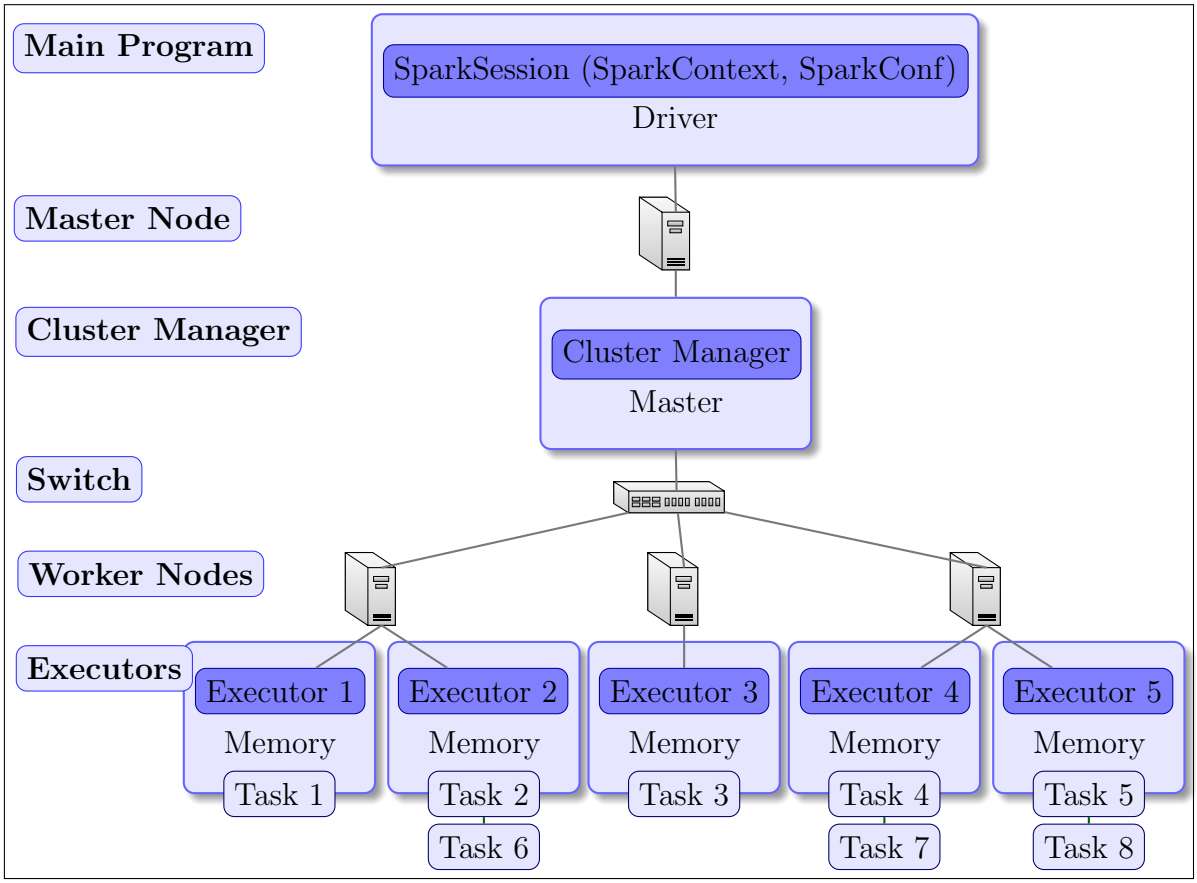
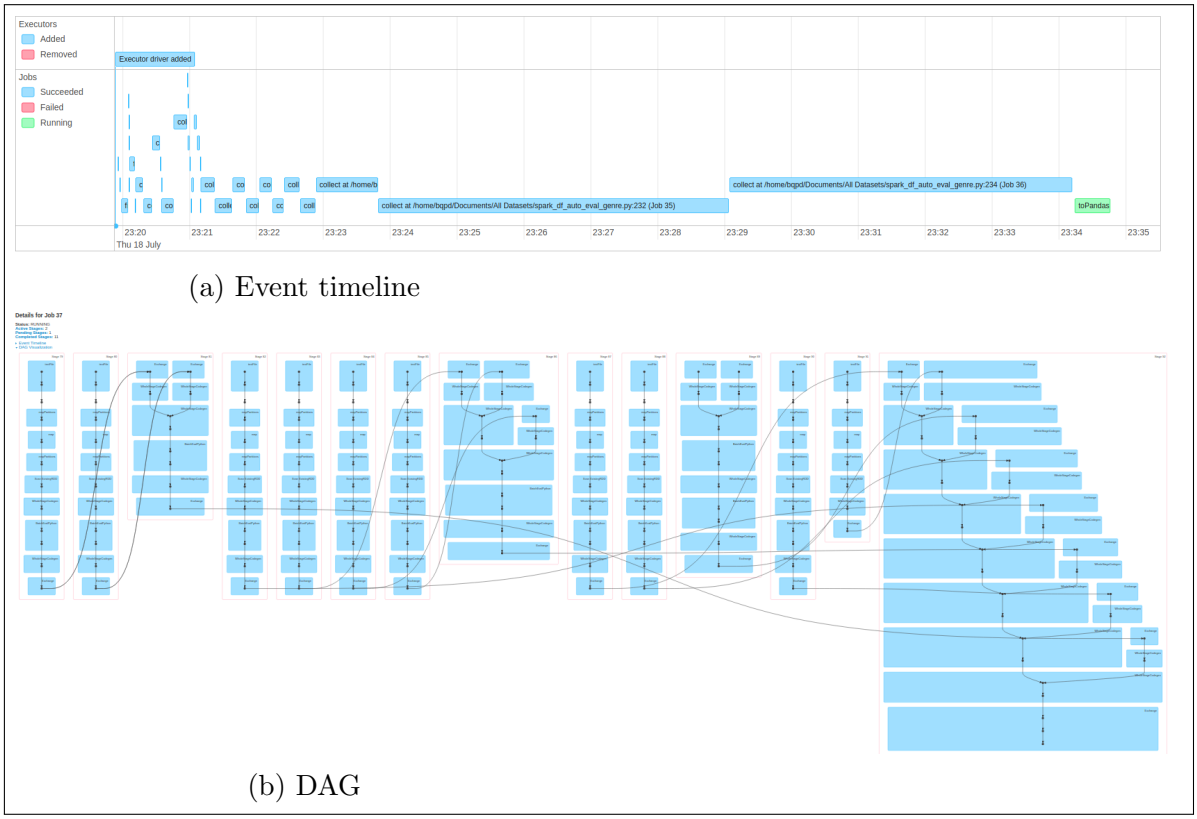


Figure 2.17: Spark cluster scheme (according to [51, p. 46])

The core components of a Spark application are the Driver, the Master, the Cluster Manager, and the Executors. The Driver is the process to which clients submit their applications. It is responsible for the planning and execution of a Spark program and returns status logs and results to the clients. It can be located on a remote client or on a node in the cluster. The SparkSession is created by the Driver and represents a connection to a Spark cluster. The SparkContext and SparkConf as child objects of the SparkSession contain the necessary information to configure the cluster parameters, e.g., the number of CPU cores and memory assigned to the Executors and the number of Executors that get spawned overall on the cluster. Up until version 2.0, entry points for

Spark applications included the SparkContext, SQLContext, HiveContext, and StreamingContext. In more recent versions these were combined into one SparkSession object providing a single entry point. The execution of the Spark application is scheduled, and directed acyclic graphs (DAG) are created by the Spark Driver. The nodes of these DAGs represent transformational or computational steps on the data. These DAGs can be visualized using the Spark application UI typically running on port 4040 of the Driver node. The Spark application UI is a useful tool to improve the performance of Spark applications and for debugging, as it also gives information about the computation time of the distinct tasks within a Spark program. [51, pp. 45ff]



(a) Event timeline

(b) DAG

Figure 2.18: Spark application UI examples taken from the recommender system

Two examples of information provided by the Spark application UI are shown in Figure 2.18, with Figure 2.18(a) showing the event timeline for a poorly optimized code snippet, where a single collect operation takes multiple minutes. Figure 2.18(b) gives an example of the heavily optimized DAG returning the recommendations from one song request into a *.csv file.

The Workers are the nodes in the cluster on which the actual computation of the Spark DAG tasks takes place. As defined within the SparkConf, the Worker nodes spawn a finite or fixed number of Executors that reserve CPU and memory resources and run in parallel. The Executors are hosted in JVMs on the Workers. Finally, the Spark

Master and the Cluster Manager are the processes that monitor, reserve and allocate the resources for the Executors. Spark can work on top of various Cluster Managers like Apache Mesos, Hadoop, YARN, and Kubernetes. Spark can also work in standalone mode, where the Spark Master also takes control of the Cluster Managers' tasks. If Spark is running on top of a Hadoop cluster, it uses the YARN ResourceManager as the Cluster Manager, and the ApplicationMaster as the Spark Master. The ApplicationMaster is the first task allocated by the ResourceManager and negotiates the resources (containers) for the Executors and makes them available to the Driver. [51, pp. 49 ff] When running on top of a Hadoop installation, Spark can additionally take advantage of the HDFS by reading data directly out of it.

Cluster Configuration and Execution

There are multiple options of passing a Spark programm to the cluster. The first one is to use a spark shell e.g. by calling `pyspark` when working with the Spark Python API or `spark-shell` for use with Scala. If the interactive option of using a spark shell is chosen, a SparkSession is automatically created and exited once the spark shell gets closed. Alternatively the Spark application can be passed to the cluster directly, using `spark-submit application.py -options` (Python). As mentioned previously, the configuration of the Spark cluster can be changed. This can either be done by using a cluster configuration file (e.g. spark-defaults.conf), by submitting the parameters as arguments passed to pyspark, spark-console or spark-submit, or by directly setting the configuration properties inside the Spark application code (see Code Snippet 2.2)

```
1 confCluster = SparkConf().setAppName("MusicSimilarity Cluster")
2 confCluster.set("spark.executor.memory", "1g")
3 confCluster.set("spark.executor.cores", "1")
4 sc = SparkContext(conf=confCluster)
5 sqlContext = SQLContext(sc)
6 spark = SparkSession.builder.master("cluster").appName("MusicSimilarity").↩
      getOrCreate()
```

Code Snippet 2.2: Example cluster configuration Python

In the code snippet, each Executor gets 1GB of RAM and 1 CPU core assigned by setting the according parameters in the `confCluster` object. The SparkContext is saved into the object `sc` and `sqlContext` contains the SQLContext object.

Spark Advantages

For this thesis, the programming language of choice is Python. With its high-level Python API, Spark applications can access commonly known and widely used Python libraries such as Numpy or Scipy. It also contains its own powerful libraries like the Spark ML library for machine learning applications or GraphX for the work with large graphs.
Spark can be used in combination with SQL (e.g., the Hive project) and NoSQL Systems like Cassandra and HBase. Spark SQL enables the transformation of RDDs to well structured DataFrames. The DataFrame concept is later used in Section 4.3.
One other important concept Spark uses is its lazy evaluation or lazy execution. Spark differentiates between data transformations (e.g. `filter()`, `join()`, and `map()`) and actions (e.g. `take()` or `count()`). The actual processing and transformation of data is deferred until an action is called.

```
1 chroma = sc.textFile("features.txt").repartition(repartition_count)
2 chroma = chroma.map(lambda x: x.split(';'))
3 chroma = chroma.filter(lambda x: x[0] == "OrbitCulture_SunOfAll.mp3")
4 chroma = chroma.count()
```

Code Snippet 2.3: Lazy evaluation

In the example Code Snippet 2.3 a text file `"features.txt"` gets read into an RDD `chroma` and repartitioned into `repartition_count` blocks. The `map()` transformation splitting the feature vectors and the `filter()` transformations that searches for a specific file ID are only executed once the `count()` action is called. Only then a DAG is created together with logical and physical execution plans and the tasks are distributed across the Executors. The lazy evaluation allows Spark to combine as many operations as possible which may lead to a drastic reduction of processing stages and data shuffling (data transferred between Executors) and thus reducing unnecessary overhead and network traffic. But the lazy execution has to be kept in mind during debugging and performance testing. [51, p.73] Another important part of Spark is its ability to process streaming data. While Hadoop is good at batch processing very large datasets but rather slow when it comes to iterative tasks on the same data due to its persistent write operations to the hard drive, Spark outperforms Hadoop with its capability to use RDDs and the main memory during iterative tasks. With Spark streaming the possibility to process data streams, e.g., from social networks, in real-time is given.
The combination of batch- and stream-processing methods is called ”Lambda architecture” in data science literatur. It describes a data-processing architecture consisting of a Batch-Layer, a Speed-Layer for real-time processing and a Serving-Layer managing

the data [56, pp. 8f]. Spark offers the possibility to take care of both, batch- and stream-processing jobs. Combined with other frameworks like the Apache SMACK stack (Spark, Mesos, Akka, Cassandra, and Kafka), Spark offers plenty possibilities for high-throughput Big Data processing [57, p. 5].
This thesis preliminary focuses on batch processing and finding similar items. But the possibility to pass song titles in real-time to Spark and getting recommendation lists of similar songs in a few seconds in return could be a long-term goal of future work.

2.6.3 Music Similarity with Big Data Frameworks

The similarities can be calculated as "one-to-many-items" similarities. That means that for only one song at a time the similarities to all other songs have to be calculated. This is the approach investigated in this thesis. The other option would be to pre-calculate a full similarity matrix (All-pairs similarity). But looking at large-scale datasets with millions of songs, this would take a considerable amount of time. A combination of both approaches would be to calculate the similarities for one song request at a time but store these similarities into a sparse similarity matrix once they got computed to speed up subsequent requests of the same songs. But this is beyond the scope of this thesis.
Given the short introduction to Big Data frameworks, the decision to use Spark for the computation of the similarities between audio features can be justified as follows.
The computation of the "one-to-many-item" similarity follows the shared nothing approach of Spark. All of the features from different songs are independent of each other, and the distances can be computed in parallel. Only the scaling of the result requires an aggregation of maximum and minimum values. And to return the top results, a means of sorting has to be performed. But apart from these operations that require data shuffling, all the features can be distributed on a cluster and the similarity to one broadcasted song can be calculated independently, following the data locality principle. This offers a fully scalable solution for very large datasets. Additionally, Spark enables efficient ways to cache the audio feature data into the main memory. Under the prerequisite that the sum of all features from all songs fit into the main memory of the cluster, interactive consecutive song requests could be answered without the need of reading the features from the hard drive every time. One limitation is that Spark itself is unable to read and handle audio files. The feature extraction itself has to be performed separately, and only the extracted features are loaded into the cluster and processed with Spark. The feature extraction process is later described in Section 4.2.

3. Similarity Analysis

This chapter introduces and evaluates different similarity measurements for timbral, melodic, and rhythmic features of music data. It explains the feature extraction, preprocessing and similarity estimation between different songs based on the different feature types.

3.1 Timbre Similarity

This section focuses on different similarity measurements and metrics based on MFCCs. Mel Frequency Cepstral Coefficients have already been introduced in Section 2.2.2 as a feature to describe timbre.

3.1.1 Euclidean Distance

To further reduce the dimensionality of the original MFCC features, a statistical summarization can be calculated. For each of the mel bands (13 in this case to reduce dimensionality) the mean and standard deviation over all frames are calculated, resulting in a vector of 13 mean values, a 13 by 13 covariance matrix ($\frac{13\cdot(13-1)}{2}$ covariance values, because of the triangular shape - the upper triangle contains the covariances and the main diagonal contains the variances) and 13 variances. These vectors are not dependent on the length of the actual song. [2, pp. 51ff]
Using such a model, the distance between two songs can be calculated using the L_p distance as in equation

$$d(x,y) = ||x-y||_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}, \tag{3.1}$$

where x and y are the n-dimensional feature vectors of two different musical pieces. Usually, the Euclidean (L_2) or the Manhattan (L_1) distance would be used in real-world scenarios [2, p. 58]. This very basic metric of timbre similarity has been refined and improved over the past years.

3.1.2 Single Gaussian Model

Symmetric Kullback-Leibler Divergence

The second approach was first proposed by Mandel and Ellis in 2005 [15] and is briefly summarized in [2, pp. 65f].
Assuming two musical pieces P and Q are given, after computing the mean value of each MFCC (resulting in the vectors μ_P and μ_Q) and the covariance matrix of the different MFCC vectors (Σ_P and Σ_Q), the Kullback-Leibler divergence (KL divergence) can be calculated as follows, with $\mathrm{tr}(\cdot)$ being the trace (i.e., the sum of the diagonal of a matrix), d being the dimensionality (number of MFCCs) and $|\Sigma_P|$ being the determinant of Σ_P [2, pp. 65f].

$$\mathrm{KL}_{(P||Q)} = \frac{1}{2}[\log\frac{|\Sigma_P|}{|\Sigma_Q|} + \mathrm{tr}(\Sigma_P^{-1}\Sigma_Q) + (\mu_P - \mu_Q)^T\Sigma_P^{-1}(\mu_Q - \mu_P) - d] \tag{3.2}$$

As a second step the result has to be symmetrized.

$$d_{SKL}(P,Q) = \frac{1}{2}(\mathrm{KL}_{(P||Q)} + \mathrm{KL}_{(Q||P)}) \tag{3.3}$$

This approach is one of the two available similarity metrics in the Musly [14] toolkit (see Section 2.3). It can be simplified and written as a closed form according to Schnitzer [22, p. 44]:

$$d_{SKL}(P,Q) = \frac{1}{4}(\mathrm{tr}(\Sigma_P\Sigma_Q^{-1}) + \mathrm{tr}(\Sigma_Q\Sigma_P^{-1}) + \mathrm{tr}((\Sigma_Q^{-1}\Sigma_P^{-1})(\mu_P - \mu_Q)^2) - 2d) \tag{3.4}$$

Jensen-Shannon-Like Divergence

The second available similarity method in the Musly toolkit by Schnitzer is using the Jensen-Shannon divergence (in a slightly adapted way). "The Jensen-Shannon (JS) divergence is another symmetric divergence derived from the Kullback-Leibler divergence. To compute it, a mixture X_m of the two distributions is defined" [22, p. 43]. "To use the Jensen-Shannon divergence [...] to estimate similarities between Gaussians, an approximation of X_m as a single multivariate Gaussian can be used [...] This approximation of X_m is exactly the same as the left-type Kullback-Leibler centroid of the two Gaussian distributions [...]" [22, p. 45]

$$\mu_m = \frac{1}{2}\mu_P + \frac{1}{2}\mu_Q \tag{3.5}$$

$$\Sigma_m = \frac{1}{2}(\Sigma_P + \mu_P\mu_P^T) + \frac{1}{2}(\Sigma_Q + \mu_Q\mu_Q^T) - \mu_m\mu_m^T \tag{3.6}$$

$$\mathrm{JS}(P,Q) = \frac{1}{2}\log|\Sigma_m| - \frac{1}{4}\log|\Sigma_P| - \frac{1}{4}\log|\Sigma_Q| \tag{3.7}$$

Mutual Proximity

After calculating a similarity matrix for all songs, Musly normalizes the similarities with mutual proximity (MP) [16]. This method aims to reduce the effect of a phenomenon called "hubness", which appears as a general problem of machine learning in high-dimensional data spaces. "Hubs are data points which keep appearing unwontedly often as nearest neighbors of a large number of other data points." [22, p. 66].
Schedl and Knees state: "To apply MP to a distance matrix, it is assumed that the distances $D_{x,i=1..N}$ from an object x to all other objects in the data set follow a certain probability distribution; thus, any distance $D_{x,y}$ can be reinterpreted as the probability of y being the nearest neighbor of x, given the distance $D_{x,y}$ and the probability distribution $P(x)$ [...] MP is then defined as the probability that y is the nearest neighbor of x given $P(x)$ and x is the nearest neighbor of y given $P(y)$" [2, p. 80]
Resulting in:

$$P(X > D_{x,y}) = 1 - P(X \leq D_{x,y}) \tag{3.8}$$

$$\mathrm{MP}(D_{x,y}) = P(X > D_{x,y} \cap Y > D_{x,y}). \tag{3.9}$$

3.1.3 Gaussian Mixture Models and Block-Level Features

Another, more compute-heavy distance measurement would make use of Gaussian Mixture Models (GMMs) of MFCCs. As Knees and Schedl state, "Other work on music audio feature modeling for similarity has shown that aggregating the MFCC vectors of each song via a single Gaussian may work almost as well as using a GMM [...] Doing so decreases computational complexity by several magnitudes, in comparison to GMM-based similarity computations" [2, p. 65]. Therefore, the usage of GMMs is not further considered in this thesis.
The last method mentioned, but not implemented in this thesis for timbral similarity uses block-level features as proposed by Seyerlehner [58] and described in short by Knees and Schedl [2, p. 67]. Instead of using single frames and summarizing them into statistical or probabilistic models, block-level features use larger, e.g., multiple-second long, audio frames. Features like fluctuation patterns (later introduced in Section 3.3.2) and spectral patterns (containing timbre information) are computed for these larger blocks of frames.

3.1.4 Validation

For this thesis, the symmetric Kullback-Leibler (SKL) divergence, the Jensen-Shannon-like (JS) divergence and the Euclidean distance are chosen and tested. There is always a trade-off between the complexity and functionality of distance computing algorithms. A re-implementation of block-level features remains left open for future research due to its rather compute heavy nature.
Using the Musly toolkit, a first evaluation using the symmetric KL divergence is presented in this section. The feature extraction and distance calculation can also be done in Python using the librosa library, and a re-implementation of the Mandel-Ellis approach was tested as well.

Genre Recall Rate / Construction Noise

In general, a good measurement for the efficiency of timbre similarity algorithms is the ability to recommend songs of the same genre. Alternatively, the example proposed by Dr. Bosse from the introduction was tested (see Chapter 1). Comparing a construction noise sound sample with the private music collection containing mostly metal, rock, pop, classical and hip hop music, the following six best results based on the JS divergence were returned in descending order:

1. Ziegenmühlen Session - Down On The Corner (Folk Musik)
2. While She Sleeps - The Divide (Metalcore)
3. Delain - Mother Machine (Live) (Symphonic Metal)
4. Within Temptation - Sanctuary (Intro Live) (Symphonic Metal)
5. Without A Martyr - Medusa's Gaze (Death Metal)
6. 100 Meisterwerke der Klassik - Orpheus In The Underworld (Orphée aux enfers) - Can-Can (Live At Grosser Saal, Musikverein) (Klassik)

Figure 3.2 show the distribution of the genres of 100 most similar songs compared to the construction noise sample.
Using an extended dataset consisting of the private music collection, private field recordings, the full FMA dataset, and the musicnet data, the following results could be achieved:

1. Born Pilot - Birds Fell (FMA, Electronic, Noise)
2. mrandmrsBrian - sun is boring (FMA, Avant-Garde, field recordings)
3. steps in snow (private field recording)
4. Sawako - Paris Children (FMA, field recordings)
5. Jeremy Gluck and Michael Dent - Olivier (FMA, Ambient Electronic)

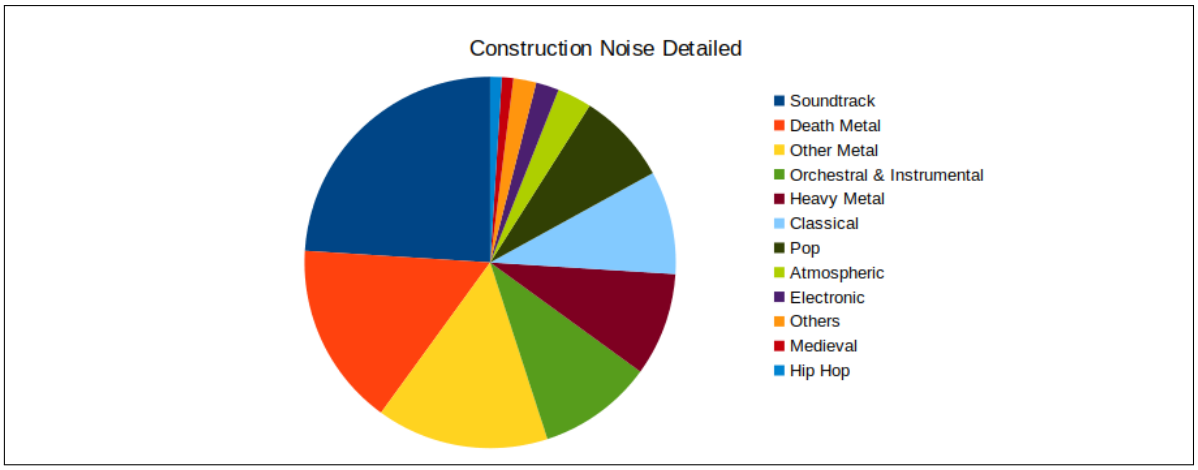


Figure 3.2: Construction noise, first 100 song recommendations based on Musly toolkit (JS)

Especially the second test shows, that the timbre based recommendations are able to recommend similar sounding audio files by returning mostly music containing ambient noises once these were included to the dataset.

Different Recordings and Cover Versions

Another experiment was to get the most similar songs to the famous 'Rondo alla Turca' by Mozart. The recording used as a starting point was taken from the CD "100 Meisterwerke der Klassik" and has a length of 3:33 minutes. This piece by Mozart appears overall four times in the dataset and is recorded by different pianists. Every recording has a different length as listed in the following overview of the recordings by CD.

- 100 Meisterwerke der Klassik (3:33)
- Piano Perlen (3:30)
- The Piano Collection - Disk 18 (3:28)
- Mozart Premium Edition - Disk 31 (4:29)

The top ten most similar songs to the 3 minutes and 33 seconds version are listed below, and the recognized cover versions are underlined:

1. Mozart - Concert No. 10 for 2 Pianos and Orchestra in E Flat Major, KV 365 - 2. Andante
2. Schubert - Sonata in B Flat, D. 960 - III. Scherzo (Allegro vivace con delicatezza)
3. Albeniz - Iberia, Book I - Evocación
4. Mozart Sonate Nr. 11 in A-Dur, K. 33 - Mozart - Alla Turca Allegretto (3:28)
5. Beethoven - Bagatellen Op 119 -Allemande in D major
6. Mozart - Rondo No. 1 in D Major, K. 485

7. Mozart - Sonata For Piano No. 8 KV 310 A Minor - Allegro Maestoso
8. Sonata For Piano No. 16 KV 545 C Major - Rondo: Allegretto
9. Mozart Sonate Nr. 11 in A-Dur, K. 33 - III. Tuerkischer Marsch (3:30)
10. Mozart - Piano Sonata No. 13 in B flat major, K. 333 (K. 315c): Allegretto grazioso

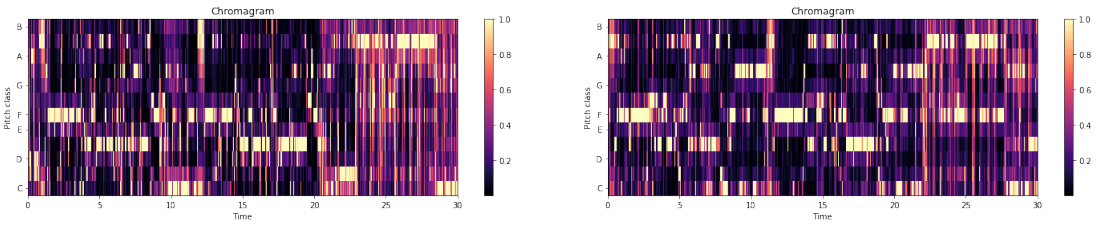
The interesting conclusion is that only two out of the three other versions were considered as most similar songs. The other recording was not even in the top 30 list of the most similar songs. However, the recommendations are all from the same genre (classical music). The inability to detect cover versions was also observable for other songs in the dataset like Serj Tankians song "Lie Lie Lie" from the CD "Harakiri" (just to give an example). This is probably due to the usage of MFCCs valuing the timbre of the music predominantly instead of the pitches and melody movements.

3.2 Melodic Similarity

As presented in Section 2.3.3, there are tools for the extraction of the pitch curve of the main melody line in a song. However, in polyphonic music these kinds of algorithms struggle to get reasonable results. In musical genres like Metal with distorted instruments it is hardly possible to get good results. In conclusion, the main pitch-line extraction and the following conversion of a song with multiple concurrent audio tracks to MIDI using up-to-date open-source toolkits does not produce very reasonable results as shown in 2.3.3. Another possible representation of melodic features is the transformation of the structural information to graphs, as Orio and Roda did [59].
But a better, and also widely used approach is to use chroma features.

3.2.1 Chroma Features Pre-Processing

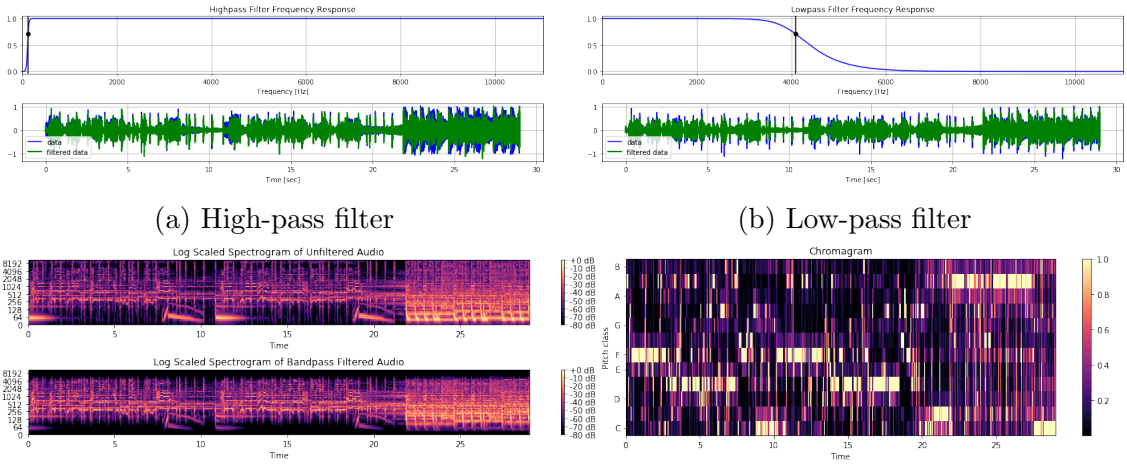
Chroma features, as described in Section 2.2.1, are a good and lower-dimensional way to describe the melody of a song. Most MIR toolkits already offer functions to extract the chromagram from audio files. The plots in this chapter were created using the Essentia [8] and librosa [60] toolkits. The reduction of dimensionality however, comes with a loss of information, especially which octaves the notes are played in. In addition to the pure computation of the chroma features, some pre- and post-processing steps were implemented and tested and will be presented throughout in this chapter.
First of all, Figure 3.3 shows the chromagram plots from two different recordings of the first thirty seconds of the song "Chandelier". Figure 3.3(a) shows the original version sung by the artist Sia and Figure 3.3(b) shows the features of a cover version by the band Pvris. In the last third of each sample the chroma features seemingly get noisier.



(a) Chroma features Sia - Chandelier (b) Chroma features Pvris - Chandelier

Figure 3.3: Chroma feature examples

At these timings in both songs, the bass and drum begin to play. To reduce the impact of rhythm elements over the melodic voice and instrument lines, the audio signal was filtered with a high-pass filter with a cut-off frequency at 128Hz (nearly equal to C3 Key) and secondly by a low-pass filter with a cut-off frequency at 4096Hz (C8 Key). This limits the frequency range to about 5 octaves. In Figure 3.4, the filter frequencies and the original audio signals are visualized in blue color, and the filtered audio signal is green. The spectrogram before and after filtering the audio signal is also shown.



(a) High-pass filter (b) Low-pass filter

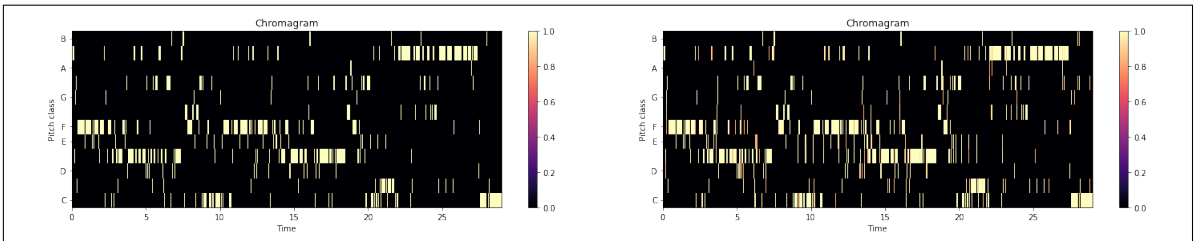
(c) FFT band-pass filter Sia (d) Band-pass filtered chromagram

Figure 3.4: Band-pass filter, Sia - Chandelier

In the chromagram of the band-pass filtered audio signal, the last 10 seconds look cleaner and the melody line is more distinct from the rest in comparison to the chromagram of the unfiltered audio in Figure 3.3.

The next step is to calculate the most dominant note value for each timeframe. Since the chromagram normalizes every timeframe to the maximum note value, the most dominant note is always assigned to value 1. The closer the rest of the notes are to 1, the more likely the timeframe contains silence. If only a few values are close to 1, a chord or harmony is played. To filter out silence the sum over all note values of every timeframe is calculated and if this sum is twice as high as the average sum of notes of

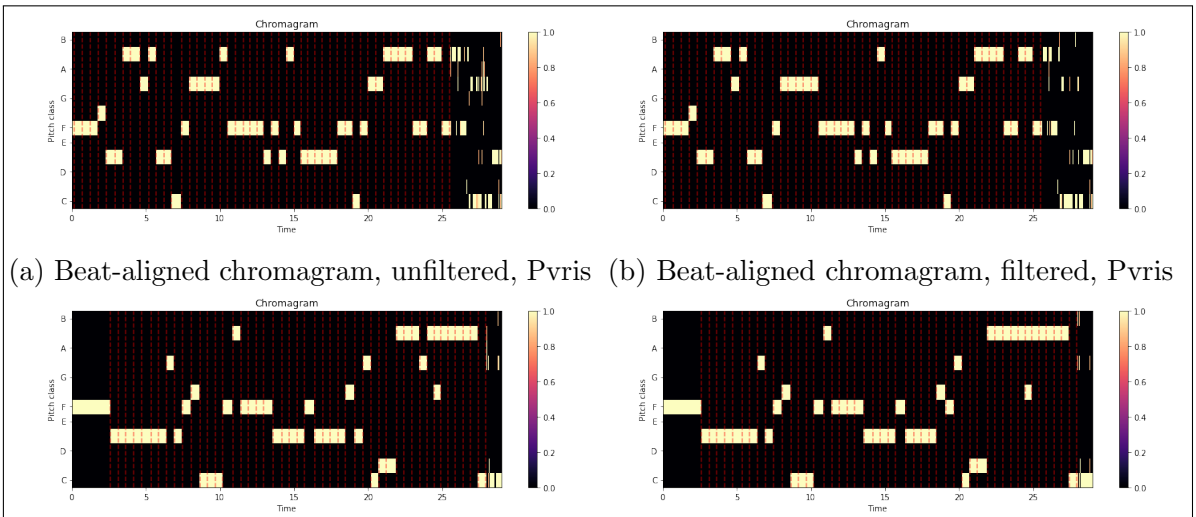
the whole song, the frame is considered as silence. Otherwise, the most dominant pitch is set to a fixed value while the rest of the notes are set to zero.
Usually only the most dominant pitch is needed to extract the main melody, but sometimes the main melody is superimposed by other accompanying instruments. To prevent this, the second most dominant pitches can also be taken into consideration if their values are greater then a specific threshold. The result is shown in Figure 3.5 with a threshold of 0.8.



(a) Single most dominant note only (b) First two most dominant notes

Figure 3.5: Thresholded chroma features, Sia - Chandelier

After that, a beat tracking algorithm is applied to the song and the count of appearances of each note between two beats is calculated. The notes that appear the most between two beats are then set to 1, while the rest is set to 0 for each section between two beats. This beat-alignment serves to make the similarity measurement invariant to the overall tempo of the song. Even if the cover of a song is played with half the tempo of the original song, the melody segment of each bar is still the same as in the faster original version.



(a) Beat-aligned chromagram, unfiltered, Pvris (b) Beat-aligned chromagram, filtered, Pvris

(c) Beat-aligned chromagram, unfiltered, Sia (d) Beat-aligned chromagram, filtered, Sia

Figure 3.6: Processed chroma features, Sia - Chandelier

Figure 3.6 shows the different beat-aligned features of both example songs with band-

pass filtered audio and unfiltered audio. The red lines resemble the detected beat events. Another option would be to separate the frames between the beats in even smaller sections. This would result in a better resolution of the melodic movement but at the same time increase the length of the data vectors that have to be compared to each other. The last processing step is to key shift the chroma features to make the similarity analysis key invariant. One way to do so would be to estimate the key in which each song is played and then shift all chroma features to the same base key, e.g., C Major or A Minor. Due to the structure of the chroma features, this can easily be done by assigning all estimated notes a new value a few keys higher or lower and thus shifting the whole song by a few semitones. The whole workflow to extract the chroma features for this thesis is shown in Figure 3.7.

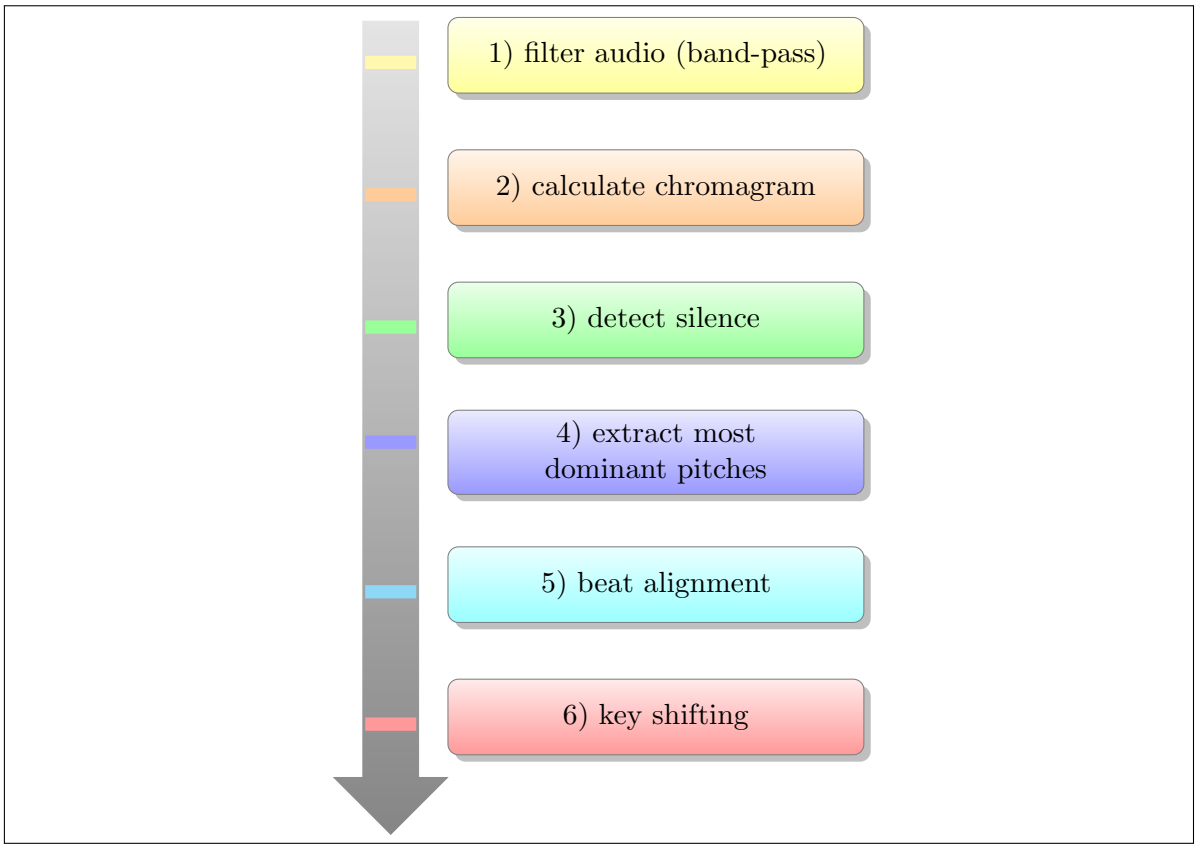
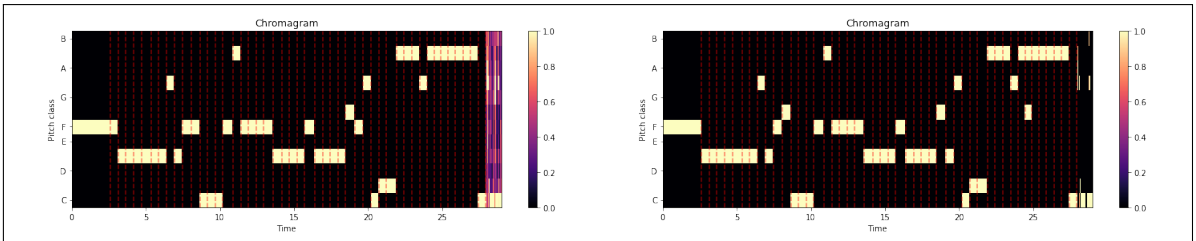


Figure 3.7: Workflow chroma feature extraction

Another consideration is to use the original chromagram without the extraction of only the most dominant keys and thus leaving the processing step 4 out. This means a possible tradeoff between accuracy and computation time. The results for the example song by Sia do not show a major impact as can be seen in Figure 3.8. In this thesis, step 4 will be used in an attempt to get rid of the pitches of the accompaniment from the main melody line.



(a) Using full chromagram (b) Using most dominant pitches

Figure 3.8: Processing step 3 of chroma features in detail

3.2.2 Similarity of Melodic Features

In this section, two completely different approaches to measure the melodic similarity between two songs will be presented. The first one as proposed by [61] or [25], uses text retrieval methods to compare the chroma features of two songs and the second evaluates the usage of cross-correlation of beat-aligned chromagrams as a signal processing approach [62] and [63]

Text Retrieval

One possibility to process the chromagrams and to estimate the similarity between the melodic features of different songs is to handle the pre-processed chromagrams as texts consisting of note values. Due to the extraction of only the main melody line in our feature vector, there is only one note for every detected beat. This main melody line gets converted into a vector of subsequent notes and the resulting vector is converted into a string. The beat- and pitch-alignment done in the previous steps makes the features relatively time- and key invariant. One problem that remains is the different length of the various feature vectors. Xia (et al) [25] mentions that this is indeed a problem when using the Levenshtein distance (also known as the edit-distance) to compute similarities. The Levenshtein distance between the first i characters of a string S and the first j characters of T can be calculated as:

$$\mathrm{lev}_{S,T}(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min(& \\ \quad \mathrm{lev}_{S,T}(i-1,j)+1, & \\ \quad \mathrm{lev}_{S,T}(i,j-1)+1, & \text{else,} \\ \quad \mathrm{lev}_{S,T}(i-1,j-1)+1_{(S_i \neq T_j)} & \\ ) & \end{cases} \tag{3.10}$$

with $1_{(S_i \neq T_j)}$ being the indicator function equal to 0 when $S_i = T_j$ and equal to 1 otherwise, following [25, p. 7].
In their paper, they use MIDI files instead of chroma features, but both contain information about the melody of songs. An adaption to chroma features is not an issue, because they can also easily be interpreted as simple strings. Xia (et al.) made some adjustments to this to be able to handle musical information. [25, pp. 7ff] For example to get rid of the problem of various lengths between the songs, they only took the first 200 and the last 200 notes of every song because it could be observed that cover songs tend to share more common notes in the beginning and at the end of each song.
Due to the fact that this thesis has no actual note information from MIDI files but rather short lists of estimated main pitches from the beat-aligned chroma features, most of the feature vectors are already smaller than 200 notes. Therefore the implemented algorithm does not split the vectors. This tends to favor cover songs that share the same length.
Englmeier (et al.) uses a more advanced text information retrieval technique called "TF-IDF weights" (term frequency - inverse document frequency) and explicit semantic analysis (ESA). "The TF-IDF weight is a measure which expresses the meaning of a term or a document within a collection of documents." [61, p. 186] To do so, "audio words" have to be created from the song database by splitting the audio signal into snippets, creating chroma features and clustering them with the k-means algorithm. The centroids are then added to a database. These audio words can then be evaluated using the TF-IDF weights and ESA. Although their approach looks promising, a re-implementation of their algorithms would exceed the frame of this thesis.

Cross-Correlation

Another possibility to handle the extracted chroma features is to view them as ordinary discrete time signals and creating opportunities to apply classical signal processing algorithms. For this approach, the pre-processing steps laid out in Figure 3.7 can be simplified by skipping steps 3 and 4 and possibly even step 6, as explained later, resulting in beat-aligned chromagrams as shown in Figure 3.11(a) and 3.11(b).
Ellis and Poliner use the cross-correlation in their 2007 published paper [63]. Serra (et al.) also references the work of Ellis and Poliner and discusses different weak points and influences of processing steps like beat tracking and key transposition to the overall performance of this similarity measurement. They also discuss and improve another approach called "dynamic time warping" (DTW) further in their paper [62]. The focus of this thesis is set on the cross-correlation method. Given two discrete-time signals $x[n]$ and $y[n]$ the cross-correlation between both signals $k[n] = (x \star y)[n]$ can be denoted as

follows:

$$k[n] = (x \star y)[n] = \sum_{m=-\infty}^{\infty} x[m]y[m-n]. \tag{3.11}$$

For two two-dimensional input matrices X with the dimensions M by N and Y as a P by Q matrix the cross-correlation result is a matrix C of size $M+P-1$ rows and $N+Q-1$ columns. Its elements are given by the equations [64]:

$$C(k,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n)\overline{Y}(m-k, n-l) \tag{3.12}$$

with

$$-(P-1) \leq k \leq M-1 \tag{3.13}$$

$$-(Q-1) \leq l \leq N-1. \tag{3.14}$$

The bar over $\overline{Y}$ denotes complex conjugation (in this case Y is a matrix with real values only). An example for the one-dimensional cross-correlation is shown in Figure 3.9 and the full two-dimensional cross-correlation of two songs is depicted in Figure 3.10 and 3.11. Ellis and Poliner did not transpose the songs in the pre-processing step to

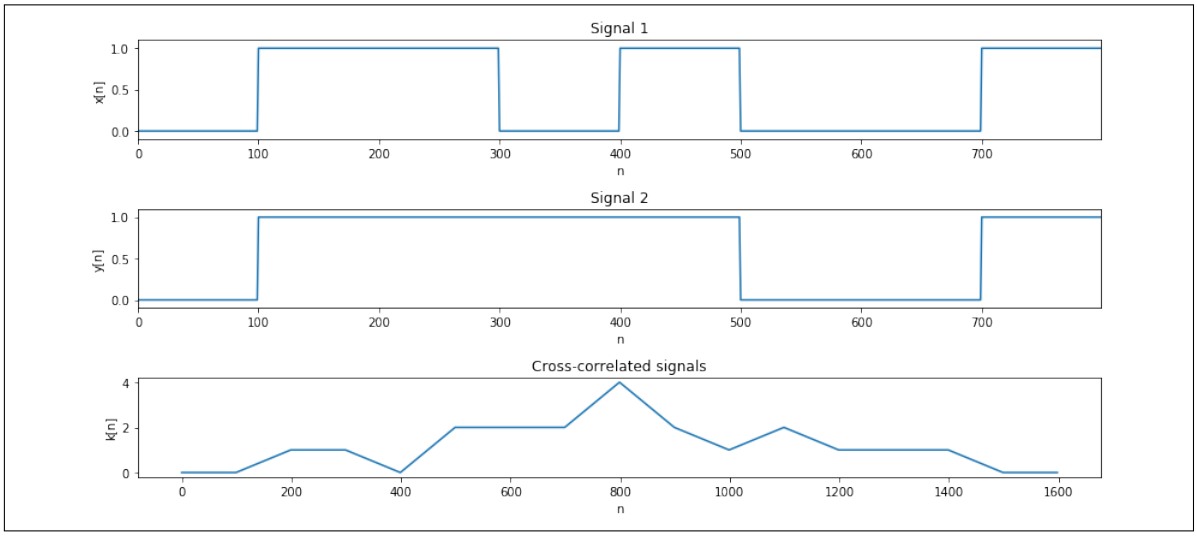


Figure 3.9: 1D cross-correlation

match the keys of both audio files. Instead, they calculated the full cross-correlation for all 12 possible transpositions and chose the best one. As input matrices, they averaged all notes of the chroma features per beat and additionally scaled them to have unit norm at each time slice (beat frame). In the original paper, the cross-correlation is normalized by the length of the shorter song segment to bind the correlation result to an interval between 0 and 1. But in a later published work from Ellis and Cotton, this step was left out as it seemingly resulted in slightly worse detection ratios of cover songs [65]. Additionally, they filtered the result of the cross-correlation with a high-pass

filter. "We found that genuine matches were indicated not only by cross-correlations of large magnitudes, but that these large values occurred in narrow local maxima in the cross-correlations that fell off rapidly as the relative alignment changed from its best value. To emphasize these sharp local maxima, we choose the transposition that gives the largest peak correlation then high-pass filter that cross-correlation function with a 3dB point at 0.1 rad/sample" [63, p. 3]. The later published paper also states that changes to the filter parameters improved the cover song recognition rate further [65]. However, the exact values, e.g., for the cutoff frequency were not given. Accordingly, in this thesis, the older parameters for the filter are used.
Serra (et al.) also discusses the effects of different pre-processing steps that improve the algorithm even further and they note that a higher chroma resolution of 3 octaves gives better results. Also, the key detection and transposition before the cross-correlation gives slightly worse results in comparison to the method Ellis and Poliner used.
In this thesis, a version where the songs are all key aligned before the cross-correlation was tested to reduce the computation time overhead when estimating the similarities on a cluster. In summary, the implementation in this thesis is similar to the approach by Ellis and Poliner [63], but some of the steps from the newer paper [65] leave some space for further improvements.
The chroma features are beat aligned, averaged per beat, and normalized to unit length as well. Additionally, all chroma features are transposed to a common key (A in this case) in the pre-processing step. The full cross-correlation according to Equation (3.12) including "key shifts" with zero padding at the edges by letting k run from $-(P-1) \leq k \leq M-1$ is shown in Figures 3.10 and 3.11. But due to the previously already performed key shift during the pre-processing steps and the fact that both input matrices share the same amount of rows (12, one per semi-tone), the full cross-correlation is not necessary and computation time can be saved by altering the computation according to Equation (3.15) resulting in a vector C with the correlation results without additional key-shifting

$$C(l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n)\overline{H}(m,n-l) \tag{3.15}$$

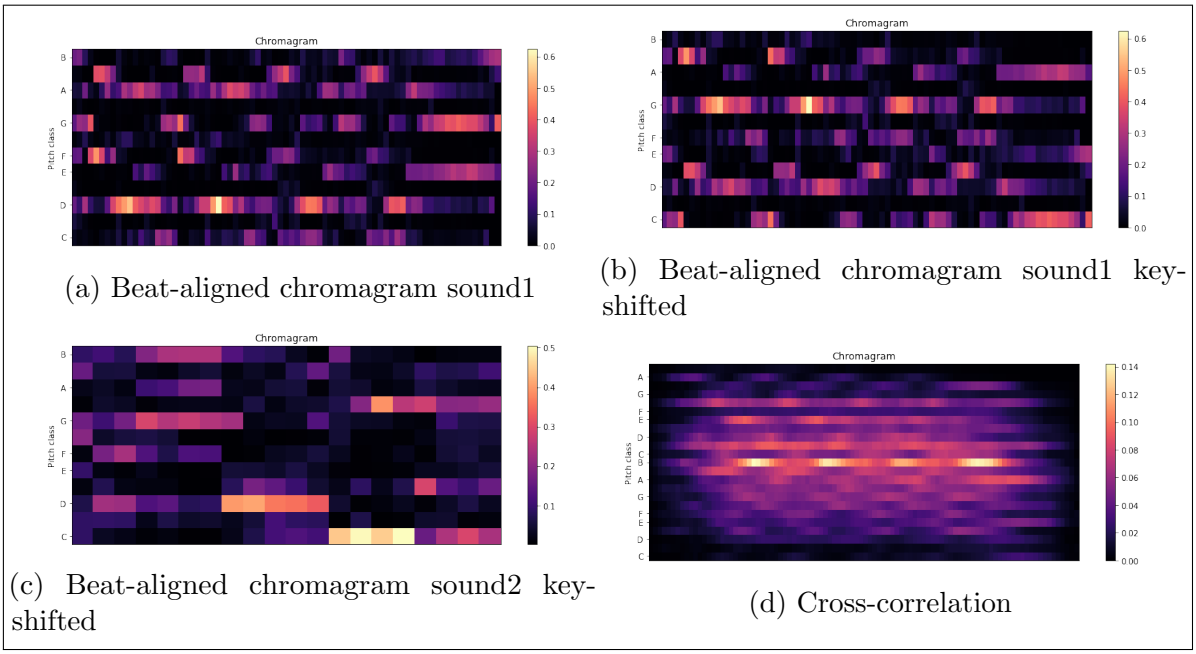
$$-(Q-1) \leq l \leq N-1 \tag{3.16}$$

or even faster without calculating the edges of the matrix (without zero-padding).

$$0 \leq l \leq N-Q \tag{3.17}$$

This simplified version relies on an accurate key detection of the songs during the pre-processing. The post-processing step from Ellis and Poliner, more precisely the

high-pass filtering of the result was also implemented.
Figure 3.10 shows two beat aligned, key-shifted and per beat averaged chroma features of two short guitar audio samples and their cross-correlation results. The most interesting row of the cross-correlation matrix is the middle row marked with the B key on the y-axis, where both chromagrams are aligned.



(a) Beat-aligned chromagram sound1

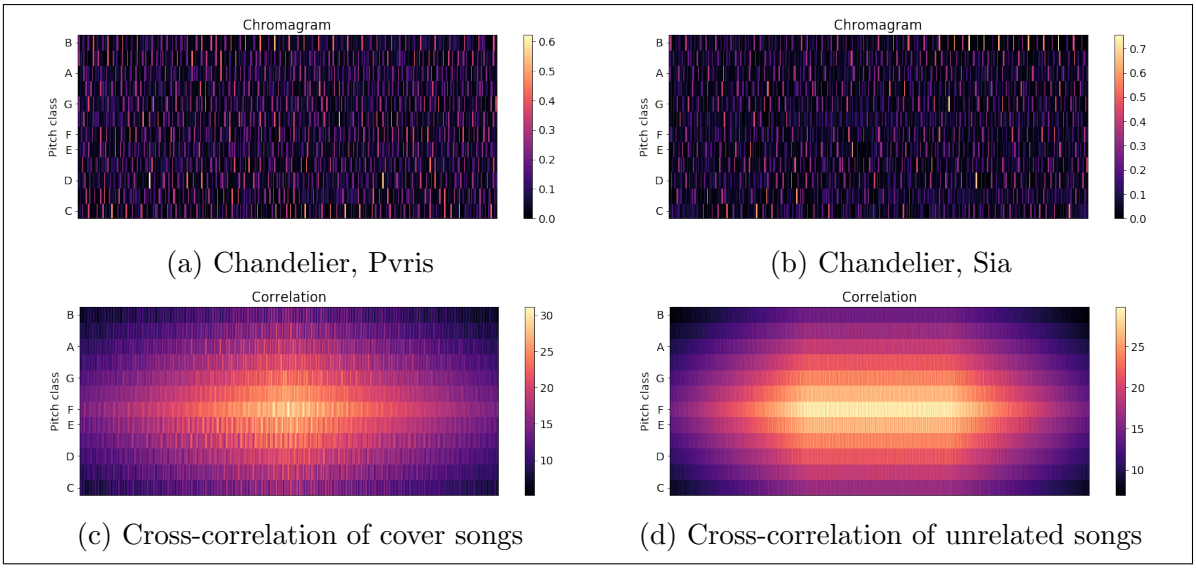
(b) Beat-aligned chromagram sound1 key-shifted

(c) Beat-aligned chromagram sound2 key-shifted

(d) Cross-correlation

Figure 3.10: 2D cross-correlation of beat-aligned and key-shifted chromagrams (audio snippets)

In Figure 3.11 the cross-correlation of the song "Chandelier" by singer Sia and covered by Pvris are shown in 3.11(c) and in contrast to this the cross-correlation of "Chandelier" with the song "Rock you like a Hurricane" by The Scorpions is shown (Figure 3.11(d)). Due to the previous key shifting, plot 3.11(c) shows the maximum peak right in the center row. Originally, the version by Sia is detected to be written in C sharp and the cover version in F sharp, but both songs are shifted to the A key during the pre-processing step.
The unrelated songs result in much smaller correlation values, especially when looking at the middle row of the matrix (marked with the F key on the y-axis in figure 3.11(d)), but also if the songs were transposed additionally even then they would not correlate well, but this is also related to the zero-padding when additional key-shifts are performed. In contrast to this, the cover songs have multiple visible peaks in the center row. The row with the maximum correlation value is extracted, and the resulting plot shows that the cover songs do correlate much better than the unrelated songs (3.12(a) and 3.12(b)). The center rows of the cross-correlation matrices from Figure 3.11 are separately pictured in Figure 3.12. After applying the high-pass filter to the extracted row with the maximum



(a) Chandelier, Pvris

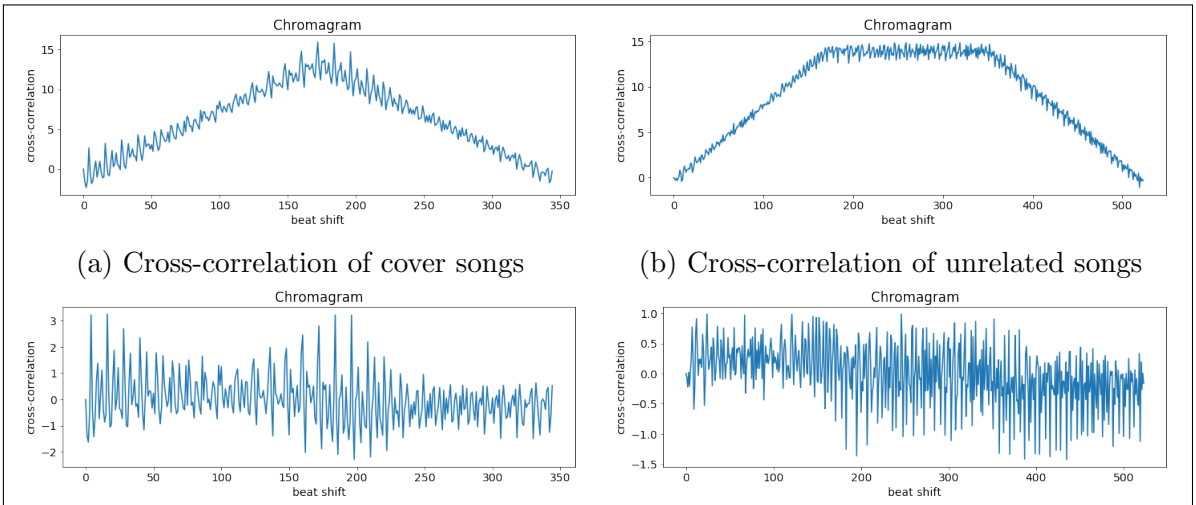
(b) Chandelier, Sia

(c) Cross-correlation of cover songs

(d) Cross-correlation of unrelated songs

Figure 3.11: 2D cross-correlation of beat-aligned chromagrams (Sia / Pvris - Chandelier)

correlation value, the peaks in 3.12(a) when cross-correlating the cover songs are clearly visible compared to the unrelated songs. An interesting detail that can be pointed out is that the song structure is also visible in plot 3.12(c) with clearly visible recurring peaks when the refrain is repeated.



(a) Cross-correlation of cover songs

(b) Cross-correlation of unrelated songs

(c) Cover songs filtered

(d) Unrelated songs filtered

Figure 3.12: Filtered cross-correlation (high-pass)

3.2.3 Validation

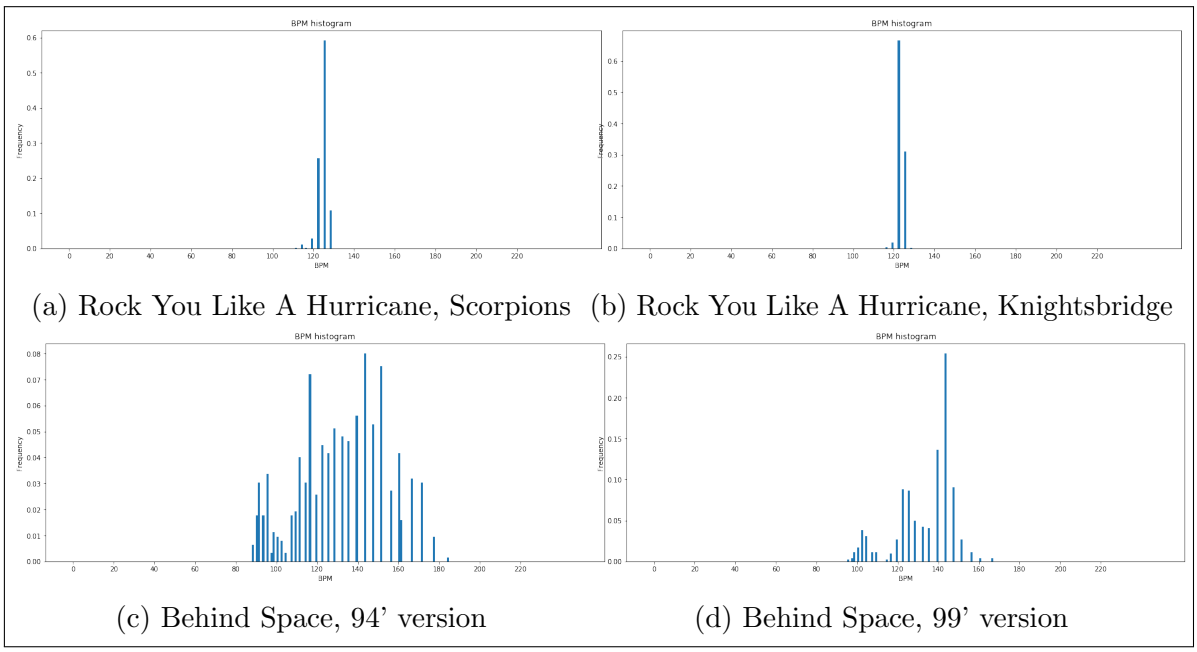
A good measurement for the efficiency of a melodic similarity algorithm is the ability to find cover songs, remixes, and different recordings of the same song. Chapter 5.1.2 evaluates the cover song recognition rate of the implemented recommender system.

3.3 Rhythmic Similarity

This chapter provides an overview of some of the possibilities for computing music similarity by focusing on rhythmic features of different songs.
Nearly every MIR toolkit provides an extraction tool for the beats per minute (BPM) and thus the tempo of each song. The most trivial solution of computing very low-level rhythmic similarities is by sorting and comparing songs only by their main tempo (BPM). There are certainly far better and more accurate solutions. This chapter presents some of the most promising approaches to compute rhythm similarities regarding the applicability in a Big Data framework.

3.3.1 Beat Histogram

One possible similarity measurement is, e.g., the usage of beat histograms as proposed by Tsanetakis and Cook [66]. The Essentia toolkit offers methods to extract the beat histogram. The different detected BPMs are normalized to 1. If a song changes its tempo, then multiple peaks can be seen. Figure 3.13 shows the beat histograms of



(a) Rock You Like A Hurricane, Scorpions (b) Rock You Like A Hurricane, Knightsbridge

(c) Behind Space, 94' version (d) Behind Space, 99' version

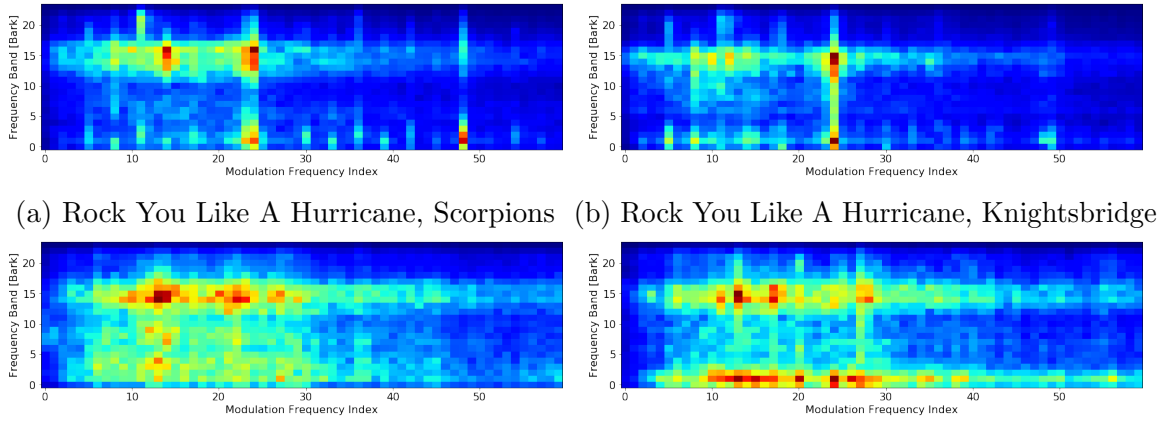
Figure 3.13: Beat histogram examples

the song "Rock you like a hurricane" by the Scorpions (Figure 3.13(a)) and covered by Knightsbridge(Figure 3.13(b)) as well as two different versions of the song "Behind Space" from the Swedish metal band In Flames, one is sung by Stanne Mikkels in 1994 (Figure 3.13(c)) and the second version was recorded with Anders Friden as the vocalist

in 1999 (Figure 3.13(d)). The 1994 version changes its tempo in the outro of the song, and the tempo change can be seen in the histogram in Figure 3.13(c) as a second large peak around 120 BPM. Similarities between two beat histograms can be computed using the Euclidean distance. Gruhne (et al.) further improved beat histograms and suggested an additional post-processing step before calculating the similarity between songs with the Euclidean distance. They found that logarithmic re-sampling of the lag axis of the histogram and cross-correlation with an artificial rhythmic grid improves the performance of this similarity measurement further (see [67, p. 182]). This thesis does not use the additional re-sampling.
Another paper that is just mentioned here (one of the older ones from 2002) uses the beat spectrum as a feature [27] to compute similarities.

3.3.2 Rhythm Patterns

A more state-of-the-art feature is the so-called rhythm pattern, also known as fluctuation patterns, evaluated by Lidy and Rauber in [68] for instance. To extract these features, the rp_extractor library for Python [69] was made publicly available by the TU Vienna [70]. Figure 3.14 shows the extracted rhythmic patterns of the previously mentioned songs "Rock you like a Hurricane" and "Behind Space". The similarities of the different versions from the same songs are quite visible, while at the same time substantial differences between the different songs are recognizable.



(a) Rock You Like A Hurricane, Scorpions (b) Rock You Like A Hurricane, Knightsbridge

(c) Behind Space, 94' version (d) Behind Space, 99' version

Figure 3.14: Rhythm pattern examples

The x-axis represents the frequency bands converted to the Bark-scale (a scale representing the human auditory system comparable to the mel scale from Equation (2.6)), and the y-axis represents the modulation frequency index representing the modulation frequencies up to 10Hz (around 600 BPM). The Bark of a frequency f can be determined

using the equation

$$\text{Bark} = 13 \arctan(0.00076 f) + 3.5 \arctan((f/7500)^2). \tag{3.18}$$

The algorithm to extract rhythm patterns, rhythm histogram, and statistical spectrum descriptors measuring the variations over the critical frequency bands, can be seen in Figure 3.15.

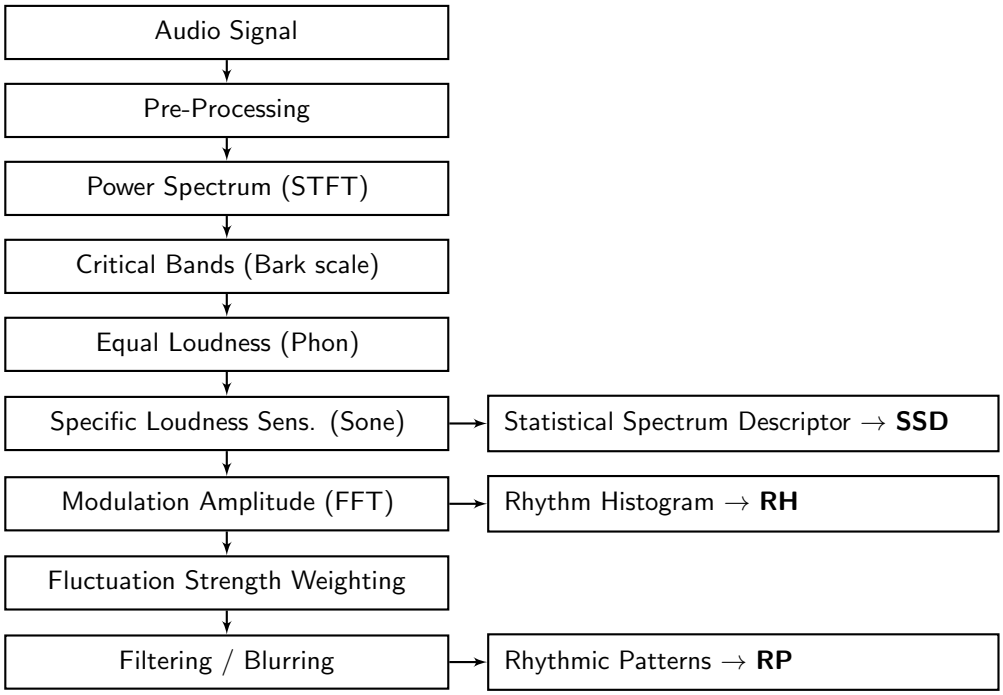


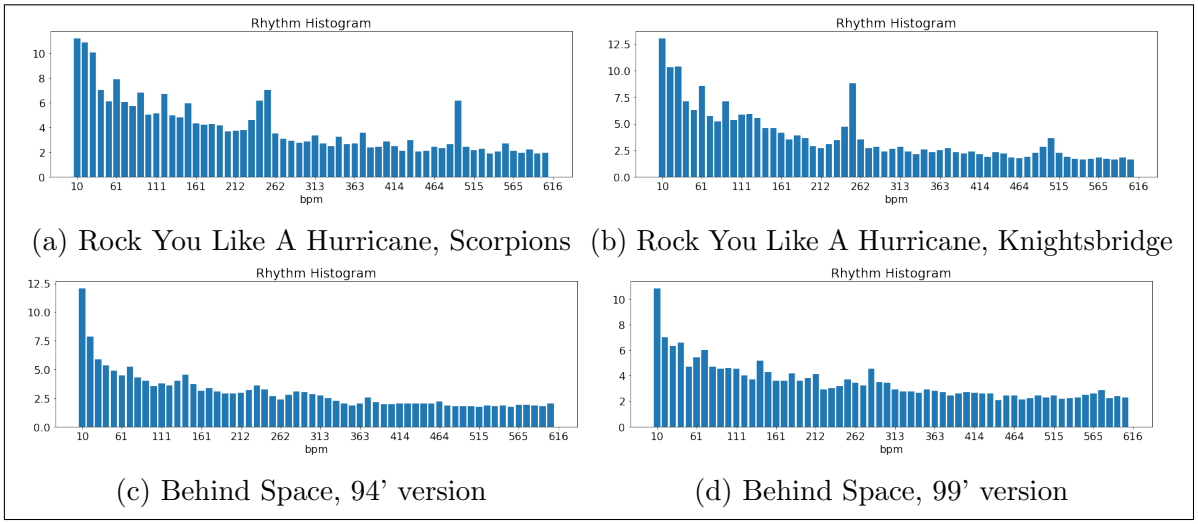
Figure 3.15: Rhythm pattern extraction procedure as suggested by [70]

In conclusion, the rhythm patterns basically represent the BPM of various frequency bands. To compare two different songs the Euclidean distance between the vectorized rhythm pattern matrices can be calculated as Pampalk suggests [71, p. 40].
Pohle, Schnitzer (et al.) [72] later refined fluctuation patterns into onset patterns, e.g., by using semitone bands instead of fewer critical bands to detect onsets. This thesis however, focuses on fluctuation-/ rhythm patterns extracted with the rp_extractor library.

3.3.3 Rhythm Histogram

A more simplistic and lower-dimensional feature coming with the rp_extract toolkit is the rhythm histogram. "The Rhythm Histogram features we use are a descriptor for general rhythmics in an audio document. Contrary to the Rhythm Patterns and the Statistical Spectrum Descriptor, information is not stored per critical band. Rather, the magnitudes of each modulation frequency bin of all 24 critical bands are summed

up, to form a histogram of "rhythmic energy" per modulation frequency. The histogram contains 60 bins which reflect modulation frequency between 0 and 10 Hz." [68, p. 3]. The difference between rhythm histogram and the earlier in Section 3.3.1 mentioned beat histogram appears to be the beat histogram focusing on the basic tempo of the whole song while the rhythm histogram takes all frequency bands and therefore the sub-rhythms of single instruments into account. Figure 3.16 shows the rhythm histograms of four example songs.



(a) Rock You Like A Hurricane, Scorpions (b) Rock You Like A Hurricane, Knightsbridge

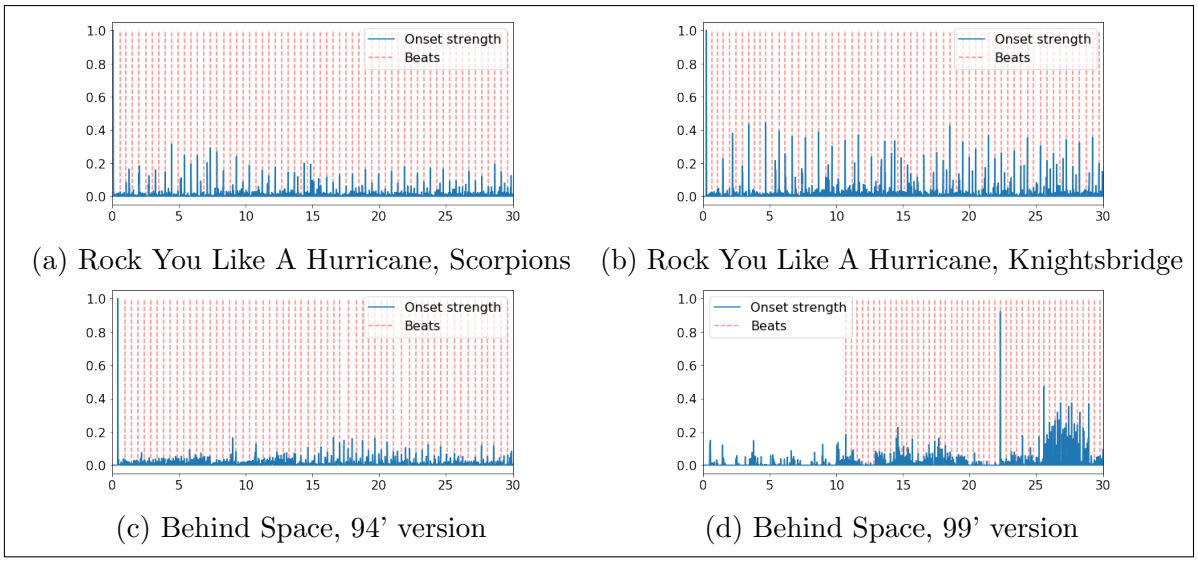
(c) Behind Space, 94' version (d) Behind Space, 99' version

Figure 3.16: Rhythm histogram examples

3.3.4 Cross-Correlation

Estimating the onset strength as introduced in section 2.2.3, averaging it per beat and creating a discrete-time signal for each song is another possibility. Similar to the chroma features, the cross-correlation of these discrete-time onset features could be used as a similarity measurement, following Equation 3.11.

Looking at the extracted onset features of the Song "Behind Space" by In Flames (sung by Anders Frieden 99' and Stanne Mikkels 94') in Figure 3.17, one can see that the quality of these signals is greatly dependent on the underlying beat extraction and onset detection algorithms. As an example, the librosa toolkit struggles to detect beats in the first 10 seconds of the song "Behind Space" recorded in 1999. Also, this representation seems to contain a lot less valuable and comparable information in contrast to fluctuation patterns. In conclusion, this approach is discarded and not further considered and tested in this thesis.



(a) Rock You Like A Hurricane, Scorpions

(b) Rock You Like A Hurricane, Knightsbridge

(c) Behind Space, 94' version

(d) Behind Space, 99' version

Figure 3.17: Detected onset examples (30 second song snippets)

3.4 Summary

After evaluating various options of similarity measurements for different aspects of music (timbre, rhythm, and melody), all of the chosen approaches that are implemented in the next chapters are summarized in this section.
The chosen similarity metrics for timbre similarity are:

- Euclidean Distance
- symmetric Kullback-Leibler divergence
- Jensen-Shannon-like divergence

For the computation of the melodic similarities, two different similarity metrics are chosen:

- Levenshtein Distance
- cross-correlation on full beat aligned and per beat averaged chroma features, key shifted to A

Three different similarity measurements are chosen for the rhythm features:

- Euclidean distance between beat histograms
- Euclidean distance between rhythm histograms
- Euclidean distance between rhythm patterns

4. Implementation

The implementation consists of two separate parts. The first contains the feature extraction and preparation of data from the audio files. The results are stored into feature files. In the second part, these feature files then have to be processed with the Big Data framework Spark to compute the similarities between songs.
Both parts are implemented in Python and can be executed on computer clusters. The source code can be found on the CD in the appendices and it can be pulled from GitHub [73]. Details for the usage of the Python scripts are also documented there.

4.1 Underlying Hardware

The first tests were performed on a single PC with 4 CPU cores (8 with HT) (Intel Core i7-3610QM CPU, 2.30GHz $\times$ 4) running Spark 2.4.0.
The cluster tests were performed on the ARA-cluster of the Friedrich Schiller University in Jena. It offers 16 compute-nodes for Spark applications with 36 CPU-cores (Dual Socket, 2 x Intel Xeon "Scalable" 6140, 2.30 GHz x 18) per node (72 with HT), and 192GB of RAM. The cluster's Spark partition is running with an older version of Spark (1.6.0). The ARA-cluster also offers a larger "Skylake" partition with 152 compute-nodes of which 36 were used to extract the audio features with. The hardware of these nodes is the same as in the Spark partition of the cluster.

4.2 Audio Feature Extraction

So far, the required audio features as well as toolkits to extract those features from the audio data have been described and selected in Chapter 2. In Section 2.5, different sources for audio files have been presented. Section 3.1, 3.2, and 3.3 presented algorithms to pre-process the low-level features and use these to compute similarities. This section focuses on the selection of fitting datasets and the performance of the feature extraction and pre-processing software implementation.

4.2.1 Test Datasets

A lot of data is needed to test the algorithms in a Big Data environment, so the Free Music Archive with its over 106000 songs is a good option for performance tests. It has to be kept in mind, that the genre distribution in the FMA dataset is quite one sided. Most of the songs are tagged as experimental, electronic, and rock. Also, this dataset may not be representative for actual popular music, a lot of the songs are live recordings with poor audio quality. The 1517-Artists dataset offers 19 different genres with songs relatively evenly distributed. For an objective evaluation of the proposed algorithms, e.g., by genre recall, this dataset is ideal. For cover song detection, the covers80 dataset is included as well. The last source used in this thesis is the private music collection. This collection is biased towards metal music, but due to the match with personal taste, it enables a subjective evaluation of the results from the implemented recommender system. In conclusion that adds up to about 117000 songs for performance tests, from which in the end 114210 could be used (see Section 4.2.2 for the details on the dropout), and about 11500 songs for a detailed evaluation of the algorithms in this thesis and the quality of the recommendations. As mentioned in Section 2.5.1, all albums from the private music collections are catalogued as well, and the associated document is in the appendices.

FMA	106.733 Songs
private	8484 Songs
1517-Artists	3180 Songs
covers80	164 Songs (80 originals + 84 covers)

Table 4.1: Selected music datasets

4.2.2 Feature Extraction Performance

After evaluating the different features in the last three chapters, this section only discusses the performance of the feature extraction process without going too much into the details of the code for feature pre- and post-processing, like the note estimation from the chroma features and the calculation of statistic features from the MFCCs. These additional steps were already explained in-depth in the previous chapters and are therefore left out here. The full code is in the appendices.

Librosa

For most of the plots in Chapter 2, the Python toolkit librosa was used because of its ease of use and very good documentation. The following code example shows the

necessary steps to extract the most important features like MFCC, chromagram, beats, and onsets.

```
1 path = ('music/guitar2.mp3')
2 x, fs = librosa.load(path)
3 mfcc = librosa.feature.mfcc(y=x, sr=fs, n_mfcc=12)
4 onset_env = librosa.onset.onset_strength(x, fs, aggregate=np.median)
5 tempo, beats = librosa.beat.beat_track(onset_envelope=onset_env,sr=fs)
6 times = librosa.frames_to_time(np.arange(len(onset_env)), sr=fs, hop_length=
      512)
7 chroma = librosa.feature.chroma_stft(x, fs)
```

Code Snippet 4.1: Librosa

First of all an audio file is read into the variable `x` and the sample rate `fs` is returned by `librosa.load(path)`. This audio file is then passed to `librosa.feature.mfcc()` for the extraction of the MFCCs, `librosa.onset.onset_strength()` for the onsets, and `librosa.feature.chroma_stft()` to extract the chromagram. The onsets are also used to detect the beats and their time signatures in the song.
When extracting features from batches of audio files, the librosa library turned out to be very slow. For a tiny dataset of 100 songs, the extraction of just the mean, variance, and covariance of the MFCCs and the estimated notes from the chromagram took about 53 minutes on a single computer (1 CPU core used). For larger datasets like the 1517-Artists dataset, the feature extraction process would have taken about 28 hours and over 940 hours for the FMA dataset.

Essentia

Moffat (et al.) [4] compare different Audio feature extraction toolboxes and show that Essentia is a much faster alternative to librosa due to the underlying C++ code and provides even more features, but it is a bit less well documented and requires more effort for the implementation at the same time. In the end, the code to extract the necessary features had to be rewritten for the usage of Essentia due to the slow performance of librosa. Essentia offers two different ways to handle audio files. The first one is to use the Essentia standard library. It provides similar methods as librosa and uses an imperative programming style. The audio file has to be read, sliced and pre-processed manually. The second way is to use Essentia streaming. Basically, a network of connected algorithms is created, and they handle and schedule the "how and when" of the execution whenever a process is called. The melodic and timbral features and the beat histograms are computed with Essentia. Only the rhythm patterns and rhythm histograms are computed in a separate step, as stated below.

Essentia Standard

In the final code for the audio feature extraction, the computation of the MFCCs and beat histogram is done with the Essentia standard library, because it offers a fast and easy way to implement the basic feature extraction tasks (see Code Snippet 4.2).

```
1 audio = es.MonoLoader(filename=path, sampleRate=fs)()
2 hamming_window = es.Windowing(type='hamming')
3 spectrum = es.Spectrum()
4 mfcc = es.MFCC(numberCoefficients=13)
5 mfccs = numpy.array([mfcc(spectrum(hamming_window(frame)))[1] for frame in es.
      FrameGenerator(audio, frameSize=2048, hopSize=1024)])
6 rhythm_extractor = es.RhythmExtractor2013(method="multifeature")
7 bpm, beats, beats_confidence, _, beats_intervals = rhythm_extractor(audio)
8 peak1_bpm, peak1_weight, peak1_spread, peak2_bpm, peak2_weight, peak2_spread,
      histogram = es.BpmHistogramDescriptors()(beats_intervals)
```

Code Snippet 4.2: Essentia standard

Again at first an audio file is read into the variable `audio` by calling `es.MonoLoader (filename=path, sampleRate=fs)()`. This audio file is then split into frames by the `es.FrameGenerator()` for the following extraction of the MFCCs with `mfcc(spectrum( hamming_window(frame)))[1]` for each frame. For the beat extraction the audio data gets passed to `rhythm_extractor()`.

Essentia Streaming

The Essentia streaming library is used to calculate the chroma features. It eases up filtering with high- and low-pass filters. The audio signal is passed through various processing stages and ultimately results in the chroma features of the band-pass filtered audio signal. In Code Snippet 4.3 the different stages get set up, e.g., the filter parameters are set by calling `ess.HighPass(cutoffFrequency=128)` and `ess.LowPass(cutoffFrequency =4096)`. The audio file is read by calling `ess.MonoLoader()`.

```
1 loader = ess.MonoLoader(filename=path, sampleRate=44100)
2 HP = ess.HighPass(cutoffFrequency=128)
3 LP = ess.LowPass(cutoffFrequency=4096)
4 framecutter = ess.FrameCutter(frameSize=frameSize, hopSize=hopSize, silentFrames
      ='noise')
5 windowing = ess.Windowing(type='blackmanharris62')
6 spectrum = ess.Spectrum()
7 spectralpeaks = ess.SpectralPeaks(orderBy='magnitude', magnitudeThreshold
      =0.00001, minFrequency=20, maxFrequency=3500, maxPeaks=60)
```

```
8  hpcp = ess.HPCP()
9  hpcp_key = ess.HPCP(size=36, referenceFrequency=440, bandPreset=False, ⤸
       minFrequency=20, maxFrequency=3500, weightType='cosine', nonLinear=False, ⤸
       windowSize=1.)
10 key = ess.Key(profileType='edma', numHarmonics=4, pcpSize=36, slope=0.6, ⤸
       usePolyphony=True, useThreeChords=True)
11 pool = essentia.Pool()
```

Code Snippet 4.3: Essentia streaming

In Code Snippet 4.4 the audio file gets passed through the various stages. At first it gets filtered with a high- and low-pass filter, resulting in a band-pass filter operation. Then the signal gets split into frames and the chromagram (harmonic pitch class profiles, HPCP) gets extracted and stored into `chroma`.

```
1  loader.audio >> HP.signal
2  HP.signal >> LP.signal
3  LP.signal >> framecutter.signal
4  framecutter.frame >> windowing.frame >> spectrum.frame
5  spectrum.spectrum >> spectralpeaks.spectrum
6  spectralpeaks.magnitudes >> hpcp.magnitudes
7  spectralpeaks.frequencies >> hpcp.frequencies
8  spectralpeaks.magnitudes >> hpcp_key.magnitudes
9  spectralpeaks.frequencies >> hpcp_key.frequencies
10 hpcp_key.hpcp >> key.pcp
11 hpcp.hpcp >> (pool, 'tonal.hpcp')
12 essentia.run(loader)
13 chroma = pool['tonal.hpcp'].T
```

Code Snippet 4.4: Essentia streaming

Essentia Performance

The calculation with the Essentia streaming and standard library for 100 songs took less than a third of the time librosa needed. This is a significant improvement, however the Essentia library uses only one CPU core so that performance was further improved by using the Parallel Python and mpi4py library.

Parallel Python

Parallel Python is a Python module that enables the execution of Python code in parallel. On a single PC, multiple CPU cores get parts of the full filelist and compute the features fully in parallel (see Code Snippet 4.5).

```
job_server = pp.Server()
job_server.set_ncpus(ncpus)
jobs = [ ]
for index in xrange(startjob, parts):
  starti = start+index*step
  endi = min(start+(index+1)*step, end)
  jobs.append(job_server.submit(parallel_python_process, (index, filelist[starti
      :endi],1,1,1,1,1)))
  gc.collect()
times = sum([job() for job in jobs])
job_server.print_stats()
```

Code Snippet 4.5: Parallel Python

The computation time takes on average approximately 18.6 seconds per song and processor core (assuming ideal data balancing).

$$\text{time} = \frac{\#\text{songs}}{\#\text{CPUs}} \cdot 18.6s \tag{4.1}$$

Using 4 CPU cores for 100 songs, the overall processing time could be reduced to about 465 seconds. Parallel Python also opens up the possibility to use a cluster instead of a single-node PC.

For convenience, every processor gets a batch of files instead of single songs. For every batch, different output files for the various features are created. The batch size determines the overall size of these feature-files. As an example, a batch size of 400 songs was chosen for the 1517-Artists dataset, which means four CPUs had to process two batches, resulting in eight different output files with the chroma feature files being the largest with about 25MB per file.

One problem that appeared when using Parallel Python was that the main memory usage increased over time. Neither explicit usage of the garbage collector nor the deletion of unwanted objects also could solve that problem. After processing a few hundred songs the processes eventually ran out of memory and had to be restarted. By replacing Parallel Python with mpi4py, this problem could later be solved (see Section 4.2.2).

Rp_extractor

For the extraction of the rhythm patterns and rhythm histogram features as described in Section 3.3, the ”rp_extractor” tool provided by the TU Wien was used. Although running in parallel on all CPU cores on a single node, the extraction of the features from 100 songs took about 344 seconds.

Performance on a Single PC

The performance of the different MIR toolkits is shown in Figure 4.1.

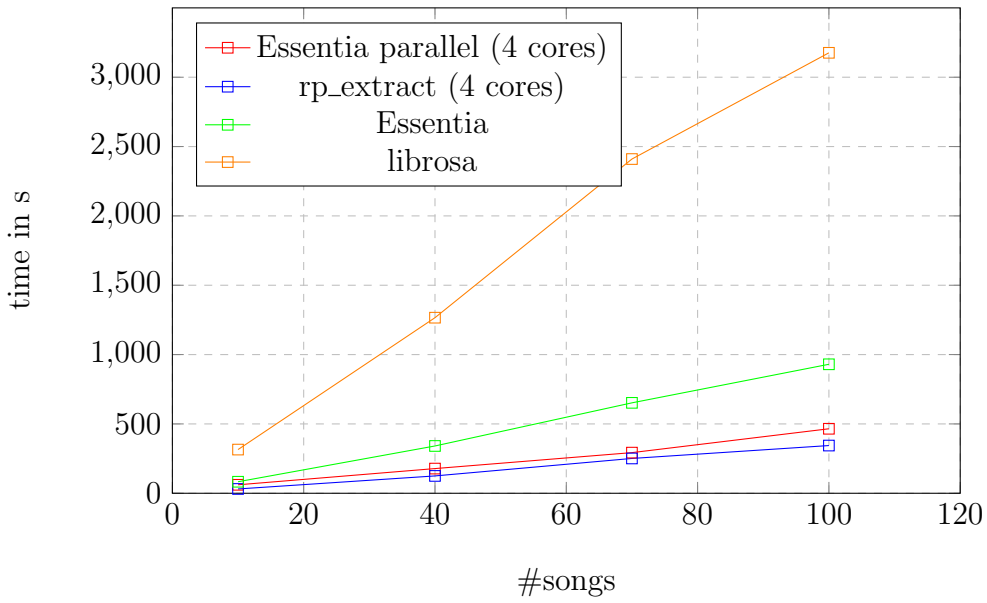


Figure 4.1: Performance of various toolkits on a single computer

In summary, the estimated time for the feature extraction of larger datasets on a single computer based on the performance measurements was extrapolated and is listed below, leading to the conclusion that the features for the full dataset including the FMA dataset can only be extracted with the help of a computer cluster.

Estimated feature extraction times

- 3h24min - 1517-Artists - Essentia parallel, single-node, 4 CPU cores
- 3h54min - 1517-Artists - rp_extract
- 9h06min - private dataset - Essentia parallel, single-node, 4 CPU cores
- 10h24min - private dataset - rp_extract
- (125h - all datasets - Essentia parallel, single-node, 4 CPU cores)
- (143h - all datasets - rp_extract)

Performance on a Cluster with mpi4py

For the extraction of the features from the audio files of the FMA dataset on the computer cluster of the Friedrich Schiller University in Jena (the "ARA-cluster"), Parallel Python had to be replaced with mpi4py (see Code Snippet 4.6). Mpi4py provides Python bindings for the Message Passing Interface standard (MPI) [74]. Every compute-process gets a rank number and recognizes the overall count of processes. With these two values, the file list of all audio files is split, and each process only processes its respective data. The audio files were stored in a parallel cluster file system called "beegfs" [75]. Similar

to the implementation using Parallel Python, every process stores the results in separate output files, each of them containing batches of 25 songs.

```
comm = MPI.COMM_WORLD # get MPI communicator object
size = comm.size # total number of processes
rank = comm.rank # rank of this process
status = MPI.Status() # get MPI status object
files_per_part = 25
start = 0
last = len(filelist)
parts = (len(filelist) / files_per_part) + 1
step = (last - start) / parts + 1
for index in xrange(start + rank, last, size):
    if index < parts:
        starti = start+index*step
        endi = min(start+(index+1)*step, last)
        parallel_python_process(index, filelist[starti:endi])
```

Code Snippet 4.6: Mpi4py

All audio files larger than 25MB were filtered out of the FMA dataset in advance, to avoid memory overflows, still leaving 102813 songs out of the 106733 songs to process. A total of 36 compute nodes were used. Every node had 192GB of RAM and 36 CPU cores (72 using hyper-threading (HT)). To increase the available memory per CPU core, only 18 CPU cores per node were used. So, overall, 648 processes were spawned. During the computation of the audio features with Essentia, 1 out of the 648 processes ran out of memory, so only 102793 out of the 102813 songs were processed in the end. For performance tests, this does not make a big difference, but for future work the feature extraction script should be adapted accordingly.

The ARA-cluster is managed with the help of the Slurm Workload Manager [76]. To submit the Essentia feature extraction script to the cluster, the following Slurm *.sbatch file was used to configure the cluster:

```
#!/bin/bash
#SBATCH --partition=s_standard
#SBATCH --time=08:00:00
#SBATCH -n 648
#SBATCH -N 36
#SBATCH --ntasks-per-node=18
#SBATCH --mem-per-cpu=10000
srun -n $SLURM_NTASKS --mpi=openmpi python mpi4py_ara_features.py
```

Code Snippet 4.7: Slurm *.sbatch file for feature extraction with Essentia on the ARA-cluster

Figure 4.2 shows the performance of the feature extraction on the ARA-cluster. The extraction of the features took between 1439 seconds (fastest process) and 1950 seconds (slowest). With better balancing and messaging between the processes, the tasks could be distributed in a way where idle tasks take parts of the file list from other tasks that are still processing.
For the extraction of the rhythm features with the rp_extract tool, the script of the TU Wien was adapted for usage with mpi4py as well. The same amount of processes gets spawned on the cluster (648), but each of the processes is able to make use of two CPU cores plus HT. The fastest process finished after 1657 seconds and the slowest one took 1803 seconds.

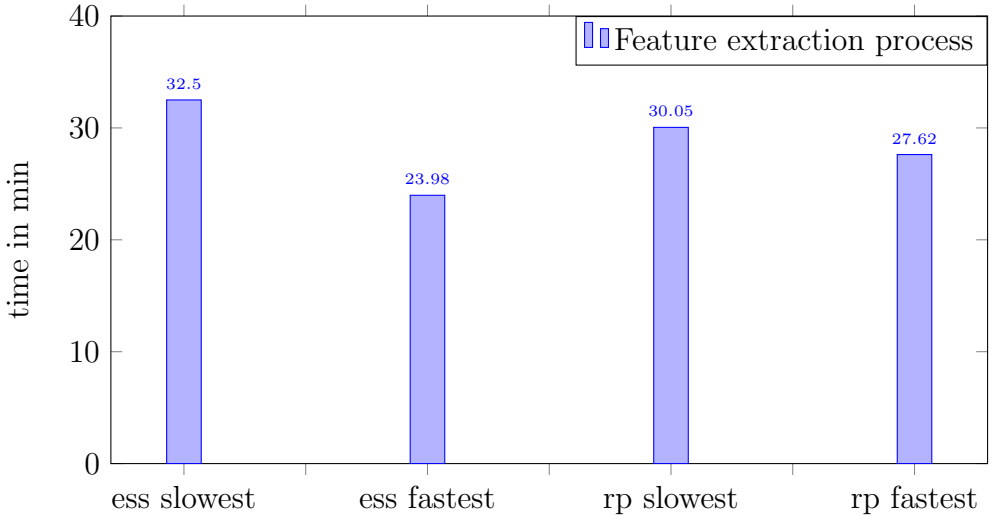


Figure 4.2: Feature extraction of the FMA dataset on the ARA-cluster (performance)

Total Amount of Songs

Due to the above-mentioned filtering of audio files larger than 25MB, the out-of-memory error of one process executing the Essentia task, and since the rhythm pattern extraction script does not handle some audio file formats like Ogg Vorbis, not all features from all songs could be extracted. So, in the end, the overall count of songs from the datasets listed in Table 4.1 where all features could be obtained is 114210.

4.3 Big Data Framework Spark

After all features are extracted, the next step is to load the feature files into the HDFS.

4.3.1 Feature Files

For the about 114000 songs all feature files sum up to about 11.2 GB (see Figure 4.3). Large streaming platforms like Spotify give access to about 50 million songs in their databases [77]. At this scale, the feature files would sum up to approximately 5 TB.

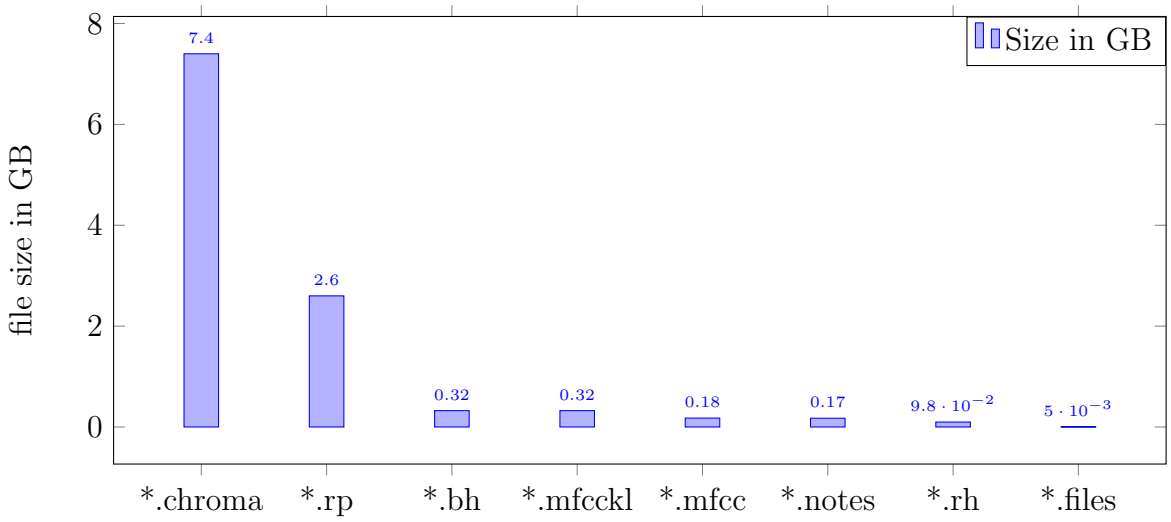


Figure 4.3: Feature file sizes

To calculate the distances based on timbral features, for each song the mean and variance vectors and covariance matrix has to be computed from the MFCCs. These are stored in two different output text files:

- out.mfcc (containing mean vector (length b), variance vector (size b) and vectorized upper triangular covariance matrix (length $\frac{b \cdot (b+1)}{2}$))
- out.mfcckl (containing mean vector (length b) and full covariance matrix (size $b \cdot b$)

The amount of mel bands chosen is $b = 13$. The second *.mfcckl file is created to dispose of the necessity to rearrange the covariance matrix inside the Big Data framework and reduce the computation time when a similarity estimation request is processed. To even further safe storage space the variance vector from the *.mfcc files could have been left out. These variance values are already stored within the main diagonal of the covariance matrix (as mentioned in Section 3.1.1) and left within the triangular matrix, leading to $\frac{b \cdot (b+1)}{2}$ instead of $\frac{b \cdot (b-1)}{2}$ values stored in the features files.
The melodic features are stored in two different output text files. The vector length is dependent on the numbers of detected beats n:

- out.notes (containing the estimated original key, the scale and a list of most dominant key per beat, key-shifted to the A key (size n))

- out.chroma (full beat-aligned and key-shifted chromagram, containing a $12 \times n$ matrix)

The rhythm features are stored in three different output text files:

- out.bh (containing the estimated overall bpm and a vector for the beat histogram normalized to one (size 250))
- out.rh (containing a vector for the rhythm histogram extracted with rp_extract (size 60))
- out.rp (containing a vectorized matrix for the rhythm patterns extracted with rp_extract (size 24×60))

An additional file containing a list of all song names is stored as

- out.files

4.3.2 Workflow

Although multiple different implementations were tested to evaluate the fastest and most efficient way to compute the similarities, all of these different approaches follow the same basic steps. These are presented in Figure 4.4.

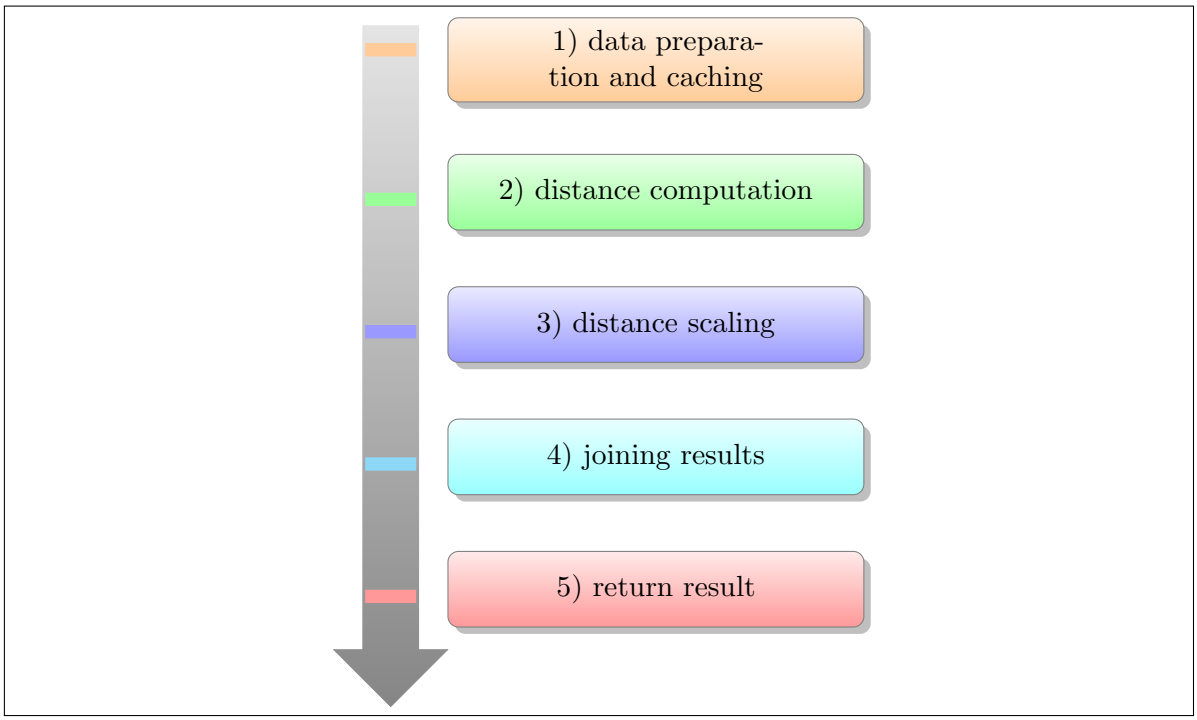


Figure 4.4: Workflow Spark

The following sections explain the various stages in more detail, also giving more details over a few subtle differences between the different implemented and tested approaches like the usage of RDDs, single DataFrames for each feature or one large DataFrame containing all features.

4.3.3 Data Preparation

The features are stored into many small text files as described in Section 3.4. Due to the fact that the features were extracted in parallel and in batches of only a few songs, each of the feature files only contains the features of a small number of songs. As many small files are inefficient to process with Spark, all files containing the same feature type are merged to one large file, before being loaded into the HDFS [51, p. 153]. By loading larger files into the HDFS, partitioning into data blocks is performed according to the standard parameters of the HDFS (e.g. 128 MB partitions). Additional repartitioning on the cluster is later performed with Spark by using the `rdd.repartition(repartition_count)` command. Finally, to work with the features a few transformations have to be performed on the data. The extracted note values are stored as lists of integers for example, each representing a certain note. To compare them using the Levenshtein distance, the lists are converted into strings (see Code Snippet 4.8).

```
1 chroma = sc.textFile("features/out[0-9]*.notes").repartition(repartition_count)
2 chroma = chroma.map(lambda x: x.split(';'))
3 chroma = chroma.map(lambda x: (x[0], x[1], x[2], x[3].replace("0",'A').replace("
      1",'B').replace("2",'C').replace("3",'D').replace("4",'E').replace("5",'F').
      replace("6",'G').replace("7",'H').replace("8",'I').replace("9",'J').replace(
      "10",'K').replace("11",'L'))).map(lambda x: (x[0], x[1], x[2], x[3].replace(
      ',','').replace(' ','')))
4 df = spark.createDataFrame(chroma, ["id", "key", "scale", "notes"])
```

Code Snippet 4.8: Notes preprocessing

All the other features are stored as lists of floats and have to be converted into vectors.

```
1 from pyspark.mllib.linalg import Vectors
2 list_to_vector_udf = udf(lambda l: Vectors.dense(l), VectorUDT())
3 rp = sc.textFile("features[0-9]*/out[0-9]*.rp").repartition(repartition_count)
4 rp = rp.map(lambda x: x.split(","))
5 kv_rp = rp.map(lambda x: (x[0].replace(";","").replace(".","").replace(",","").
      replace(" ",""), list(x[1:])))
6 rp_df = spark.createDataFrame(kv_rp, ["id", "rp"])
7 rp_df = rp_df.select(rp_df["id"],list_to_vector_udf(rp_df["rp"]).alias("rp"))
```

Code Snippet 4.9: Rhythm patterns preprocessing

The Spark ML library and the older MLlib library offer sparse and dense vectors as data types. The only feature type that contains a lot of zeros, where sparse vectors could improve performance, is the beat histogram. Compared to other features like the chromagram, the beat histogram vectors are relatively small, with a length of only 200 values, so all lists including the beat histograms are converted to dense vectors

by calling `Vectors.dense()`. An example is given for the rhythm pattern features in Code Snippet 4.9. The data is read out of the HDFS into an RDD and repartitioned with `sc.textFile("feaures.txt").repartition(repartition_count)`. The repartitioning is optional but improves the overall performance (see Section 4.3.7). After the execution of the pre-processing steps, the RDD can be converted into a Spark SQL DataFrame by calling `spark.createDataFrame(rdd)`, to ease up the access to data and improve code readability. The features can then be accessed via column names instead of the RDD indices.
For performance tests, three different kinds of implementations were tested. The first one merges all audio features into one large DataFrame in the beginning and persists this to the main memory. The second implementation uses single DataFrames for each feature set, and the third uses RDDs instead of DataFrames. The results of the performance analysis of DataFrames vs. RDDs are given in Section 4.3.7.

4.3.4 Distance Computation

After the data preparation, the similarities between a requested single song and all other songs in the database can be calculated. The code differs slightly when RDDs instead of DataFrames are used. As already mentioned, the full source code is attached in the appendices on the included CD and can be checked out from GitHub [73]. Most of the following code examples were written for the usage with DataFrames. The examples for usage with RDDs are annotated accordingly.

Euclidean Distance

```
1 from scipy.spatial import distance
2 from pyspark.sql import functions as F
3 distance_udf = F.udf(lambda x: float(distance.euclidean(x, comparator_value)), ↩
      FloatType())
4 result = feature_vec_df.withColumn('distances', distance_udf(F.col('features')))
5 result = result.select("id", "distances").orderBy('distances', ascending=True)
6 result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.10: Euclidean distance DF

The Euclidean distance is used as a metric to compute the distances between vectors of beat histograms, rhythm histograms, rhythm patterns, and MFCCs, making it the most versatile distance measurement introduced in this thesis. To compute the Euclidean distance in Spark, a user-defined function (UDF) gets declared (see Code Snippet 4.10). This UDF is then applied to all elements of the `'features'` column. Inside the UDF,

the Euclidean distance is computed using Python's scipy library. The `comparator_value` variable contains the feature of the requested example song to which the distances are calculated. Assuming that all features were merged into one large DataFrame (`fullFeatureDF`) and cached to the main memory, the `comparator_value` can be found by filtering the DataFrame for the requested song's ID (e.g., the pathname of the original song).

```
1 song = fullFeatureDF.filter(fullFeatureDF.id == songname).collect()
2 comparator_value = song[0]["mfccEuc"]
```

Code Snippet 4.11: Filter for requested song

When working with RDDs instead of DataFrames, the computation of the distances between the feature vectors is performed with a `map()` instead of a UDF (see Code Snippet 4.12).

```
1 resultRP = rp_vec.map(lambda x: (x[0], distance.euclidean(np.array(x[1]), np.
      array(comparator_value))))
```

Code Snippet 4.12: Euclidean distance RDD

Bucketed Random Projection

As an alternative to the Euclidean UDF, Spark offers an implementation of a locality-sensitive hashing (LSH) family for the Euclidean distance called "Bucketed Random Projection" (BRP). The Spark API documentation describes the idea behind LSH: "The general idea of LSH is to use a family of functions ("LSH families") to hash data points into buckets, so that the data points which are close to each other are in the same buckets with high probability, while data points that are far away from each other are very likely in different buckets" [78]. The BRP projects the feature vectors x onto a random unit vector v and portions the projected result into hash buckets with the bucket-length r, resulting in the equation

$$h(x) = \left\lfloor \frac{x \cdot v}{r} \right\rfloor . \tag{4.2}$$

"A larger bucket length (i.e., fewer buckets) increases the probability of features being hashed to the same bucket (increasing the numbers of true and false positives)." [78] The method `model.approxNearestNeighbors(dfA, key, k)` searches for the `k` nearest neighbors of `dfA` to the `key`, but the Spark API documentation mentions that, "Approximate nearest neighbor search will return fewer than k rows when there are not enough candidates in the hash bucket." [78] This means that the smaller (and therefore more precise) the

bucket length is, the fewer nearest neighbors get returned by this function. This is problematic when searching for the nearest neighbors of different feature sets because the resulting distances calculated from the different kinds of features have to be joined to get the resulting similarities as a combination of different distance measurements (see Section 4.3.6). If the BRP only returns a handful of nearest neighbors, the overall distances to all the other songs can not be determined.
Due to the fact that the ARA-cluster is running with PySpark version 1.6.0 and the Bucketed Random Projection was introduced later with PySpark version 2.2.0, the algorithm could only be tested on the single-node test platform using Code Snippet 4.13, where it performed worse than the naive Euclidean implementation from Code Snippet 4.10 on a dataset consisting of about 11500 songs. Whether the BRP outperforms the naive approach on a cluster with larger datasets could be investigated further.

```
1 from pyspark.ml.feature import BucketedRandomProjectionLSH
2 brp = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", seed
      =12345, bucketLength=100.0)
3 model = brp.fit(feature_vec_df)
4 comparator_value = Vectors.dense(comparator[0])
5 result = model.approxNearestNeighbors(feature_vec_df, comparator_value,
      feature_vec_df.count()).collect()
6 rf = spark.createDataFrame(result)
7 result = rf.select("id", "distCol").rdd.flatMap(list).collect()
```

Code Snippet 4.13: Bucketed Random Projection

Cross-Correlation

As laid out in Section 3.2, there are different options to calculate the cross-correlation of the beat-aligned chroma features. The chroma features are already key-shifted to a common key, but the possibility to perform a full 2D-cross-correlation with additional key-shifting as explained in Equations (3.12), (3.13), and (3.14) still exists. Due to the fact that the computation of the cross-correlation already takes the longest time, even without additional key-shifting (see Section 4.3.7), the implementation on the cluster and in Code Snippet 4.15 calculates the simplified cross-correlation (Equations (3.15) and (3.17)). Whether or not the results are compromised because of that is left open and requires further investigation.
The cross-correlation was used to detect cover songs on the same dataset Ellis and Cotton used in their paper [65]. The results are presented in Section 5.1.2. There are some differences in the resulting recommendations compared to the original paper. These can be explained with the different underlying beat tracking, different filter parameters, and a few improvements that are left out compared to the implementation

of Ellis [65] as mentioned in Section 3.2.1.

```
corr = scipy.signal.correlate2d(chroma1, chroma2, mode='valid')
```

Code Snippet 4.14: Cross-correlation scipy

Concerning the actual implementation of the cross-correlation, two different libraries were tested. Code Snippet 4.14 shows the cross-correlation function coming with the scipy library.

```
from scipy.signal import butter, lfilter, freqz, correlate2d, sosfilt
import numpy as np
def cross_correlate(chroma1, chroma2):
    length1 = chroma1_par.size/12
    chroma1 = np.empty([12, length1])
    length2 = chroma2_par.size/12
    chroma2 = np.empty([12, length2])
    if(length1 > length2):
        chroma1 = chroma1_par.reshape(12, length1)
        chroma2 = chroma2_par.reshape(12, length2)
    else:
        chroma2 = chroma1_par.reshape(12, length1)
        chroma1 = chroma2_par.reshape(12, length2)
    correlation = np.zeros([max(length1, length2)])
    for i in range(12):
        correlation = correlation + np.correlate(chroma1[i], chroma2[i], "same")
    #remove offset to get rid of initial filter peak (highpass filter jump 0-20)
    correlation = correlation - correlation[0]
    sos = butter(1, 0.1, 'high', analog=False, output='sos')
    correlation = sosfilt(sos, correlation)[:]
    return np.max(correlation)
distance_udf = F.udf(lambda x: float(cross_correlate(x, comparator_value)), ↩
    DoubleType())
result = df_vec.withColumn('distances', distance_udf(F.col('chroma')))
result = result.select("id", "distances").orderBy('distances', ascending=False)
result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.15: Cross-correlation numpy

The parameter `mode='valid'` when using scipy, determines whether or not additional key shifting is included. The `'valid'` option already includes additional key-shifting but without zero-padding. Other options would be `mode='same'`(no key-shifting) and `mode='full'` (with zero-padding).

The other variant is shown in Code Snippet 4.15. It uses the numpy library. Although

numpy only offers a 1D-cross-correlation function, which had to be nested inside a for-loop to get the 2D-cross-correlation, performance tests showed that the numpy version was faster than the scipy version by orders of magnitude. Calculating and scaling the distances of the chroma features from one song to about 114000 other songs took about 22 seconds with numpy and approximately 725 seconds with scipy on the ARA-cluster.

Jensen-Shannon-Like Divergence

While computing the Jensen-Shannon-like divergence, for some of the MFCC features, a problem with negative determinants was encountered. Because the logarithm of negative numbers is not defined, no similarity for these features could be calculated. Schnitzer mentioned a problem with "skyrocketing values of determinants, which lead to inaccurate results" [22, p.45]. He proposed a solution by using the sum of the upper triangular matrix of the Cholesky decomposition to compute the logarithm of the determinant of the covariance matrix in Equation (3.7). This approach was also considered for the encountered issue mentioned above but ultimately did not work out because of the covariance matrices causing the error not being positive definite.

```
1  import numpy as np
2  def jensen_shannon(vec1, vec2):
3    #preprocessing: splitting vec1 and vec2 into mean1, mean2, cov1 and cov2
4    mean_m = 0.5 * (mean1 + mean2)
5    cov_m = 0.5 * (cov1 + mean1 * np.transpose(mean1)) + 0.5 * (cov2 + mean2 * np.
         transpose(mean2)) - (mean_m * np.transpose(mean_m))
6    div = 0.5 * np.log(np.linalg.det(cov_m)) - 0.25 * np.log(np.linalg.det(cov1))
         - 0.25 * np.log(np.linalg.det(cov2))
7    if np.isnan(div):
8      div = np.inf
9    return div
10 distance_udf = F.udf(lambda x: float(jensen_shannon(x, comparator_value)),
       DoubleType())
11 result = df_vec.withColumn('distances', distance_udf(F.col('features')))
12 result = result.filter(result.distances_js != np.inf)
13 result = result.select("id", "distances").orderBy('distances', ascending=True)
14 result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.16: Jensen-Shannon-like divergence

Because no immediate solution to that problem was found, the rows where this issue appears just get filtered out by setting the distance to `np.inf` and later dropping these rows. This problem seems to appear for about 5-10% of the distances calculated with

the Jensen-Shannon divergence. Further investigation to solve this problem would be necessary. An example is given in Code Snippet 4.16.

Symmetric Kullback-Leibler Divergence

When implementing the symmetric Kullback-Leibler divergence a few interesting observations were made. First of all, this metric seems to be prone to outliers. While only very few distances get disproportionately large (around 10^6), most of the distances lie between 0 and 100. The large outliers lead to problems when scaling the resulting distances to an interval between 0 and 1 (see Section 4.3.5 and 5.1.1). As a temporary solution all distances larger than a certain threshold get filtered out.
Secondly when using the FMA dataset a few of the songs returned an error, where the covariance matrix could not be inverted. These songs get filtered out as well. The example code for the calculation of distance using DataFrames can be seen in Code Snippet 4.17.

```
1  import numpy as np
2  def symmetric_kullback_leibler(vec1, vec2):
3    #preprocessing: splitting vec1 and vec2 into mean1, mean2, cov1 and cov2
4    try:
5      d = 13
6      div = 0.25 * (np.trace(cov1 * np.linalg.inv(cov2)) + np.trace(cov2 * np.
           linalg.inv(cov1)) + np.trace((np.linalg.inv(cov1) + np.linalg.inv(cov2))
            * (mean1 - mean2)**2) - 2*d)
7    catch:
8      div = np.inf
9      print("ERROR: NON INVERTIBLE SINGULAR COVARIANCE MATRIX\n")
10   return div
11 distance_udf = F.udf(lambda x: float(symmetric_kullback_leibler(x,
       comparator_value)), DoubleType())
12 result = df_vec.withColumn('distances', distance_udf(F.col('features')))
13 #thresholding for outliers
14 result = result.filter(result.distances <= 100)
15 result = result.select("id", "distances").orderBy('distances', ascending=True)
16 result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.17: Kullback-Leibler divergence

After implementing this similarity measurement in Spark, some tests and comparisons to the results of the Musly toolkit [14] were done. While overall the genre recall is quite good (see Section 5.1.3) and the results seem reasonable, they do differ from the ones returned by Musly. These differences could be explained with the choice of only 13 mel bands during the computation of the MFCCs in this thesis compared to the 25 bands

in Musly [14] and some other decisions like omitting the normalization with mutual proximity (Section 3.1.2). The same applies for the Jensen-Shannon-like divergence.

Levenshtein Distance

Spark already offers a function for the computation of the Levenshtein distance if the feature vectors are stored in a DataFrame. The Levenshtein distance can then be computed between two columns for all rows. Code Snippet 4.18 shows a minimal example.

```
1 from pyspark.sql.functions import levenshtein
2 df_merged = featureDF.withColumn("compare", lit(comparator_value))
3 df_levenshtein = df_merged.withColumn("word1_word2_levenshtein", levenshtein(col
      ("notes"), col("compare")))
4 df_levenshtein.sort(col("word1_word2_levenshtein").asc()).show()
```

Code Snippet 4.18: Levenshtein DataFrame

As an alternative for the RDD based variant of the Spark application, the Python wrapper [79] for the C/C++ library "edlib" [80] was used. During initial tests, when experimenting with a naive implementation of the Levenshtein distance using a Python function with numpy, immense performance issues were encountered. Due to the underlying C/C++ code of the edlib the computation of the Levenshtein distance in Code Snippet 4.19 performes comparably well as the Spark-native DataFrame equivalent and offers a good alternative.

```
1 import edlib
2 def naive_levenshtein(seq1, seq2):
3     result = edlib.align(seq1, seq2)
4     return(result["editDistance"])
5 #...
6 resultNotes = notes.map(lambda x: (x[0], naive_levenshtein(str(x[1]), str(
      comparator_value)), x[1], x[2]))
```

Code Snippet 4.19: Levenshtein RDD

Lazy Evaluation and Data Caching

As described in Section 2.6.2, Spark's main advantage is its ability to use the main memory of the nodes in a cluster to safe intermediate data without the need of writing it back to the disk. However Spark does not automatically cache the data. RDDs and DataFrames have to be explicitly assigned to the main memory, by either calling `persist()` (optionally with the parameter `storageLevel=StorageLevel.MEMORY_ONLY_SER`) or

`cache()` and even then Spark only takes this as a suggestion. If not enough main memory is available, the data is still written onto the hard drives.
As introduced in Section 2.6.2, Spark also uses an optimization technique called "lazy evaluation" that differentiates between transformations on data and actions. The `cache()` and `persist()` commands both do not count as actions. Instead they are executed only when an actual action on the data is called. This has to be kept in mind when optimizing Spark applications and evaluating the performance by measuring execution times. The Code Snippet 4.20 gives a short example.

```
1  import time
2  #...
3  featureDf = preprocess_features().persist() #p3
4  print(featureDf.first()) #l4
5  tic1 = int(round(time.time() * 1000))
6  neighbors = get_distances(songname, featureDf).persist() #p6
7  neighbors = neighbors.orderBy('scaled_dist', ascending=True).persist() #p7
8  neighbors.show()
9  neighbors.toPandas().to_csv("neighbors.csv", encoding='utf-8')
10 neighbors.unpersist()
11 tac1 = int(round(time.time() * 1000))
12 time_dict['time: ']= tac1 - tic1
13 print time_dict
```

Code Snippet 4.20: Spark lazy evaluation

The function `preprocess_features()` is a function, where the chroma features get read into RDDs, pre-processed, repartitioned, and converted into a DataFrame. The function `get_distances()` calculates all distances between the song belonging to the ID `songname` and the other 114209 songs in the database. Within this function, the results are then scaled to an interval between 0 and 1 by dividing all distances by their maximum value. The result is stored in the DataFrame `neighbors` and after that, two actions are performed subsequently on this result. The first (`show()`) prints the 20 nearest neighbors to the standard output (e.g., the pyspark shell). The second action (`toPandas().to_csv()`) prints the whole list of all 114210 distances into a *.csv file.
In a simple experiment, the impact of ineffective caching and the impact of the lazy evaluation on time and performance tests is shown. The results are plotted in Figure 4.5. The first bar (labeled with "opt") shows the `print time_dict` output when executing the full code from Code Snippet 4.20. In the second bar (labeled with "p6"), the `persist()` command in line 6 got removed. Due to the fact that the scaling of the distances inside the function `get_distances()` requires an action on the data stored in `neighbors` but the results are no longer persistent in the cache, this part of the code has to be executed twice (in line 6 and again in line 7). For the third bar (labeled with "p7"), the `persist()`

command in line 7 is removed as well. The result `neighbors` is no longer stored in the main memory, and every time an action requires the results of this DataFrame, it has to be recalculated which is the case for both actions in line 8 and line 9 in the code example.
When further removing the print command in line 4, the lazy evaluation no longer executes line 3 before starting to measure the time in line 5 because the action `first()` is no longer executed on the DataFrame. Instead, line 3 gets called later, when `get_distances()` is executed, because only then an action on the `featureDf` DataFrame is called for the first time. This is shown in the bar labeled with "l4". Up until this point, the original `featureDf` still gets persisted to the main memory but if the `persist()` command in line 3 gets removed as well in the last test labeled with "p3", `preprocess_features()` has to be executed every time the `featureDf` is needed.

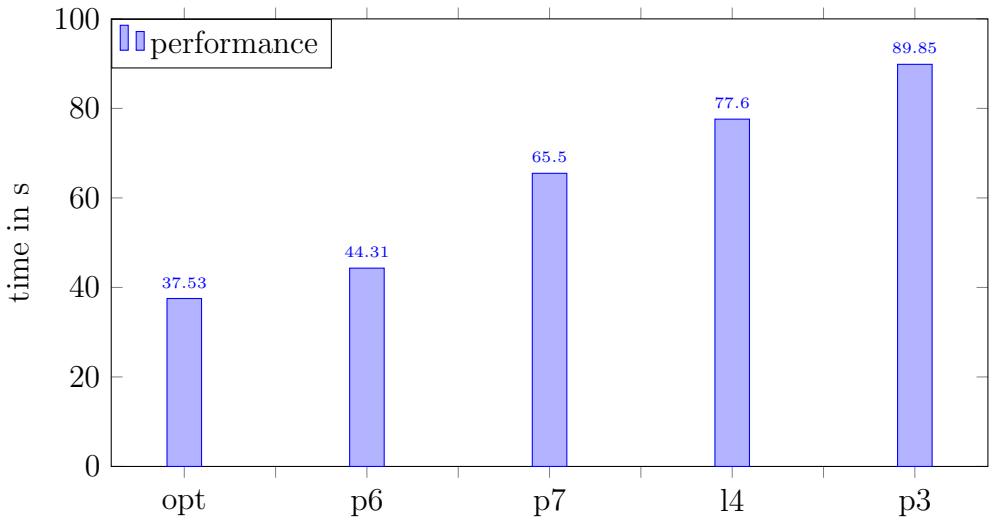


Figure 4.5: Lazy evaluation and caching optimization

In summary, finding the correct way of caching the data is a tricky task. Writing everything into the main memory is no solution because then the cluster will run out of memory eventually. As a rule of thumb, the best way to persist data is to cache it every time more than one subsequent action is performed on it.
That means that especially in this application in the area of music similarity, all pre-processed features have to fit into the main memory of the cluster to speed up consecutive song requests.

4.3.5 Distance Scaling

To combine different distance measurements into one combined distance, the various results from different kinds of features have to be rescaled to avoid biasing the overall distance. The easiest way is to subtract the minimum $\min(d)$ from all distances d and

to divide by the difference between the maximum $\max(d)$ and the minimum distance, as described in the equation

$$d' = \frac{d - \min(d)}{\max(d) - \min(d)}. \tag{4.3}$$

The minimum distance should always be the self-similarity of the requested song with a value of 0. But in the implementation of the symmetric Kullback-Leibler distance, this is not always the case. Sometimes the self-similarity is just very close to zero. The analysis of the distances in Section 5.1.1 also shows that, e.g. the Levenshtein distances and cross-correlation results are unevenly distributed over the unit interval $[0, 1]$. Dropping the self-similarity out of the distance vector and rescaling it afterwards with a new minimum distance unequal to zero could solve this, but was not tested in this thesis. A second issue was already mentioned in Section 4.3.4, where outliers tend to bias the results. These can get filtered out before rescaling the distances. This is further evaluated in Section 5.1.1.

Another option to rescale the features, laid out by Sebastian Stober in [3, pp. 543ff], but not implemented in this thesis, would be to rescale all distances to have a mean value of 1 by using

$$d' = \frac{d}{\mu_f}, \tag{4.4}$$

and by dividing the distances d by the mean distance μ_f. Outliers should be detected and removed before calculating the mean distance. A better way to rescale the data could be evaluated in future research.

Implementation-wise the aggregation of the minimum and maximum value went through different tests.

```
1 max_val = result.agg({"distances": "max"}).collect()[0]
2 max_val = max_val["max(distances)"]
3 min_val = result.agg({"distances": "min"}).collect()[0]
4 min_val = min_val["min(distances)"]
```

Code Snippet 4.21: Minimum and maximum aggregation separate

During the first tests, the aggregation of minimum and maximum value were performed separately (see Code Snippet 4.21). This turned out to be very inefficient because the data had to be accessed multiple times. An improved version, shown in Code Snippet 4.22, only uses one action to gather minimum and maximum value, which improved the overall performance significantly. Another alternative would be the usage of the `df.describe()` function for DataFrames. For the implementation using RDDs the `rdd.stats()` function was used, returning minimum, maximum, mean, and variance values all at once.

```
from pyspark.sql import functions as F
aggregated = result.agg(F.min(result.distances),F.max(result.distances))
max_val = aggregated.collect()[0]["max(distances)"]
min_val = aggregated.collect()[0]["min(distances)"]
```

Code Snippet 4.22: Minimum and maximum aggregation optimized

4.3.6 Combining Different Measurements

To finally compute the overall similarity of what Stober calls the facet distances (the different distances computed using different feature sets) in [3, pp. 543ff], the weighted arithmetic mean of the previously scaled facet distances is calculated by using the equation

$$\text{dist} = \frac{\sum_{m=0}^{M-1} w_m \cdot d_m}{\sum_{m=0}^{M-1} w_m} \tag{4.5}$$

given M different distances $d_1, d_2, ..., d_m$ and weights $w_1, w_2, ..., w_m$. In this thesis, only binary weights were tested by either including a facet distance with a weight of one or just leaving it out of the overall similarity by setting its weight to zero. The impact of different weights is left open for future research.

4.3.7 Performance

Cluster Configuration

The first optimization step was to alter the spark cluster configuration for the ARA-cluster, as described in Section 2.6.2. The cluster configuration in the Code Snippet 4.23 turned out to perform well compared to other test configurations. The cluster is configured in a way where between 16 and up to 32 Executors are spawned with each Executor requesting as many CPU cores and memory resources as possible. The `repartition_count` variable is used with the `repartition()` method during the data preparation stage to evenly distribute all chunks of feature files across the cluster.
With the help of the `spark.dynamicAllocation` parameters, the number of Executors spawned can be determined [51, p. 153]. While normally the Executors are spawned and then retained for the life span of the application, dynamic allocation allows Spark to free resources of idling Executors and to then reassign the pertinent system resources. It should be mentioned that normally `spark.shuffle.service.enabled` should also be set to true when the dynamic allocation is used, and an external shuffle service should be configured to avoid the loss of shuffle data in case an Executor gets deleted. During the tests this option was disabled, though. This should not be a problem because

the dynamic allocation is only used to ensure that a certain fixed minimum amount of Executors is spawned. With this configuration, no more than 16 Executors can be spawned anyway because of the missing resources on the ARA-cluster, so for this configuration, the Executors never actually get killed and no shuffling data gets lost.

```
confCluster = SparkConf().setAppName("MusicSimilarity Cluster")
confCluster.set("spark.driver.memory", "64g")
confCluster.set("spark.executor.memory", "64g")
confCluster.set("spark.driver.memoryOverhead", "32g")
confCluster.set("spark.executor.memoryOverhead", "32g")
#confCluster.set("yarn.nodemanager.resource.memory-mb", "196608")
confCluster.set("spark.yarn.executor.memoryOverhead", "4096")
confCluster.set("spark.driver.cores", "32")
confCluster.set("spark.executor.cores", "32")
#confCluster.set("spark.shuffle.service.enabled", "True")
confCluster.set("spark.dynamicAllocation.enabled", "True")
#confCluster.set("spark.dynamicAllocation.initialExecutors", "16")
#confCluster.set("spark.dynamicAllocation.executorIdleTimeout", "30s")
confCluster.set("spark.dynamicAllocation.minExecutors", "16")
confCluster.set("spark.dynamicAllocation.maxExecutors", "32")
confCluster.set("yarn.nodemanager.vmem-check-enabled", "false")
repartition_count = 32
```

Code Snippet 4.23: Cluster setup

The Spark driver program is executed on the ARA-cluster login-node on which also software from other clients runs, possibly influencing the results of the performance tests.

Fine-tuning the cluster settings is a tricky task. Increasing the number of Executors also increases the additional overhead of managing the Executors and shuffling the data, while on the other side more unique tasks are being distributed over the compute nodes. To get a performant cluster configuration, various other cluster settings were tested. Increasing the `repartition_count` and the amount of Executors spawned (with fewer resources each) seemingly increased the overhead and network traffic on the cluster without reducing the overall computation time. Increasing the `repartition_count` while keeping the Executors the same size as in the Code Snippet turned out to be slower as well.

Although each node on the ARA-cluster has 36 CPU cores, only 32 cores were assigned to each Executor because this turned out to perform just a little bit better when calculating the similarities for only one song in the first tests. Therefore the cluster configuration was set as described in the Code Snippet 4.23 for the following tests in

this section, to keep the tests comparable to each other.
Later, when calculating the similarities on already cached feature data for consecutive song requests, 36 cores per Executor performed slightly better than 32 cores. Increasing the CPU core count to 72 per Executor performed far worse.

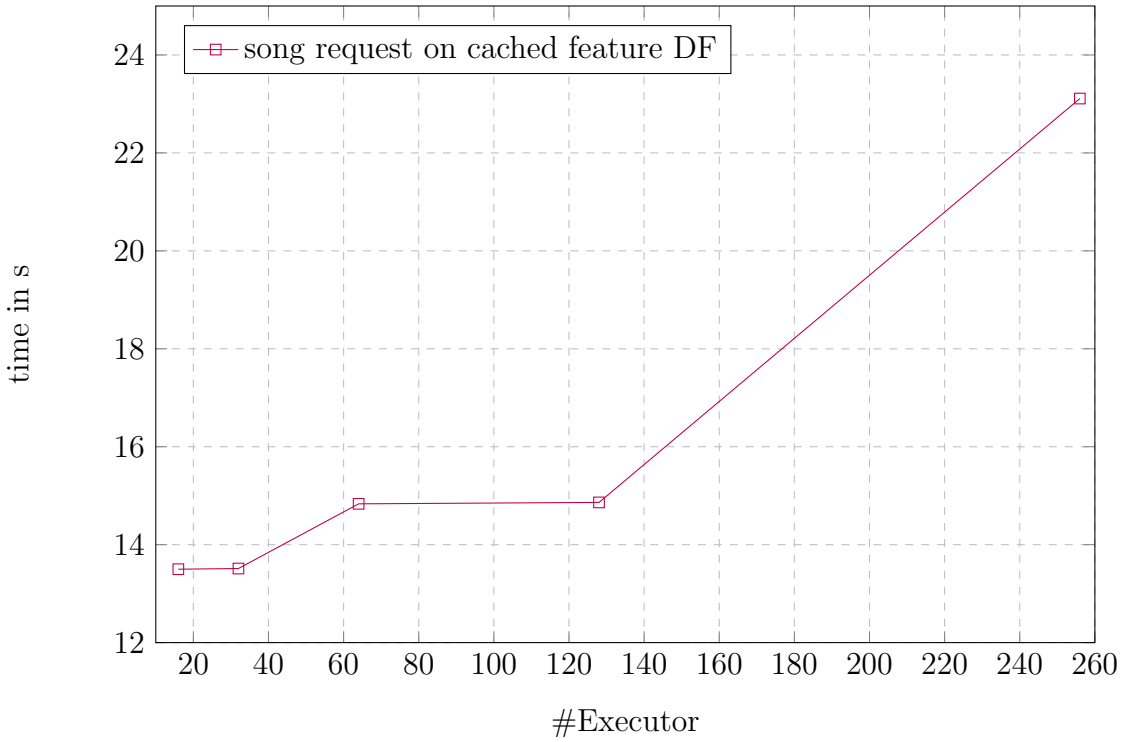


Figure 4.6: Performance depending on the #Executors spawned

Figure 4.6 shows the execution time of one full song request for all features on all 114210 songs on an already cached large DataFrame containing all of the different features (this approach is explained more detailed later on, see Figure 4.10). The x-axis shows the numbers of Executors that are spawned on the cluster. Since there are limited resources on the cluster, the number of CPU cores assigned to each Executor decreases when more Executors get spawned. In total there are 576 cores on 16 nodes, so the number of CPU cores per Executor can be calculated as $\#\text{CPUs} = \frac{576}{\#\text{Executors}}$. The available main memory per node (192GB) is split equally. The large DataFrame is cached and split in twice as many parts as Executors are spawned. Thus each Executor has to handle two data chunks.

Differences Between the Feature Types

Due to the different complexity of the various similarity measurements and metrics, the time needed to calculate the distances between all songs and a single requested song

differs for the various feature types. The computation time for all feature types (with respect to the lazy evaluation as described in Section 4.3.4) is pictured in Figure 4.7.

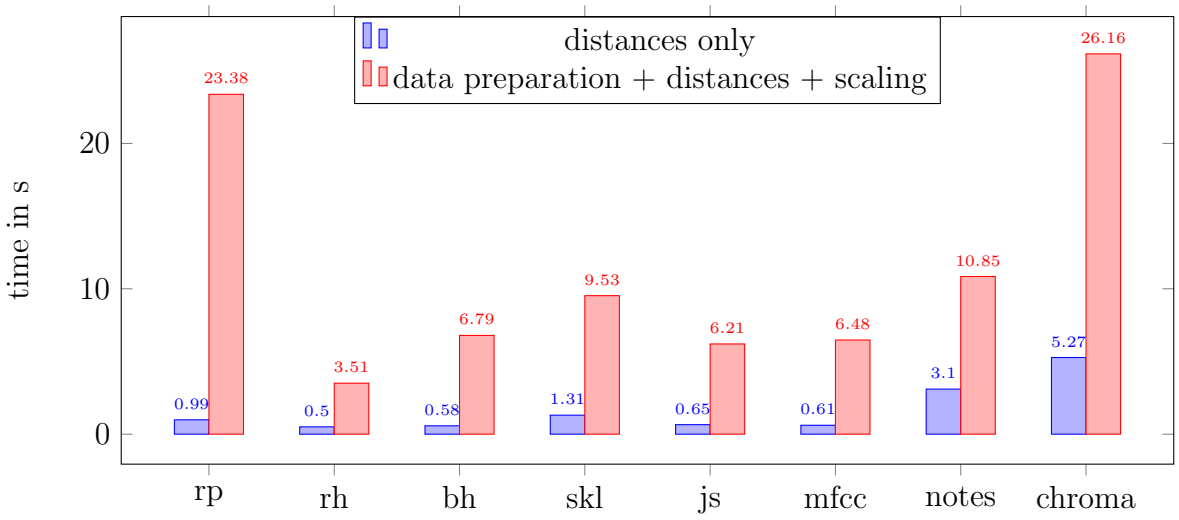


Figure 4.7: Performance of different feature types

The blue bars figure the computation time required to compute the distances between one requested song and all 114210 songs in the dataset without loading the data and without scaling. That means the features are already stored in the main memory. The measured times for the whole computation of the similarities for each feature set, including the data time taken for pre-processing and the scaling of the results to the unit interval, are shown in the red bar. The plot shows the importance of proper caching for fast response times. The labels on the x-axis represent the different distance measurements and are used further throughout this thesis, mainly in different plots.

- rp (rhythm patterns, Euclidean distance)
- rh (rhythm histogram, Euclidean distance)
- bh (beat histogram, Euclidean distance)
- skl (MFCCs, symmetric Kullback-Leibler divergence)
- js (MFCCs, Jensen-Shannon-like divergence)
- mfcc (MFCCs, Euclidean distance)
- notes (notes, Levenshtein distance)
- chroma (beat-aligned chromagram, cross-correlation)

Data Representation

Figure 4.8 and 4.9 show the performance of three different approaches on the ARA-cluster for different combinations of features (see caption).
For the approach annotated with ”Merged DF” all features are pre-processed, joined and stored in one large DataFrame that then gets repartitioned across all nodes and

cached into the main memory. The idea behind this approach is to reduce shuffling operations during the computation of similarities by bringing all feature types of the same songs to the same compute nodes. The downside of this method is a higher initial workload that has to be endured during the pre-processing stage.

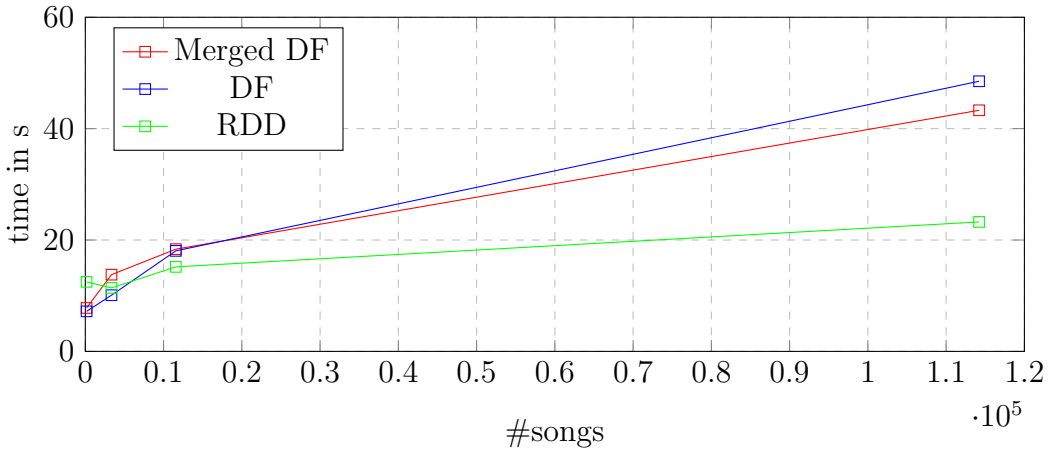


Figure 4.8: Performance ARA, full workload, (MFCC + Notes + RP)

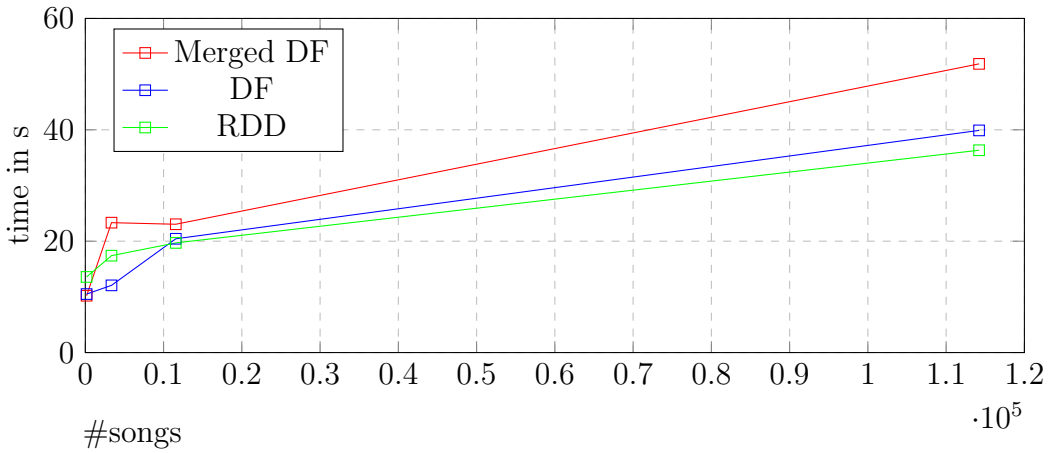


Figure 4.9: Performance ARA, full workload, (JS + Chroma + RP)

Once the pre-processing of the features is done, the similarities between the songs are computed, and the results are stored in new, smaller DataFrames, one for each feature type. Due to the previous joining of feature data by their song IDs, repartitioning, and caching, the distances of the same songs but for different feature types are in theory calculated on the same node, reducing unnecessary shuffling operations during the compute time. The resulting small DataFrames containing the facet distances of one feature set are then joined by song IDs once all similarities are computed. Then the joined results are scaled using only one `agg()` call for all feature types (see Section 4.3.5), and the combined distances are summed up and sorted. Figure 4.10 shows the adapted

workflow (original, see Figure 4.4) of this approach.

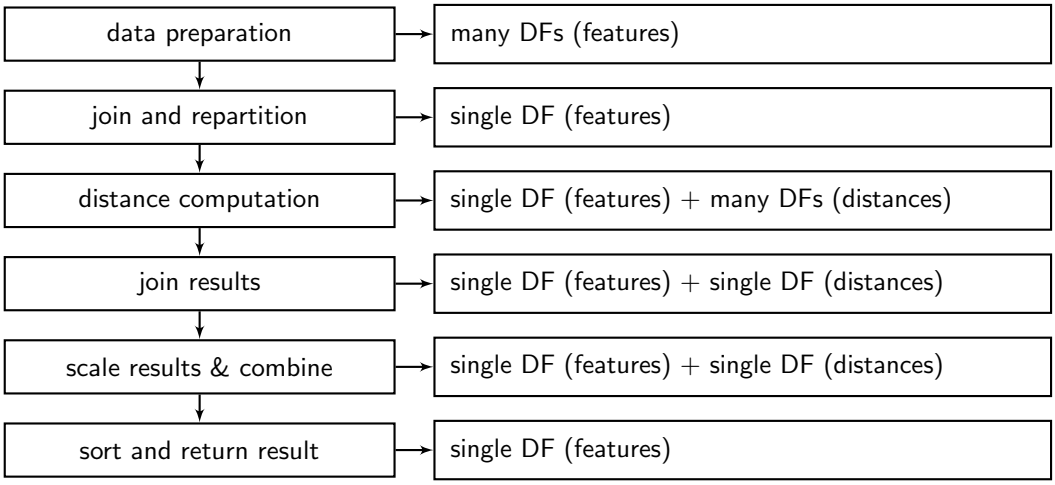


Figure 4.10: Workflow of Merged DF approach

The second approach annotated with "DF" also uses DataFrames, but stores the different pre-processed feature types in separate smaller DataFrames instead. This increases data shuffling during the computation of the similarities but has less initial overhead during the pre-processing stage.
The third approach does not use DataFrames at all but uses single RDDs for the pre-processed features instead. This approach has no additional overhead during the pre-processing stage, but the code is harder to read, and the workload during the computation of the similarities is also higher.
Each of the timespans measured in Figure 4.8 and 4.9 cover the full workflow, including data pre-processing, calculating, scaling, and combining all similarities for a single song request. The plots show the time required to compute the similarities for that single requested song for growing datasets starting from 163 (covers80) to 114210 songs (all datasets combined). Unsurprisingly the Merged DF approach performed relatively poorly compared to the other approaches due to its initial overhead. The next section will show this poor performance balanced out when presenting the performance on the calculation of subsequent song requests on the same, already cached and pre-processed features.

Performance of Subsequent Song Requests

In contrast to the performance analysis from the last section, Figure 4.11 shows the time measured to process two subsequent song requests. That means that the second consecutive song request is able to use the already pre-processed and cached feature data.

The plots annotated with ”Merged DF total”, ”DF total” and ”RDD total” depict the overall computation time including the pre-processing and the handling of both song requests. The other graphs show the computation time of only the second song request on persisted data, including calculation of distances, scaling, and join operations of the different result-types.

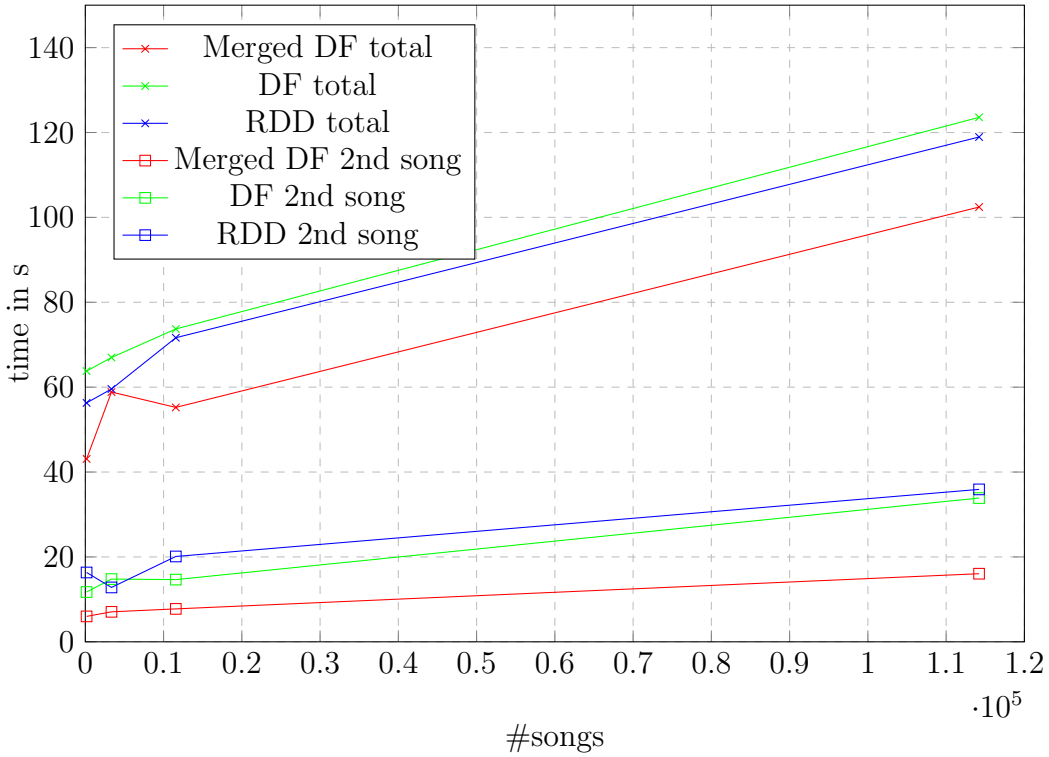


Figure 4.11: Performance of two subsequent song requests, all features

The results show that the pre-merged DataFrame approach performs best, returning the 20 nearest neighbors for the second song request in about 16 seconds and 14 seconds when using 36 cores per Executor (as mentioned in Section 4.3.7).

Descending Importance Filter and Refine

To improve performance even further, a filter and refine method was tested. The similarities are computed for one feature set at a time, and all songs to which the distance is larger than the mean value of these distances get filtered out of the feature DataFrame. From the thinned-out dataset, another less important feature set is chosen, and this is repeated until all feature sets were used. The implementation is based on the ”Merged DF” approach described and pictured in Figure 4.10 earlier, but with a few changes applied. After all features are pre-processed, joined and repartitioned, this large feature DataFrame gets cloned and persisted to the main memory as well. It is important that the cluster has enough main memory available to cache the full feature

DataFrame twice. The first feature set is chosen, distances are calculated and appended to the cloned version of the full feature DataFrame. Then the column with the original features gets dropped out of the cloned DataFrame to free some memory. In the next step, all rows of the DataFrame where the freshly calculated distances are larger than a certain threshold (the mean value of the distance column in this case) get dropped out of the DataFrame, drastically reducing the size of all feature sets remaining. When using the mean value, about half of the songs get dropped out of the DataFrame, reducing the problem size for the next feature set to half the size. This is also the reason why the data had to be copied in-memory because now the clone can be altered and thinned out without impacting the original DataFrame. Copying of the data on the other hand, is an additional overhead and requires more memory.

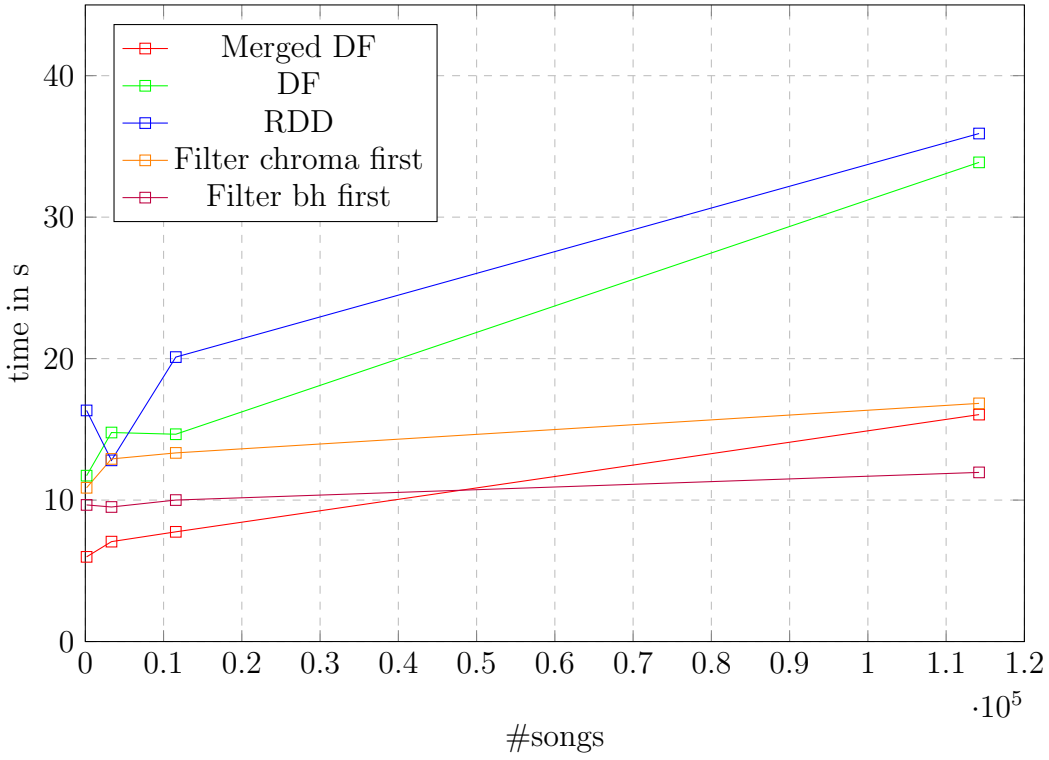


Figure 4.12: Performance of descending importance filter and refine, all features

When looking at the results in Figure 4.12, it shows that the filter and refine method scales very well with increasing sizes of the dataset. The plots show the performance of full song requests (all feature types) on already cached feature DataFrames or RDDs. The graphs of the filter and refine tests include the necessary time to create a copy of the cached feature DataFrames, so the additional overhead is taken into account. The order of filter operations in the filter chain for the plot labeled with "Filter chroma first" is:

$chroma \rightarrow (js \rightarrow skl \rightarrow mfcc) \rightarrow rp \rightarrow rh \rightarrow bh \rightarrow notes$

and $bh \rightarrow rh \rightarrow notes \rightarrow rp \rightarrow (js \rightarrow skl \rightarrow mfcc) \rightarrow chroma$ for the plot labeled with

"Filter bh first". The order of the different filter and refine operations is very important. When searching for cover songs for example, the cross-correlation and the Levenshtein distance should be calculated at the very beginning of the filter chain or otherwise the cover songs could be filtered out during the first stages. When running a simple test with the song "Für Elise" by Beethoven that appears three times in the full dataset, the filter and refine method starting with the chroma features was still able to detect one alternative recording as the top recommendation and the other recording was placed as recommendation number 14, scoring even higher than in a test without the filter and refine method because other non-matching songs got filtered out.
Admittedly, the computation of the cross-correlation between chroma features is the most compute-intensive one; for performance reasons, it would be better to start with a distance measurement like the Euclidean distance of the beat histograms. Later when the more demanding computations follow the data set is already thinned out. This is also the reason this approach is called "descending importance filter and refine" in this thesis because the client, who requests the song recommendations, has to define which aspect is most important to him (speed, melody, rhythm, timbral features or cover song detection), before choosing an order for the filter chain (descending importance). The results get better the further the application progresses in the filter chain (filter and refine).

Cluster Size

The runtime and its dependencies on cluster configuration, size of the input dataset, and implementation details were presented in the previous Sections. With about twelve seconds response time for the filter and refine method and 14 seconds for the merged DataFrame approach on 16 compute nodes, and for 114000 songs, the runtime is reasonably fast but not yet fast enough for real-time processing.
To simulate the impact of growing cluster sizes in Figure 4.13, the cluster configuration was changed from 1 up to 16 Executors spawned, each reserving 36 CPU cores (the maximum number of available cores on one node (without HT)) and 64GB (+ 32GB overhead) of main memory. To do this, the parameters of the dynamic allocation were changed. When setting the minimum Executor count above 16 without there being enough resources on the cluster, the Spark Driver only spawns as many as it is able to (16 on the ARA-cluster with 36 CPU cores/Executor). As a test algorithm, the merged DataFrame approach (repartitioned in 32 chunks) with two subsequent song requests was chosen. The computation time of the second song request for all feature-sets is shown in Figure 4.13.

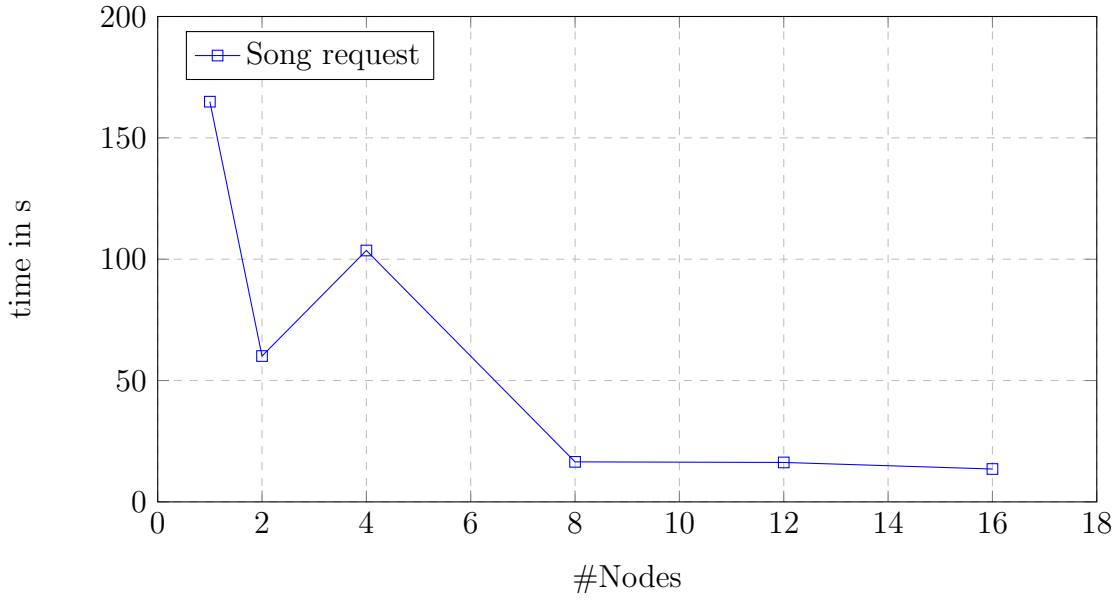


Figure 4.13: Performance depending on #Executors (36 CPU cores each)

4.3.8 Possible Improvements and Additions

Spark offers a few other, interesting alternatives to compute similarities that are only mentioned here and not further evaluated. The so-called ”DIMSUM all-pairs similarity” (Dimension Independent Similarity Computation using MapReduce) is a MapReduce algorithm to compute full similarity matrices (”all-pairs” similarity instead of the ”one-to-many-items” similarity implemented here) and could be of interest as well.
Also, an implementation of the TF-IDF weights is already part of the Spark framework, possibly enabling a future addition of the melodic similarity computation using the mentioned approach in Section 3.2.2. The Alternating Least Squares algorithm to perform collaborative filtering (see Section 2.4.5) would be an interesting addition. Although this thesis only focuses on audio features, a future additional implementation of metadata and listening behavior information could provide valuable information.

5. Results

In this chapter, the results concerning the quality of the recommendations are shown. An attempt to quantify the results and the quality of the recommendations is made by choosing objective tests like genre recall and cover song recognition. The second part comprises some subjective impressions, including personal taste and listening preferences.

5.1 Objective Evaluation

At first, for the objective, scientific evaluation, the resulting distances are analyzed and visualized in Section 5.1.1. To evaluate the quality of the resulting song recommendations returned by the Spark application, some tests were made. As mentioned in Section 3.2.3, a way of evaluating the quality of the melodic similarities is the ability to recognize cover songs. This will be examined in Section 5.1.2. To test the quality of the timbre and rhythm based distances, the genre recall rate is examined in Section 5.1.3. Another indicator of the quality of rhythm features is the ability to recommend songs around the same BPM count (see Section 5.1.4).

5.1.1 Feature Correlation and Distance Distribution

This section evaluates the results from the similarity analysis to determine how the distances from different feature sets correlate with each other, and how they are distributed over the unit interval $[0, 1]$. To analyze this, a test dataset consisting of distances returned by the Spark application had to be created. Ninety-five songs (five songs from every genre) were randomly chosen from the 1517-Artists dataset, and the distances to all other songs of the 1517-Artists dataset were calculated. The dataset contains 3180 songs evenly distributed over 19 different genres (see Figure 2.13(c)). Sampling of distances from different genres is vital for the analysis of the distribution of distances. Distances and their distribution vary, depending on where in the feature space the actual song is located. A song taken from the edge of the distribution of the feature space will end up with different distances than a song taken from the center. To

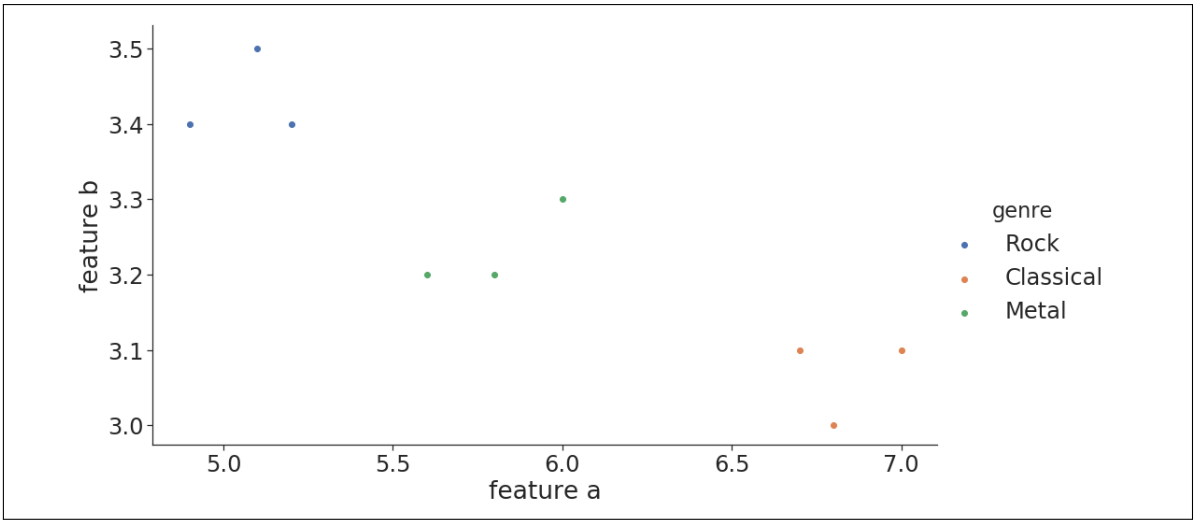


Figure 5.1: Feature space example

demonstrate this, Figure 5.1 shows a minimal example. While the distances from songs tagged with ”Metal” to the songs tagged with ”Rock” and ”Classical” are about the same, the distances from a song taken off the Classical genre to the ”Rock” or ”Metal” songs are different in this example. The Rock songs are twice as far from the Classical songs than from the Metal songs.

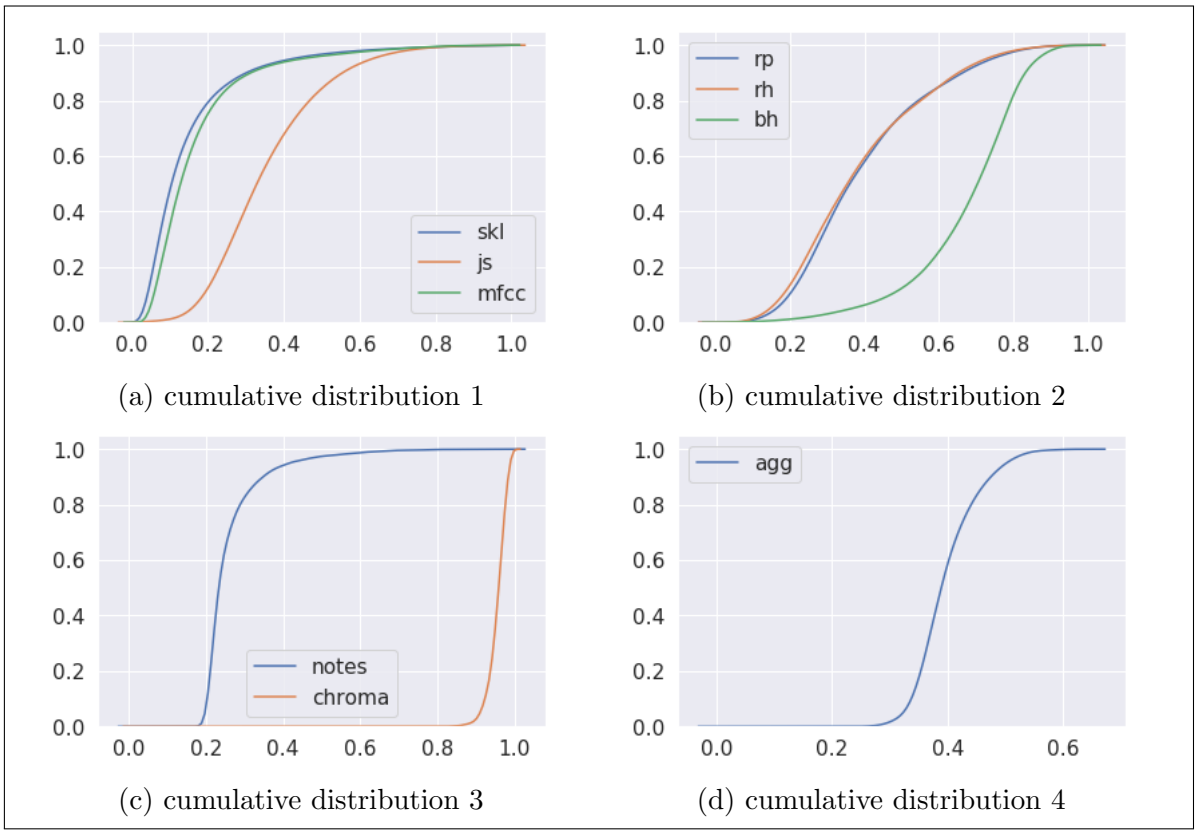
Figure 5.2 shows the correlation between the distances from the various feature types. The eight different distances for each song pair are summed up into one new combined distance (following Equation (4.5) with all weights $w = 1$). This combined distance is labeled as ”agg” in the following plots. Unsurprisingly the various rhythm features correlate well with each other and the JS and SKL features do so as well. The melodic features on the other hand are only weakly correlated.

	rp	rh	bh	js	skl	mfcc	chroma	notes	agg
rp	1	0.918345	0.258626	-0.0131253	0.0357719	0.105182	0.0455418	0.00375641	0.752988
rh	0.918345	1	0.192452	0.0207377	0.0443187	0.150032	0.0396717	-0.00201152	0.7558
bh	0.258626	0.192452	1	-0.203041	-0.160113	-0.0695903	0.0286554	-0.00464233	0.323581
js	-0.0131253	0.0207377	-0.203041	1	0.747947	0.0894321	-0.021468	-0.00046403	0.435151
skl	0.0357719	0.0443187	-0.160113	0.747947	1	0.0580153	-0.0458679	0.0222944	0.461898
mfcc	0.105182	0.150032	-0.0695903	0.0894321	0.0580153	1	0.047422	0.0705918	0.378666
chroma	0.0455418	0.0396717	0.0286554	-0.021468	-0.0458679	0.047422	1	0.169881	0.142827
notes	0.00375641	-0.00201152	-0.00464233	-0.00046403	0.0222944	0.0705918	0.169881	1	0.25369
agg	0.752988	0.7558	0.323581	0.435151	0.461898	0.378666	0.142827	0.25369	1

Figure 5.2: Correlation matrix, 95 random songs, 19 genres (5 each), 1517-Artists

The correlation of a feature type with the overall distance is a sign of the impact of the feature type on the overall distance from the weighted sum. But because not all

distances are equally distributed over the unit interval, different feature types have different impacts on the sum of distances. This problem was already mentioned in Section 4.3.5 and Section 4.3.4. Figure 5.3 shows how the distances are distributed with the cumulative histograms over the unit interval. It is apparent that especially the cross-correlation distances are not evenly distributed. In Section 4.3.5, a few proposals were already given as to how this problem could be solved in the future.



(a) cumulative distribution 1

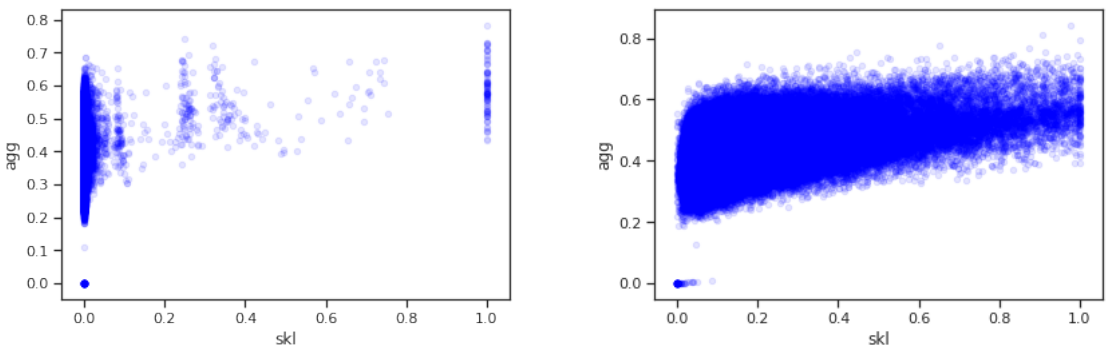
(b) cumulative distribution 2

(c) cumulative distribution 3

(d) cumulative distribution 4

Figure 5.3: Cumulative distributions of distances

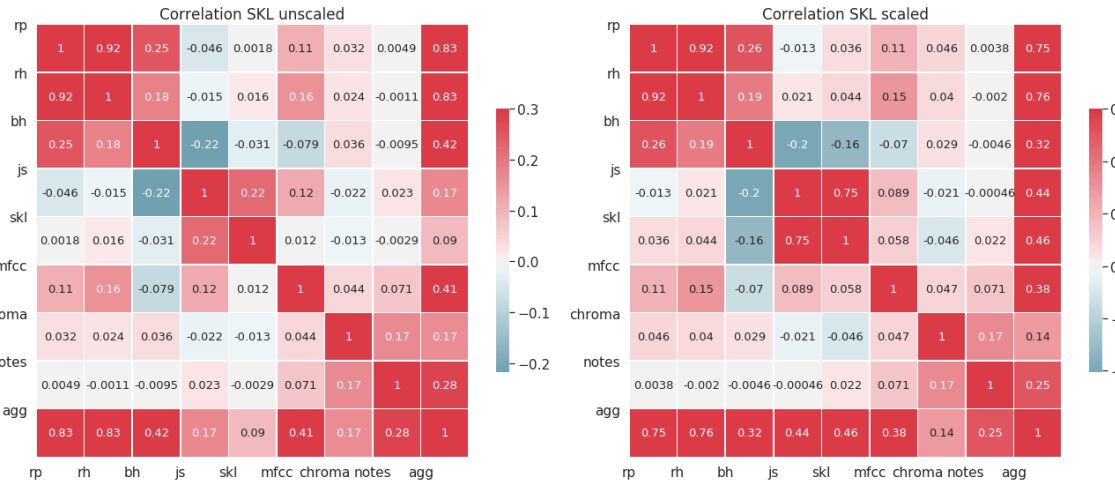
As mentioned in Section 4.3.4, the SKL divergence was also prone to outliers and had shortcomings when scaling distances to the unit interval. The solution was to filter out all song pairs with an SKL divergence larger than a certain threshold before scaling the distances. If this filter operation is left out, nearly all distances calculated with the symmetric Kullback-Leibler divergence are close to zero after the scaling. The impact can be seen in Figure 5.4 and Figure 5.5. If the outliers are not filtered, the correlation between the unfiltered SKL distances and the combined distance from the weighted sum ("agg") decreases significantly (see Figure 5.5). Interestingly also the correlation between the Jensen-Shannon-like divergence and the combined distance ("agg") is decreasing.
A possible explanation could be that the SKL and JS distances are highly correlated, but due to the bad scaling, the SKL has no impact on the overall distance. The results from the JS divergence alone are not able to impact the weighted sum of the combined



(a) SKL unscaled (b) SKL scaled

Figure 5.4: Impact of SKL scaling on the weighted sum

distance in the same way both features together could.



(a) SKL unscaled (b) SKL scaled

Figure 5.5: Correlation of features depending on SKL scaling

Finally, Figure 5.6 shows the full scatter plot matrix of the various distances for the 95 song sample from different genres to visualize the correlation and distribution of the distances. The main diagonal shows the histograms of the distances from the respective unique feature-sets. It shows that besides the chroma features all feature types correlate well with the weighted sum of all features. The strong correlation between the rhythm patterns and rhythm histograms as well as the Jense-Shannon-like divergence and the symmetric Kullback-Leibler divergence for all genres is clearly visible in the scatter plots.

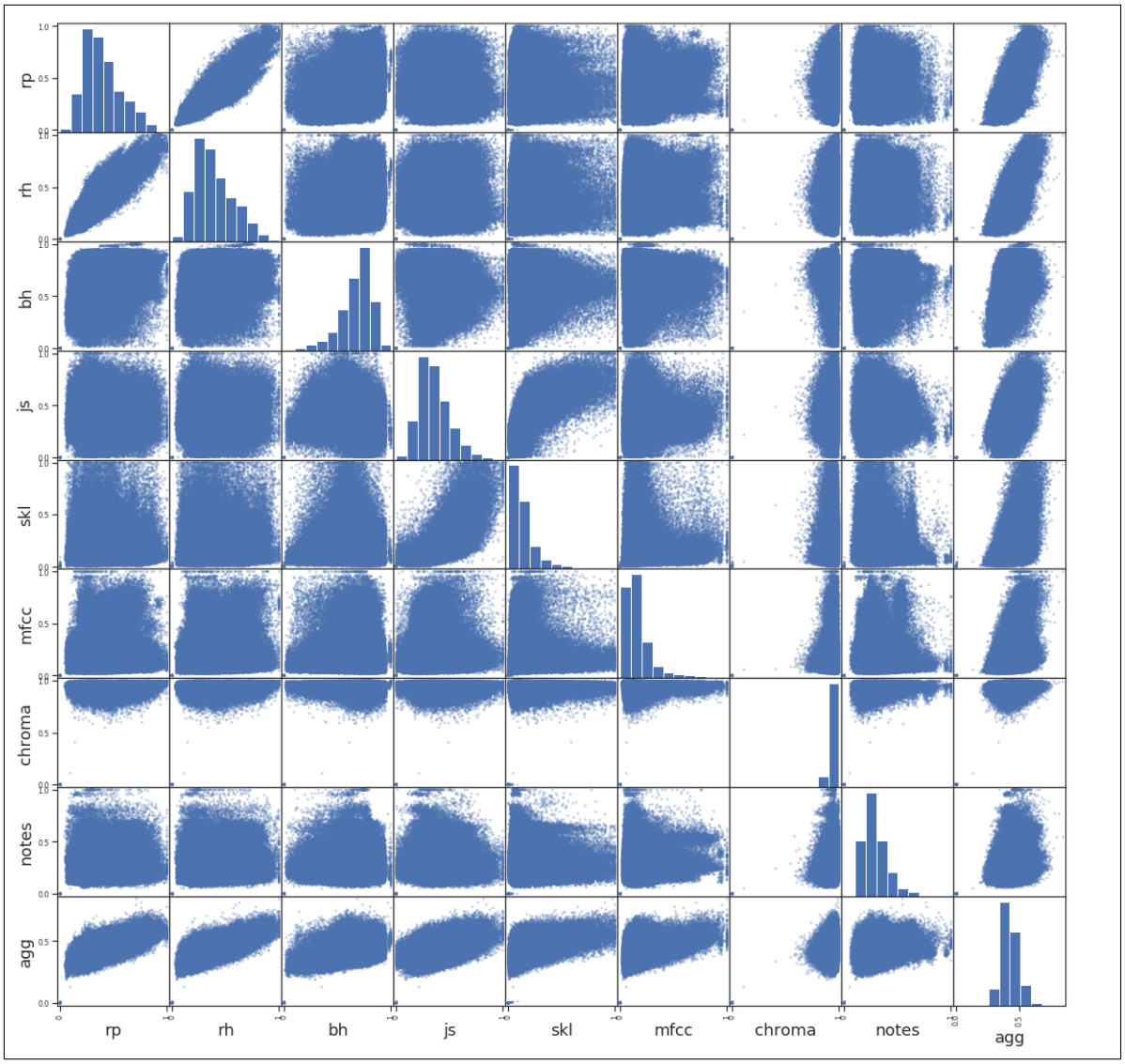


Figure 5.6: Scatter matrix, correlation 95 songs, 19 genres (5 each), 1517-Artists

5.1.2 Cover Song Identification

As mentioned in Section 3.1.4, purely MFCC based recommender systems lack the capacity to detect cover songs. Melody based similarity algorithms like the cross-correlation approach by Ellis and Poliner (see Section 3.2.2) and the approach using the Levenshtein distance by Xia (et al.) in Section 3.2.2, were primarily implemented to detect cover songs. Running the first tests on the full dataset consisting of 114210 songs, the Spark implementation was able to find the cover of "Rock you like a Hurricane" by the Scorpions and covered by Knightsbridge as the top recommendation when using the cross-correlation.
The application was also able to find an alternative recording of the piece "Für Elise" cover as a top recommendation in over 114210 songs, even when using the filter and refine algorithm (starting with chroma features) presented in Section 4.3.7.
As a third example the famous "Rondo Alla Turca (Allegretto)" also known as the Turkish March by Mozart was tested. This song was also used in Section 3.1.4 where the capacity of the Musly toolkit to detect cover songs was tested. Two different versions were detected as the top results, and the fourth recommendation even listed a variation of the original song theme. For this test, a combination of js, chroma, and rp features was used. The top five results are listed below.

Song request: 100 Meisterwerke der Klassik - Mozart - Alla Turca (Allegretto) (private collection), JS + RP + CHROMA

1. Piano Perlen / Mozart - Türkischer Marsch (private collection)
2. FRITZ STEINEGGER - RONDO ALLA TURCA KV 331 (1517-Artists)
3. 136071 (2Kutup - We Shall Cuddle Up And Sleep) (FMA dataset)
4. Sean Bennett - Variations on the Turkish March (1517-Artists)
5. Mozart - Fantasie in D minor (1517-Artists)

Although the private music collection contains two additional versions of this song (see Section 3.1.4), the other versions could not be detected because the rp_extract tool failed during the extraction of the features from these songs due to file format issues. In a second test, the rhythm patterns were left out and only js and chroma features were used. The six top recommendations are again listed below:

Song request: 100 Meisterwerke der Klassik - Mozart - Alla Turca (Allegretto) (private collection), JS + CHROMA

1. Mozart Collection / CD31 / KV331-3 Alla turca allegretto (private collection)
2. Piano Collection / CD25 - Mozart - Alla Turca Allegretto (private collection)

3. Piano Perlen / Mozart - Türkischer Marsch (private collection)
4. FRITZ STEINEGGER - RONDO ALLA TURCA KV 331 (1517-Artists)
5. 136071 (2Kutup - We Shall Cuddle Up And Sleep) (FMA dataset)
6. Sean Bennett - Variations on the Turkish March (1517-Artists)

In a third request where only the Jensen-Shannon-like divergence was tested to detect the alternative recordings, the first alternative recording appeared as the 13th recommendation. This confirmed the presumption that timbral features and the Jensen-Shannon-like divergence nor the symmetric Kullback-Leibler divergence are appropriate for cover song recognition.
But there are also song requests where the cross-correlation fails to detect the cover song, one example being the song Chandelier by Sia and its cover version by Pvris that was used in Section 3.2.1 to explain the computation of the chroma features.
To further quantify the ability to detect cover songs after the promising first tests, the covers80 dataset introduced in Section 2.5.1 was loaded onto the cluster. The 80 "A-versions" songs were passed to the Spark application as song requests, and the resulting nearest neighbors were analyzed.

features	detected covers
chroma	30
chroma + notes	27
chroma + skl	26
chroma + notes + rp	24
chroma + rp	22
chroma + skl + rp	22
chroma + mfcc	19
chroma + js + rp	17
chroma + js	17
notes	17
chroma + mfcc + rp	15
all	15
notes + rp	13
mfcc + notes + rp	7
rp	7
mfcc + js + skl	3

Table 5.1: Cover recognition rate - Top 1

features	detected covers
chroma	33
chroma + notes	31
chroma + notes + rp	30
chroma + skl	29
chroma + rp	29
chroma + skl + rp	26
chroma + mfcc + rp	24
notes	23
all	23
chroma + mfcc	22
chroma + js + rp	22
chroma + js	21
notes + rp	19
rp	15
mfcc + notes + rp	14
mfcc + js + skl	10

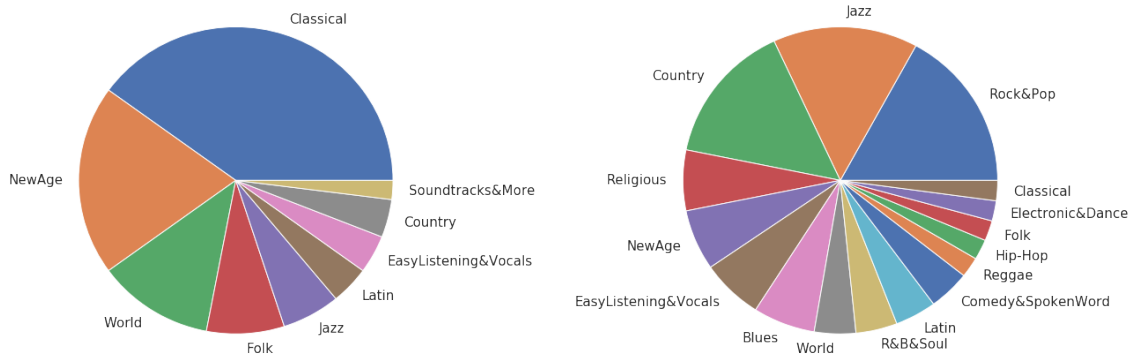
Table 5.2: Cover recognition rate - Top 5

Table 5.1 counts the appearance of the "B-version" songs as the first recommendations

while Table 5.2 lists the count of the recommended cover versions in the top five results, when using different combinations of feature sets. As expected, the approaches using melodic similarity features perform best. The combination of different timbre based features performs worst. Interestingly the distances based on rhythm patterns also detect some cover songs.
Although 30 out of 80 detected cover songs does not seem like a surprisingly good hit rate at first and is not quite as good as the results from the original paper, it has to be mentioned that most of the cover versions in the cover80 dataset differ significantly from the original recordings in musical style, instrumentation, rhythm and even genre from the original recordings. These differences in musical style were also mentioned in the original paper from Ellis and Cotton [65, p. 3]. As an interesting side note it has to be mentioned that the detected cover versions of the ”chroma-” and ”notes-only” requests were mostly the same. Aside from two songs, the chroma feature cross-correlation approach detected all of the cover songs that the Levenshtein distance also detected. So in conclusion, the cross-correlation is more precise but also more compute heavy.

5.1.3 Genre Similarity

Another way to quantify the quality of the distances and therefore the quality of the music recommendations is to measure the genre recall rate. In a simple test on the 1517-Artists dataset, five classical songs are passed to Spark, and the nearest neighbors based on rhythm and timbre features (skl, js, mfcc, rp, rh, and bh) are calculated. Then the genres of the top ten recommendations from all five song requests are analyzed. The result is pictured in Figure 5.7(a).



(a) 1517-Artist classical recommendations (b) 1517-Artist rock recommendations

Figure 5.7: Genre recall rate on 1517-Artists dataset

Although not all recommendations are classical songs, the recommended other genres like New Age, Wold, Folk and Jazz music are closely related to classical music. Not a

single song from more ”modern” genres like Hip-Hop, Rock & Pop, Electronic & Dance or Reggae appears. The same was tested with five songs from the Rock & Pop genre (see Figure 5.7(b)). The results are scattered across 16 out of 19 different genres from 1517-Artists dataset. A possible explanation for this is, that the songs annotated with ”Rock & Pop” in this dataset come from a wider variety of sub-genres. When taking a closer look at the dataset, it shows that, e.g., Metal songs are also tagged as Rock & Pop.

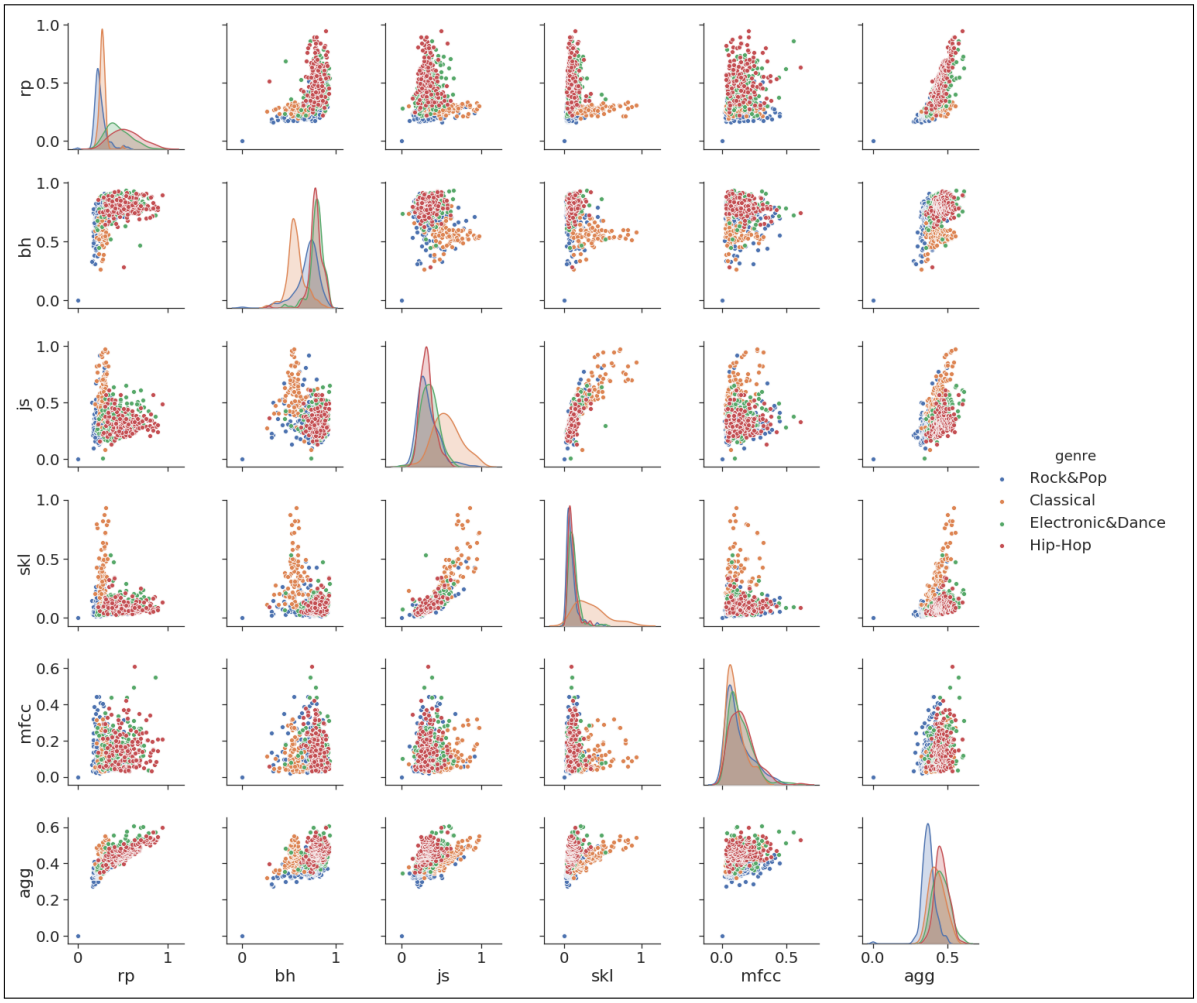


Figure 5.8: Scatter matrix, distances 1 random Rock&Pop song, 1517-Artists, 4 genres

To investigate the impact of different feature types on the overall recommendations and to visualize the distribution of distances for different genres, another test was performed. For single song requests, all distances to the songs from a subset of the 1517-Artists dataset containing the genres ”Classical”, ”Hip-Hop”, ”Electronic & Dance” and ”Rock & Pop” were computed. Figure 5.8 shows the scatter matrices of all distances from one song request taken from the genre Rock & Pop. The different distances of the recommendations are colored by the genre of the recommended song. On the main

diagonal the Kernel Density Estimation of the respective feature type is shown. One interesting detail that should be pointed out is that the JS distance alone is unable to distinguish between Rock/Pop songs and Hip-Hop songs but is able to separate between classical music and the rest. On the other hand, the rhythm patterns alone can not separate classical music from rock and pop. But when both feature types are combined, all three genres can be separated. The scatter plot of the distances from the rhythm patterns and Jensen-Shannon-like divergence in combination shows three clusters of songs belonging to different genres. The fourth genre, "Electronic & Dance" however can not be separated from hip-hop songs no matter what feature-set is used. But it has to be kept in mind that all these distances are distances coming from a song request of the Rock/Pop genre.

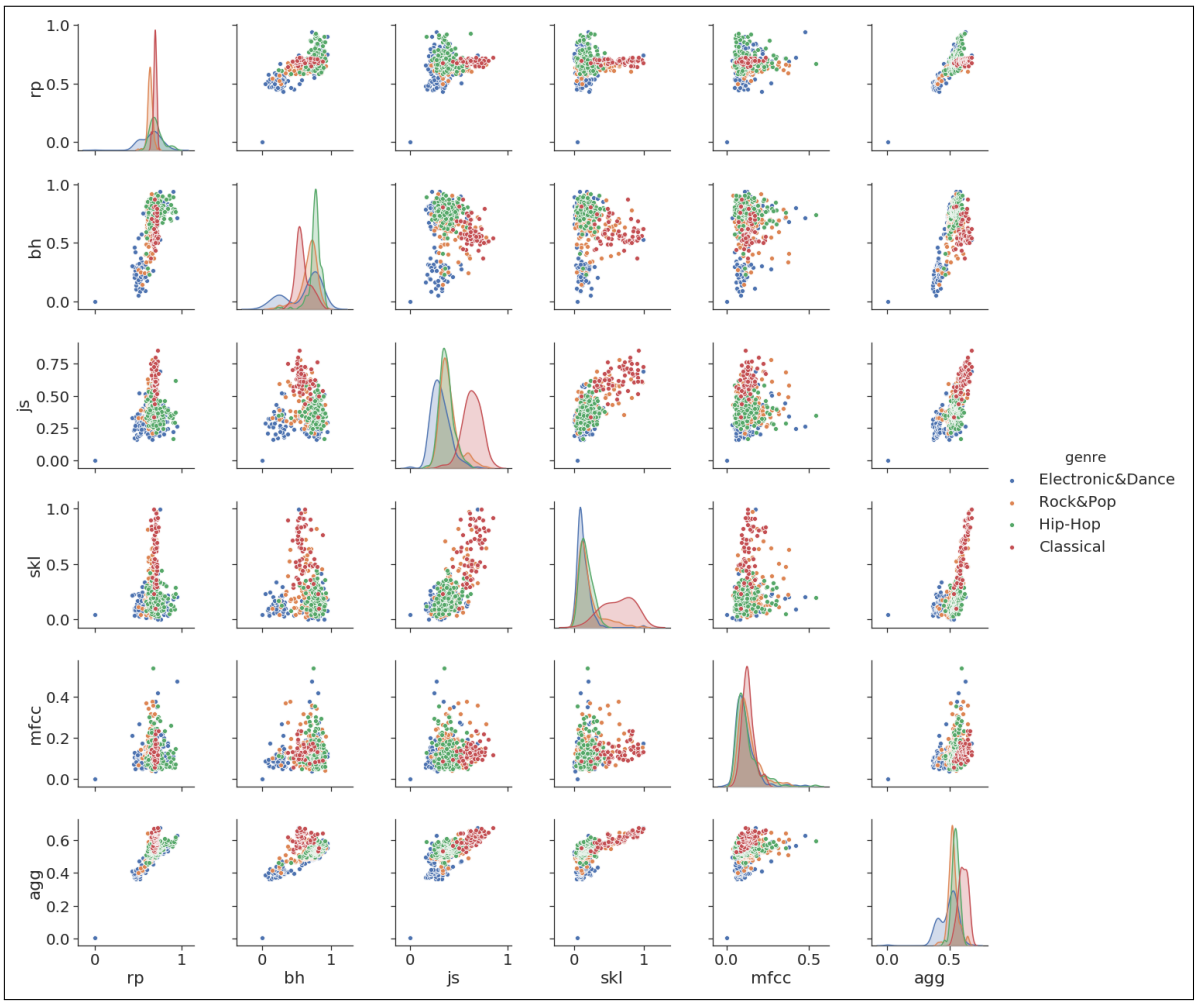
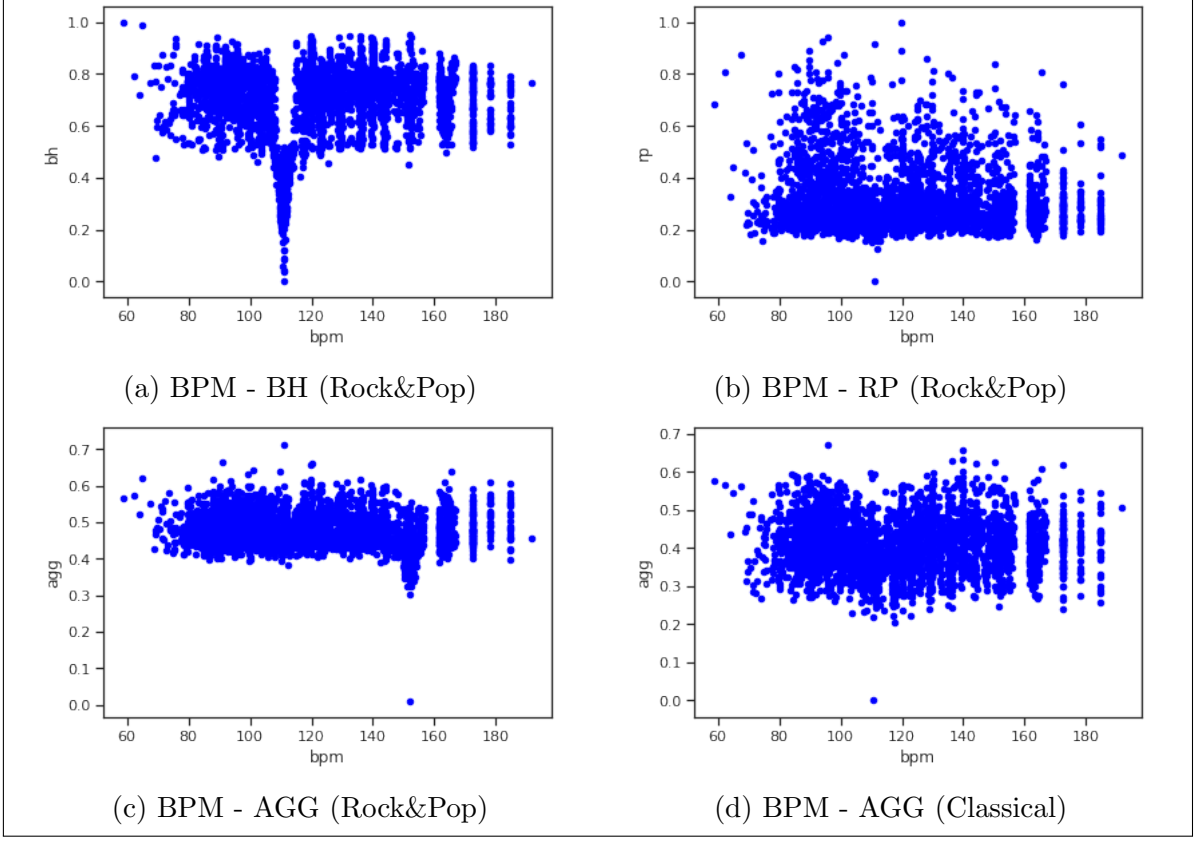


Figure 5.9: Scatter matrix, distances 1 random Electronic song, 1517-Artists, 4 genres

As mentioned in Section 5.1.1 and visualized in Figure 5.1, the distribution of the distances varies depending on where in the feature space the song request is located. Apparently the songs of the Hip-Hop and Electronic/Dance genre are on average all about the same distance away from the requested Rock/Pop song. When requesting a

song from the genre Electronic/Dance, the distribution of the distances look entirely different (see Figure 5.9). The ”agg” - plots represent the weighted sum of all features combined (also including cross-correlation and Levenshtein distances not shown in the plots). After the combination of all feature types, the returned results primarily recommend other Rock & Pop songs in Figure 5.8 and Electronic & Dance songs in Figure 5.9.
When using only one feature type, the Spark recommendation engine would not be able to separate all four of the different genres from each other. Only due to the combination of different rhythmic and timbral features an overall satisfying list of recommendations can be retrieved.

5.1.4 Rhythm Features



(a) BPM - BH (Rock&Pop)

(b) BPM - RP (Rock&Pop)

(c) BPM - AGG (Rock&Pop)

(d) BPM - AGG (Classical)

Figure 5.10: Scatter plots rhythm features / BPM for random Rock&Pop and Classical songs

Another critical requirement for the recommendation engine is the ability to obtain songs that are about the same tempo. To investigate the capabilities of the rhythm features, Figure 5.10 shows the resulting distances of two song requests performed

on the 1517-Artists dataset. The scatter plots show that the beat histogram and the rhythm patterns are closely related to the overall BPM of the songs. The ”agg” value (the weighted sum) includes all eight different feature types, so the overall impact of the rhythm features on the recommendations can be seen. All in all, the Spark recommendation engine is more likely to recommend songs that have similar BPM when rhythm features are included in the weighted sum. The classical song request in Figure 5.10(d) also shows that the overall distances are not exclusively dominated by the BPM but rather slightly influenced.

5.2 Subjective Evaluation

This section includes the personal opinion and music taste of the author. Although these results are not ”scientific”, music taste is something personal and judging music recommendation solely from an objective perspective would be the wrong approach for this thesis. The core strength of this Spark-based recommender system is that its parameters can be used to personalize the music recommendations.

5.2.1 Beyond Genre Boundaries

The main reason for the choice of the topic of this thesis was that recommender systems as they come with streaming platforms like Spotify tend to value the music context information over music content. For example, the ”Song Radio”- option coming with Spotify stays in the boundaries of genres and is heavily influenced by other people's listening behavior. Although this is not necessarily a bad thing, this thesis tried to focus directly on the timbral, rhythmic, and melodic properties. As a result, songs from other genres are recommended as can be seen in the following example. When searching for the nearest neighbors of the ”Prelude in C- Sharp Minor (Op. 3 No. 2)” by the Russian composer Sergei Rachmaninoff based on the Euclidean distance of MFCCs, the following results were returned:

1. Klassik/Rachmaninoff - Piano Concerto No2 In C Minor Op18-1 Moderato
2. Klassik/Liszt - Piano Concerto No 1 in E flat major S124(LWH4) Allegro maestoso
3. Klassik/Brahms - Piano Sonata No2 in F sharp minor Op2 - III Scherzo allegro
4. Metal&Rock/Steve Moore - Intro & Credits
5. Klassik/Liszt - Piano Concerto No 1 in E flat major S124(LWH4) Allegro animato

The ”Metal & Rock” recommendation seems out of place at first glance, but when taking a closer look, the recommended song is called ”Intro & Credits” and it is not a typical Metal song. When listening to it, some similarities are recognizable; it is a calm,

dark instrumental piece made of synthesizer sounds. The primarily requested Prelude is a dark piano piece. Of course, this is just one example, and the recommendation is arguably not perfect. In general some of the timbre based recommendations seem out of place. This might be due to the choice of 13 MFCC bands over 25 as the Musly toolkit uses, or potentially there are some unnoticed issues with the implementation left, which would have to be investigated in future work. But as also stated in Section 5.1.3, the overall performance concerning the genre recall rate is reasonably good aside from a few outliers.

5.2.2 Personal Music Taste

As a last side note on personal music taste, a song request using one of my favorite songs was made. As already mentioned, my private music collection was a part of this thesis. To retain some kind of reproducibility the whole collection is documented, and the pertinent list of albums and songs is on a document on the CD in the appendix. On the last pages of this document, there is also a list containing my personal song favorites in the metal music genre. One of these songs was chosen, and recommendations were calculated for the private music collection. The song is called "The Art Of Dying" by the band Gojira. The recommendations based purely on rhythm patterns are listed below. Another track from my personal list of favorite song appears as a recommendation.

- Numenorean - Adore
- Shylmagoghnar - Transience
- Amon Amarth - The Last Stand Of Frej
- Delain - We Are the Others
- Ensiferum - Descendants Defiance Domination

This could be an indication that my taste in music is closely related to the rhythmic properties of the music. An idea for future research could be to reverse engineer a user's musical taste by looking at a list of favorite songs. The information which songs a user likes the most is already available to all streaming platforms because most likely the songs a user listens to the most are also the best liked songs. Spark could be used to calculate the similarities between these favorite songs of a user and analyze the distances. Whether or not these songs are more similar in rhythm, melody or timbre could enhance the parametrization of a recommender engine and further personalize music recommendations by adapting the weights of a recommendation engine.
Of course, the field of personalized music recommendation is an already existing one, but maybe the addition of Spark and Big Data opportunities of using audio content instead of contextual information and collaborative filtering could enhance these existing systems.

6. Summary

In this last chapter, the results of this thesis are summarized and a short outlook on open tasks and possibilities for future work are given.

6.1 Conclusion

Looking back at the content of this thesis, Chapter 2 provided an overview of the field of music information retrieval. Different high- and low-level audio features were explained, and various ways to measure the similarities between audio files based on the audio features were introduced. Additionally, a short introduction to Big Data frameworks, especially Apache Spark and Hadoop was given, and different audio data sources were gathered. Chapter 3 presented ways to extract and pre-process timbre, rhythm, and melodic features from audio files. Multiple algorithms for calculating the distances between the extracted features were given. With the theoretic knowledge from the first chapters, the implementation could be planned. Data was collected; over 1TB of music files containing 114000 different songs were aggregated.
In the first part of the implementation, the necessary audio features were extracted and pre-processed (e.g., by extracting the melody from chroma features) in parallel using MPI on a computer cluster, paving the way for the usage with the Big Data processing framework Spark. The features were loaded into the HDFS of a cluster, and multiple similarity measurements were implemented, tested, evaluated, and improved using the Spark framework. With Spark, multiple approaches (RDD, DataSet, Filter and Refine, Cluster Configurations) were tested, and the runtime was measured. The resulting distances were presented, analyzed, and visualized.

The final application handles the recommendation of songs similar to a song request by computing the distances based on melodic, rhythmic and timbral properties of the music. The recommendations are parameterized, giving the user the option to prioritize different aspects of the music. The system is scalable. More songs can be added, the cluster size can be increased, and the possibility to add different kinds of audio features and more state-of-the-art similarity measurements is also given.

6.2 Performance

The extraction of the features on a single PC would have taken approximately 258 hours for about 100000 songs using the Essentia toolkit. By using a computer cluster with 648 concurrent threads and Mpi4py the computation time could be reduced to about half an hour (32 minutes and 30 seconds). This is approximately 476 faster than on a single PC core. The extraction of the rhythm patterns and rhythm histograms with the rp_extractor tool provided by the TU Wien for the same number of songs takes about the same amount of time on the ARA-cluster (also parallelized with Mpi4py).
The computation of similarities using the Big Data framework Spark on a 16 node computer cluster takes approximately 14 seconds for all of the 8 features types combined. This processing duration could be reduced to about 12 seconds by using a filter and refine method. It can also be reduced by using only subsets of the features types.

6.3 Outlook

There are still a few minor flaws, especially when looking at the implementation of the symmetric Kullback-Leibler divergence and the Jensen-Shannon divergence and the scaling of the distances. The different starting points for possible future research were laid out during the whole thesis and are summarized here. First of all the file format issues with *.wav and *.ogg audio files when using the rp_extract tool from the TU Wien should be fixed to allow the computation of all features from all the songs of a dataset (see Section 4.2.2). The next step would be to re-evaluate the Jensen-Shannon-like divergence and the symmetric Kullback-Leibler divergence and fix the issues with outliers. The issues with non-invertible or non-singular covariance matrices should be investigated as well (see Section 4.3.4). The proposed enhancements by Schnitzer [22] of reducing the hubness with mutual proximity and by using more mel bands for the computation of the MFCCs might also be sufficient to improve the quality of recommendations (see Section 3.1.2). Scaling of the different features could be improved in a way where all features are evenly distributed over the unit interval (see Section 4.3.5).
Tests of the performance on larger clusters and with more songs would be critical to assess the scaling of the problem. An implementation of the Spark streaming abilities to enable real-time computation of similarities instead of using batch-processing jobs would be the next logical step if the objective was to develop a system able to run with music streaming platforms. When evaluating the genre recall rate with Spark, an issue with the garbage collection running out of memory after about 40 subsequent song

requests was encountered and should be fixed first.
As another way of improving the presented Spark application, more state-of-the-art similarity measurements like block-level features (Section 3.1.3) or the TF-IDF weights (Section 3.2.2) for melodic similarity could be added. The most promising enhancement for the developed recommendation engine in this thesis would be the addition of genre and metadata information, genre-specific features, collaborative filtering, and lyrics (see Section 2.4.5). All the contextual music information that would typically be processed by a Big Data framework was not included in this thesis but could significantly enhance recommendations. Most streaming services already have all the information needed like user's listening behavior or audio metadata available. Services like Spotify are already using Spark for collaborative filtering so the Spark application presented in this thesis could be added and integrated into running streaming systems. A last suggestion for future enhancements is to investigate the proposal from Section 5.2.2 of personalized music recommendation based on the audio feature similarities of a user's favorite songs made available by the Big Data framework.

References

[1] SoundCloud Go+ tracks, URL: https://blog.soundcloud.com/2017/03/27/soundcloudliveinthenetherlands/.

[2] Knees, P. and Schedl, M., Music Similarity and Retrieval, An introduction to web audio- and web-based strategies, Springer, 2016, ISBN: 9783662497203.

[3] Weihs, C. et al., Music Data Analysis: Foundations and Applications, 1st, New York: Chapman and Hall/CRC, 2016, ISBN: 1498719562, 9781498719568, URL: https://doi.org/10.1201/9781315370996.

[4] Moffat, D., Ronan, D., and Reiss, J., An Evaluation of Audio Feature Extraction Toolboxes, in: Nov. 2015, DOI: 10.13140/RG.2.1.1471.4640.

[5] Mathieu, B. et al., YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software. In: Jan. 2010, pp. 441–446.

[6] Python, URL: https://www.python.org.

[7] MATLAB, URL: https://de.mathworks.com/help/matlab/index.html.

[8] Bogdanov, D. et al., ESSENTIA: an Audio Analysis Library for Music Information Retrieval, in: International Society for Music Information Retrieval Conference (ISMIR'13), 2013, pp. 493–498.

[9] Mandel, M. I. and Ellis, D. P. W., LABROSA's audio music similarity and classification submissions, in: Music Information Retrieval Information Exchange (MIREX), 2007.

[10] Jupyter, URL: https://jupyter.org/index.html.

[11] Lartillot, O. and Toiviainen, P., MIR in Matlab (II): A Toolbox for Musical Feature Extraction from Audio, in: Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR 2007, Vienna, Austria, September 23-27, 2007.

[12] Eaton, J. W. et al., GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computations, in: 2016, URL: http://www.gnu.org/software/octave/doc/interpreter/.

[13] Hartmann, M. A., A port of MIRToolbox for Octave, in: 2016, URL: https://github.com/martinarielhartmann/mirtooloct.

[14] Schnitzer, D., Audio Music Similarity, URL: http://www.musly.org/index.html.

[15] Mandel, M. I. and Ellis, D. P. W., Song-level features and support vector machines for music classification, in: Proceedings of the 6th International Conference on Music Information Retrieval, ISMIR, 2005.

[16] Schnitzer, D. et al., Using Mutual Proximity to Improve Content-Based Audio Similarity. In: Jan. 2011, pp. 79–84.

[17] Für Elise, URL: https://upload.wikimedia.org/wikipedia/commons/a/a9/BH_116_Vergleich.png.

[18] Prélude cis-Moll (Rachmaninow), URL: https://upload.wikimedia.org/score/5/m/5m046ksmmlpu0s4xbwna50qx9nmoh8v/5m046ksm.png.

[19] Brossier, P. et al., aubio/aubio: 0.4.8 (Version 0.4.8), in: 2018, URL: http://doi.org/10.5281/zenodo.1494152.

[20] Salamon, J. and Gómez, E., Melody Extraction from Polyphonic Music Signals using Pitch Contour Characteristics, in: IEEE Transactions on Audio, Speech and Language Processing, 20(6):1759-1770, 2012.

[21] Cannam, C., Landone, C., and Sandler, M., Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files, in: Proceedings of the ACM Multimedia 2010 International Conference, Firenze, Italy, 2010, pp. 1467–1468.

[22] Schnitzer, D., Dealing with the Music of the World: Indexing Content-Based Music Similarity Models for Fast Retrieval in Massive Databases, 1st ed. PhD thesis, 2012, ISBN: 9781477494158.

[23] McFee, B. and Lanckriet, G., Large-scale music similarity search with spatial trees, in: ISMIR '11, URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.226.5060.

[24] Marolt, M., A Mid-level Melody-based Representation for Calculating Audio Similarity, in: ISMIR 2006, 7th International Conference on Music Information Retrieval, Victoria, Canada, 8-12 October 2006, 2006.

[25] Xia, G. et al., MidiFind: Similarity Search and Popularity Mining in Large MIDI Databases, in: Oct. 2013, pp. 259–276, ISBN: 978-3-319-12975-4, DOI: 10.1007/978-3-319-12976-1_17.

[26] Kim, J. et al., Crepe: A Convolutional Representation for Pitch Estimation, in: Apr. 2018, pp. 161–165, DOI: 10.1109/ICASSP.2018.8461329.

[27] Foote, J., Cooper, M., and Nam, U., Audio Retrieval by Rhythmic Similarity, in: Proceedings of the International Conference on Music Information Retrieval, 2002, pp. 265–266.

[28] Yufeng, Z. and Xinwei, L., Design and Implementation of Music Recommendation System Based on Hadoop, in: Second International Conference of Sensor Network and Computer Engineering (ICSNCE 2018), 2018, DOI: 10.2991/icsnce-18.2018.36..

[29] Gulati, S., Serra, J., and Serra, X., An evaluation of methodologies for melodic similarity in audio recordings of Indian art music, 678-682, in: 2015, DOI: 10.1109/ICASSP.2015.7178055..

[30] Erkut, C. et al., Extraction of Physical and Expressive Parameters for Model-Based Sound Synthesis of the Classical Guitar, in: Audio Engineering Society Convention 108, 2000, URL: http://www.aes.org/e-lib/browse.cfm?elib=9224.

[31] Defferrard, M. et al., FMA: A Dataset For Music Analysis, in: 2016, arXiv: 1612.01840 [cs.SD].

[32] Soundcloud bqpd, URL: https://soundcloud.com/bq_pd.

[33] Thickstun, J., Harchaoui, Z., and Kakadetitle, S. M., Learning Features of Music from Scratch, in: International Conference on Learning Representations (ICLR), 2017, URL: https://arxiv.org/abs/1611.09827.

[34] Seyerlehner, K., 1517-Artists Dataset, 2010, URL: http://www.seyerlehner.info/index.php?p=1_3_Download.

[35] Bittner, R. et al., MedleyDB: A Multitrack Dataset for Annotation-Intensive MIR Research, in: 15th International Society for Music Information Retrieval Conference, 2014.

[36] Bittner, R. et al., MedleyDB 2.0: New Data and a System for Sustainable Data Collection, in: New York, NY, USA: International Conference on Music Information Retrieval (ISMIR-16), 2016.

[37] Man, B. D., Mora-Mcginity, M., and Reiss, J. D., The Open Multitrack Testbed, in: In 137th Convention of the Audio Engineering Society, 2014.

[38] Ellis, D. P. W., The "covers80" cover song data set, in: 2007, URL: available: http://labrosa.ee.columbia.edu/projects/coversongs/covers80/.

[39] Spotify API, URL: https://developer.spotify.com/documentation/.

[40] Spotipy - a Python client for The Spotify Web API, URL: https://github.com/plamere/spotipy.

[41] Spotify Terms and Conditions of Use, URL: https://www.spotify.com/lt/legal/end-user-agreement/plain/#s9.

[42] Spotify Commercial Restrictions, URL: https://developer.spotify.com/legal/commercial-restrictions/.

[43] Bertin-Mahieux, T. et al., The Million Song Dataset, in: Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011), 2011.

[44] Schreiber, H., Improving Genre Annotations for the Million Song Dataset, in: Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR), pages 241-247, Málaga, Spain, Oct. 2015.

[45] Last.fm dataset, the official song tags and song similarity collection for the Million Song Dataset, URL: http://labrosa.ee.columbia.edu/millionsong/lastfm.

[46] The Echo Nest Taste profile subset, the official user data collection for the Million Song Dataset, URL: http://labrosa.ee.columbia.edu/millionsong/tasteprofile.

[47] SecondHandSongs dataset, the official list of cover songs within the Million Song Dataset, URL: http://labrosa.ee.columbia.edu/millionsong/secondhand.

[48] The Echo Nest, URL: http://the.echonest.com/.

[49] Zaharia, M. et al., Apache Spark: A Unified Engine for Big Data Processing, in: Commun. ACM 59.11 (Oct. 2016), pp. 56–65, ISSN: 0001-0782, DOI: 10.1145/2934664.

[50] Apache Hadoop, URL: https://hadoop.apache.org/.

[51] Aven, J., Data Analytics with Spark Using Python, 1st, Addison-Wesley Professional, 2018, ISBN: 013484601X, 9780134846019.

[52] Ghemawat, S., Gobioff, H., and Leung, S.-T., The Google File System, in: vol. 37, Dec. 2003, pp. 29–43, DOI: 10.1145/945445.945450.

[53] Dean, J. and Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters, in: vol. 51, Jan. 2004, pp. 137–150, DOI: 10.1145/1327452.1327492.

[54] MapReduce, URL: https://commons.wikimedia.org/wiki/File:Mapreduce.png.

[55] Scala, URL: https://www.scala-lang.org/.

[56] Quinto, B., Next-Generation Big Data : A Practical Guide to Apache Kudu, Impala, and Spark, Berkeley, CA: Apress, 2018, ISBN: 978-1-4842-3147-0, DOI: 10.1007/978-1-4842-3147-0.

[57] Estrada, R. and Ruiz, I., Big Data SMACK : A Guide to Apache Spark, Mesos, Akka, Cassandra, Springer Science+Business Media, 2016, ISBN: 978-1-4842-2174-7, DOI: 205310.1007/978-1-4842-2175-4.

[58] Seyerlehner, K. and Schedl, M., Block-Level Audio Features for Music Genre Classification, 2009.

[59] Orio, N. and Rodà, A., A Measure of Melodic Similarity based on a Graph Representation of the Music Structure, in: 10th International Society for Music Information Retrieval Conference (ISMIR 2009), 2009, pp. 543 –548.

[60] McFee, B. et al., LibROSA: Audio and Music Signal Analysis in Python, in: Proceedings of the 14th Python in Science Conference, pp. 18–24, DOI: DOI: 10.25080/Majora-7b98e3ed-003.

[61] Englmeier, D. et al., Musical similarity analysis based on chroma features and text retrieval methods, in: Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, ed. by Ritter, N. et al., Bonn: Gesellschaft für Informatik e.V., 2015, pp. 183–192.

[62] Serra, J. et al., Chroma Binary Similarity and Local Alignment Applied to Cover Song Identification, in: 16 (Sept. 2008), pp. 1138 –1151, DOI: 10.1109/TASL.2008.924595.

[63] P.W. Ellis, D. and E. Poliner, G., Identifying 'Cover Songs' with Chroma Features and Dynamic Programming Beat Tracking, in: vol. 4, May 2007, pp. IV–1429, DOI: 10.1109/ICASSP.2007.367348.

[64] Mathworks, xcorr2, URL: https://www.mathworks.com/help/signal/ref/xcorr2.html.

[65] P W Ellis, D. and Cotton, C., The 2007 LabROSA cover song detection system, in: (Jan. 2007), DOI: DOI:10.7916/D8959SXK.

[66] Tzanetakis, G. and Cook, P., Musical Genre Classification of Audio Signals, in: IEEE Transactions on Speech and Audio Processing 10 (Aug. 2002), pp. 293 –302, DOI: 10.1109/TSA.2002.800560.

[67] Gruhne, M., Dittmar, C., and Gärtner, D., Improving Rhythmic Similarity Computation by Beat Histogram Transformations. In: ISMIR, Jan. 2009, pp. 177–182.

[68] Lidy, T. and Rauber, A., Evaluation of Feature Extractors and Psycho-Acoustic Transformations for Music Genre Classification. In: ISMIR, Jan. 2005, pp. 34–41.

[69] Audio Feature Extraction, URL: https://github.com/tuwien-musicir/rp_extract.

[70] Audio Feature Extraction - Rhythm Patterns, URL: http://www.ifs.tuwien.ac.at/mir/audiofeatureextraction.html.

[71] Pampalk, E., Computational Models of Music Similarity and their Application in Music Information Retrieval, PhD thesis, 2006.

[72] Pohle, T. et al., On Rhythm and General Music Similarity. In: 10th International Society for Music Information Retrieval Conference (ISMIR'09), Jan. 2009, pp. 525–530.

[73] Schoder, J., MusicSimilarity-Spark, URL: https://github.com/oObqpdOo/MusicSimilarity-Spark.

[74] mpi4py, URL: https://pypi.org/project/mpi4py/.

[75] beegfs, URL: http://www.beegfs.io/content/latest-release/.

[76] Slurm Workload Manager, URL: https://slurm.schedmd.com/documentation.html.

[77] Spotify company info, URL: https://newsroom.spotify.com/company-info/.

[78] Locality Sensitive Hashing, URL: https://spark.apache.org/docs/2.4.0/ml-features.html#lsh-operations.

[79] edlib, URL: https://github.com/Martinsos/edlib.

[80] Šošić, M. and Šikić, M., Edlib: a C/C++ library for fast, exact sequence alignment using edit distance, in: bioRxiv (2016), DOI: 10.1101/070649, eprint: https://www.biorxiv.org/content/early/2016/08/23/070649.full.pdf, URL: https://www.biorxiv.org/content/early/2016/08/23/070649.

7. Appendix

7.1 Feature Analysis

Scatter Matrix, 1 Song (Soundtack) from 50 (5 genres) song sample in Figure 7.1
Main diagonal = Kernel Density Estimation

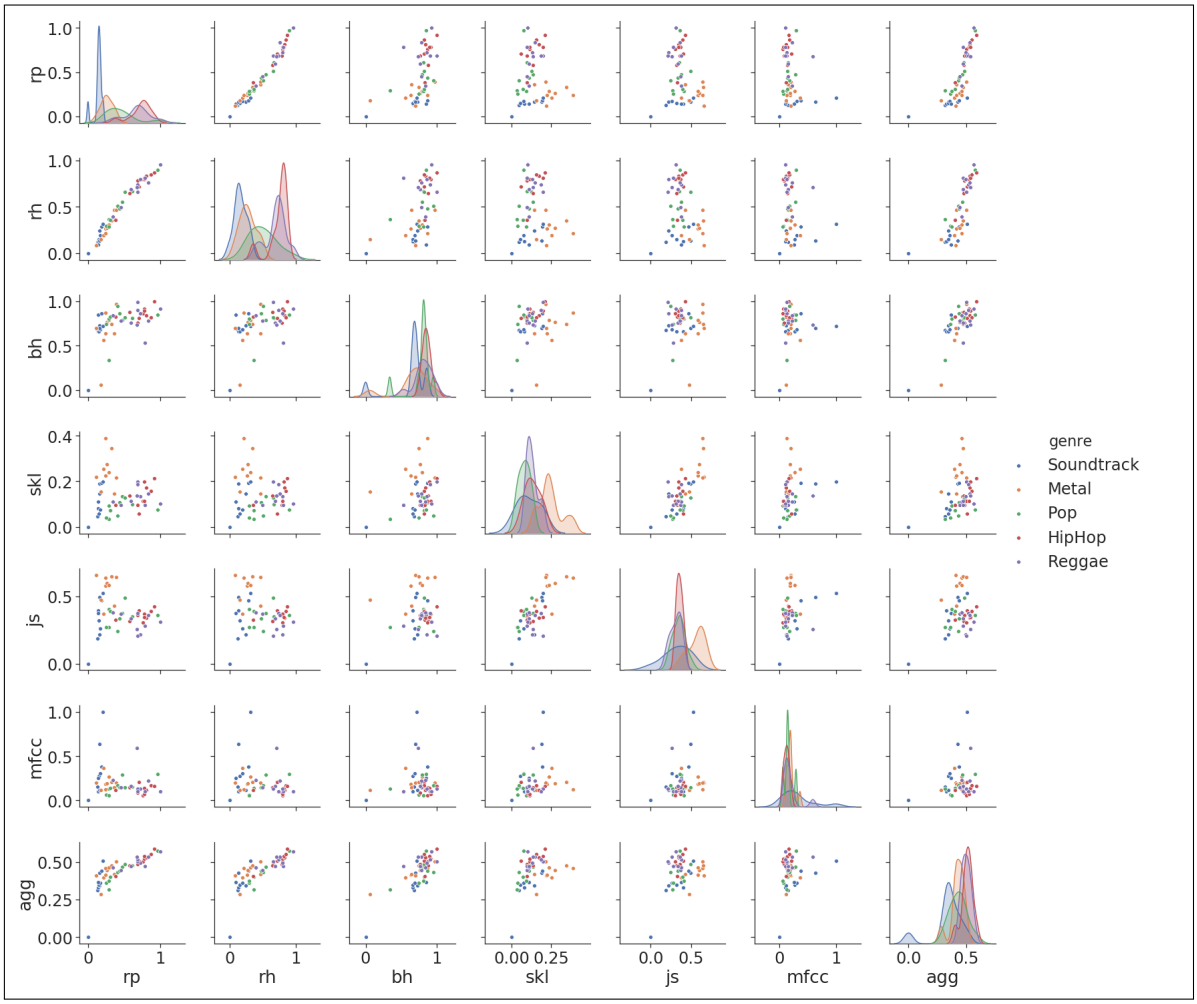


Figure 7.1: Distances 1 random song (Soundtrack), 5 genres (10 songs each)

7.2 Spotipy Data Extraction

```
from __future__ import print_function
from spotipy.oauth2 import SpotifyClientCredentials
import json, sys, spotipy, time, os.path
import requests, urllib
import matplotlib.pyplot as pl
import h5json, scipy
import numpy as np
from scipy.spatial import distance

reload(sys)
sys.setdefaultencoding('utf8')
client_credentials_manager = SpotifyClientCredentials()
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
if len(sys.argv) > 1:
  uri = sys.argv[1]
else:
  uri = 'spotify:user:bqpd:playlist:5oF8D71X38WwzeRUdyvpmd'
username = uri.split(':')[2]
playlist_id = uri.split(':')[4]
playlist = sp.user_playlist(username, playlist_id)
results = sp.user_playlists(username, limit=50)
playlist_length = playlist['tracks']['total']
path = os.getcwd()
path = path + "/crawled_data"
playlist_name = playlist['name']
directory = path + "/" + playlist_name
if not os.path.exists(directory):
os.makedirs(directory)
t_start = time.time()
f_feat = open(path + "/" + playlist_name + "/featurevector.txt","w")
f_feat.write("Features: \n")
f_feat.close()
feat_vec = []
feat_num = []
feat_name = []
for num in range(0, playlist_length, 100):
  results = sp.user_playlist_tracks(username, playlist_id, limit=100, offset=int(num
))
  tracks = results
  for i, item in enumerate(tracks['items']):
    track = item['track']
    track_id = str(track['id'])
    path = os.getcwd()
```

```
    path = path + "/crawled_data"
    artist = str(track['artists'][0]['name'])
    songtitle = str(track['name'])
    artist = artist.replace("/", "")
    songtitle = songtitle.replace("/", "")
    artist = artist.replace("$", "")
    songtitle = songtitle.replace("$", "")
    number = i + num
    name = str(number) + " - " + artist + " - " + songtitle
    directory = path + "/" + playlist_name + "/" + name
    prev_url = track['preview_url']
    if not prev_url == None:
      if not os.path.exists(directory):
        os.makedirs(directory)
      filename = directory + "/" + artist + " - " + songtitle + ".mp3"
      urllib.urlretrieve(prev_url, filename)
      tid = 'spotify:track:' + track['id']
      analysis = sp.audio_analysis(tid)
      with open(directory + "/" + songtitle + '_analysis.json', 'w') as outfile:
        json.dump(analysis, outfile)
      outfile.close()
      segments = analysis["segments"]
      bars = analysis["bars"]
      beats = analysis["beats"]
      tid = str(tid)
      features = sp.audio_features(tid)
      with open(directory + "/" + songtitle + '_features.json', 'w') as outfile:
        json.dump(features, outfile)
      outfile.close()
      acousticness = features[0]['acousticness']
      danceability = features[0]['danceability']
      energy = features[0]['energy']
      instrumentalness = features[0]['instrumentalness']
      liveness = features[0]['liveness']
      loudness = features[0]['loudness']
      speechiness = features[0]['speechiness']
      valence = features[0]['valence']
      feat_vec.append(scipy.array([acousticness, danceability, instrumentalness,
liveness, loudness, speechiness, valence]))
    else:
      print("no url - entry: " + artist + " - " + songtitle)
      print(track_id + "\n")
t_delta = time.time() - t_start
print ("features retrieved in %.2f seconds" % (t_delta,))
dist = distance.euclidean(feat_vec[0], feat_vec[1])
```

7.3 CD Contents

Feature Extraction Code

- mpi4py_ara_features.py
- mpi4py_ara_files.py
- mpi4py_ara_rhythm.py
- *.sbatch files

Spark Recommender Code

- spark_ara_df.py (unique DataFrames approach)
- spark_ara_filter_refine.py (single merged DataFrame approach)
- spark_ara_mergeddf.py (filter and refine approach)
- spark_ara_rdd.py (single RDDs approach)

PDFs

- PDF private music collection
- digital copy of this thesis

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This thesis was not previously presented to another examination board and has not been published in German, English or any other language. The author has no objections to make the present master's thesis available for public use in the University Archives.

Jena, Sunday 22nd September, 2019

Johannes Schoder