Faculty of Mathematics and Computer Sciences

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Master's Thesis

Music Similarity Analysis Using the Big Data Framework Spark

presented by:	Johannes Schoder
born:	03. September 1994
ID:	169197
course of studies:	M.Sc. Informatik
supervisors:	Prof. Dr. Martin Bücker, Ralf Seidler

Contents

Abbreviations **iv**

List of Figures **v**

List of Tables **vii**

List of Code Snippets **viii**

1 Introduction **1**
- 1.1 Why and how? . . . 1
- 1.2 Overview . . . 2
- 1.3 Conclusions . . . 3

2 Music Information Retrieval and Big Data **4**
- 2.1 Audio Features . . . 4
  - 2.1.1 Fourier Transformation . . . 4
  - 2.1.2 MFCC . . . 5
  - 2.1.3 other audio features . . . 8
- 2.2 MIR Toolkits . . . 9
  - 2.2.1 Low-level audio feature extraction . . . 9
  - 2.2.2 Music Similarity . . . 9
  - 2.2.3 Melody/ pitch extraction . . . 10
- 2.3 Music Similarity Measurements . . . 12
  - 2.3.1 Timbre based . . . 12
  - 2.3.2 Pitch based . . . 12
  - 2.3.3 Rhythm based . . . 12
  - 2.3.4 Metadata based/ Collaborative Filtering . . . 12
  - 2.3.5 Note based . . . 13
  - 2.3.6 Genre specific features . . . 13
  - 2.3.7 summary . . . 14
- 2.4 Data aggregation . . . 14

2.4.1 Datasets 14
2.4.2 Alternatives 16
2.5 Big Data 19
2.5.1 Hadoop 20
2.5.2 Spark 21
2.5.3 Music Similarity with Big Data Frameworks 26

3 Similarity Analysis **27**
3.1 Timbre Similarity 27
3.1.1 Euclidean Distance 27
3.1.2 Single Gaussian Model 28
3.1.3 Gaussian Mixture Models and block-level features 29
3.1.4 Validation 29
3.2 Melodic Similarity 32
3.2.1 Representation 32
3.2.2 Chroma Features pre-processing 32
3.2.3 Similarity of chroma features 35
3.2.4 Validation 41
3.3 Rhythmic Similarity 41
3.3.1 Beat histogram 41
3.3.2 Rhythm patterns 42
3.3.3 Rhythm Histogram 44
3.3.4 cross-correlation 44
3.4 Summary 45
3.4.1 Timbre Similarity 45
3.4.2 Melodic Similarity 46
3.4.3 Rhythm Similarity 46
3.4.4 Feature files 47

4 Implementation **48**
4.1 Audio Feature Extraction 48
4.1.1 Test Datasets 48
4.1.2 Feature Extraction Performance 49
4.2 Big Data Framework Spark 55
4.2.1 underlying hardware 55
4.2.2 workflow 56
4.2.3 Data preparation 56
4.2.4 Distance Computation 58
4.2.5 distance scaling 66

4.2.6 Combining different measurements . . . 67
4.2.7 performance . . . 68
4.2.8 possible improvements and additions . . . 75

5 Results **76**
5.1 Runtime . . . 76
5.2 feature separation quality . . . 76
5.3 Cover song identification . . . 83
5.4 Genre similarity . . . 84
5.5 ... and beyond genre boundaries . . . 84
5.6 Summary and outlook . . . 85

References **86**

6 Appendix **92**
6.1 Spotipy Data Miner . . . 92

Abbreviations

BH	beat histogram
BPM	beats per minute
BRP	bucketed random projection
DAG	directed acyclic graph
DCT	discrete cosine transformation
DF	Spark DataFrame
DTW	dynamic time warping
ESA	explicit semantic analysis
FFT	fast fourier transformation
GMM	gaussian mixture model
HDFS	hadoop distributed file system
HT	hyperthreading
JS divergence	Jensen Shannon
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
SQL	structured query language
UDF	user defined function
YARN	yet another resource negotiator

List of Figures

2.1 Frequency Space 5
2.2 Sweep Sinal 6
2.3 MFCCs 6
2.4 Timbre guitar vs. piano 7
2.5 MFCC statistics guitar vs. piano 7
2.6 MFCC statistics guitar vs. piano 8
2.7 Features of the song Layla by Eric Clapton 8
2.8 Features of the song Layla by Eric Clapton 9
2.9 Original Scores 10
2.10 Aubio 11
2.11 Melodia 11
2.12 Transcription 12
2.13 Datasets 15
2.14 Spotify API 17
2.15 million song dataset genre distribution 19
2.16 MapReduce [57] 21
2.17 Spark Cluster 22
2.18 Spark Application UI 23

3.1 Construction Noise 30
3.2 Chroma Features 32
3.3 Bandpass - Sia 33
3.4 Thresholded Chroma Features - Sia 34
3.5 Processed Chroma Features - Sia 34
3.6 Workflow chroma feature extraction 35
3.7 Processing Step 3 Chroma Features 35
3.8 cross-correlation 38
3.9 beat-aligned chromagram 39
3.10 Cross-correlation 40
3.11 Cross-correlation 40

3.12 Cross-correlation filtered . . . 41
3.13 Beat Histogram . . . 42
3.14 Rhythmic Patterns . . . 43
3.15 Rhythm Pattern extraction [71] . . . 43
3.16 Rhythm Histogram . . . 44
3.17 Detected Onsets (first 30 seconds) . . . 45

4.1 Performance of various toolkits on a single computer . . . 53
4.2 File sizes . . . 55
4.3 Workflow Spark . . . 56
4.4 Lazy evaluation and caching . . . 66
4.5 Performance on different feature types . . . 69
4.6 Workflow Merged DF . . . 71
4.7 Performance ARA, full workload, (MFCC + Notes + RP) . . . 71
4.8 Performance ARA, full workload, (JS + Chroma + RP) . . . 72
4.9 two subsequent songs, all features . . . 72
4.10 Descending importance filter and refine, all features . . . 74

5.1 Correlation of features . . . 76
5.2 Correlation of features . . . 77
5.3 correlation 95 songs, 19 genres (5 each), 1517 artists . . . 77
5.4 correlation 95 songs, 19 genres (5 each), 1517 artists . . . 78
5.5 distances 1 song (Soundtrack), 5 genres (10 songs each) . . . 79
5.6 distances 1 song, Rock/ Metal, 1517 artists, 4 genres . . . 80
5.7 distances 1 song, electronic, 1517 artists, 4 genres . . . 81
5.8 distances 1 song (Classical), 4 genres (10 songs each) . . . 82
5.9 genre recall 100 songs . . . 83
5.10 genre recall 100 songs . . . 84

List of Tables

2.1 music datasets 16

4.1 appropriate music datasets 49

List of Code Snippets

2.1 MIR Toolkit Similarity 10
2.2 example cluster configuration python 24
2.3 lazy evaluation 25
4.1 librosa 49
4.2 essentia standard 50
4.3 essentia streaming 51
4.4 parallel python 52
4.5 mpi4py 54
4.6 notes preprocessing 57
4.7 rp preprocessing 57
4.8 euclidean distance DF 58
4.9 Filter for requested song 59
4.10 euclidean distance RDD 59
4.11 bucketed random projection 60
4.12 cross-correlation scipy 61
4.13 cross-correlation numpy 61
4.14 Jensen-Shannon-like Divergence 62
4.15 Kullback-Leibler Divergence 63
4.16 Levenshtein DataFrame 64
4.17 Levenshtein RDD 64
4.18 Spark lazy evaluation 65
4.19 Minimum and maximum aggregation separate 67
4.20 Minimum and maximum aggregation optimized 67
4.21 cluster setup 68

Abstract

Estimating music similarity by merging various similarity measurements with the help of the Big Data Framework Spark.

This thesis is about the comparison of construction noise and modern day music. The field of music information retrieval (MIR) in computer science is mostly a data driven and purely mathematical topic. The goal of this thesis is to merge the fields of computer science with music theoretical knowledge and to find potential weak spots of current music similarity algorithms focusing on single aspects like melody, rhythm and timbre.

1. Introduction

The idea originated from Dr. T. Bosse from the chair for advanced computing at the Friedrich-Schiller-University (FSU) in Jena. When proposing the idea for a master thesis with the topic of "Music similarity measurement using genre specific features" 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. This thesis is meant to evaluate how music similarity algorithms compare construction noise to common musical genres and how to possibly improve these algorithms.

1.1 Why and how?

Why is music similarity research even necessary?

First of all, there is no fixed definition of music similarity so far. This is one of the first problems, dealing with music similarity. This topic offers multifaceted approaches. Merging multiple approaches with different weights can offer a more diverse music recommendation system. To do this, a lot of different data is required.
Content (music features) and context (listener behaviour) data can be fed into a big data framework to speed up operations. Collecting this data for large amounts of songs results in big datasets that need to be explored efficiently.

What to improve?

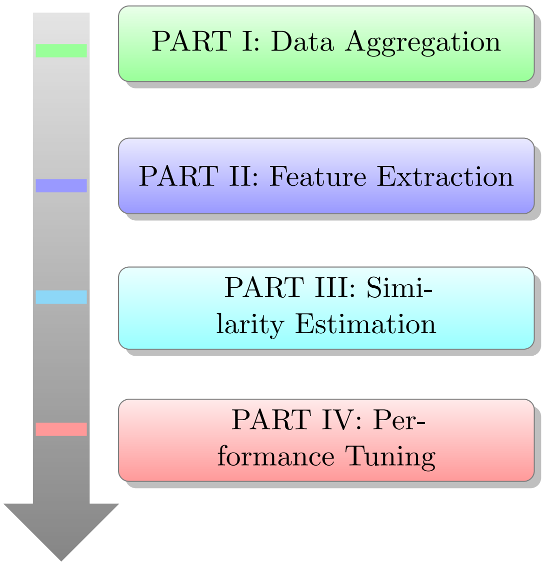
"[...] Spotify Radio, iTunes Radio, Google Play Access All Areas and Xbox Music. Recommendations are typically made using (undisclosed) content-based retrieval techniques, collaborative filtering data or a combination thereof." [1, p. 9] The goal of this work is, to propose a transparent music similarity retrieval method based on various weighted contextual and content-based data. Applying different weights to different features allows similarity retrieval methods to search for different kind of similarity. E.g. weighing the tempo and beat of a song more then melodic similarity allows the creation of playlists for workout and sport, whilst melodic/ timbre etc. similarities allows to search for similar songs from musical subgenres. The user would get to decide what kind of playlist he wants to create. Adding contextual data and feeding it to an algorithm could add more or less popular music to the playlist with the goal to discover new upcoming artists or get other popular and most listened music. For this thesis however, the focus lies on content-based data/ audio features.

How to approach this?

First of all, a lot of data is required. In the first part, different scientific datasets are evaluated. Secondly, the available features are shown and explained. In the third part, different metrics are explained using the previously explored features. Lastly, a big data approach to efficiently use the gathered features is proposed and evaluated. A way to evaluate the results is also proposed.

1.2 Overview

Structure:



1.3 Conclusions

Why using a big data framework would help: music similarity is not well defined. It is a rather subjective value that differs 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. The usage of a big data framework allows to create a variable/ fuzzy metric definition. Various parameters could easily be taken into consideration when calculating the musical distance of two different pieces. Using a Big Data framework, the problem of the fuzzy definition of music similarity could be avoided, if a metric can be found, that takes multiple of the accounted features of this thesis into consideration. Available information includes metadata, user data, audio features, sheet music and more.

The idea of using genre specific features could be evaluated any further
Another important question is how to measure similarity algorithms. There are a few possibilities like genre, composer/ interpret or cover song identification. Or actual user data. MSD Challenge Dataset usable?[2]

2. Music Information Retrieval and Big Data

The field of music information retrieval (MIR) is a large research area combining studies in computer science like signal processing and machine learning as well as 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 approaches to music similarity. Various datasets are presented and an overview over Big Data frameworks is given as well.

2.1 Audio Features

This section provides an overview about different music similarity measurements, audio features and metrics.
More in-depth information about the different metrics is given in chapter 3.1, 3.2 and 3.3

2.1.1 Fourier Transformation

Most of the algorithms start with switching from the time domain to the frequency domain, by performing a Discrete Fourier transform as described in equation 2.1 and then compute the power spectrum (equation 2.2)

$$X_m = \sum_{k=0}^{K-1} x_k \cdot e^{-\frac{K}{2 \cdot \pi \cdot i} \cdot k \cdot m} \tag{2.1}$$

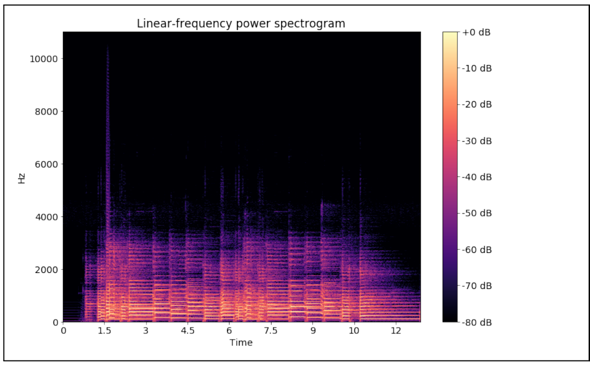
$$|X_m| = \sqrt{Re(X_m)^2 + Im(X_m)^2} \tag{2.2}$$

Figure 2.1a shows the spectrogram (spectrum of frequencies over time) of the first bars of the song Layla by Eric Clapton. The sound sample was recorded on an electric guitar. Due to the fact that the human ear perceives sound in a non-linear matter, a logarithmic or Mel-scale is better to represent different pitches.

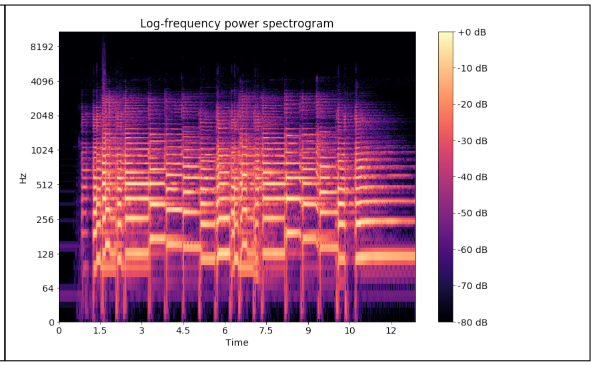
For example the note A4 is perceived at a frequency of 440Hz, the A note of next octave (A5) is at 880Hz and the next one is at 1600Hz and so on. The Mel-scale [1, pp. 53f] was introduced to resemble the human perception of frequency (equation 2.3)

$$m = 1127 \cdot ln(1 + \frac{f}{700}) \tag{2.3}$$

The following plots were created with the librosa library [3]. The high dimensionality of



(a) Spectrogram



(b) log scaled spectrogram

Figure 2.1: Frequency Space

the data is a problem for machine learning applications and music similarity tasks, as computation based on a vector with such a high dimensionality of the data would take to long. Given a sample rate of $f_s = 44,1kHz$ (usual CD sample-rate) and a length of a song of about $t = 180s$, the time domain contains 7938000 data points usually with 16-bit resolution.

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

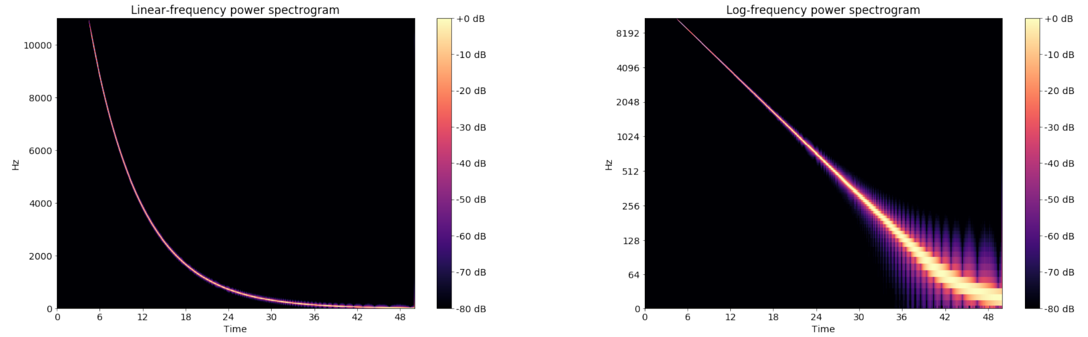
Calculating a FFT with a window size of 1024 samples and a hop size of 512 samples (resulting in the factor 1.5 in equation 2.5)[1], the full resulting spectrogram would contain 11627 frames with 1024 frequency values per frame. (eq: 2.5)

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

To reduce the dimensionality of the feature vector, a typical approach in MIR would be to calculate the so called Mel Frequency Cepstral Coefficients (MFCCs). They are described in more detail in the next section.

2.1.2 MFCC

This section gives a brief overview over the computation of the MFCC as stated in [1, pp. 55ff] because out 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 features. Figure 2.2 shows the magnitude spectrum of a frequency sweep signal.



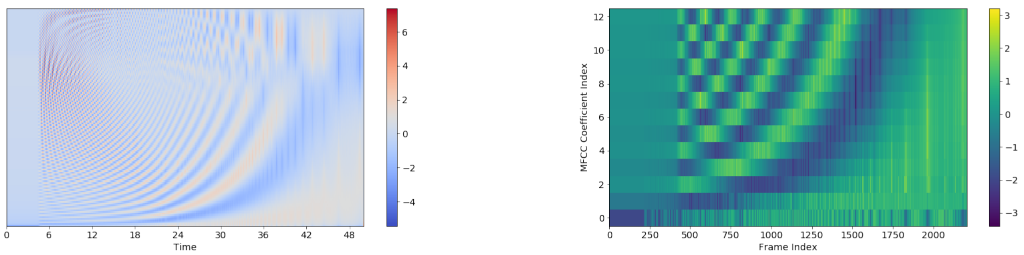
(a) Sweep signal linear (b) Sweep signal logarithmic

Figure 2.2: Sweep Sinal

First of all the magnitude spectrum is transformed to the Mel-scale by assigning each frequency value to a Mel- band. Doing this a dimension reduction can be done, 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.

$$X_k = \sum_{n=0}^{N-1} x_n cos\left[\frac{\pi}{N}(n+\frac{1}{2})k\right] \tag{2.6}$$

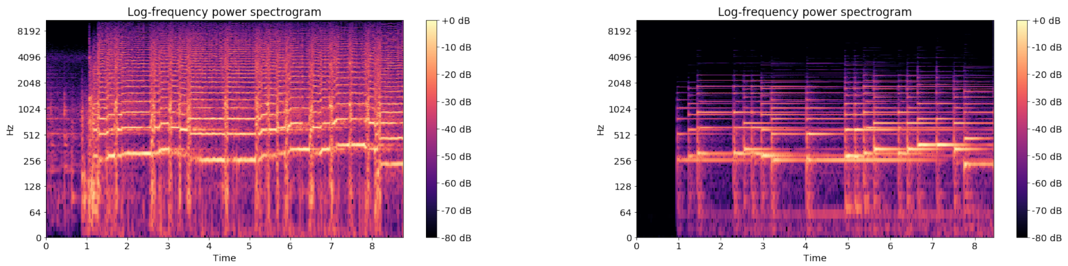
Figure 2.3a shows the resulting MFCCs with a high resolution of 1024 Mel bands. This is not what in a usual application would be done, because this is nearly as high dimensional as the original spectrogram. Figure 2.3b shows the MFCC reduced to 13 Mel Bands. To better visualize the MFCC features all values are typically scaled to have a standard deviation of 1 and a mean value of 0 per band in the plots.



(a) MFCC High Resolution (b) MFCC 12 Bands scaled

Figure 2.3: MFCCs

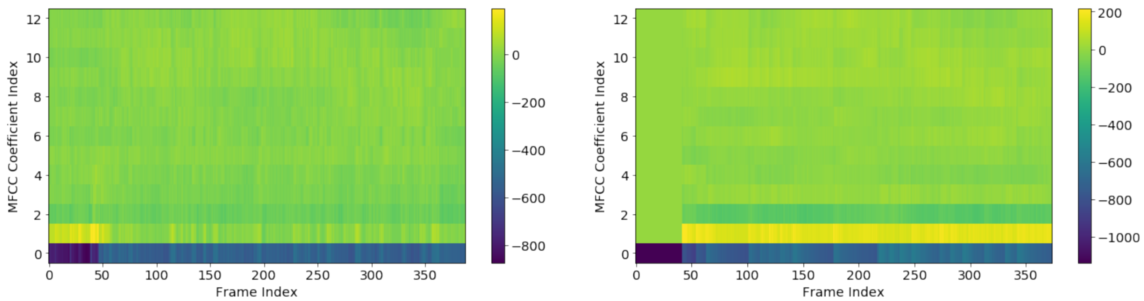
MFCCs were found to be suited to represent timbral properties of music [1, p. 55 ff]. To describe a tone, three moments can be used according to [4, pp. 15]: tonal intensity perceived as loudness, the tonal quality perceived as the pitch and the timbre or tonal color as the third moment. Looking at an example melody line played on a electric distorted guitar and a piano, distinct differences can be seen. Due to the physical properties of a string every note played consists of the main frequency (the actually played key) and so called harmonic overtones because of the way a string e.g. in a piano vibrates and the wooden body resonates. Typically the overtones 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 decline faster or slower or don't appear at all. The electrically amplified guitar amplifies these overtones as well, which is visible in figure 2.4b.



(a) Guitar (b) Piano

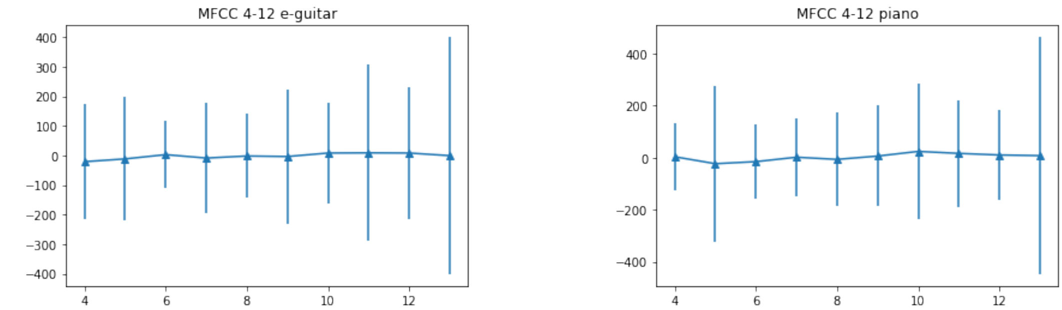
Figure 2.4: Timbre guitar vs. piano

Looking at the MFCCs these differences are also visible. This time the MFCC plots are pictured without the previously mentioned scaling and the mean value and standard deviance of the MFCCs four to 13 are pictured. This calculation of statistical features is later explained in chapter 3.1. Although both times the exact same melody is played in the same tempo, these features vary due to the different timbral properties of the instruments.



(a) Guitar (b) Piano

Figure 2.5: MFCC statistics guitar vs. piano



(a) Guitar (b) Piano

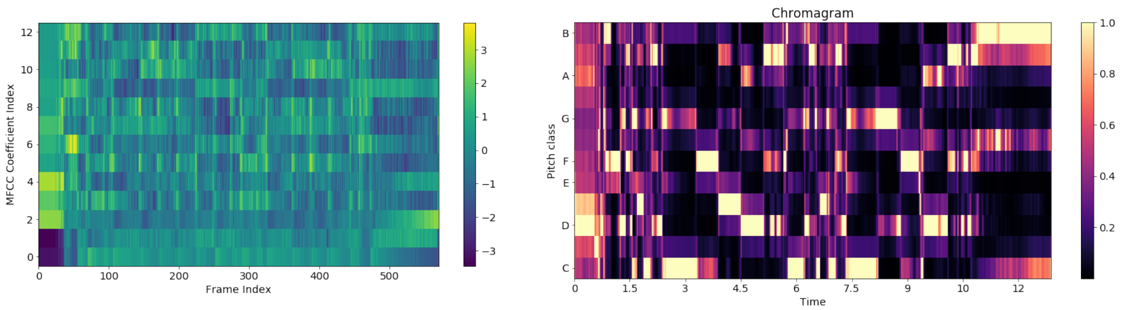
Figure 2.6: MFCC statistics guitar vs. piano

2.1.3 other audio features

As another, better comprehensible, higher-level set of features, the chromagram represent the tonal properties of a song. The chroma plot (Figure 2.7b) shows the distribution of the different pitches mapped to the various keys in one octave. The values are normalized to one by the strongest dimension. So if all values are close to one, it is most likely that there is only noise or silence at that frame in the recording, as depicted in the first frames of figure 2.7b. 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 extend.

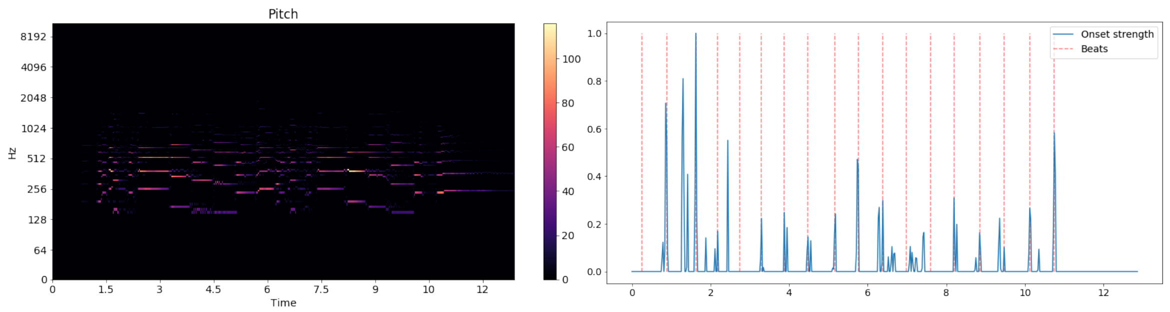
Figure 2.8a figures the pitch curve of the recording. None but the most dominant frequencies are shown. Pitches below a certain threshold are filtered out. In contrast to the chromagram the pitch curve provides information over the whole spectrum and is not limited to one octave. These Pitch curves can be used to estimate and transcribe musical notes from audio data as presented in 2.2.3.

The rhythmic low-level features of a song include the estimation of the overall tempo, beats and onset events. The Plot in figure 2.8b shows the onsets and estimated beats in the first 10 seconds from a recording of the song Layla by Eric Clapton.



(a) MFCC (b) Chroma Features

Figure 2.7: Features of the song Layla by Eric Clapton



(a) Pitch

(b) Rhythm/ Beat

Figure 2.8: Features of the song Layla by Eric Clapton

2.2 MIR Toolkits

2.2.1 Low-level audio feature extraction

To extract audio features like the ones presented in section 2.1.1 (mfccs, chromagram, beats, onsets), a wide variety of toolkits is publicly available, a few are presented in [5]. The YAAFE toolkit [6] is able to extract a lot of different audio features like energy, mfcc or loudness directly into the hadoop file format h5 making it ideal for big data frameworks to use. It can be used with C++, Python or Matlab.
The Essentia toolkit [7] is pretty similar to YAAFE, extending it by the calculation of the rhythm decriptors, bpm etc. It can also be used in C++ and Python
The Librosa Toolkit provides similar functionality [8] as Essentia. It is user-friendly, well documented and can be called from a Jupyter-Notebook [9], allowing rapid prototyping and testing of different algorithms. Most of the plots from section 2.1.1 were created using librosa. The code for the extraction of low level features with essentia and librosa is given in chapter 4.1 as well as a performance analysis.

2.2.2 Music Similarity

The easiest way to test state of the art music similarity algorithms is to use the open source toolkit Musly [10]. It is based on statistical models of MFCC features and calculates the distances between songs very fast, supporting OpenMP acceleration. It offers the classical mandel-ellis similarity method [11] and a timbre based improved version of the mandel-ellis algorithm using a jenson-shannon-like divergence [12]. More details and a re-implementation of this toolkit is presented in chapter 3.1
As another option, the MIR Toolkit [13] is a toolbox for Matlab [14]. A port to GNU

Octave [15] is also available [16]. The short code snippet below 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: MIR Toolkit Similarity

2.2.3 Melody/ pitch extraction

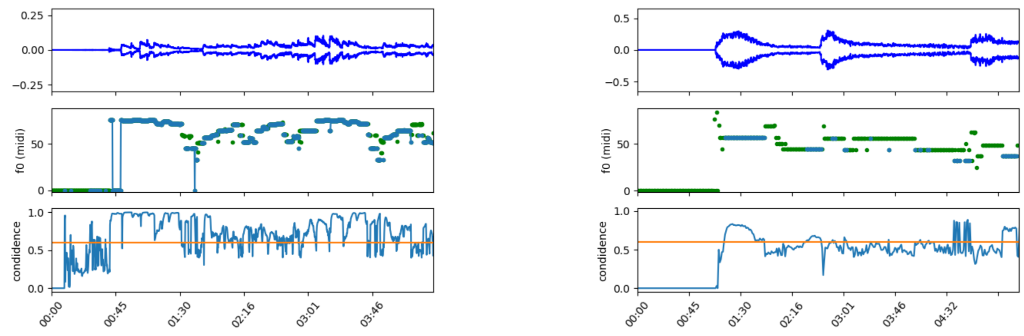
To test the various pitch extraction toolkits, a piece by Rachmaninoff and by Beethoven was used. The first three bars of Rachmaninoffs Prelude can be found in figure 2.9a Figure 2.9b shows the first five bars of Beethovens Bagatelle in A Minor (Für Elise). The first toolkit tested is Aubio [19]. The result can be seen in figure 2.10a and figure

(a) Rachmaninoff Prelude in C Sharp Minor [17]

(b) Für Elise [18]

Figure 2.9: Original Scores

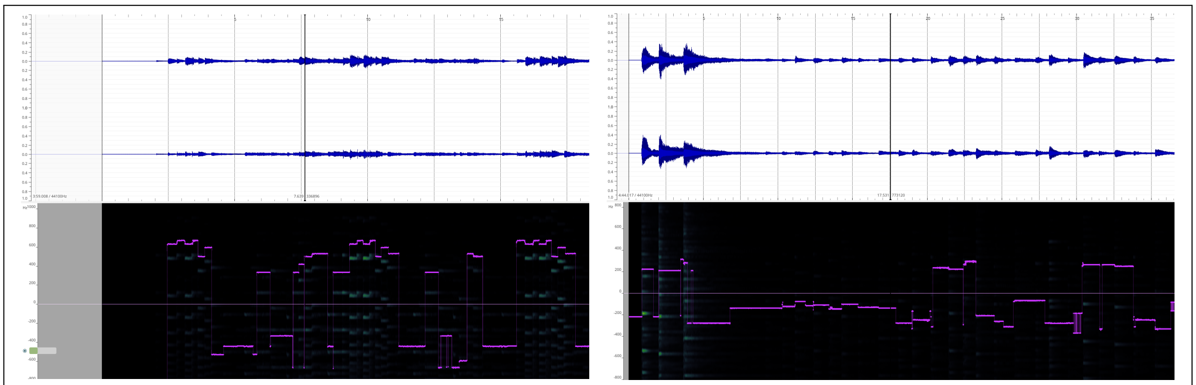
2.10b The upper subplot shows the waveform of the first few seconds of each piece. The second plot figures the estimated pitch with green dots. If the pitch is zero, then no pitch could be estimated, most likely because the associated frame contains silence. The blue dots resemble the estimated pitches, where the confidence (shown as the blue graph in the third subplot) is above a certain threshold (the orange line).



(a) Für Elise Aubio Pitch (b) Rachmaninoff Prelude Aubio Pitch

Figure 2.10: Aubio

The other melody extraction tool is Melodia[20], which is available as a VAMP plugin and can be used together with the Sonic Visualizer[21]
The results are shown in figure 2.11a and 2.11b. The purple line is the estimated pitch, however there are large jumps between different octaves of the harmonics.



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

Figure 2.11: Melodia

Sadly the conversion to MIDI does not work flawlessly. It is clearly visible in figure 2.12 that the transcription does not work accurately enough even for a classical music piece with only one instrument. Figure 2.12a shows the output of a python script using the Melodia VAMP plugin to calculate a MIDI file containing the main melody line and figure 2.12b 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: Transcription

2.3 Music Similarity Measurements

2.3.1 Timbre based

As done by: [22]
The proposed approach by [22] is, to take mfccs as low level features and compute statistical features like mean, standard deviation and covariance of the different mfccs to reduce dimensionality before computing similarities. Various similarity measurements are proposed and evaluated in chapter 3.1 The algorithm used in this thesis is described in [22, pp. 17ff]. The framework used to determine the song similarity for the first tests in chapter 3.1 is the MUSLY toolkit [10].

2.3.2 Pitch based

As done by: [23]
The proposed approach by [23] is, to take mid-level features like the chromagram or estimated main pitchline instead of high-level features like sheet music or low-level features like gaussian mixture models of MFCCs. A more detailed analysis of this topic is given in chapter 3.2

2.3.3 Rhythm based

As done by: [24]
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 2.8b could be used as a starting point for rhythmic similarity retrieval. More advanced approaches make use of the beat histogram, rhythm histogram and rhythm patterns. An in depth overview is given in chapter 3.3

2.3.4 Metadata based/ Collaborative Filtering

As done by: [25] and [26] using the Million Song Dataset (MSD) [27]
In 2012 the MSD Challenge was brought to the MIR community. The researches were

challenged to give a list of song recommendations based on a large set of user data. So if user X listened a lot to artist A and artist B and user Y listens mostly to artist A and artist C, then maybe user X would like artist C as well. These kind of collective listening behavior based recommendations are called collaborative filtering [1, p. 192f.] and are pretty common in large music streaming services, although not necessarily representing direct musical similarity. These kind of recommendation systems tend to propose commonly well known artists rather than not so well known ones biasing the result. On the other hand these kind 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 the lyrics and also hybrid recommendation systems that combine lyrics, metadata and collaborative filtering are possible. However all of these recommendation strategies are not directly based on musical features and are not considered for this thesis. They are however a possible addition for a hybrid recommendation engine for future research.

2.3.5 Note based

As done by: [28]
For comparing musical pieces by their symbolic represenation (notes, tabulatures etc.) different text retrieval methods could be used. The MIDI datatype could be used, as it is a form of digital representation of music information. [28] uses a variation of the Levinshtein distance measurement. The problem with notation based algorithms is, that there are not many datasets available containing audio and MIDI information. As shown in 2.2.3 the automatic transcription of notes from raw audio does not work flawlessly. There is ongoing research to automatically annotate musical notes with the help of neural networks.[29] In chapter 3.3 an attempt by Xia (et. al) [30] to extract note information as text features from chromagrams and calculating the similarity by using the levinshtein distance is shown and evaluated.

2.3.6 Genre specific features

As done by: [31] for indian art music, by using 560 different combinations of different features. They state that: "We evaluate all possible combinations of the choices made at each step of the melodic similarity computation discussed in Section 2. 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" [31, p. 3]. This evaluation showed, that the choice of features and parameters for music similarity measurement is a critical point.

In Rock, Pop and Metal music, extraction of different guitar playstyles would be imaginable. Guitar Tab Extraction [32] Toolkits could be used to extract information if the guitar in a song is mostly plucked or strummed for instance. 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.

2.3.7 summary

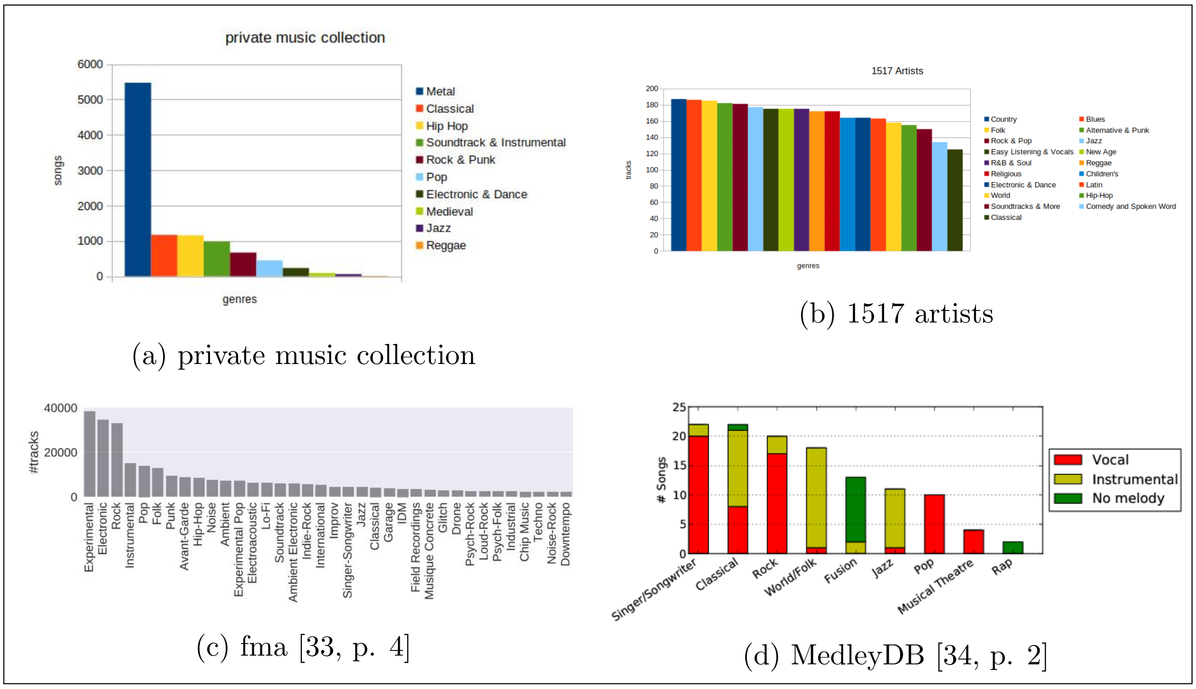
In this thesis music similarity measurements based on three different types of features are evaluated. The first is based on MFCCs to represent timbral features of the songs and therefor offering a set of features to make recommendations that are similar in tone color and should be able to make recommendations inside the boundaries of different genres. The second is based on chroma features/ note information collapsed to one octave to represent a measurement of melodic similarity. With these features the detection of cover versions should be possible. The third set of features is based on the rhythmic properties of a song. This should enable recommendations of songs with the same tempo and rhythmic structure, possibly also enabling the recommendation of songs within the same genre.

The usage of MIDI-files is not considered further due to the rather poor performance of score extraction tools for songs with multiple instruments and melody lines. Also the melodic component of the songs is already represented by the chroma features, although that limits the representation to one or at most a few octaves. Collaborative filtering is left out due to the fact that it does not necessarily represent the musical features and properties but the personal taste of music. And secondly no dataset with the required information and music files was found (2.4). Genre specific features are also not an option because this field is not yet very well researched and the development of algorithms and the extraction of features would go beyond the scope of this thesis.

2.4 Data aggregation

To evaluate the music similarity algorithms and metrics a lot of music data is needed.

2.4.1 Datasets



(a) private music collection

(b) 1517 artists

(c) fma [33, p. 4]

(d) MedleyDB [34, p. 2]

Figure 2.13: Datasets

Free Music Archive

The largest dataset is the Free Music Archive- dataset (fma) consisting of 106733 different songs totalling an amount of nearly one terabyte of music data from all kinds of different music genres [33]. There is also a lot of metadata information available for most of the songs.

Private music collection

The private music collection used in this work consist mainly of metal music. The music was legally purchased, all rights belong to the respective owners. The distribution of different songs per genre for this dataset is visualized in figure 2.13a.
Additionally a private recording dataset was used, consisting of ambient recordings and self produced music. Most of this music can be downloaded from soundcloud [35].
The private music collection is fully cataloged and the according pdf file is the appendices.

1517 artists and Musicnet

Another source of music is the Musicnet dataset [36]. It includes 330 pieces of classical music with musical notes as annotations. Other sources of musical information would be the 1517-Artists dataset containing 3180 songs of multiple genres. [37]

covers80

For a cover song detection system, the covers80 dataset is available [38] containing 80 original songs mostly from the musical genres rock and pop and 84 cover versions. These cover versions tend to differ significantly in musical style, rhythm and timbre from the original.
The ability to detect cover songs or different versions/ recording of a musical piece could be a good measurement for the efficiency of music similarity algorithms. In the next section one test case is presented, showing, that a MFCC based music similarity algorithm isn't able to detect different recordings of the same piano piece as most similar to each other.

MedleyDB

For a melody/ pitch based similarity analysis, multitrack datasets could provide useful data, due to the fact that the pitch estimation can be done instrument by instrument. Datasets available are the MedleyDB [34] and MedleyDB2 [39] datasets as well as the Open Multitrack Dataset [40] currently consisting of 593 multi-tracks in which the MedleyDB dataset is already included, leaving 481 other tracks for analysis.

Overview and other sources

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

fma	106.733 Songs
private	8484 Songs
1517 artists	3180 Songs
Maestro	1184 Songs (piano) + MIDI
musicnet	330 Songs (classical) + note annotation
Open Multitrack Testbed	593(481) Songs/ Multitracks
covers80	164 Songs (80 originals + 84 covers)
MedleyDB	122 Songs/ Multitracks
MedleyDB2	74 Songs/ Multitracks

Table 2.1: music datasets

2.4.2 Alternatives

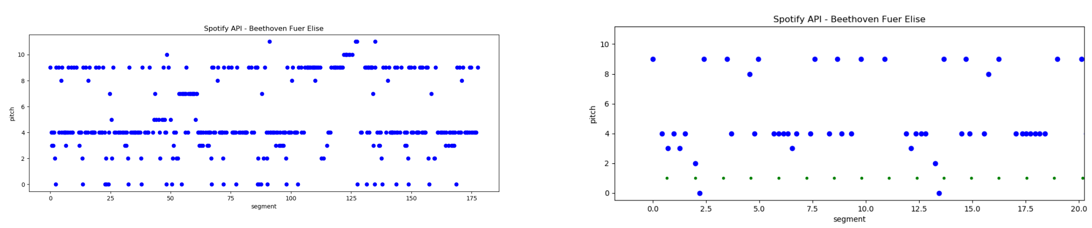
Spotify API/ Echonest

Another way of getting music information, audio analysis and metadata is by using the Spotify API[41] Part of the available audio features comes from the Echo Nest[42].

The Downside using the Spotify API is, that there is no packed and ready to use test dataset containing the relevant features. So for scientific purposes, a test dataset would have to be created first. With a small Python library named Spotipy, the available information can very easily be used and accessed. [43]
For the purpose of this thesis, the option of creating an own dataset using the Spotify API and spotipy was considered. Ten very small test playlists of different genres were created using the Spotify Playlist Miner [44]. Appendix 6.1 lists a small script, that is able to download all audio features and analysis data from all of the songs of a playlist, that contains a preview URL with a 30 second audio snippet. The audio features and analysis data is saved as a JSON file containing information over:

- acousticness
- danceability
- instrumentalness
- liveness
- loudness
- speechiness
- valence
- predicted key
- tempo

as well as pitch and timbre information, beats and bars.
In figure 2.14a the chroma features of the piano piece Für Elise by Beethoven are shown and figure 2.14b 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 zero representing the key C and 11 is representing a B. The Spotify API actually returns a chroma feature value for every single one of the keys per segment, where one segment is a section of samples that are relatively uniform in timbre and harmony. In the plots only the most dominant key per segment is shown.



(a) Für Elise Spotify pitch

(b) Für Elise detail

Figure 2.14: Spotify API

Together with the 30 second audio sample from which more features like MFCCs could be extracted. This data miner could provide all the information needed to build a large dataset for MIR. However the terms and conditions explicitly prohibits 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;" [45]

Therefore a larger user created dataset can not be used without the risk of legal infringements. However one could argue, that there is a difference between data mining and data crawling and for small datasets with the purpose of creating Spotify playlists, these restrictions may not apply.

In the sense of the Spotify Developer Terms of Service [46] there may be no legal infringements by creating a non-commercial playlist creation tool. [47] states, that by creating algorithmically-generated playlists similar to the "Discover Weekly" Playlists one may run into challenges if using such features commercially. However it does not prohibit the usage for non-commercial cases. Upon request the Spotify API developer team did not respond and therefor in this thesis the Spotify API wont be used to create a test dataset.

Million Song Dataset

Another outstanding and very large dataset is available with the Million Song Dataset (MSD)[27]. 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)[48], the last.fm dataset[49] and the Echo Nest API dataset[50]. Although the MSD does not contain any music files in the first place, 30 second samples could be gathered through simple scripts from 7digital.com when the dataset was made publicly available. On top of that the Echo Nest API data already contains a lot of audio features like pitch, loudness, energy and danceability to name just a few.

Another addition is the secondhand dataset, containing a list of cover songs in the million song dataset[51]

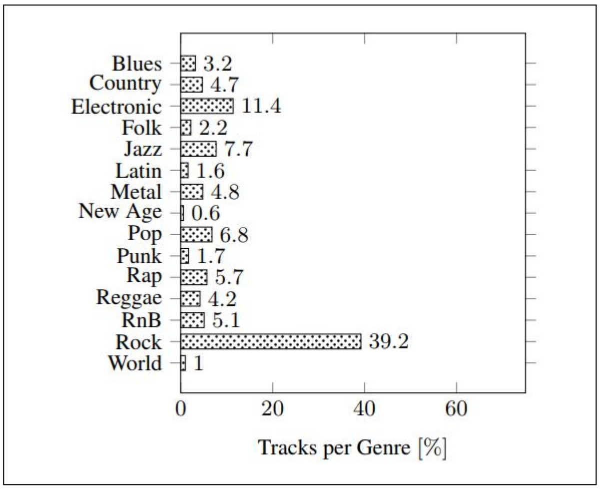


Figure 2.15: million song dataset genre distribution

Due to the fact that the Spotify API[41] also works with audio features from the Echo Nest[42], the MSD could be used in a big data environment to simulate the work with Spotify data, without manually mining the actual data. The MSD was actually already used in Big Data frameworks for music similarity retrieval based on metadata and user information[26]

Sadly 7digital does not offer the download of the 30 second sample files any more which makes this dataset unusable for this thesis, because missing audio features like mfccs can not be computed from the audio files itself.

2.5 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 [52] and Hadoop [53]. 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 [54].

Later chapter 4.2 deals with the implementation of the various similarity measurements with Spark, the handling of larger amounts of data, runtime analysis and the combination of multiple similarity measurements, while chapter 5 gives short overview over the achieved results using the Big Data framework to compare audio features.

2.5.1 Hadoop

With the ever growing availability of huge amounts of high dimensional data the need for toolkits and efficient algorithms to handle these grew as well over the past years. They key to handle Big Data is to use parallelity.
Search engine providers like Google and Yahoo firstly ran into the problem of using "internet-scale" data in the early 2000s when being faced with the problem of storing and processing the ever growing amount of indexes from documents in the internet. In 2003, Google presented their whitepaper called "The Google File System" [55]. MapReduce as a programming paradigm was 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 [56]. 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. [54, p. 6]
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 network at compute time when working with very large-scale data/ Big Data. One prerequisite is that the operations on the data are independent from each other. Hadoop follows this approach called "shared-nothing". The data can be processed locally on many nodes at the same time in parallel by splitting the data in independent small subsets without the need of communicating 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). [54, p. 7]
Hadoop is a scalable solution able to run 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 rate defining how many copies of each data chunk are stored on other nodes. This enables an error management where faulty operations can simply be restarted.
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. The core idea is to split a problem into many independent tasks.

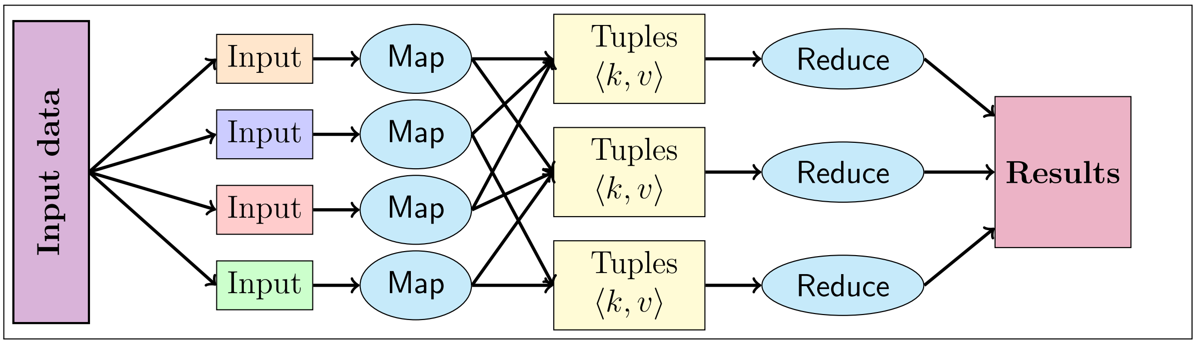


Figure 2.16: MapReduce [57]

In the first stage the input data is split in many chunks and distributed over the nodes of a cluster. This is usually managed by the distributed file system like the HDFS. One master nodes stores the addresses of all data chunks.
The data is fed into the mapper who operates 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 reducer. The reducer applies another method to all tuples with the same key.
The amount of key-value pairs at the output from the mapper 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 and small q (highly parallel with more network traffic) and small r and larger q (less network traffic but worse parallelism due to an overall smaller reducer count).

2.5.2 Spark

Hadoop as a Big Data processing framework has some downsides compared to other and newer options. Apache Spark was developed as an alternative to the implementation of MapReduce in Hadoop. The Spark project was started in 2009 and was created as a part of the Mesos research project. Spark is written in the programming language Scala 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 the hard drive. While Hadoop has to read all data from the disk and writes the results back to the disk, Spark is able to efficiently take advantage of the RAM memory, 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). RDDs are able to use the main memory across multiple machines in a cluster. [54, p. 13]
Figure 2.17 shows the simplified architecture of a compute cluster running Spark.

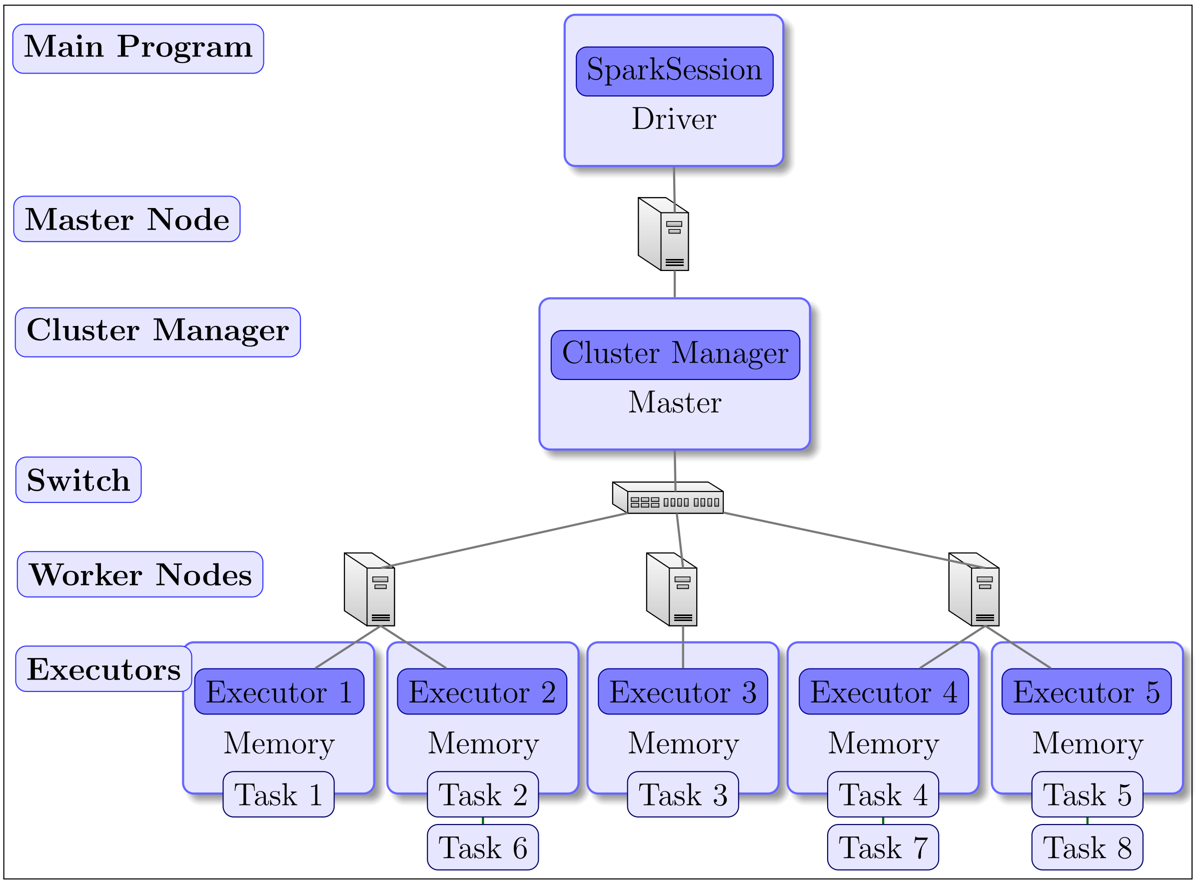
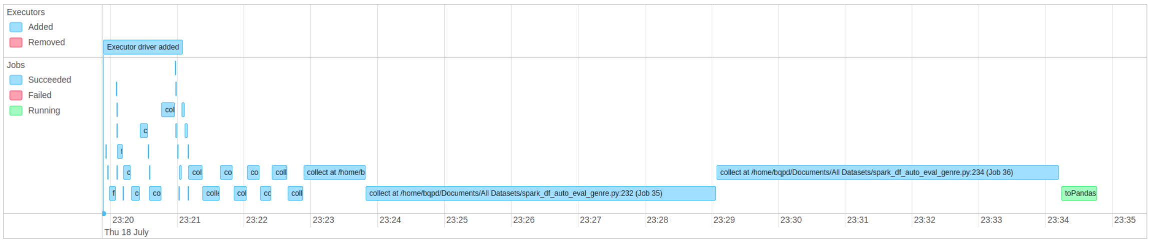


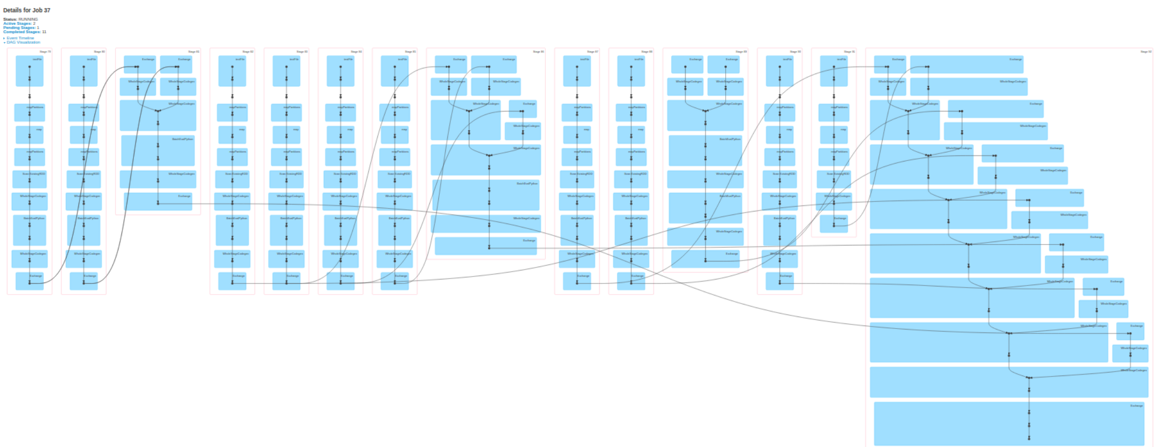
Figure 2.17: Spark Cluster

The core components of a Spark application are the Driver, the Master, the Cluster Manager and the Executors. The Driver is the process where the client submit their applications to. It is responsible for the panning and execution of the Spark program and returns status logs and results to the client. 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 SparkSession contain the necessary information to configure the cluster parameters, e.g. the amount of CPU cores and memory assigned to the executors and how many executors are spawned overall. 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 planned and directed acyclic graphs (DAG) with nodes that represent transformational or computational steps are created by the Spark Driver. These DAGs can be visualized by 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, as it also gives information about the computation time of the distinct tasks within a Spark program. [54, pp. 45 ff]
An Example is shown in figure 2.18



(a) Event Timeline



(b) DAG

Figure 2.18: Spark Application UI

The Worker nodes are the nodes in the cluster where the actual computation of the Spark DAG tasks take place. As defined within the SparkConf the Worker nodes spawn a finite or fixed number of Executors that reserve CPU and memory resources on the slave nodes and run in parallel. The Executors are hosted in JVMs on the Worker nodes. 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 ressources (containers) for the Executors and makes them available to the Driver. [54, 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. If the interactive option of using a spark shell is chosen, a SparkSession is automatically created and exited once the spark-shell is closed. 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)

```python
confCluster = SparkConf().setAppName("MusicSimilarity Cluster")
confCluster.set("spark.driver.memory", "1g")
confCluster.set("spark.executor.memory", "1g")
confCluster.set("spark.executor.memoryOverhead", "500m")
#Sum of the driver or executor memory plus the driver or executor memory overhead is always less than the value of yarn.nodemanager.resource.memory-mb
#confCluster.set("yarn.nodemanager.resource.memory-mb", "8192")
confCluster.set("spark.yarn.executor.memoryOverhead", "512")
#set cores of each executor and the driver -> less than avail -> more executors spawn
confCluster.set("spark.executor.cores", "1")
confCluster.set("spark.dynamicAllocation.enabled", "True")
confCluster.set("spark.dynamicAllocation.minExecutors", "4")
confCluster.set("spark.dynamicAllocation.maxExecutors", "8")
confCluster.set("yarn.nodemanager.vmem-check-enabled", "false")
sc = SparkContext(conf=confCluster)
sqlContext = SQLContext(sc)
spark = SparkSession.builder.master("cluster").appName("MusicSimilarity").
  getOrCreate()
```

Code Snippet 2.2: example cluster configuration python

Spark advantages

For this thesis the programming language of choice is Python. With its high-level Python API, Spark applications can take advantage of commonly known and widely used Python libraries such as Numpy or Scipy. It also contains 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 chapter 4.2
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. In the example code snippet 2.3 the map and filter operation is only executed once the count() operation 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 data/ network traffic. The lazy execution has to be kept in mind during debugging and performance testing. [54, p.73]

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

Code Snippet 2.3: lazy evaluation

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 already outperforms Hadoop with its ability 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, a data-processing architecture consisting of a Batch-Layer, a Speed-Layer for real-time processing and a Serving-Layer managing the data [58, pp. 8 f]. Spark already has 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 many possibilities for

high-throughput big data processing [59, p. 5].
This thesis preliminary only 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.

2.5.3 Music Similarity with Big Data Frameworks

Given the short introduction to Big Data frameworks the decision to use Spark for the computation of the similarities seems justifiable. The computation of the one-to-many-item similarity follows the shared nothing approach of Spark. All the features are independent from each other. Only the scaling of the result requires an aggregation of maximum and minimum values and to return the top results some kind of sorting has to be performed. But apart from these operations, all the features can be distributed on a cluster and the similarity to one broadcasted song can be calculated independently, following the data locality approach. This offers a fully scalable solution for very large datasets. Additionally Spark enables efficient ways of caching the features. Under the prerequisite that the sum of all features fit into the main memory of the cluster, interactive requests could be answered without the need of reading the features from the disk every time.
One limitation is, that Spark itself is unable read and handle audio files. So the features extraction itself has to be performed separately and only the extracted features are loaded into the cluster and processed with Spark. The features extraction is described in chapter 4.1.

3. Similarity Analysis

This needs an introduction

3.1 Timbre Similarity

Mel Frequency Cepstral Coefficients have already been introduced in chapter 2.1.2. This section focuses on the different similarity metrics.
To reduce the dimensionality of the data even further, a statistical summarization of the MFCC feature can be calculated [1, pp. 51ff]

3.1.1 Euclidean Distance

For each of the Mel- Bands (12 in this case) the mean and standard deviation over all frames is calculated, resulting in a vector of 12 mean values, a 12 by 12 co-variance matrix ($\frac{12*(12-1)}{2}$ covariance values, because of the triangular shape - the upper triangle contains the co-variances and the main diagonal contains the variances) and 12 variances. These vectors are therefore not dependent on the length of the actual song. Using such a model, the distance between two songs can be calculated as in equation 3.1, where x and y are the n-dimensional feature vectors of two different musical pieces:

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

also known as the L_p distance. Most of the times, the Euclidean (L_2) or the Manhattan (L_1) distance would be used in real world scenarios. This very basic approach has been refined and improved over the past years. [1, p. 58]

3.1.2 Single Gaussian Model

Symmetric Kullback-Leibler Divergence

This approach was first proposed by Mandel and Ellis [11] in 2005 and is briefly summarized in [1, pp. 65f].
After computing the mean value of each MFCC (μ_P and μ_Q) and the covariance matrix of the different MFCC vectors (Σ_P and Σ_Q) of two musical pieces P and Q, the Kullback-Leibler divergence (KL divergence) can be calculated as follows, with $tr(\cdot)$ 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

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

As a second step the result has to be symmetrized.

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

This approach is one of the two available similarity metrics of the musly [10] toolkit introduced in section 2.2. It can be simplified and written as a closed form according to [22, p. 44]:

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

Jensen-Shannon-like Divergence

The second available metric in the musly toolkit by Schnitzer is using the Jenson-Shannon Divergence (in an 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. 44]

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

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

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

Mutual Proximity

After calculating a similarity matrix for all songs, musly normalizes the similarities with mutual proximity (MP). [12]. This method wants to reduce the effect of a phenomenon called hubness that 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 discance $D_{x,y}$ 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 $P(y)$" [1, p. 80]
Resulting in:

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

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

according to [1, p. 80]

3.1.3 Gaussian Mixture Models and block-level features

Another, more compute heavy distance measurement would make use of Gaussian Mixture Models 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." [1, p. 65] Therefore the usage of GMMs is not further considered in this thesis.
The last method mentioned in this thesis for timbral similarity is to use block-level features as proposed by Seyerlehner [60] and described in short by Knees and Schedl [1, 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 and features like fluctuation patterns are computed for these frames.

3.1.4 Validation

For this thesis the Kullback-Leibler divergence, the Jensen-Shannon divergence and the Euclidean distance are chosen and tested. There is always a trade-off between the complexity and functionality of the distance computing algorithms and a re-implementation of the block-level features remains left open for future research due to its rather compute

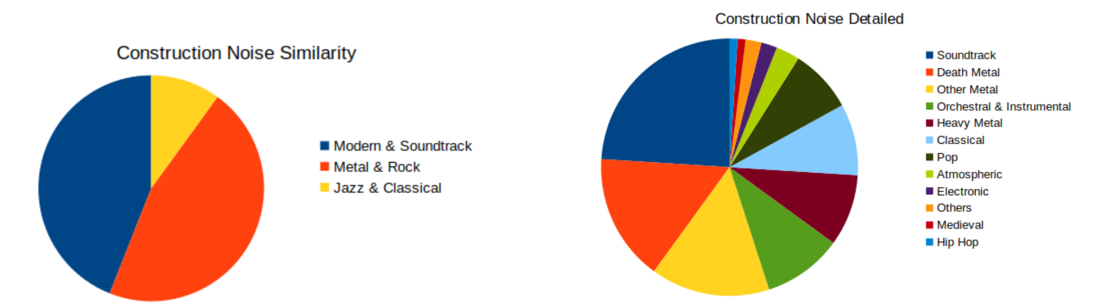
heavy nature.
Using the musly toolkit, a first evaluation is presented in the next section. The feature extraction and distance calculation can also be done in python using the librosa library and a small reimplementation of the Mandel-Ellis approach was tested in advance.
A possible measure for the efficiency of a timbre similarity algorithm is the ability to find songs of the same genre.

Construction Noise

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 could be achieved in descending order:

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

Figure 3.1a and 3.1b show the distribution of the genres of 100 most similar songs compared to the construction noise sample.



(a) Similar genres to construction noise sample

(b) Similar genres to construction noise sample (detailed)

Figure 3.1: Construction Noise

Using the full dataset consisting of the private music collection, private field recordings, the full fma-dataset and the musicnet data, the following results could be achieved:

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

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 is 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:

- Mozart - Concert No. 10 for 2 Pianos and Orchestra in E Flat Major, KV 365 - 2. Andante
- Schubert - Sonata in B Flat, D. 960 - III. Scherzo (Allegro vivace con delicatezza)
- Albeniz - Iberia, Book I - Evocación
- <u>Mozart Sonate Nr. 11 in A-Dur, K. 33 - Mozart - Alla Turca Allegretto (3:28)</u>
- Beethoven - Bagatellen Op 119 -Allemande in D major
- Mozart - Rondo No. 1 in D Major, K. 485
- Mozart - Sonata For Piano No. 8 KV 310 A Minor - Allegro Maestoso
- Sonata For Piano No. 16 KV 545 C Major - Rondo: Allegretto
- <u>Mozart Sonate Nr. 11 in A-Dur, K. 33 - III. Tuerkischer Marsch (3:30)</u>
- Mozart - Piano Sonata No. 13 in B flat major, K. 333 (K. 315c): Allegretto grazioso

The interesting conclusion is that only 2 out of the 3 other versions were considered as most similar songs. The slower recording wasn't even in the top 30 list of the most similar songs. The same was observable for other cover versions of songs in the dataset like Serj Tankians song "Lie Lie Lie" from the CD "Harakiri" and an orchestral recording of the same piece. This is probably due to the usage of GMMs of MFCCs representing and valuing the timbre of the music predominantly instead of the pitches and melody movements.

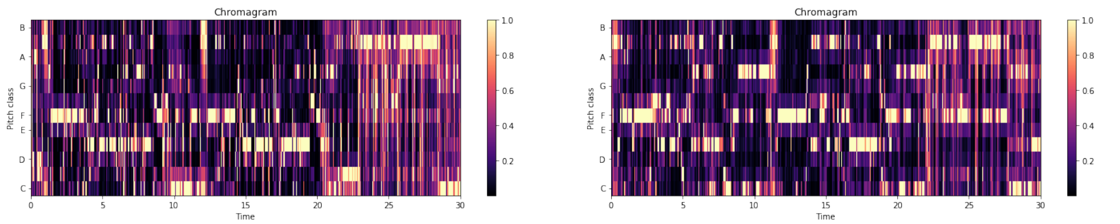
3.2 Melodic Similarity

3.2.1 Representation

As presented in section 2.2.3 there are tools for pitch curve extraction of the main melody line. However in polyphonic music these kind of algorithms struggle to get reasonable results, even in pop music. In musical genres like Metal it gets even worse. 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 doesn't produce very reasonable results as shown in 2.2.3. Another possible representation for melodic features could be to transform the structural information to graphs and use graph comparing algorithms to estimate the similarity between songs. [61] A better and widely used approach is to use chroma features as described in the next section 3.2.2.

3.2.2 Chroma Features pre-processing

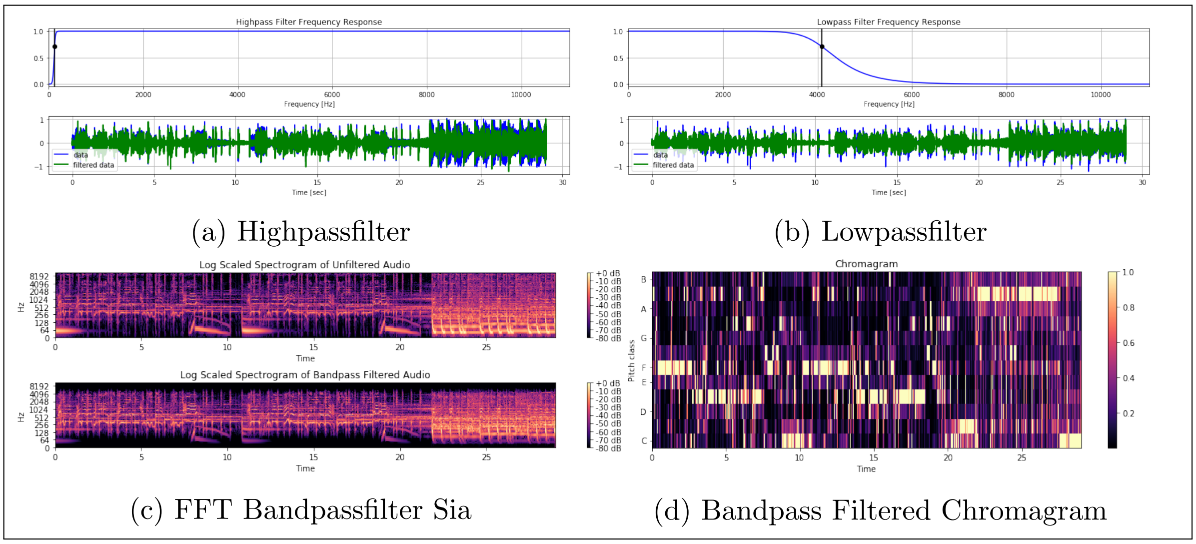
Chroma Features as described in section 2.1.1 are a good and low dimensional way to describe the melodic features of a song. The reduction of dimensionality however comes with a loss of information, especially what octaves the notes are played in. To compute chroma features, most MIR toolkits already offer methods to do so. The plots in this chapter were created using the essentia [7] and librosa [3] toolkits. In addition to the pure computation of the chroma features, some pre- and post-processing steps were implemented and tested and will be presented later in this chapter. First of all figure 3.2 shows the chroma feature plots from 2 recordings of the first thirty seconds of the song "Chandelier". Figure 3.2b shows the original version sung by the artist Sia and figure 3.2a shows the features of a cover version from the band Pvris. In the last



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

Figure 3.2: Chroma Features

third of each sample, the chroma features seemingly get noisier. At these timings in both songs the bass and drum set in. To reduce the effect of rhythm elements over the melodic voice and instrument lines, the audio signal was filtered firstly by a high-pass filter with a cut-off frequency of 128Hz (nearly equal to C3 Key) and secondly by a low-pass filter with a cut-off frequency of 4096Hz (C8 Key). This limits the frequency range to about 5 octaves. In figure 3.3 the filter frequency and the original audio signals are visualized in blue color and the filtered audio signal is green. The FFT plot before and after filtering the audio signal is also shown. In the chromagram of the bandpass



(a) Highpassfilter

(b) Lowpassfilter

(c) FFT Bandpassfilter Sia

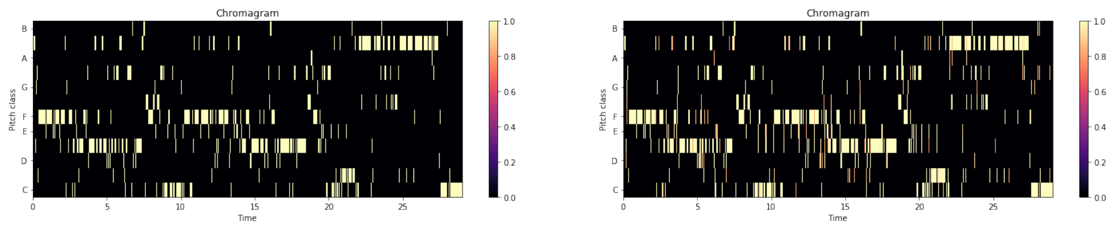
(d) Bandpass Filtered Chromagram

Figure 3.3: Bandpass - Sia

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 3.2. The next step is to calculate the most dominant note value for each timeframe. Due to the fact that the chromagram normalizes every timeframe to the maximum note value, the most dominant note always gets the value 1. The closer the rest of the notes are to zero the more likely the timeframe contains silence. If only a few values are close to one, then 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, then 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.

To extract the main melody in most cases only the most dominant pitch is needed, but sometimes the main melody is superimposed by other accompanying instruments. To prevent this the second most dominant pitch is also taken into consideration if its value is greater then a specific threshold. The result is shown in figure 3.4 with the threshold 0.8.

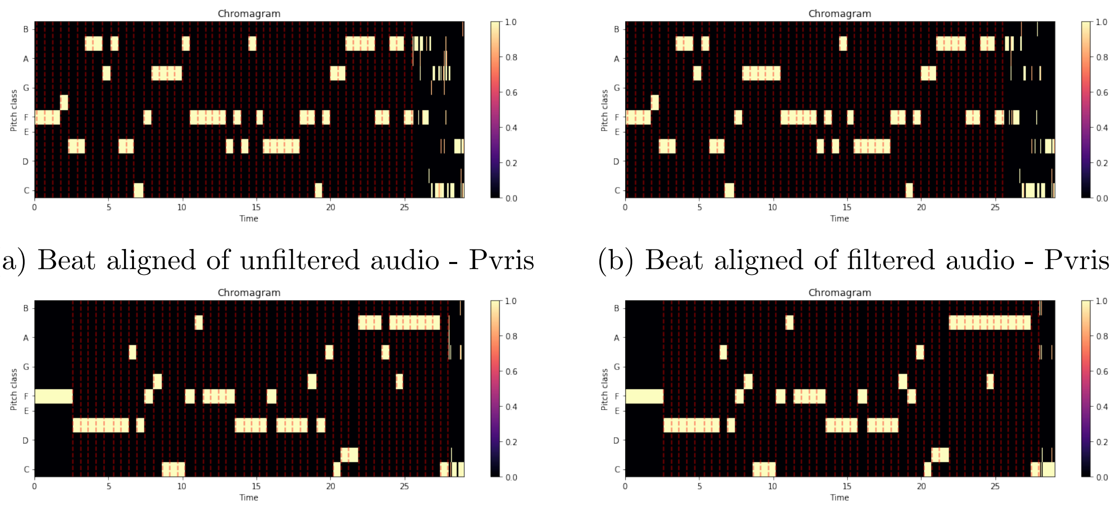
After that a beat tracking algorithm is applied to the song and the count of appearances of each note between two beats is calculated. The notes that appear the most are then set 1 while the rest is set to 0 for each section between two beats. This beat-alignment serves to make the similarity measurement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



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

Figure 3.4: Thresholded Chroma Features - Sia

the melody of each bar is still the same as in the faster version. Figure 3.5 show the different beat aligned features of both songs with bandpass filtered audio and unfiltered audio. The red lines are 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



(a) Beat aligned of unfiltered audio - Pvris (b) Beat aligned of filtered audio - Pvris

(c) Beat aligned of unfiltered audio - Sia (d) Beat aligned of filtered audio - Sia

Figure 3.5: Processed Chroma Features - Sia

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 the song is played in 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 3.6. Another consideration is to use the original chromagram without filtering out the least dominant keys and thus leaving the processing step 3 out. This means a possible tradeoff between accuracy and computation time. The result in the example song by

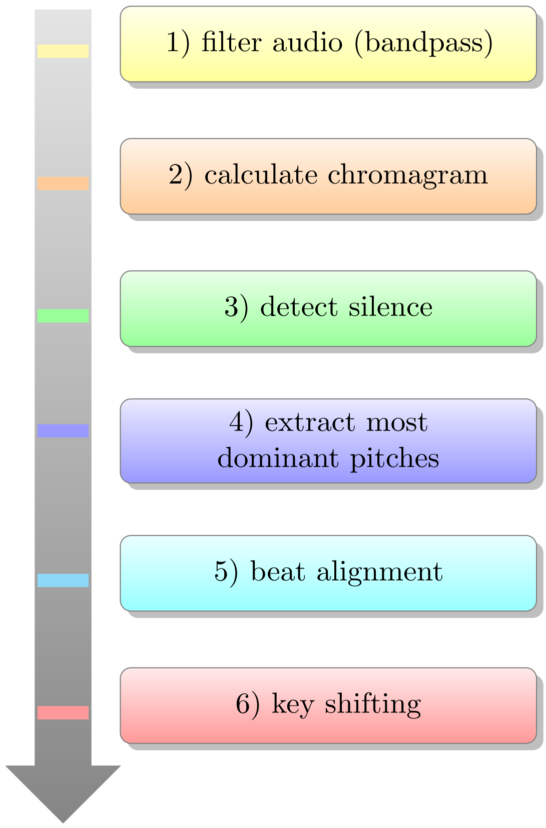
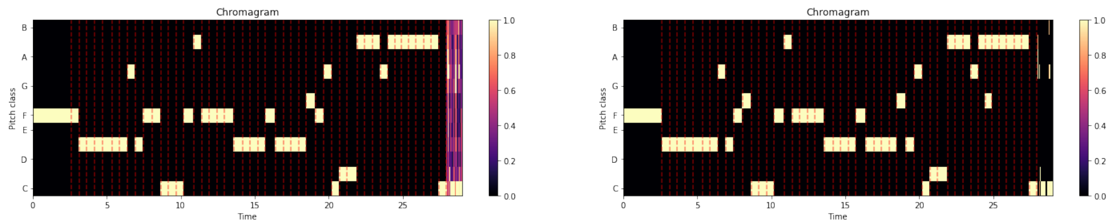


Figure 3.6: Workflow chroma feature extraction

Sia doesn't show a major impact, as can be seen in figure 3.7. In this thesis step 3 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 3.7: Processing Step 3 Chroma Features

3.2.3 Similarity of chroma features

In this section, two completely different approaches to measure the melodic similarity of two songs will be presented. The first one as proposed by [62] or [30] uses text retrieval methods to compare the chroma features of two songs and the second evaluates the usage of cross-correlation as a signal processing approach [63] and [64]

Text retrieval

One possibility to process the chromagrams and to estimate similarity between the features of different songs is by handling the features like texts. Due to the extraction of only the main melody line in our feature vector there is only one note for every detected beat. 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. [30] mentions that this is indeed a problem when using the levenshtein distance (also known as the edit-distance) to compute similarities. In their paper they use MIDI files instead of chroma features, but both contain information about the melody of songs so an adaption to chroma features is not an issue, because they can also easily be interpreted as simple strings. The levenshtein distance between the first i characters of a string S and the first j characters of T can be calculated as follows [30, p. 7]

$$lev_{S,T}(i,j) = \begin{cases} max(i,j), & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{S,T}(i-1,j)+1 \\ lev_{S,T}(i,j-1)+1 \\ lev_{S,T}(i-1,j-1) \\ +cost[S_i \neq T_j] \end{cases} & \text{else} \end{cases} \tag{3.10}$$

Xia (et al.) made some adjustments to this to be able to handle musical information.[30, 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 in 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. Therefor the implemented algorithm does not split the vectors. This tends to favor cover songs that share the same length.

Englmeier (et al.) use more advanced information retrieval techniques called TF-IDF weights 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." [62, p. 186] To do so, they have to create "audio words" 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 by viewing them as ordinary signals and creating opportunities to apply classical signal processing algorithms. Ellis and Poliner use cross-correlation in their 2007 published paper [64]. Serra (et al.) also reference the work of Ellis and Poliner and discuss 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 an other approach called dynamic time warping (DTW) further in their paper [63]. The focus in this thesis is set on the cross-correlation method. Given two discrete time signals $x[n]$ and $y[n]$ the cross-correlation between the 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 2-dimensional input matrices X with the dimensions M by N and Y as an 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 equation 3.12 [65], the bar over H denotes complex conjugation (in this case H is a matrix with real values only).

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

with

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

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

An example for the one dimensional cross-correlation is shown in figure 3.8 and the full two dimensional cross-correlation of two songs is figured in figure 3.9 and 3.10. Ellis and Poliner did not transpose the songs in the pre-processing step to 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 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 [66]. Additionally they filtered the result of the correlation with a high-pass filter. ”We found that genuine matches were indicated not only by cross-correlations of large magnitudes, but that

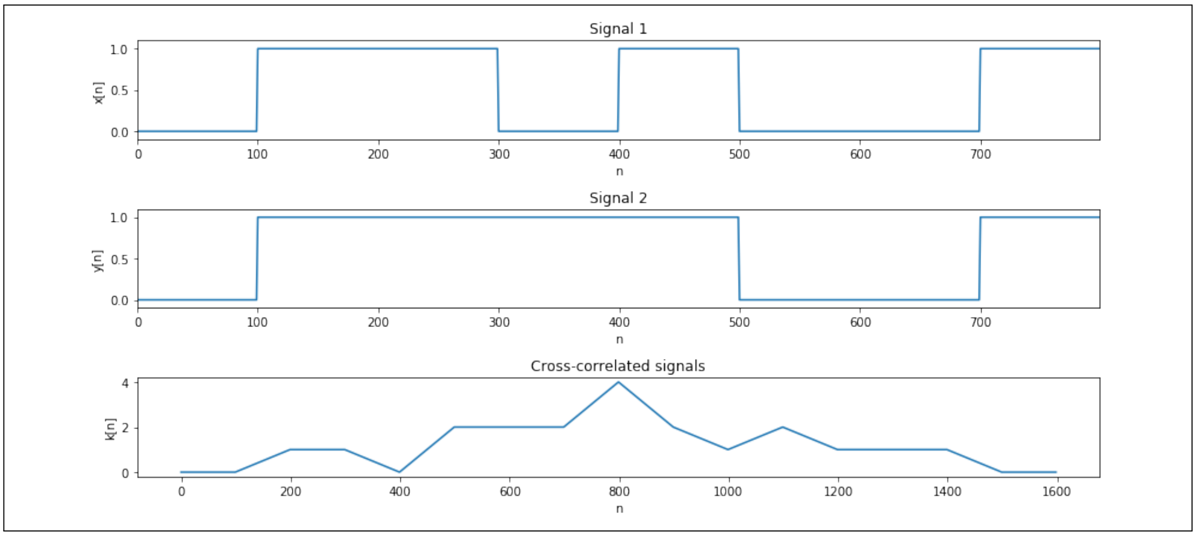


Figure 3.8: cross-correlation

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" [64, p. 1431]. The later published paper [66] also states, that changes to the filter parameters improved the cover song recognition rate further, however the exact values e.g. for the cutoff frequency weren't give, so this thesis uses the older parameters for the filter

Serra (et al.) discussed various effects of pre-processing steps that improve the algorithm even more. E.g they note that a higher chroma resolution of 3 octaves gives better results. Also a key detection and transposition before cross-correlation gives slightly worse results in comparison to the method Ellis and Poliner used.

In this thesis a versions where the songs are all key aligned before the cross-correlation was tested, but due to the fact, that the key detection algorithm in the labrosa and essentia frameworks weren't always correct a second version where addionally the cross-correlation for all key transpositions is calculated, was also implemented. In summary, the implementation in this thesis is similar to the approach by Ellis and Poliner [64] but some of the steps from the newer paper [66] leave some room 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 including key shifts by letting k run from $-(P-1) \leq k \leq M-1$ in equation 3.12 is shown in the figures 3.9 and 3.10 but due to the previous pre-processing key shift and the fact that both input matrices share the same amount of rows (12, one per key) these aren't ultimately necessary and computation time can be safed by altering the equation to equation 3.15 resulting in a vector C with the correlation results without additional key-shifting but this version is reliant on an accurate key detection of the songs.

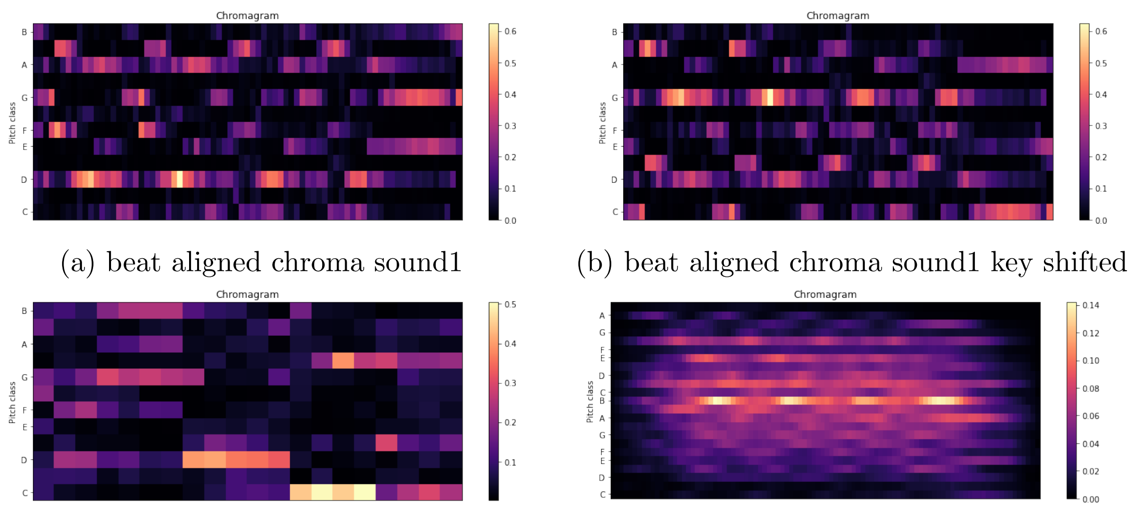
$$C(l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m,n)\overline{H}(m,n-l) \tag{3.15}$$

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

or even faster without calculating the edges of the matrix.

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

The post-processing step from Ellis and Poliner, namely the high-pass filtering of the result was also implemented.
Figure 3.9 shows two beat aligned, key shifted and per beat averaged chroma features of two short guitar snippets and their cross-correlation. The interesting row of the cross-correlation matrix is the middle row marked with the C key. It shows that both already key shifted melodies do not correlate well. In figure 3.10 the cross-correlation of



(a) beat aligned chroma sound1

(b) beat aligned chroma sound1 key shifted

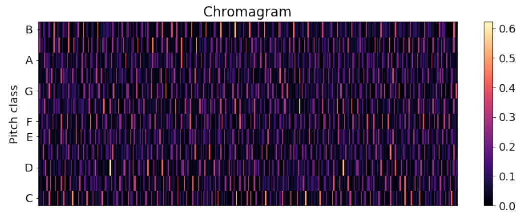
(c) beat aligned chroma sound2 key shifted

(d) cross-correlation

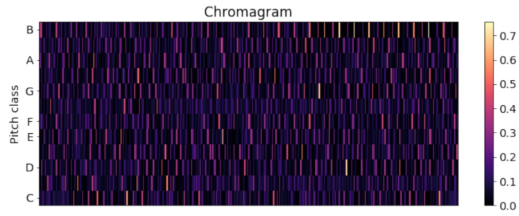
Figure 3.9: beat-aligned chromagram

the song "Chandelier" by the singer Sia and covered by Pvris are shown in 3.10c and in contrast to this the cross-correlation of "Rock you like a Hurricane" with the song "Chandelier" by Sia are shown. Due to the previous key shifting, plot 3.10a 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 in the pre-processing step.
The unrelated songs result in much smaller correlation values, especially when looking

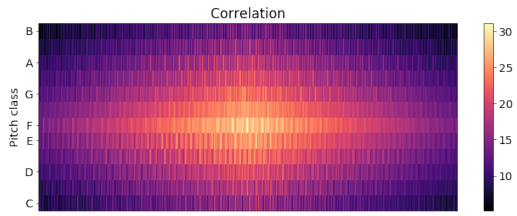
at the middling row of the matrix (marked as the B-key), but also if the songs are transposed additionally even then they do not correlate well. In contrast to this the cover songs have multiple visible peaks in the center row. The row with the maximum



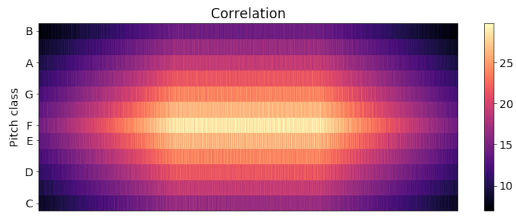
(a) Pvris



(b) Sia



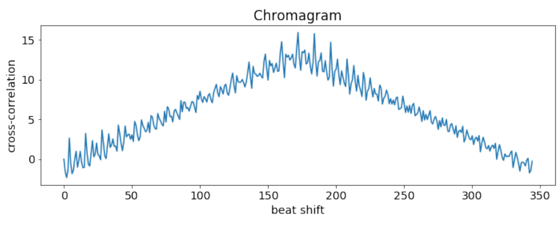
(c) cross-correlation of cover songs



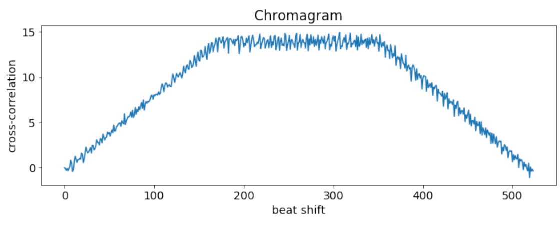
(d) cross-correlation of unrelated songs

Figure 3.10: Cross-correlation

correlation value is extracted and the resulting plot shows, that the cover songs do correlate much better than the unrelated songs (3.11a and 3.11b). The center rows of the cross-correlation matrices from figure 3.10 are separately pictured in figure 3.11 and 3.12. After applying the high-pass filter to the extracted row with the maximum correlation value, the peaks in 3.11a when cross-correlating the cover songs is 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 3.12a with clearly visible recurring peaks when the refrain is repeated.

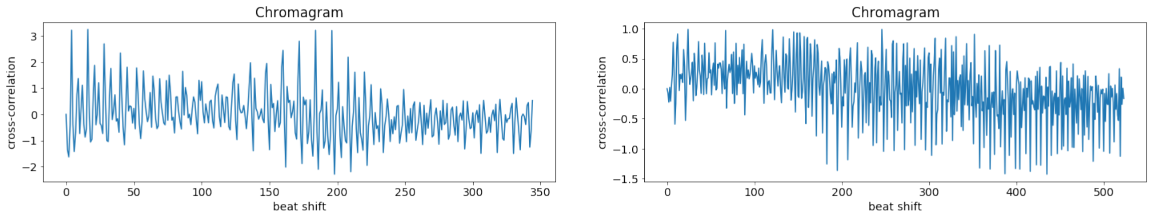


(a) cross-correlation of cover songs



(b) cross-correlation of unrelated songs

Figure 3.11: Cross-correlation



(a) cover songs filtered

(b) unrelated songs filtered

Figure 3.12: Cross-correlation filtered

3.2.4 Validation

A good measure for the efficiency of a melodic similarity algorithm is the ability to find cover songs, remixes and recordings of the same song from different artists.

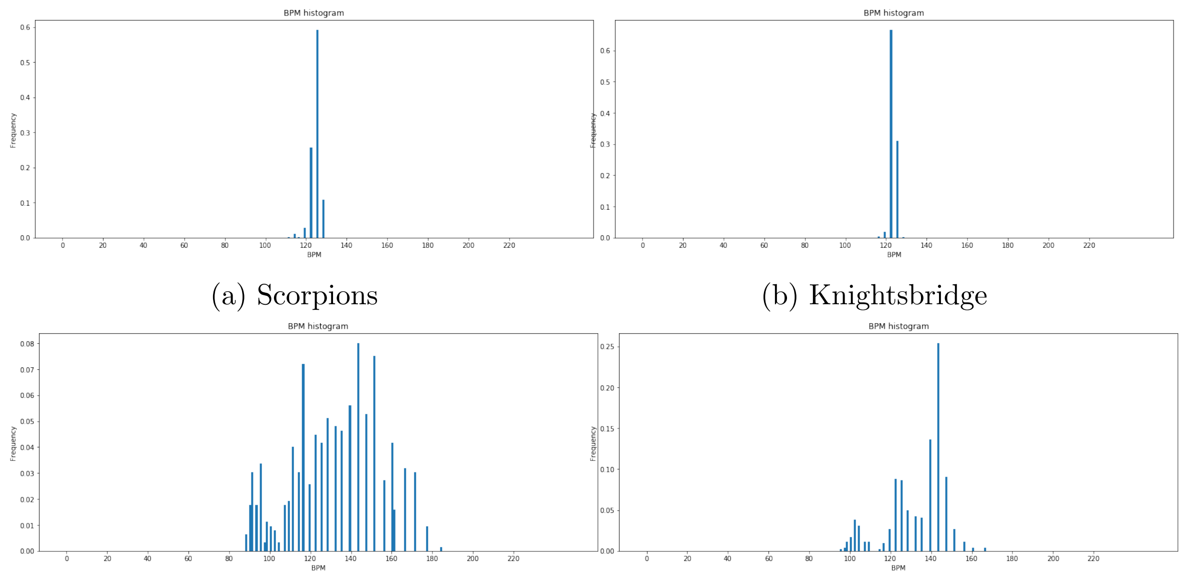
3.3 Rhythmic Similarity

This chapter provides an overview over 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 to computing very low level rhythmic similarities is by sorting and comparing songs by their tempo. Of course there are far better and more accurate solutions. By just comparing the tempo of songs a lot of rhythm information is lost e.g. the rhythmic structures of songs like the time signature, up- and downbeats, etc.
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

Other similarity measurements are e.g. the usage of beat histograms as proposed by Tsanetakis and Cook [67]. These are relatively similar to the later evaluated Rhythm Histograms. Gruhne (et al.) further improved the beat histograms and suggested an additional post processing step for the beat histogram before calculating the similarity between songs with the euclidean distance, to improve a comparison of two songs with different tempi by transforming the beat histograms into the logarithmic lag domain. 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 [68, p. 182]. The essentia toolkit offers methods to extract the

beat histogram. The different detected potential 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 the song "Rock you like a hurricane" by the Scorpions and covered by Knightsbridge 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 and the second version was recorded with Anders Friden as a singer in 1999. The 1994 version changes its tempo in the outro of the song as can be seen in the figure.



(a) Scorpions (b) Knightsbridge

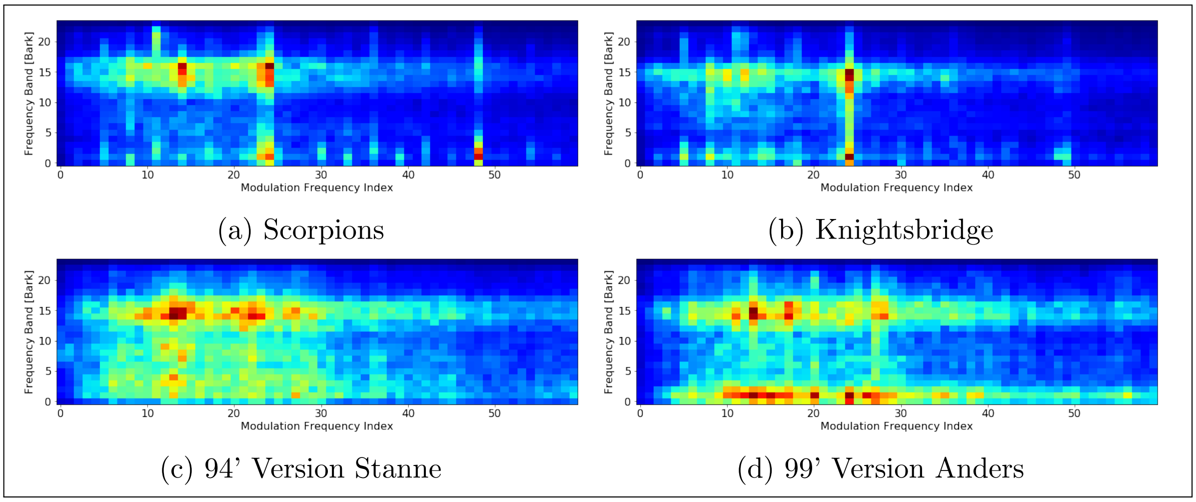
(c) 94' Version Stanne (d) 99' Version Anders

Figure 3.13: Beat Histogram

Another feature that will just be mentioned here (one of the older ones from 2002) uses the beat spectrum as a feature [24].

3.3.2 Rhythm patterns

A more state-of-the-art feature is the so called rhythm pattern, also known as fluctuation patterns for instance mentioned by [69]. To extract these features the rp_extractor library [70] was made publicly available by the TU Vienna [71]. 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.
The x-axis represents the frequency band converted to the bark scale (a scale representing the human auditory system comparable to the mel scale) and the y-axis represents the modulation frequency index representing the modulation frequencies up to 10Hz (600



(a) Scorpions (b) Knightsbridge

(c) 94' Version Stanne (d) 99' Version Anders

Figure 3.14: Rhythmic Patterns

BPM). The Bark of a frequency f can be determined using formula 3.18.

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

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

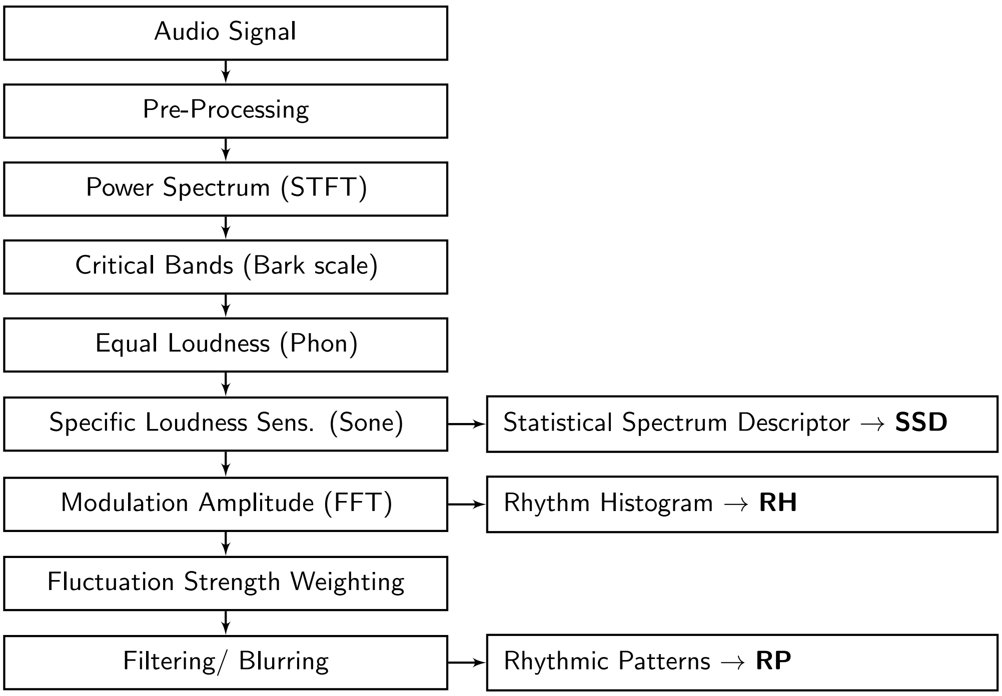


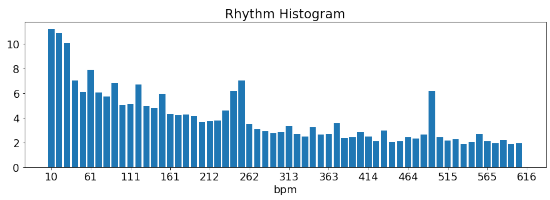
Figure 3.15: Rhythm Pattern extraction [71]

So in conclusion the Rhythm Patterns basically represent the BPM of various frequency bands. To compare two different songs the euclidean distance between the vectorized

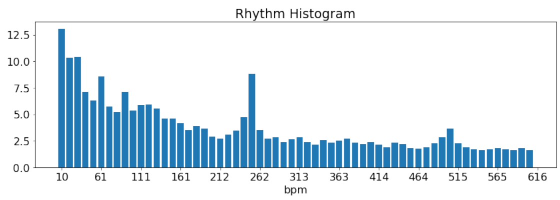
rhythm pattern matrices can be calculated as Pampalk suggests [72, p. 40]
Pohle, Schnitzer et al. refined Fluctuation Patterns into Onset Patterns e.g. by using semitone bands instead of fewer critical bands to detect onsets. [73]
This thesis however focuses on Fluctuation/ Rhythm patterns extracted with the rp_extractor library.

3.3.3 Rhythm Histogram

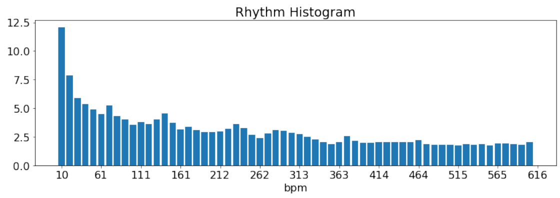
A more simplistic and lower dimensional feature coming with the rp_extract toolkit is the Rhythm histogram. "The Rhythm Histogram features we use are a descriptor for general rhythmics in an audio document. Contrary to the Rhythm Patterns and the Statistical Spectrum Descriptor, information is not stored 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." [69, p. 3].
The difference in comparison to the beat histogram mentioned earlier in section 3.3.1 appears to be, that the beat histogram focuses on the basic tempo of the whole song while the rhythm histogram takes all frequency bands and therefor the sub-rhythms of single instruments into account.



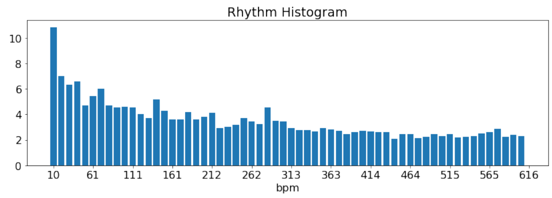
(a) Scorpions



(b) Knightsbridge



(c) 94' Version Stanne

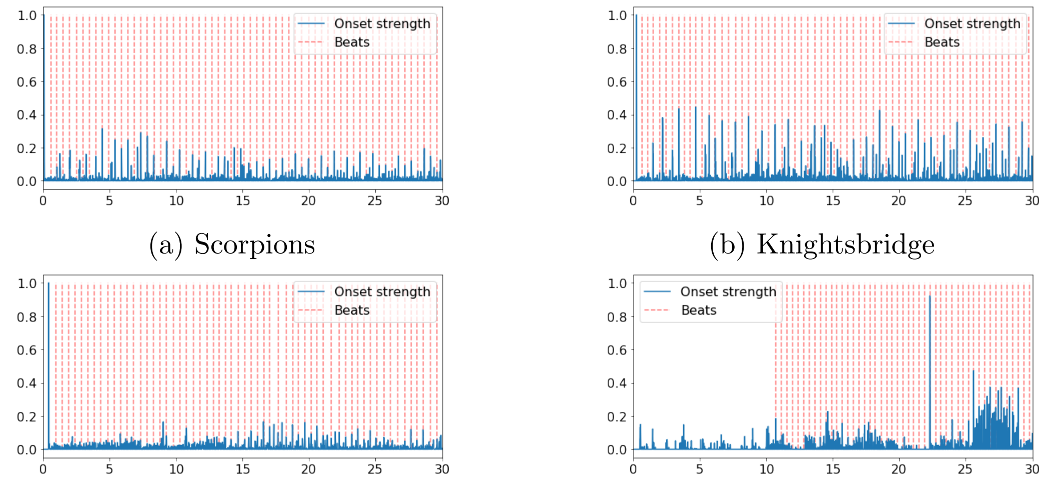


(d) 99' Version Stanne

Figure 3.16: Rhythm Histogram

3.3.4 cross-correlation

Estimating the onset strength per beat and creating a discrete-time signal for each song is an other option. Similar to the chroma features the cross-correlation of the onset functions could be used as a similarity measurement.



(a) Scorpions (b) Knightsbridge

(c) 94’ Version Stanne (d) 99’ Version Stanne

Figure 3.17: Detected Onsets (first 30 seconds)

Looking at the extracted onset features of the Song ”Behing 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 are greatly dependent on the underlying beat extraction and onset detection algorithms. E.g. the librosa toolkit struggles to detect beats in the first 10 seconds of the version of the song from the year 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.

3.4 Summary

3.4.1 Timbre Similarity

The chosen similarity metrics are :

- Euclidean Distance
- Symmetric Kullback-Leibler Divergence
- Jensen-Shannon like Divergence

To calculate the distances, for each song the mean vector, variance and covariance matrix has to be computed for each mfcc band. These are stored in two different output text files:

- out.mfcc (containing mean vector (length b), variance vector (length 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 (length $b \cdot b$)

The amount of mfcc bands chosen is $b = 13$.
The second *.mfcckl file is created to get rid of the necessity to rearrange the covariance matrix inside the Big Data Framework and reduce the computation time when a similarity computation request is processed.
To even further safe storage space the variance vector from the *.mfcc files could have been left out because these values are also stored within the main diagonal of the covariance matrix and left within the triangular matrix, leading to $\frac{b \cdot (b+1)}{2}$ instead of $\frac{b \cdot (b-1)}{2}$ values (as mentioned in section 3.1.1) in the triangular matrix.

3.4.2 Melodic Similarity

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

These 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 (length n))
- out.chroma (full beat aligned chromagram, containing a $12 \times n$ matrix)

3.4.3 Rhythm Similarity

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

These 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 (length 250))
- out.rh (containing a vector for the rhythm histogram extracted with rp_extract (length 60))
- out.rp (containing a vectorized matrix for the rhythm patterns extracted with rp_extract (length 24×60))

3.4.4 Feature files

The feature files contain strings like the following:
out.mfcc: *music/song.mp3; [-498.03763, ... ,4.321189]; [8943.487,... ,61.624344]; [8944.3907652, ... ,74.17548092]*
out.mfcc: *music/song.mp3; [-498.03763, ... ,4.321189]; [[6568.27958735, ... ,74.64776425], ... , [74.64776425, ... ,69.1589048]]*
out.notes: *music/song.mp3; G; minor; [6, 2, 5, 7, 7, 7, 7, 2, 2, 2, 2, 2, 0, 0, 0, 0, 3, 0, 0, 0]*
out.chroma: *music/song.mp3; [[0.5209161 ,0.82440507,... ,0.68443549] ... [0.31470749,0.02552716,... ,0.01234249]]*
out.bh: *music/song.mp3; 86.9380264282; [0. ... 0.01453488 ... 0. 0.]*
out.rh: *music/song.mp3,15.2521291416,10.10441871, ... ,2.2519330706*
out.rp: *music/song.mp3,0.0237481782333,0.0208784207788, ... ,0.00204177442894*

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

4. Implementation

The implementation consists of two separate parts. The first one contains the feature extraction and preparation of the data from the audio files. The result are stored in feature files. These features files then have to be processes with the Big Data framework Spark to compute the similarities between songs.
Both parts are implemented in Python and are able to be executed on computer clusters. The source code can be found in the appendices and can be pulled from github [74]. Details for the usage of the python scripts are also documented there.

4.1 Audio Feature Extraction

So far the required audio features have been selected in chapter 2 as well as toolkits to extract those features from the audio data. In chapter 2.4 different sources for audio files have been presented. Chapter 3.1, 3.2 and 3.3 presented algorithms to pre-process the low-level features and use these to compute similarities. This chapter focuses on the selection of datasets to extract features from and the performance of the feature extraction and pre-processing software implementation.

4.1.1 Test Datasets

Chapter 2.4.1 introduced a range of MIR datasets but not all are fitting to the problems this thesis evaluates. To test the algorithms on the one hand a lot of data is needed, so the Free Music Archive with its over 100000 songs is a solid option for performance tests. However on the other hand 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 really representative for actual popular music, a lot of the songs are live recordings with poor audio quality, possibly influencing the results. The 1517 artists dataset offers 19 different genres with songs relatively equally 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 offers a subjective evaluation of the results of the similarity analysis. In conclusion that sums up to about 117000 songs for performance tests and about 12000 songs for a detailed evaluation of the algorithms in this thesis. As mentioned in 2.4.1 all albums from the private music collections are cataloged as well and the associated document is in the appendices. For the first tests an even smaller sample dataset containing 10 songs out of 10 different genres was created from the private music collection and the list with the belonging songs is also in the appendices.

fma	106.733 Songs
private	8484 Songs
1517 artists	3180 Songs
covers80	164 Songs (80 originals + 84 covers)

Table 4.1: appropriate music datasets

4.1.2 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 the feature post-processing. The post-processing of the features like the note estimation from the chroma features and the calculation of statistic features from the MFCCs was already explained in-depth in the previous chapters and is therefor left out here. The full code is in the appendices.

Librosa

For most of the plots in the introduction section 2 the python toolkit librosa was used because of its ease of use and very good documentation. The code example shows the necessary methods to extract the most important features like mfcc, chromagram and beats/ onsets.

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

Code Snippet 4.1: librosa

But when extracting features from batches of audio data the librosa library turned out to be very slow. For a very small 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 48 minutes. For larger datasets like the 1517 artists dataset the feature extraction process would have taken about 22 hours.

Essentia

[5] compares different Audio feature extraction toolboxes and shows 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 in 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 offers similar methods to librosa and uses an imperative programming style. The audio file has to be read, sliced and preprocessed by hand. 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" whenever a process is called. The melodic and timbral features and the beat histograms are all computed with essentia. Only the rhythm patterns and rhythm histograms are computed in a separate step as stated below.

Essentia Standard

In the final extractor code the mfcc calculation and beat histogram estimation is done with the essenia standard library, because it offers a fast and easy way to implement the basic feature extraction tasks.

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

Code Snippet 4.2: essentia standard

Essentia Streaming

The essentia streaming library is used to calculate the chroma features in the final code. It eases up the filtering with the high- and a lowpass filter. The audio signal is passed through various stages of processing and ultimately resulting in the chroma features of the high-pass filtered audio signal.

```
loader = ess.MonoLoader(filename=path, sampleRate=44100)
HP = ess.HighPass(cutoffFrequency=128)
LP = ess.LowPass(cutoffFrequency=4096)
framecutter = ess.FrameCutter(frameSize=frameSize, hopSize=hopSize,
  silentFrames='noise')
windowing = ess.Windowing(type='blackmanharris62')
spectrum = ess.Spectrum()
spectralpeaks = ess.SpectralPeaks(orderBy='magnitude', magnitudeThreshold=0.00001,
  minFrequency=20, maxFrequency=3500, maxPeaks=60)
hpcp = ess.HPCP()
hpcp_key = ess.HPCP(size=36, referenceFrequency=440, bandPreset=False,
minFrequency=20,
  maxFrequency=3500, weightType='cosine', nonLinear=False, windowSize=1.)
key = ess.Key(profileType='edma', numHarmonics=4, pcpSize=36, slope=0.6,
  usePolyphony=True, useThreeChords=True)
pool = essentia.Pool()
loader.audio >> HP.signal
HP.signal >> LP.signal
LP.signal >> framecutter.signal
framecutter.frame >> windowing.frame >> spectrum.frame
spectrum.spectrum >> spectralpeaks.spectrum
spectralpeaks.magnitudes >> hpcp.magnitudes
spectralpeaks.frequencies >> hpcp.frequencies
spectralpeaks.magnitudes >> hpcp_key.magnitudes
spectralpeaks.frequencies >> hpcp_key.frequencies
hpcp_key.hpcp >> key.pcp
hpcp.hpcp >> (pool, 'tonal.hpcp')
essentia.run(loader)
chroma = pool['tonal.hpcp'].T
```

Code Snippet 4.3: essentia streaming

Essentia performance

The calculation with the essentia library for 100 songs took less than half 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 library as presented in the next code snippet.

parallel python

Multiple CPU cores get a part of the filelist of all songs and can compute the features fully parallel.

```
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 4.4: parallel python

The computation time takes about 15.4 seconds per song and processor core. Using 4 CPU cores for 100 songs, the overall processing time could be reduced to about 385 seconds.

$$time = \frac{\#songs}{\#CPUs} \cdot 15.4s \tag{4.1}$$

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. For example for the 1517 artists dataset a batch size of 400 songs was chosen, so overall 4 CPUs had to process 2 batches, resulting in 8 different output files with the chroma feature files being the largest with about 25MB per file.

One problem that appeared by using parallel python was that the memory usage was increasing over time. The explicit usage of the garbage collector and the deletion of unwanted objects also couldn't solve that problem. So after calculating a few hundred features the process ran out of memory and had to be restarted. By replacing parallel python with mpi4py this problem could be solved later.

rp_extractor

For the extraction of the rhythm patterns and rhythm histogram features as described in chapter 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 takes about 442 seconds.

performance on a single pc

The extraction of the rhythm patterns and the rhythm histogram is performed by the rp_extractor tool. The feature extraction and processing of all the other features (beat histogram, mfcc statistics, notes and beat-aligned chromagram) had to be implemented separately and different MIR toolkits were tested.

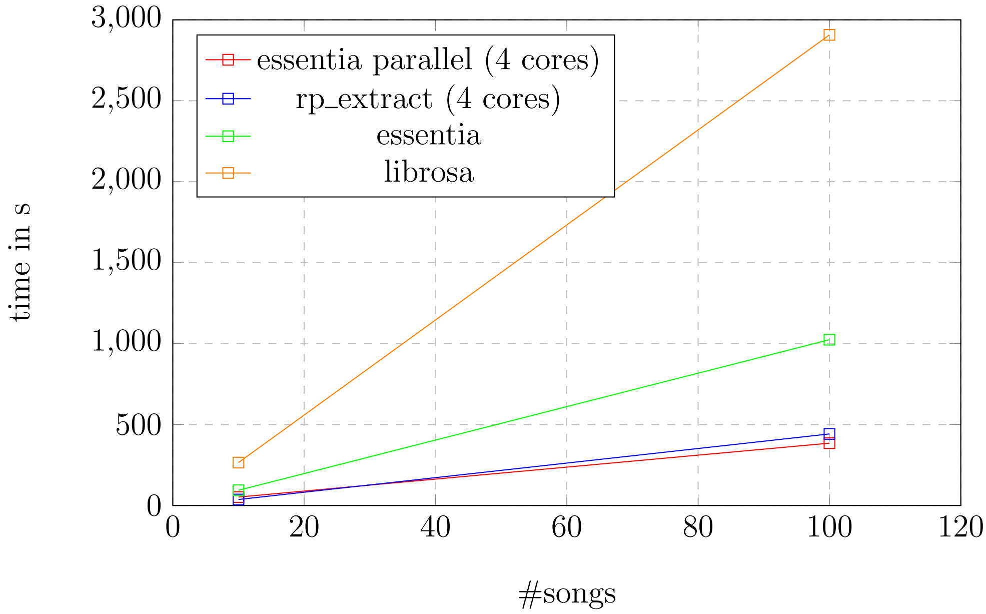


Figure 4.1: Performance of various toolkits on a single computer

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

Estimated feature extraction times

- 3h24 - 1517 artists - essentia parallel, single node, 4 CPU cores
- 3h54 - 1517 artists - rp_extract
- 9h06 - private dataset - essentia parallel, single node, 4 CPU cores
- 10h24 - private dataset - rp_extract
- (125h - full dataset - essentia parallel, single node, 4 CPU cores)
- (143h - full dataset - rp_extract)

performance on a cluster with mpi4py

For the extraction of the features from 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. Mpi4py provides Python bindings for the Message Passing Interface standard (MPI) [75]. Every compute process gets a rank number and is aware of the overall count of all processes. With these two values the file list of all audio files is split and each process only processes the according files. The audio files were stored in a parallel cluster file system called beegfs [76]. Equally to the implementation using parallel python every process stores the results in separate output files, each of them containing batches of 25 songs.
All audio files larger than 25MB were filtered out of the fma dataset 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, only 18 CPU cores per node were used. Overall 648 processses were spawned. During the computation of the audio features with essentia one out of the 648 processes ran out of memory, so only 102793 out of the 102813 songs were processed. For performance tests this doesn't make a large difference but for future work, the feature extraction script should be adapted accordingly. The extraction of the features took 1439s (fastest process) to 1950s (slowest). With a better balancing and messaging between the processes, the task could be distributed in a way where idle tasks take parts of the file list from tasks that are still processing.

```python
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 4.5: mpi4py

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 is spawned on the cluster (648), but each of the processes is able to make use of 2 CPU cores plus HT. The fastest process finished after 1657s and the slowest one took 1803s.

Total amount of songs

Due to the above mentioned filtering of audio files larger than 25MB and due to the fact that the rhythm pattern extraction script is not able to handle some audio file formats like Ogg Vorbis, not all features from all songs could be extracted. So in the end the overall amount of songs where all features could be extracted is 114210.

4.2 Big Data Framework Spark

After all features are extracted, the next step is to load the feature files into the HDFS. All feature files of the same type (forming the feature sets) get merged into one large file. For the about 114000 songs all feature files sum up to about 11.2 GB (see figure 4.2).

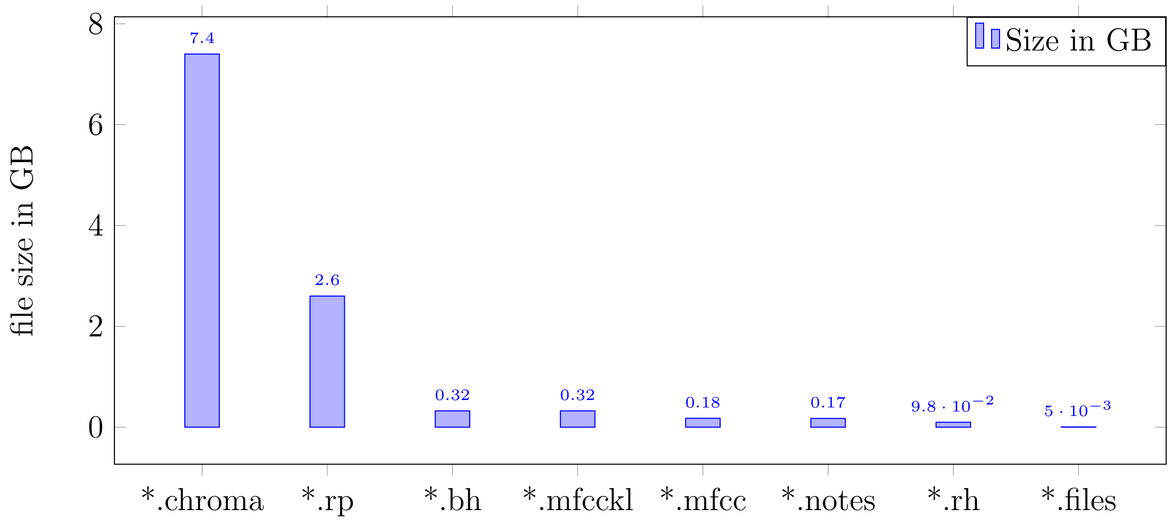


Figure 4.2: File sizes

Large streaming platforms like Spotify give access to about 30 million songs in their databases. At this scale, the feature files would approximately sum up to about 3 TB.

4.2.1 underlying hardware

The first tests with Spark 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 test were performed on the ARA-cluster, that offers 16 compute-nodes with 32 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 was running an older version of Spark (1.6.0)

4.2.2 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 can be seen in figure 4.3.

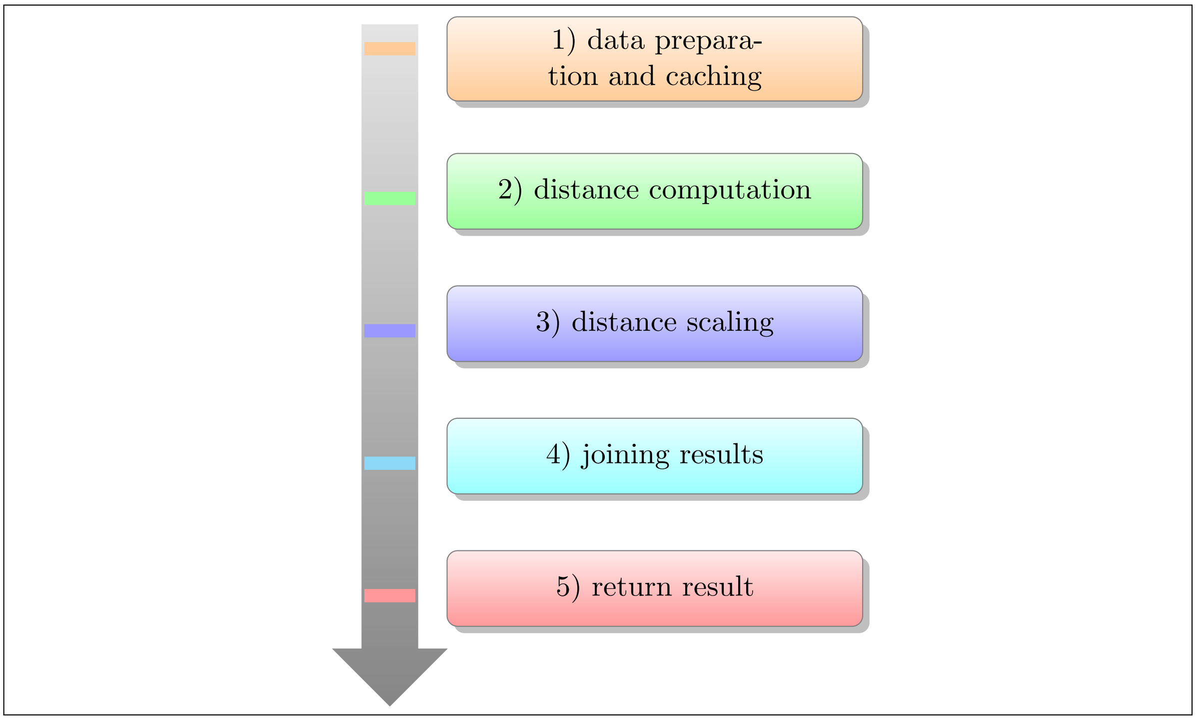


Figure 4.3: Workflow Spark

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.

4.2.3 Data preparation

The features are stored in text files as described in chapter 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 contain the features of a small number of songs. Because many small files are inefficient to process with Spark [54, p. 153] all files containing the same feature type are merged to one large file, before being loaded into the HDFS. By loading larger files into the HDFS, the 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. For example the extracted note values are stored as lists of integer numbers, each representing a certain note. To compare these using the Levenshtein distance, these lists are converted into strings.

```
chroma = sc.textFile("features/out[0-9]*.notes").repartition(repartition_count)
chroma = chroma.map(lambda x: x.split(’;’))
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(’
 ’,’’)))
df = spark.createDataFrame(chroma, ["id", "key", "scale", "notes"])
```

Code Snippet 4.6: notes preprocessing

All the other features are stored as lists of floats and have to be converted to vectors. The Spark ML library and the older MLlib library offer sparse and dense vectors as a data type. The only feature type that contains a lot of zeros and where sparse vectors could improve performance is the beat histogram. But compared to other features like the chromagram they 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(l)`. An example is given for the rhythm pattern features in code snippet 4.7.

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

Code Snippet 4.7: rp preprocessing

The data is read out of the HDFS into an RDD and repartitioned with `sc.textFile("name.txt")`. The repartitioning is optional but improves the overall performance (see 4.2.7). After the pre-processing steps are performed the RDD can be converted into a

Spark SQL DataFrame by calling `spark.createDataFrame()` to ease up the access to the data and improve the code readability. The features can then be accesses via column names instead of the RDD indices, making the code better readable and understandable. For the performance tests three different kind 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 4.2.7

4.2.4 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 using RDDs instead of DataFrames. The full source code is attended in the appendices on the included CD and can be checked out from github [74]. Most of the following code examples were written for the usage with DataFrames. The examples for the usage with RDDs are annoted accordingly.

Euclidean Distance

The euclidean distance is used as a metric to compute the distances between the 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) is declared (see 4.8). 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. Alternatively the numpy library could be used as well (`numpy.linalg.norm(x-comparator_value)`).

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

Code Snippet 4.8: euclidean distance DF

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 songs ID (e.g. the path name of the original song).

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

Code Snippet 4.9: 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 4.10).

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

Code Snippet 4.10: 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. The Spark API documentation describes the idea behind LSH as stated: "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." [77] 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. "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)." [77]

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

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." [77] This means that the smaller (and therefor 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 features sets because the resulting distances calculated from the different kind of features have to be joined to get the resulting similarities as a combination of different distance measurements

(see section 4.2.6). 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 (BRP) was introduced with PySpark version 2.2.0, the algorithm could only be tested on the single node test platform where it performed worse than the naive euclidean implementation from code snippet 4.8 on a dataset consisting of about 11500 songs. Whether the BRP outperforms the naive approach on a cluster with bigger datasets could be investigated further.

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

Code Snippet 4.11: bucketed random projection

Cross-correlation

As laid out in chapter 3.2 there are different options to calculate the cross-correlation of the beat-aligned chroma features. The chroma features are allready key-shifted to a common key but the possibility to perform a full 2D-cross-correlation with additional key-shifting as explained in equation 3.12, 3.13 and 3.14 still exists. But due to the fact that the computation of the cross-correlation already takes the most time even without additional key-shifting (see section 4.2.7) the implementation on the cluster and in code snippet 4.13 calculates the simplified cross-correlation (equations 3.15 and 3.16). 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[66]. The results are presented in chapter 5.3. There are some differences in the results 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 [66] as mentioned in 3.2.2.
Concerning the actual implementation of the cross-correlation, two different libraries were tested. Code snippet 4.12 shows the cross-correlation function coming with the scipy library.

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

Code Snippet 4.12: cross-correlation scipy

The parameter `mode='valid'` 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 4.13. 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, but 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 around 725 seconds with scipy on the ARA-cluster.

```
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 4.13: cross-correlation numpy

Jensen-Shannon-like Divergence

While computing the Jensen-Shannon-like Divergence for 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 tested for the encountered issue mentioned above but ultimately did not work out because of the covariance matrices not being positive definite.
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 code snippet is given in 4.14.

```python
import numpy as np
def jensen_shannon(vec1, vec2):
  #preprocessing: splitting vec1 and vec2 into mean1, mean2, cov1 and cov2
  mean_m = 0.5 * (mean1 + mean2)
  cov_m = 0.5 * (cov1 + mean1 * np.transpose(mean1)) + 0.5 *
    (cov2 + mean2 * np.transpose(mean2)) - (mean_m * np.transpose(mean_m))
  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))
  if np.isnan(div):
    div = np.inf
  return div
distance_udf = F.udf(lambda x: float(jensen_shannon(x, comparator_value)),
  DoubleType())
result = df_vec.withColumn('distances', distance_udf(F.col('features')))
result = result.filter(result.distances_js != np.inf)
result = result.select("id", "distances").orderBy('distances', ascending=True)
result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.14: Jensen-Shannon-like Divergence

Symmetric Kullback-Leibler Divergence

When implementing the Symmetric Kullback-Leibler Divergence a few interesting observations could be made. First of all this metric seems to be prone to outliers. While only very few distances get disproportionately large, most of the distances are between

0 and 100. The large outliers lead to problems when scaling the resulting distances to an interval between 0 and 1 (see section 4.2.5 and 5.2). 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 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 4.15.

```
import numpy as np
def symmetric_kullback_leibler(vec1, vec2):
  #preprocessing: splitting vec1 and vec2 into mean1, mean2, cov1 and cov2
  if (is_invertible(cov1) and is_invertible(cov2)):
    d = 13
    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)
  else:
    div = np.inf
    #print("ERROR: NON INVERTIBLE SINGULAR COVARIANCE MATRIX\n")
  return div
distance_udf = F.udf(lambda x: float(symmetric_kullback_leibler(x,
comparator_value)), DoubleType())
result = df_vec.withColumn('distances', distance_udf(F.col('features')))
#thresholding for outliers
result = result.filter(result.distances <= 100)
#result = result.filter(result.distances != np.inf)
result = result.select("id", "distances").orderBy('distances', ascending=True)
result = result.rdd.flatMap(list).collect()
```

Code Snippet 4.15: Kullback-Leibler Divergence

After implementing this similarity measurement into Spark some tests and comparisons to the results of the musly toolkit [10] were done. While overall the genre recall is quite good (see chapter 5.4) and the results are reasonable they do differ from the ones returned from musly. These differences in the results to the original musly tool could be explained with to the choice of only 13 MFCC bands in this thesis compared to the 25 bands in musly [10] and some other decisions like leaving the normalization with mutual proximity (3.1.2) out. The same goes for the Jensen-Shannon-like Divergence.

Levenshtein distance

Spark already offers a function for the computation of the Levenshtein distance when the feature vectors are stored in a DataFrame. The Levenshtein distance can then be computed between two columns for all rows. Code snippet 4.16 shows a minimal example.

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

Code Snippet 4.16: Levenshtein DataFrame

An alternative for the RDD based variant of the Spark application the python wrapper for the C/C++ library edlib was used. During initial tests when experimenting with an 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 4.17 performes comparably well as the Spark-native DataFrame equivalent and is a good alternative.

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

Code Snippet 4.17: Levenshtein RDD

Lazy evaluation and caching data

As described in chapter 2.5.2 Sparks 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 RAM by either calling `persist()` (optionally with the parameter `storageLevel=StorageLevel.MEMORY_ONLY_SER`) or `cache()` and even then Spark only takes this function only as a suggestion. If not enough main memory is available, the data is still written onto the hard drives. As introduced in chapter 2.5.2 Spark 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 have to be kept in mind when optimizing Spark applications and evaluating the performance by measuring execution times. The code snippet 4.18 gives a short example.

```python
1 import time
2 #...
3 featureDf = preprocess_features().persist() #p3
4 print(featureDf.first()) #p4
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 4.18: Spark lazy evaluation

`preprocess_features()` is a function where the chroma features get read into RDDs, pre-processed, repartitioned and converted to a DataFrame.`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 the maximum distance. 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. 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 4.4. The first bar shows the `print time_dict` output when executing the full code from code sample 4.18. 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 but the results are no longer persistent in the cache, this part of the code has to be executed twice. 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 this DataFrame, it has to be recalculated which is the case for both actions in line 8 and line 9 of the code example.

When further removing the print command in line 4, the lazy execution 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 when calling `get_distances()` 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” the `preprocess_features()` has to be executed every time `get_distances()` is called.

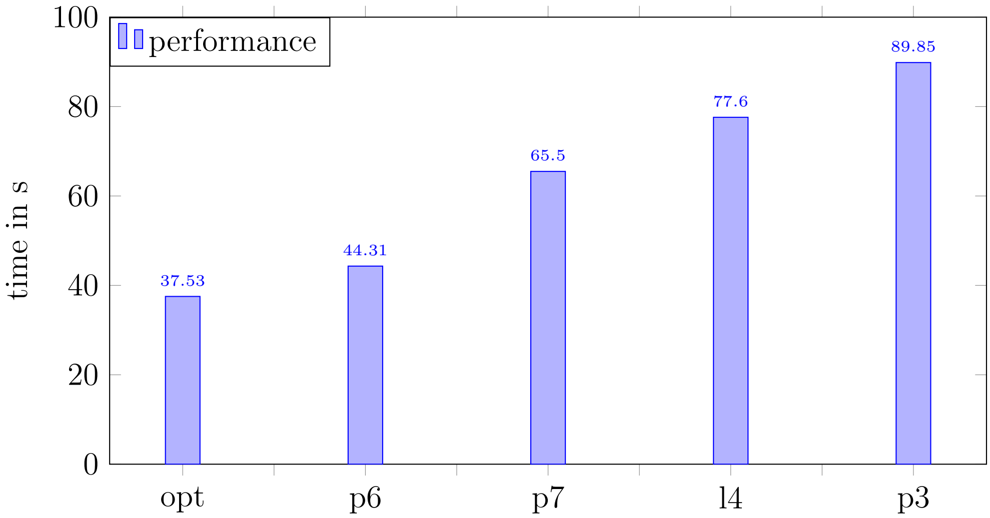


Figure 4.4: Lazy evaluation and caching

So in summary the correct way of caching data is a tricky task. Writing everything into the main memory is no solution either 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.
For the field of music similarity hat means that especially all pre-processed features have to fit into the main memory of the cluster to speed up consecutive song requests.

4.2.5 distance scaling

To combine different distance measurements into one combined distance the various results from the different kind of features have to be rescaled to avoid biasing the overall distance. The easiest way is to subtract the minimum of all distances d and dividing by the difference between the maximum and the minimum distance as described in equation 4.3.

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

The minimum distance should always be the self-similarity of the requested song with a value of 0 but in the implementation the Symmetric Kullback-Leibler distance this isn’t always the case. The analysis of the distances in chapter 5.2 also shows that e.g. the levenshtein distances and cross-correlation results are unequally distributed over

the interval between 0 and 1 (unit interval $[0, 1]$). Dropping the self-similarity out of the distance vector and rescaling it afterwards with a new miniumum distance unequal to zero could solve this but was not tested in this thesis. A second issue was already mentioned in section 4.2.4 where outliers tend to bias the results. These can get filtered out before rescaling the distances. This is further evaluated in chapter 5.2.
Another option to rescale the features laid out by Sebastian Stober in [4, pp. 543ff] but not implemented in this thesis would be to rescale all distances to have a mean value of 1 by using equation 4.4 by dividing the distance 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.

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

Implementation-wise the aggregation of the minimum and maximum value went through different tests. During first tests the aggregation of minimum and maximum value were performed separately (see 4.20). This turned out to be very inefficient because the data had to be accessed multiple times.

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

Code Snippet 4.19: Minimum and maximum aggregation separate

An improved version shown in code example 4.20 only uses one action to gather minimum and maximum value, which improved the overall performance significantly.

```
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 4.20: Minimum and maximum aggregation optimized

Another alternative would be the usage the `describe()` function for DataFrames. For the implementation using RDDs the `stats()` function was used, returning minimum, maximum, mean and variance values all at once.

4.2.6 Combining different measurements

To finally compute the overall similarity of what Stober calls the facet distances in [4, pp. 543ff] (the different distances computed using different feature sets) the weighted arithmetic mean of the previously scaled facet distances is calculated by using equation 4.5.

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

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.

4.2.7 performance

Cluster configuration

The first thing that had to be done was to alter the spark cluster configuration for the ARA-cluster as described in chapter 2.5.2. With the help of the `spark.dynamicAllocation` parameters the amount of Executors spawned can be determined. While normally the executors are spawned and then retained for the life of the application, dynamicAllocation is able to free the resources of Executors that are idle for a long time and to reassign the belonging system resources.
The cluster configuration in the code example 4.21 turned out to perform best. The cluster is configured in a way where 16 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 file across the cluster.

```
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.dynamicAllocation.enabled", "True")
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 4.21: cluster setup

To get the most performant cluster configuration various cluster settings were tested. Increasing the `repartition_count` and the amount of Executors spawned (with fewer ressources each) seemingly only increased the overhead and network traffic on the cluster without reducing the overall computation time. Only increasing the `repartition_count` while keeping the Executors as large as in the code snippet turned out to be slower as well.
Although each node on the ARA-cluster has 36 CPU cores (without hyper-threading), 32 cores assigned to each Executor turned out to perform just a little bit better when calculating the similarities for only one song in the first tests. Therefor the cluster configuration was set as described in the code snippet 4.21 for the following tests in this section.
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.

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. The computation time of all feature types (with respect to the lazy evaluation as described in section 4.2.4) is pictured in figure 4.5.

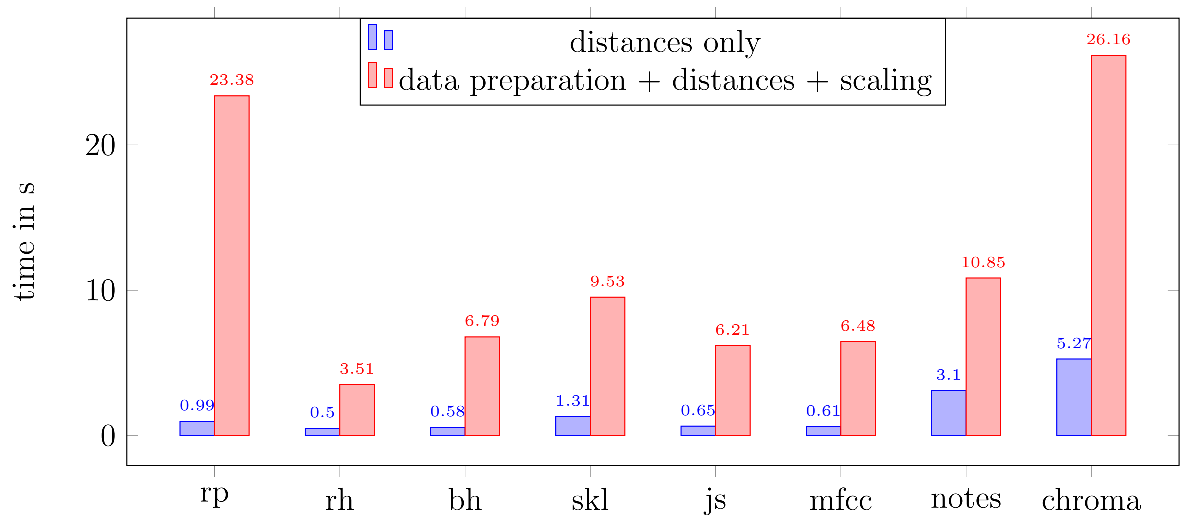


Figure 4.5: Performance on different feature types

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, that the features were stored in the main memory already.

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 is 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.

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

Data representation

Figure 4.7 and 4.8 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 storeed in one large DataFrame that then gets repartitioned across all nodes and cached. The idea behind this approach is to reduce shuffling operations during the computation of the 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 tolerated during the pre-processing stage. 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 the feature data by the song ID, the repartitioning and the caching, distances of the same songs but for different feature types are calculated on the same node in theory, reducing unnecessary shuffling operation during the compute time. The resulting small DataFrames containing the facet distances for one feature set are joined by the song IDs once all similarities are computed. Then the joined results are scaled using only one `agg()` call for all feature types (see section 4.2.5) and the combined distances are summed up and sorted. Figure 4.6 shows the adapted workflow (original see figure 4.3) of this approach

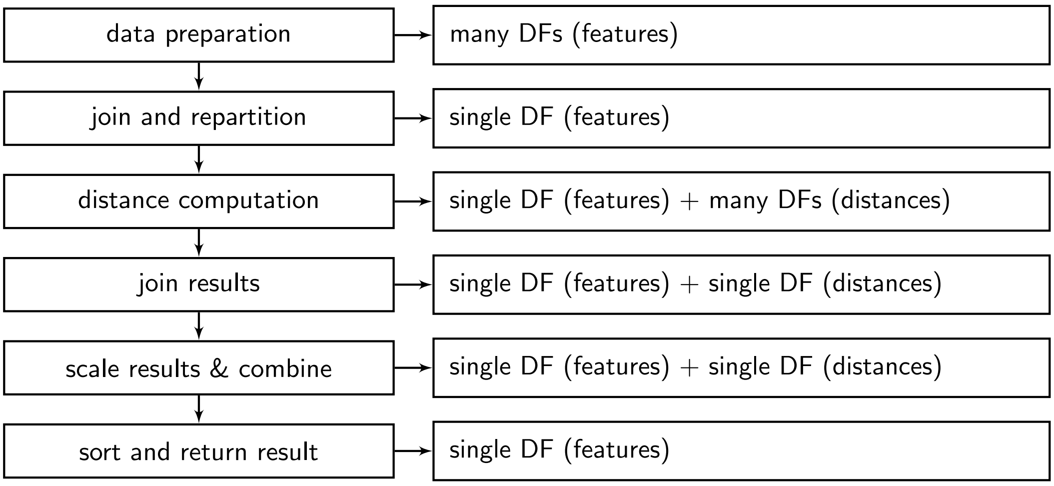


Figure 4.6: Workflow Merged DF

The second approach annotated with "DF" also uses DataFrames, but stores the different pre-processed feature types in separate smaller DataFrames instead. The third approach doesn't use DataFrames at all and uses single RDDs for the pre-processed features. Each of the times measured include the full workflow including data pre-processing, calculating, scaling and combining the 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 make up for this when presenting the performance on the calculation of subsequent song requests on the same features.

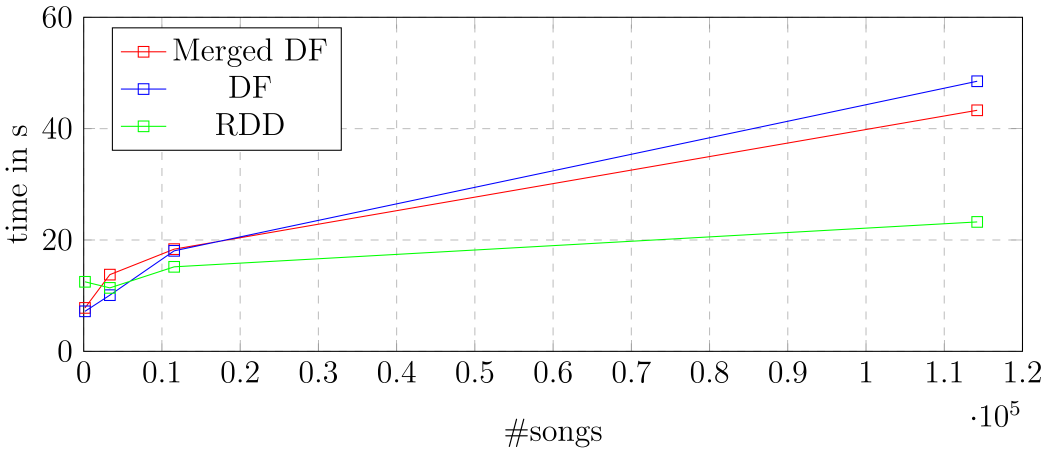


Figure 4.7: Performance ARA, full workload, (MFCC + Notes + RP)

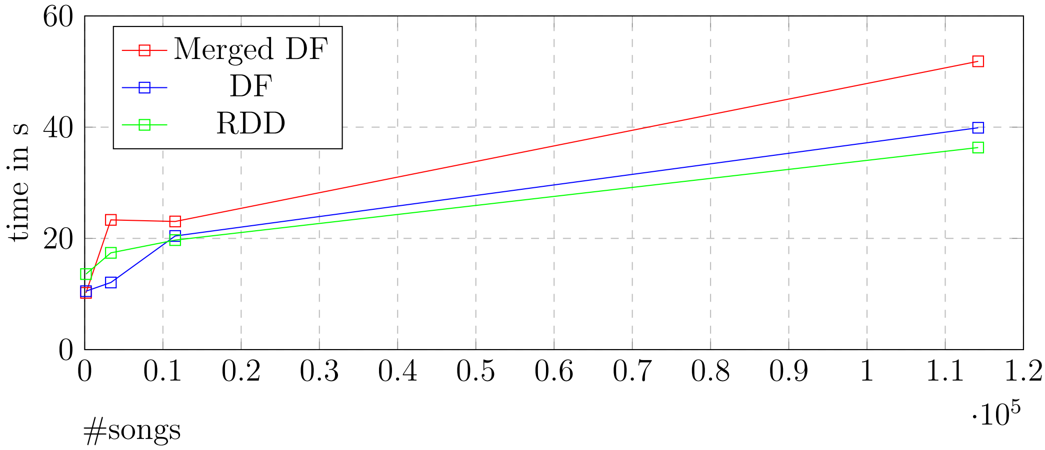


Figure 4.8: Performance ARA, full workload, (JS + Chroma + RP)

Performance for subsequent song requests

In contrast to the performance analysis from the last section, figure 4.9 shows the time measured to process 2 subsequent song requests. That means that the second consecutive song request is able to use the already pre-processed and cached feature data. Figure 4.9 shows the according results.

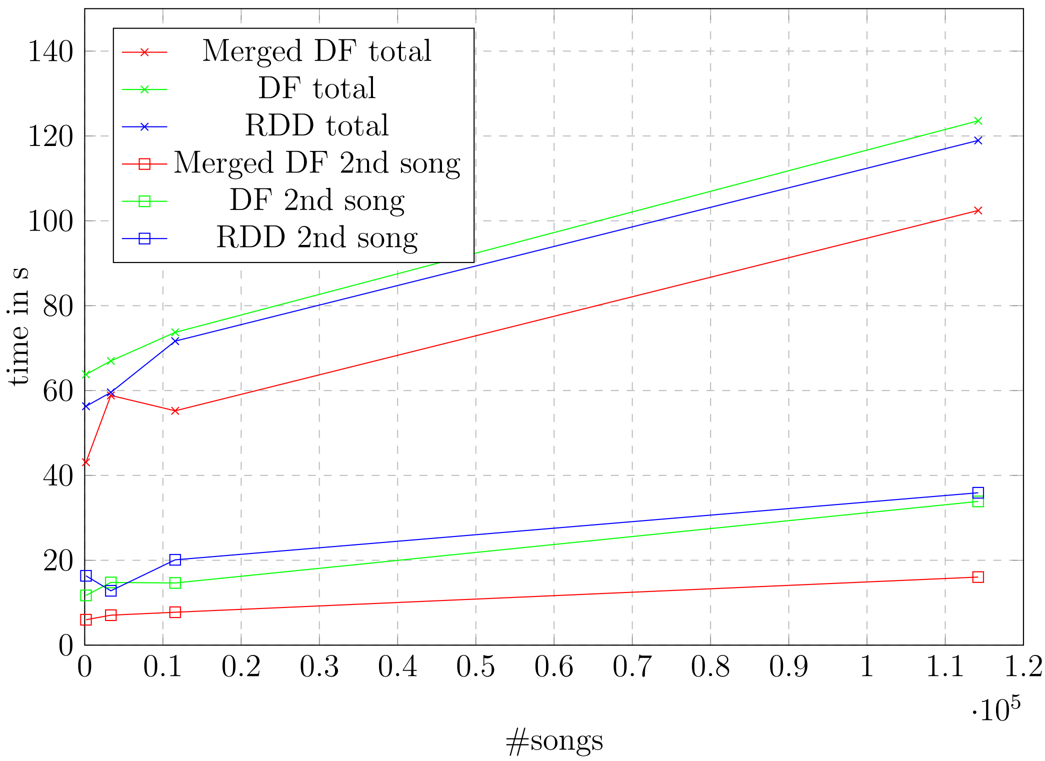


Figure 4.9: two subsequent songs, all features

The plots annoted with Merge DF total, DF total and RDD total depicture 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. The measured times include the calculation of the distances, the scaling and the join operation of the different result-types. 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 4.2.7 cluster configuration).

Descending importance filter and refine

To improve the performance even further a filter and refine method was tested where the similarities are computed for one feature set and all songs to which the distance is larger than the mean value of all 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 "Merge DF" approach described and pictured in 4.6 earlier on but a few changes had to be made. 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 compute cluster has enough RAM available to cache the full feature DataFrame twice. The first feature set is chosen and the distances are calculated and appended to the cloned version of the full feature DataFrame. In the next step all rows of the DataFrame where the freshly calculated distances are larger than some 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. 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. Of course the copying of the data on the other hand is an additional overhead. But when looking at the results in figure 4.10 it shows that the filter and refine method scales very well with increasing sizes of the dataset. The plots show the performance of full requests on already cached feature DataFrames or RDDs but the plots of the filter and refine tests include the time necessary to create a copy of the cached feature DataFrames so the additional overhead is taken into account.

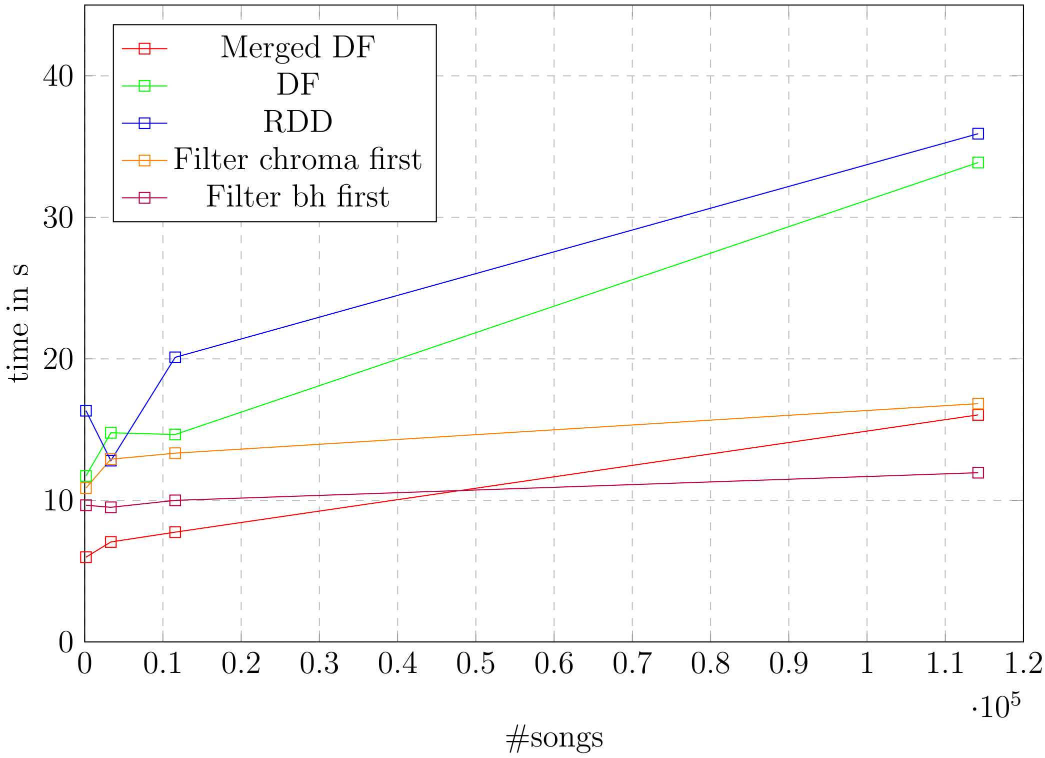


Figure 4.10: Descending importance filter and refine, all features

The order of filtering for the plot labeled with "Filter chroma first" is:
chroma → mfcc → js → skl → rp → rh → bh → notes
and:
bh → rh → notes → rp → mfcc → js → skl → chroma
for the plot labeled with "Filter bh first". The order of the different filter and refine operations is very important. For example when searching for cover songs, the cross-correlation and the levenshtein distance should be calculated in the very beginning of the filter chain or otherwise the covers songs could be filtered out. 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 able to still detect one alternative recording as the top recommendation and the other recording was placed as recommendation number 14, even scoring higher than in a test without the filter and refine method because other non-matching songs got filtered out.
Of course the computation of the cross-correlation between the chroma features is the most compute-intensive one so for performance reasons it is better to start with a distance measurement like the euclidean distance of the beat histograms because 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 requesting song recommendations has to define which aspect is most important to him (speed, melodic/ rhythm/ timbral features or cover

song detection) before choosing an order for the filter chain. The results get better the further it gets in the filter chain.

4.2.8 possible improvements and additions

Spark offers some other interesting alternatives to compute similarities that are only mentioned here and not further evaluated. The Alternating Least Squares to perform collaborative filtering (see section 2.3.4) 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 informations.
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 implementation of the melodic similarity computation using the mentioned approach in chapter 3.2.3

5. Results

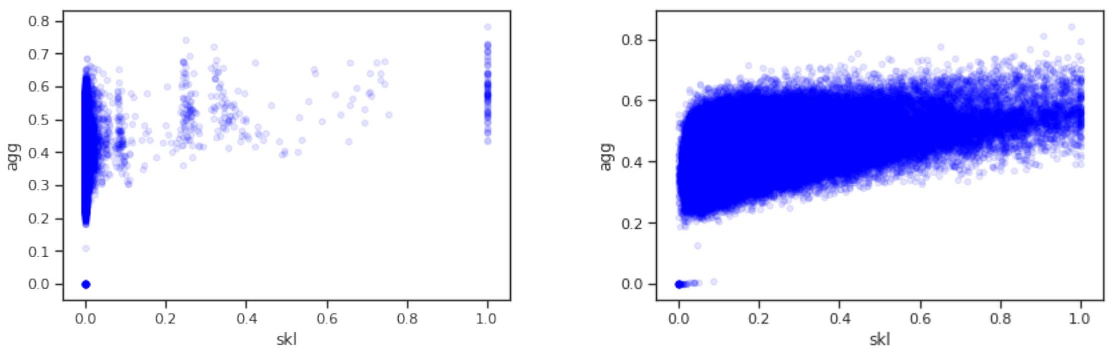
Very small 100 Song Dataset plus 20 Cover Songs to evaluate and compare similarity metrics. Full similarity matrices. High performance/ throughput tests with full 12000 song testset to evaluate full load performance.

5.1 Runtime

5.2 feature separation quality

Which features are useful, which not (looking at you rhythm histogram)

skl scaled and unscaled5.1
problem with outliers 4.2.4

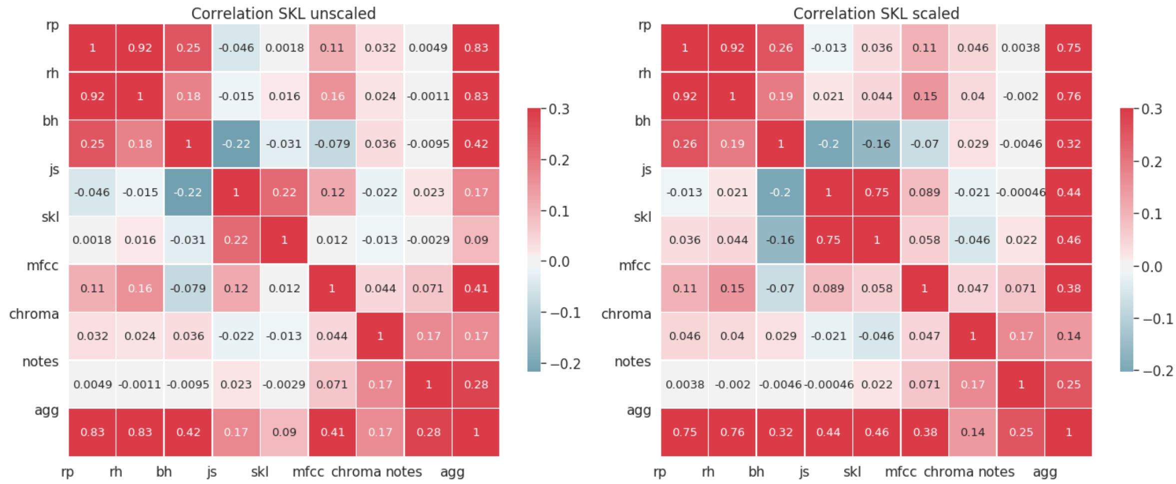


(a) SKL unscaled (b) SKL scaled

Figure 5.1: Correlation of features

correlation of features in figure 5.2

Scatter Matrix, 5 songs of each genre from 1517 artists - distances combined, main diagonal = Kernel Density Estimation in figure 5.4



(a) SKL unscaled (b) SKL scaled

Figure 5.2: Correlation of features

	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 5.3: correlation 95 songs, 19 genres (5 each), 1517 artists

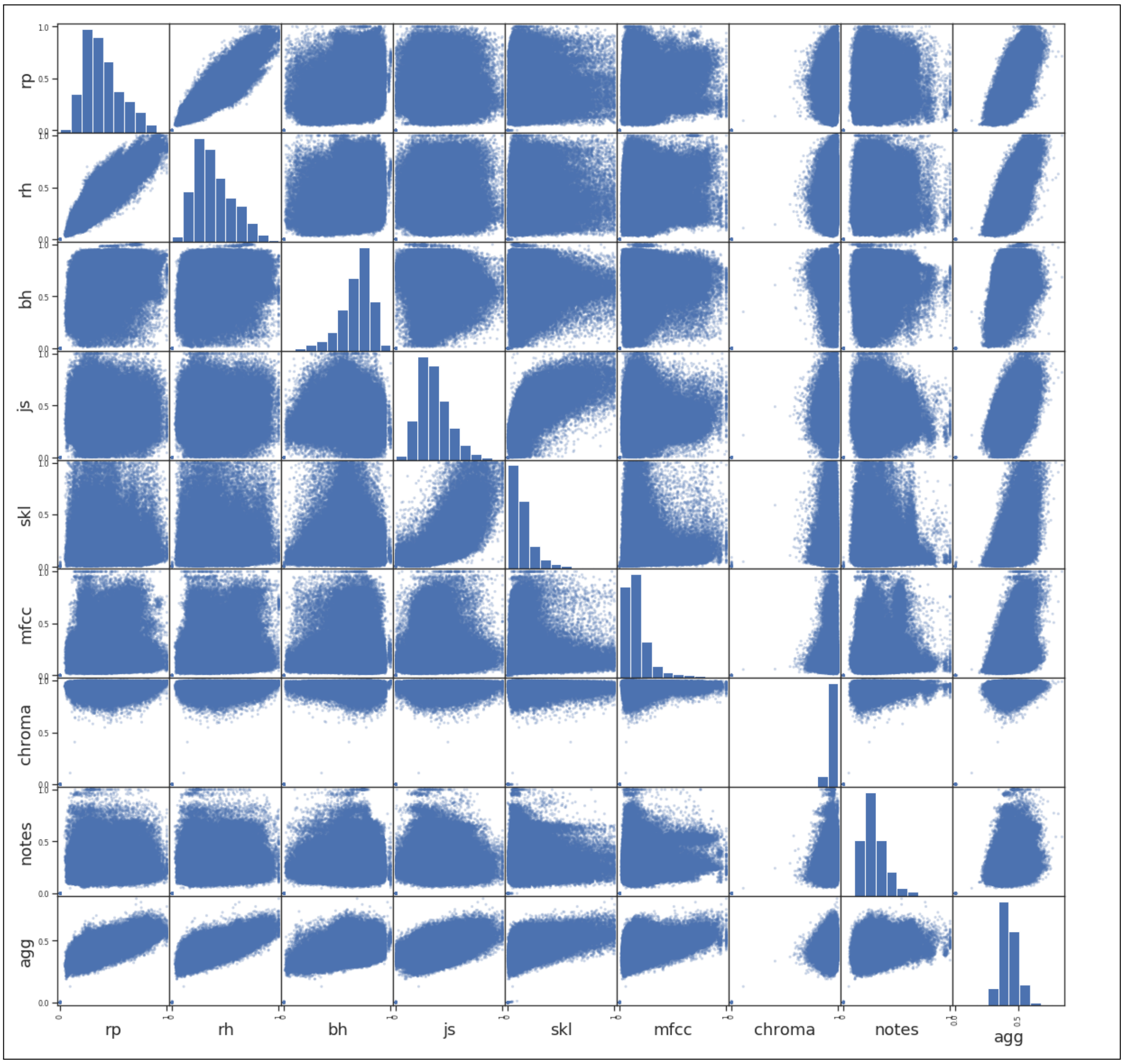


Figure 5.4: correlation 95 songs, 19 genres (5 each), 1517 artists

Scatter Matrix, 1 Song (Soundtack) from 100 song sample - distances, main diagonal = Kernel Density Estimation, picked subset of 5 genres in figure 5.5

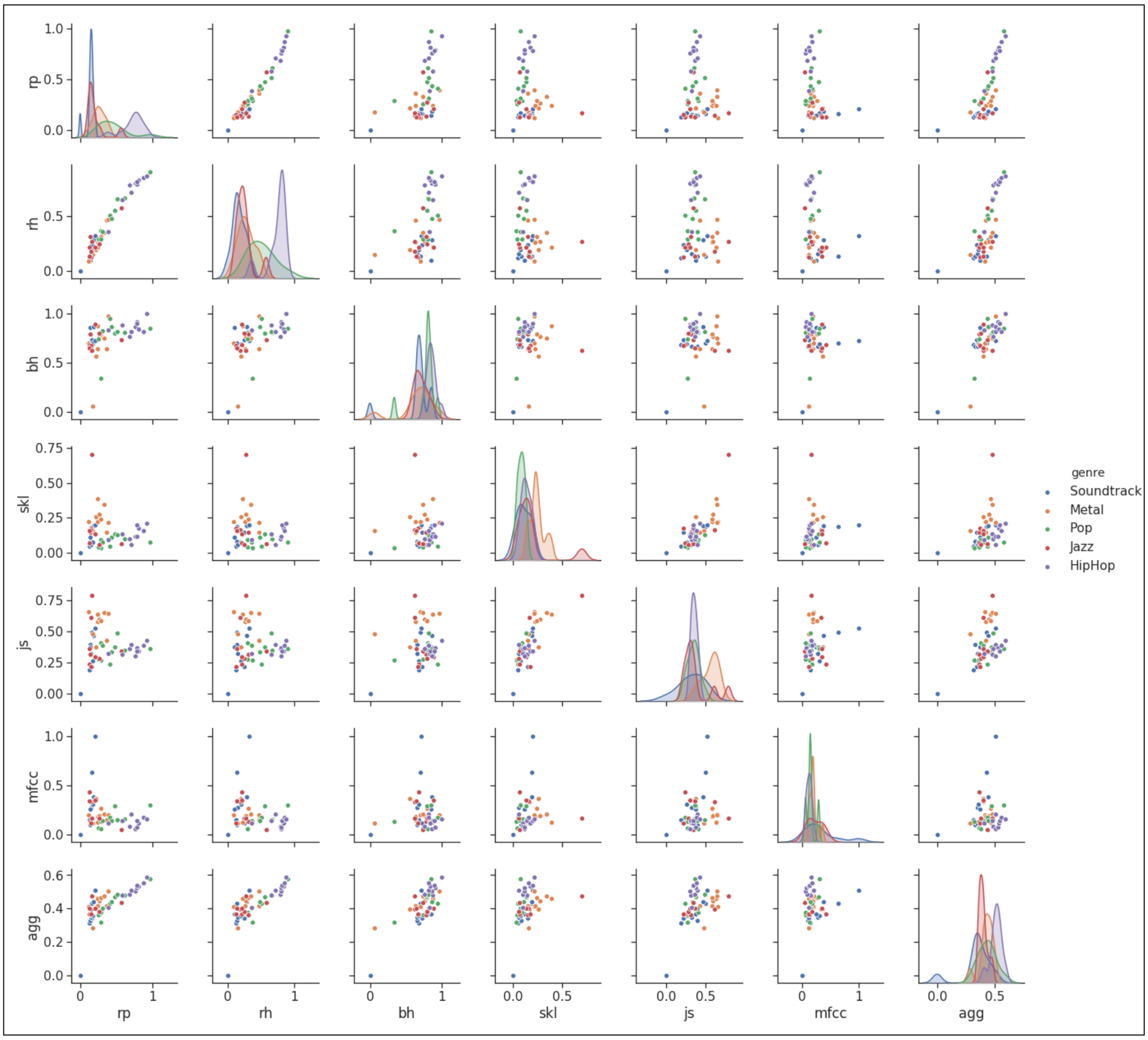


Figure 5.5: distances 1 song (Soundtrack), 5 genres (10 songs each)

Scatter Matrix, 1 Song (Rock/ Metal) from 1517 artists - distances, main diagonal = Kernel Density Estimation, picked subset of 3 genres
All features combined recommends Rock primarily
Most interesting Detail: Plot JS - RP: overall similarity JS between Rock and Hip Hop - overall similarity RP between Classical and Metal - only one feature would not be able to separate all 4 genres in figure 5.6
Scatter Matrix, 1 Song (Electronic) from 1517 artists - distances - figure 5.7 No combination of features is able to accurately separate HipHop from Rock-Pop
Scatter Matrix, 1 Song (Classics) from 1517 song sample - distances, main diagonal = Kernel Density Estimation, picked subset of 4 genres in figure 5.8
distances from classical song out of 1517 artists dataset in figure 5.9

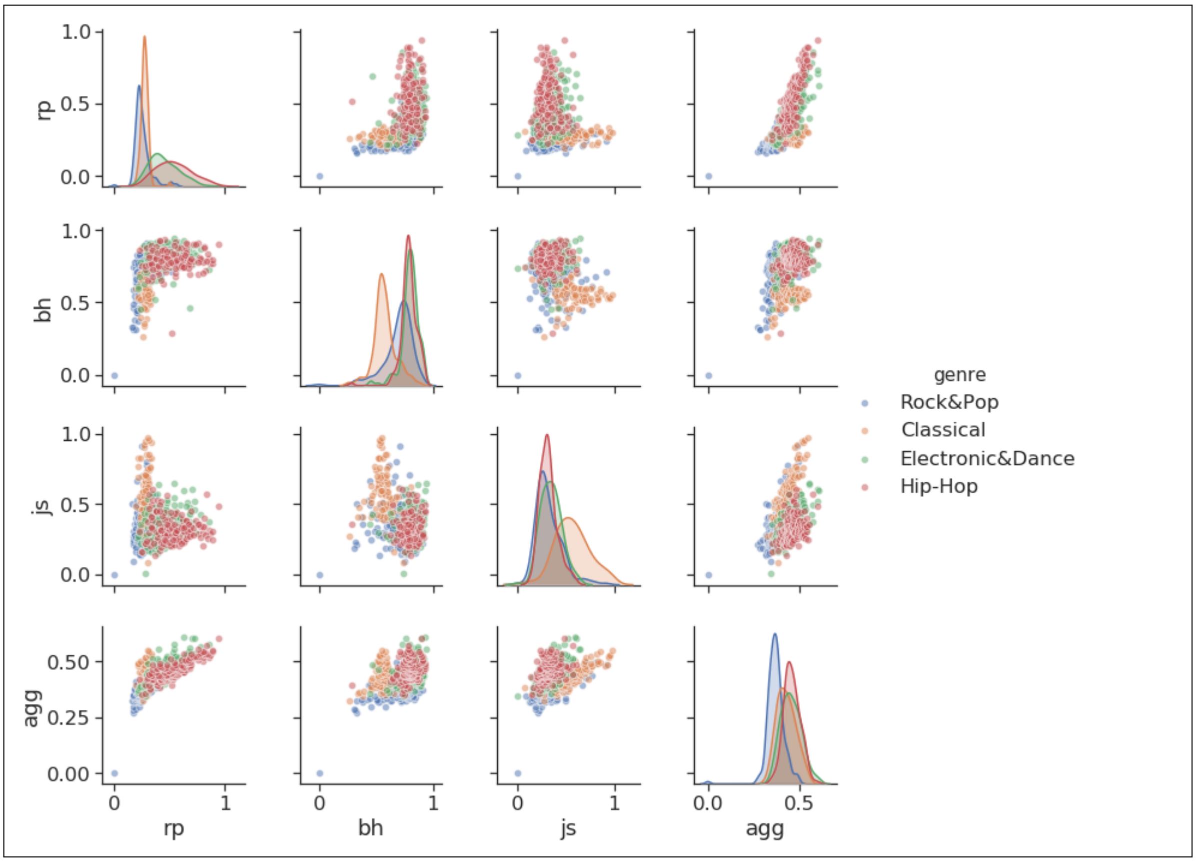


Figure 5.6: distances 1 song, Rock/ Metal, 1517 artists, 4 genres

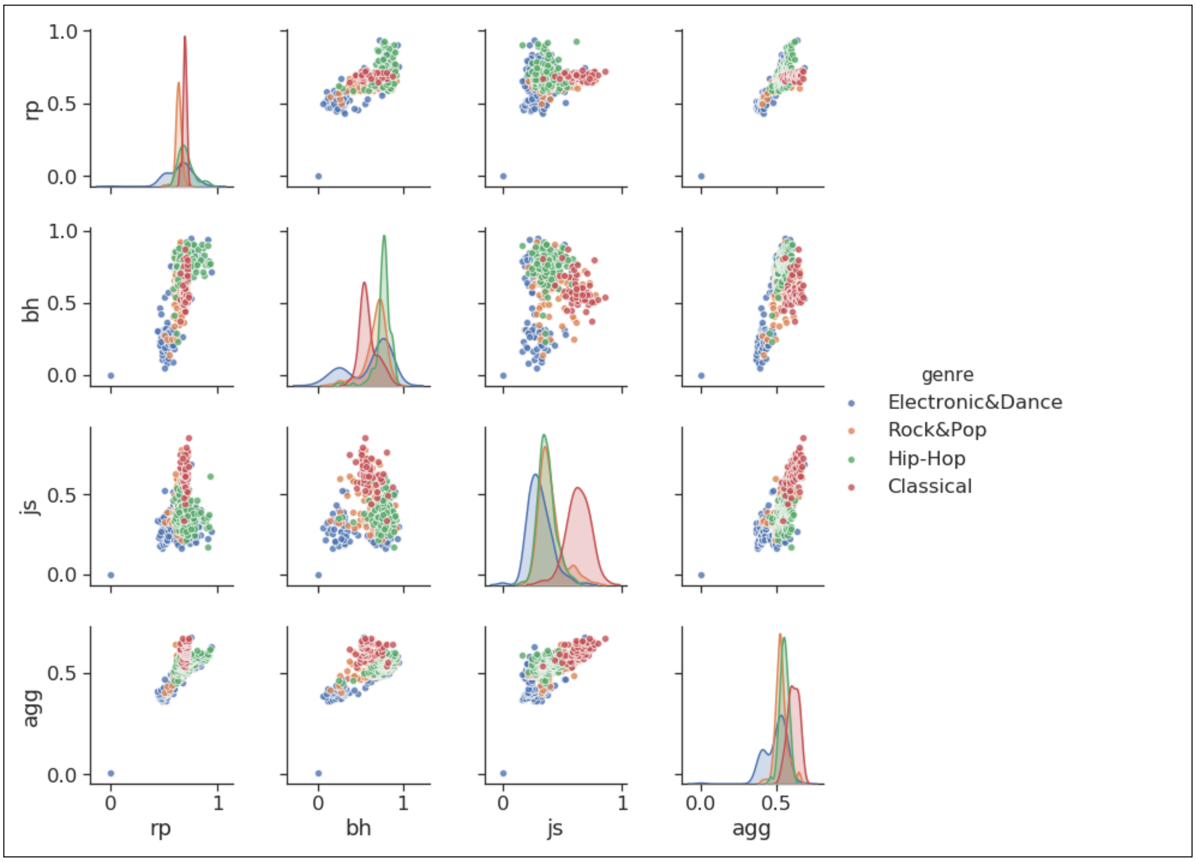


Figure 5.7: distances 1 song, electronic, 1517 artists, 4 genres

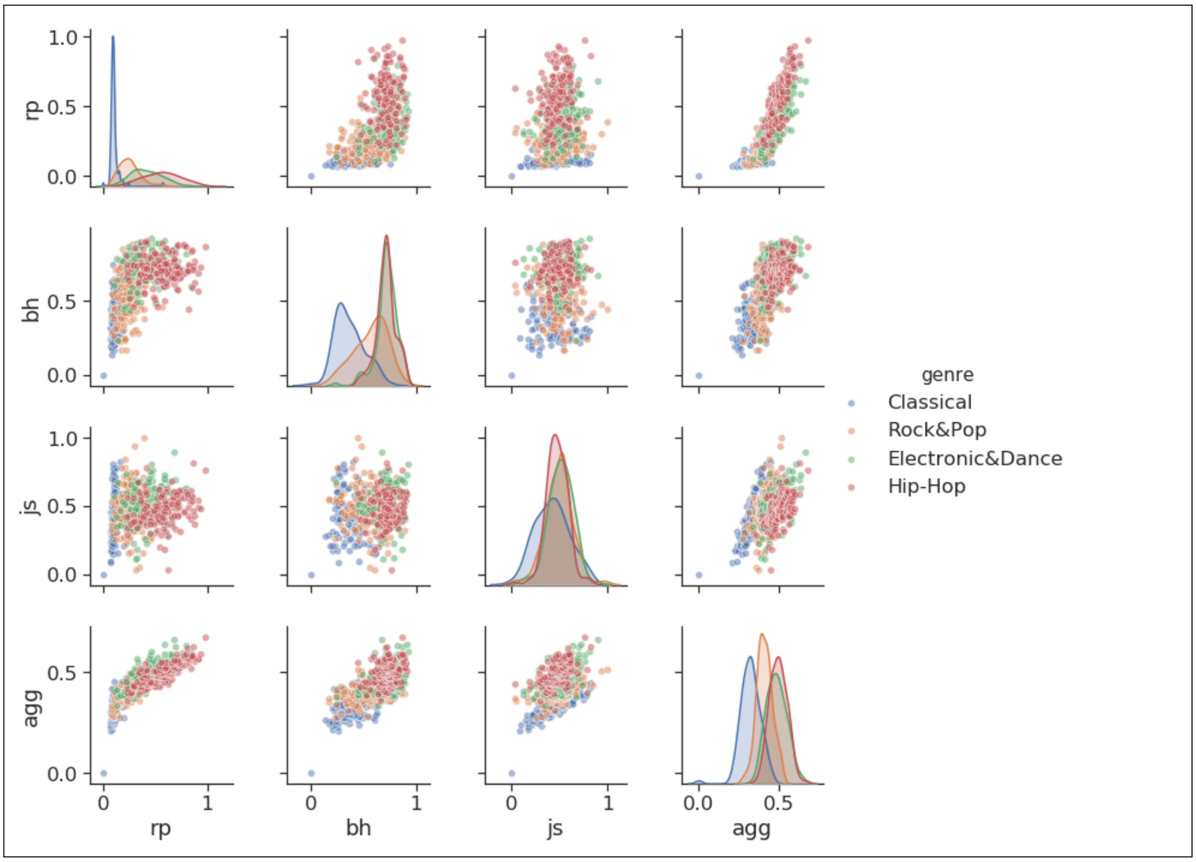
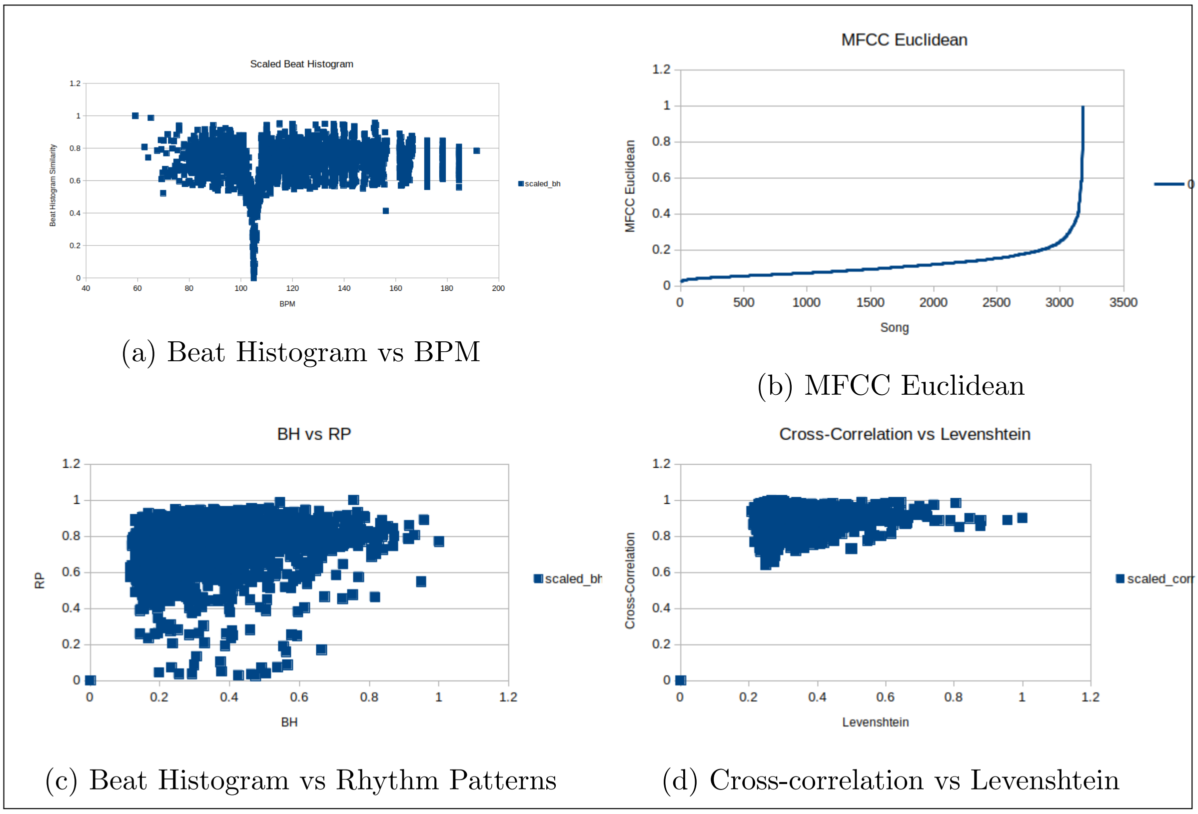


Figure 5.8: distances 1 song (Classical), 4 genres (10 songs each)



(a) Beat Histogram vs BPM

(b) MFCC Euclidean

(c) Beat Histogram vs Rhythm Patterns

(d) Cross-correlation vs Levenshtein

Figure 5.9: genre recall 100 songs

5.3 Cover song identification

Able to find "Rock you like a Hurricane" cover as top recommendation in over 11500 songs

Able to find "Für Elise" cover as top recommendation in over 11500 songs, even with filter and refine

counting the hits in the top 10 results of 80 requested songs on 164 song dataset covers80:

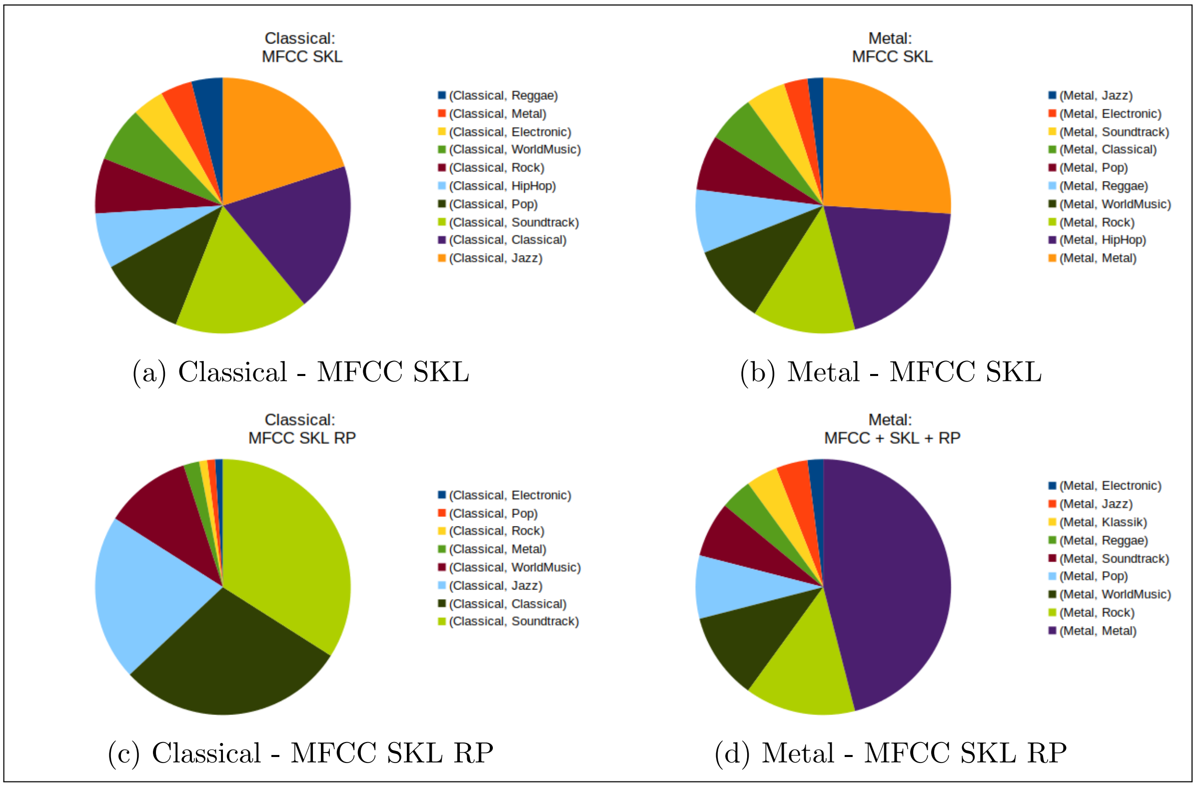
- chroma + notes + rp: 28
- chroma + notes: 24
- notes: 24
- notes + rp: 22
- mfcc + notes + rp: 20 (mfcc faulty)

- chroma: 16

interesting sidenote: results of chroma only and notes only different! Nearly ever

5.4 Genre similarity

100 Song Testset, 10 genres:



(a) Classical - MFCC SKL

(b) Metal - MFCC SKL

(c) Classical - MFCC SKL RP

(d) Metal - MFCC SKL RP

Figure 5.10: genre recall 100 songs

5.5 ... and beyond genre boundaries

Rachmaninoff Prelude C# minor full dataset Top 5 MFCC Euclidean

Titel: Klassik/Rachmaninoff - Idil Biret - Op 3 No 2 Prelude in C - Sharp Minor

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

5.6 Summary and outlook

In the first part of this thesis an overview over the field of MIR was given, different audio features were explained, similarity measurements evaluated and multiple algorithms for timbre similarity were presented.
Data was collected, over 1TB of music data, 114000 songs. The necessary audio features were extracted and pre-processed (Melody Estimation) in parallel using MPI on a cluster paving the way for usage with big data processing frameworks.
The features were loaded into the HDFS of a cluster and multiple similarity measurements were implemented, tested, evaluated and improved. With spark multiple approaches (RDD, DataSet, Filter and Refine, Cluster Configurations) were tested.
The results were presented and visualized.

outlook:

more state of the art similarity (blocked)
performance improvements
spark streaming, real-time
jensen-shannon investigation
garbage collecting issue
Genre and metadata, Genre specific features, combinations and variable model, Collaborative Filtering, Lyrics
reduce hubness

References

[1] Peter Knees and Markus Schedl. Music Similarity and Retrieval. An introduction to web audio- and web-based strategies. In: Springer, 2016. ISBN: 9783662497203.

[2] B. McFee et al. The Million Song Dataset Challenge, AdMIRe '12. In:

[3] Brian McFee et al. LibROSA: Audio and Music Signal Analysis in Python. In: Proceedings of the 14th Python in Science Conference.

[4] C. Weihs; D. Jannach; I. Vatolkin; G. Rudolph. Music Data Analysis. New York: Chapman and Hall/CRC, 2017. URL: https://doi.org/10.1201/9781315370996.

[5] David Moffat, David Ronan, and Joshua Reiss. An Evaluation of Audio Feature Extraction Toolboxes. In: Nov. 2015. DOI: 10.13140/RG.2.1.1471.4640.

[6] B.Mathieu et al. YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software. In: Proceedings of the 11th ISMIR conference, Utrecht, Netherlands, 2010.

[7] D. Bogdanov 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.

[8] Michael I. Mandel and Daniel P. W. Ellis. Labrosa's audio music similarity and classification submissions. In: 2007.

[9] Jupyter. In: URL: https://jupyter.org/index.html.

[10] Dr. Dominik Schnitzer. Audio Music Similarity. In: URL: http://www.musly.org/index.html.

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

[12] D. Schnitzer et al. Using mutual proximity to improve content-based audio similarity. In: Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR, 2011.

[13] Olivier Lartillot and Petri Toiviainen. 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.

[14] MATLAB. in: URL: https://de.mathworks.com/help/matlab/index.html.

[15] John W. Eaton 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/.

[16] Martin Ariel Hartmann. A port of MIRToolbox for Octave. In: 2016. URL: https://github.com/martinarielhartmann/mirtooloct.

[17] Prélude cis-Moll (Rachmaninow). In: URL: https://imslp.org/wiki/Morceaux_de_fantaisie\%2C_Op.3_(Rachmaninoff\%2C_Sergei).

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

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

[20] J. Salamon and E. Gómez. 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] C. Cannam, C. Landone, and M. Sandler. 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] Dr. Dominik Schnitzer. Dealing with the Music of the World: Indexing Content-Based Music Similarity Models for Fast Retrieval in Massive Databases, 1st ed. In: 2012. ISBN: 9781477494158.

[23] Matija Marolt. 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.

[24] Jonathan Foote, Matthew Cooper, and Unjung Nam. Audio Retrieval by Rhythmic Similarity. In: Jan. 2002.

[25] Zhao Yufeng and Li Xinwei. Design and Implementation of Music Recommendation System Based on Hadoop. In: 2018. DOI: 10.2991/icsnce-18.2018.36..

[26] B. McFee and G.R.G. Lanckriet. 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.

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

[28] Guangyu Xia et al. MidiFind: Similarity Search and Popularity Mining in Large MIDI Databases, 259-276. In: 2013. DOI: 10.1007/978-3-319-12976-1_17..

[29] Jong Wook Kim et al. CREPE: A Convolutional Representation for Pitch Estimation. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2018.

[30] Guangyu Xia 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.

[31] Sankalp Gulati, Joan Serra, and Xavier Serra. An evaluation of methodologies for melodic similarity in audio recordings of Indian art music, 678-682. In: 2015. DOI: 10.1109/ICASSP.2015.7178055..

[32] Cumhur Erkut et al. Extraction of Physical and Expressive Parameters for Model-based Sound Synthesis of the Classical Guitar. In: 2002.

[33] Michäel Defferrard et al. FMA: A Dataset for Music Analysis. In: 18th International Society for Music Information Retrieval Conference, 2017. URL: https://arxiv.org/abs/1612.01840.

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

[35] Soundcloud bqpd. In: URL: https://soundcloud.com/bq_pd.

[36] John Thickstun, Zaid Harchaoui, and Sham M. Kakadetitle. Learning Features of Music from Scratch. In: International Conference on Learning Representations (ICLR). 2017. URL: https://arxiv.org/abs/1611.09827.

[37] Klaus Seyerlehner. 1517-Artists Dataset. In: 2010. URL: http://www.seyerlehner.info/index.php?p=1_3_Download.

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

[39] R. Bittner 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.

[40] B. De Man et al. The Open Multitrack Testbed. In: 137th Convention of the Audio Engineering Society, 2014.

[41] Spotify API. in: URL: https://developer.spotify.com/documentation/.

[42] The Echo Nest. In: URL: http://the.echonest.com/.

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

[44] Spotify Playlist Miner. In: URL: https://developer.spotify.com/community/showcase/playlist-miner-spotify/.

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

[46] Spotify Developer Terms of Service. In: URL: https://developer.spotify.com/terms/.

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

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

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

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

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

[52] Matei Zaharia 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. URL: http://doi.acm.org/10.1145/2934664.

[53] Apache Hadoop. In: URL: https://hadoop.apache.org/.

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

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In: vol. 37. Dec. 2003, pp. 29–43. DOI: 10.1145/945445.945450.

[56] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In: vol. 51. Jan. 2004, pp. 137–150. DOI: 10.1145/1327452.1327492.

[57] MapReduce. In: URL: https://commons.wikimedia.org/wiki/File:Mapreduce.png.

[58] Butch Quinto. 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.

[59] Raúl Estrada and Isaac Ruiz. 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.

[60] Klaus Seyerlehner and Markus Schedl. Block-Level Audio Features for Music Genre Classification. In: 2009.

[61] Nicola Orio and Antonio Rodà. A Measure of Melodic Similarity based on a Graph Representation of the Music Structure. In: ISMIR. 2009.

[62] David Englmeier 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 Norbert Ritter et al. Bonn: Gesellschaft für Informatik e.V., 2015, pp. 183–192.

[63] Joan Serra 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.

[64] Daniel P.W. Ellis and Graham E. Poliner. 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.

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

[66] Daniel P W Ellis and Courtenay Cotton. The 2007 LabROSA cover song detection system. In: (Jan. 2007).

[67] George Tzanetakis and Perry Cook. 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.

[68] Matthias Gruhne, Christian Dittmar, and Daniel Gärtner. Improving Rhythmic Similarity Computation by Beat Histogram Transformations. In: Jan. 2009, pp. 177–182.

[69] Thomas Lidy and Andreas Rauber. Evaluation of Feature Extractors and Psycho-Acoustic Transformations for Music Genre Classification. In: Jan. 2005, pp. 34–41.

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

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

[72] Elias Pampalk. Computational Models of Music Similarity and their Application in Music Information Retrieval. In: 2006.

[73] Tim Pohle et al. On Rhythm and General Music Similarity. In: Jan. 2009, pp. 525–530.

[74] Johannes Schoder. MusicSimilarity-Spark. In: URL: https://github.com/oObqpdOo/MusicSimilarity-Spark.

[75] mpi4py. In: URL: https://pypi.org/project/mpi4py/.

[76] beegfs. In: URL: http://www.beegfs.io/content/latest-release/.

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

6. Appendix

6.1 Spotipy Data Miner

```
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 = []
```

```python
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])
```

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.

Jena, August 23, 2019

Johannes Schoder