##### 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 ?y*)[*n*] can be denoted as follows:

*∞*

*k*[*n*] = (*x ? y*)[*n*] = X *x*[*m*]*y*[*m − n*] (3.11)

*m*=*−∞*

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*) = X X *X*(*m,n*)*H*(*m − k,n − l*)  *m*=0 *n*=0  with | (3.12) |
| *−* (*P −* 1) *≤ k ≤ M −* 1 | (3.13) |
| *−* (*Q −* 1) *≤ l ≤ N −* 1 | (3.14) |

*M−*1 *N−*1

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

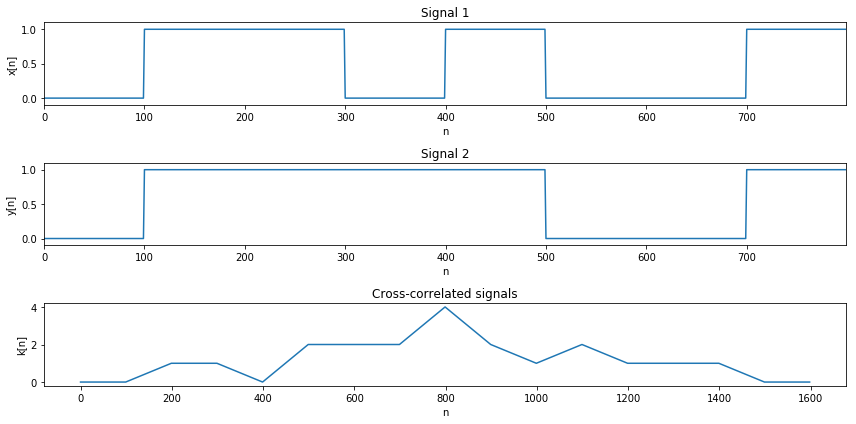


Figure 3.8: cross-correlation

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.

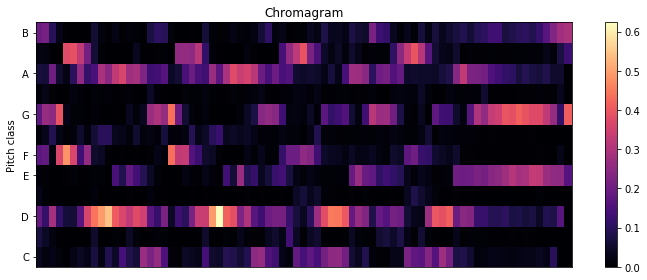
In this thesis a versions where the songs are all key aligned before the cross-correlation was tested, but due to the fact, that the key detection algorithm in the labrosa and essentia frameworks weren’t always correct a second version where addionally the crosscorrelation for all key transpositions 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) *≤ k ≤ 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*) = X X *X*(*m,n*)*H*(*m,n − l*)  *m*=0 *n*=0 | (3.15) |
| *−* (*Q −* 1) *≤ l ≤ N −* 1  or even faster without calculating the edges of the matrix. | (3.16) |
| 0 *≤ l ≤ N − Q* | (3.17) |

*M−*1 *N−*1

The post-processing step from Ellis and Poliner, namely the high-pass filtering of the result was also implemented.

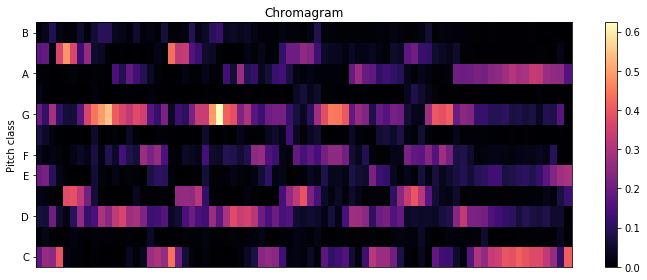
Figure 3.9 shows two beat aligned, key shifted and per beat averaged chroma features of two short guitar snippets and their cross-correlation. The interesting row of the cross-correlation matrix is the middle row marked with the C key. It shows that both already key shifted melodies do not correlate well. In figure 3.10 the cross-correlation



(

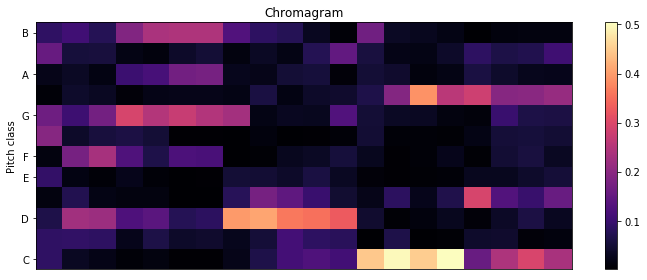
a) beat aligned chroma sound

1



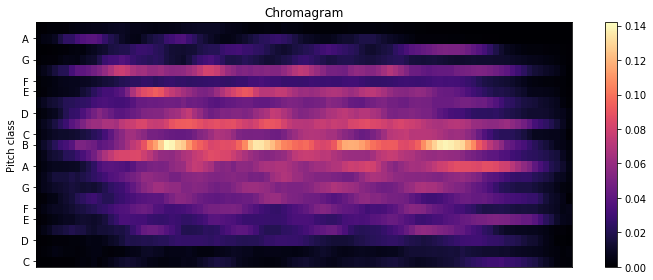
(

b) beat aligned chroma sound1 key shifted



(

c) beat aligned chroma sound2 key shifted



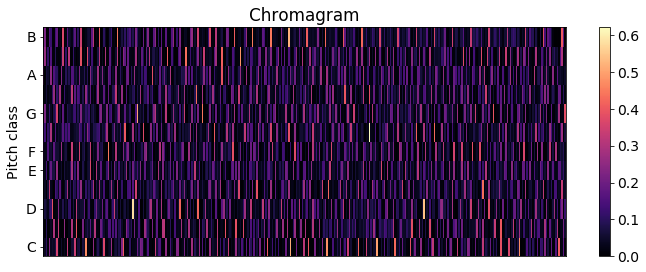
(

d) cross-correlation

Figure 3.9: beat-aligned chromagram

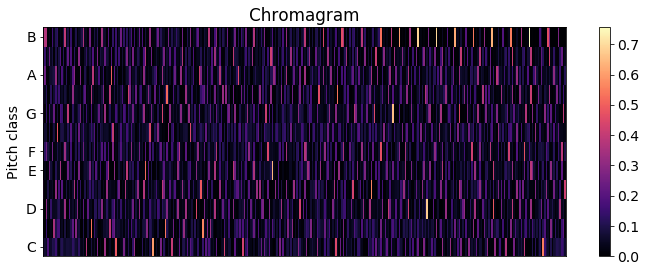
of the song ”Chandelier” by the singer Sia and covered by Pvris are shown in 3.10c and in contrast to this the cross-correlation of ”Chandelier” with the song ”Rock you like a Hurricane” by The Scorpions are shown. Due to the previous key shifting, plot 3.10a shows the maximum peak right in the center 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



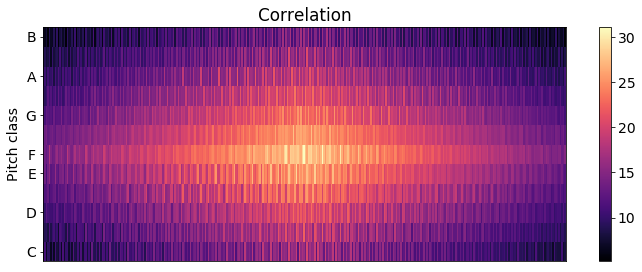
(

a) Pvris



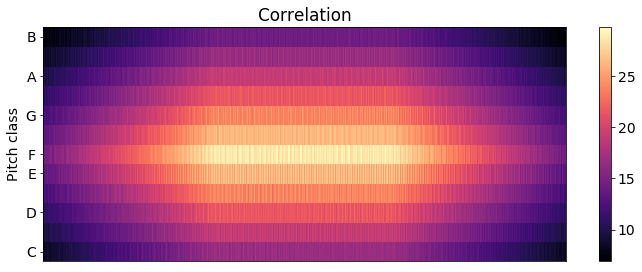
(

b) Sia



(

c) cross-correlation of cover songs

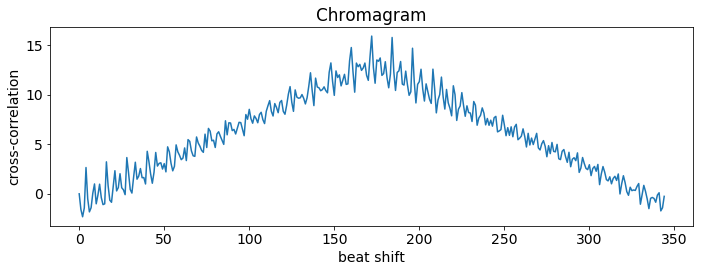


(

d) cross-correlation of unrelated songs

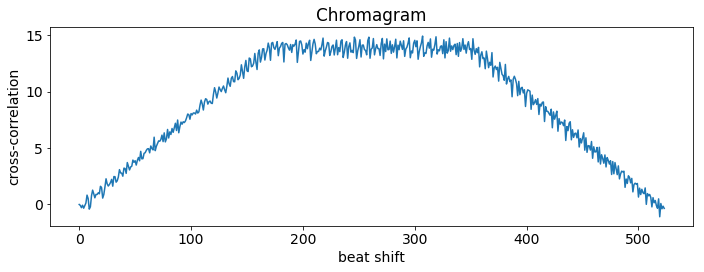
Figure 3.10: Cross-correlation

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.



(

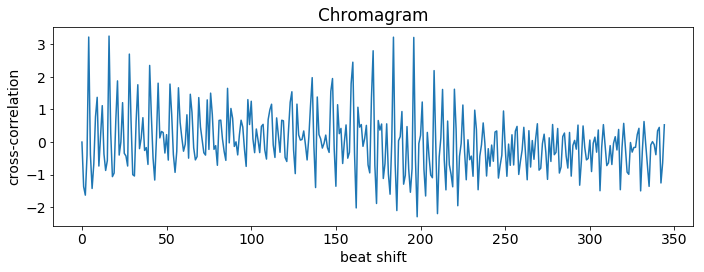
a) cross-correlation of cover songs



(

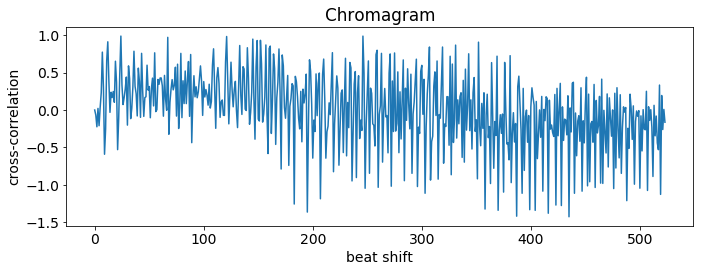
b) cross-correlation of unrelated songs

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

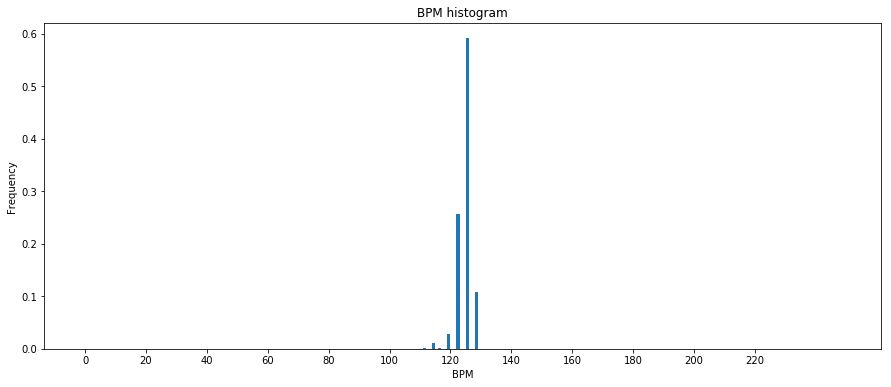
Other similarity measurements are e.g. the usage of beat histograms as proposed by

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 crosscorrelation 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

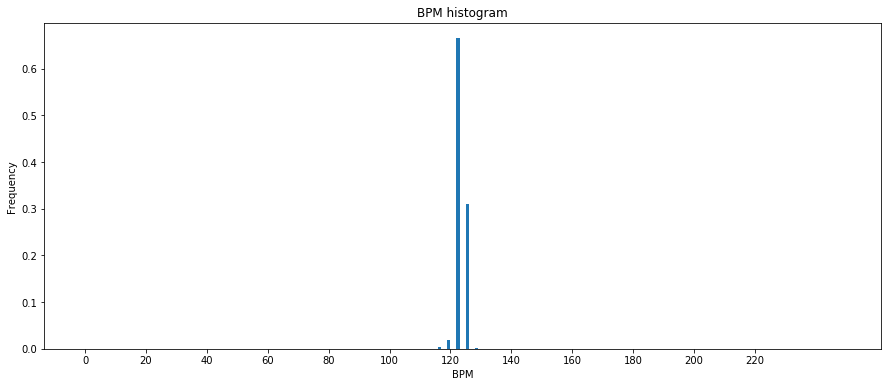
Scorpions and covered by Knightsbridge as well as two different versions of the song

”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.



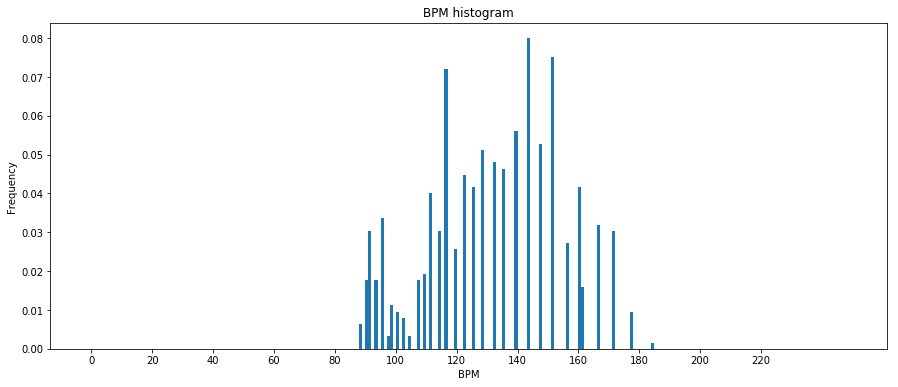
(

a) Scorpions



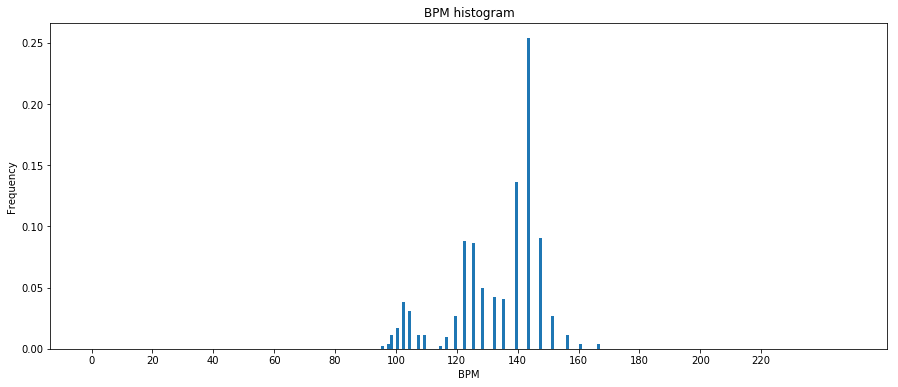
(

b) Knightsbridge



(

c) 94’ Version Stanne



(

d) 99’ Version Anders

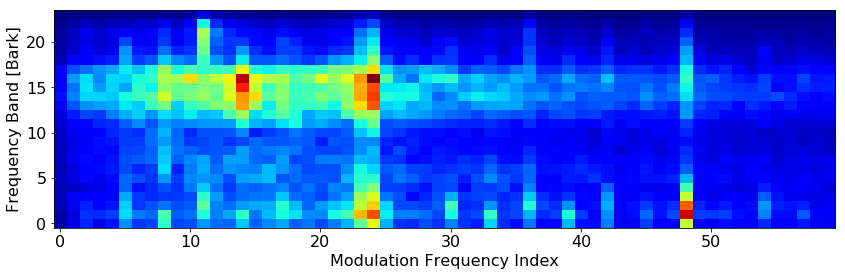
Figure 3.13: Beat Histogram

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

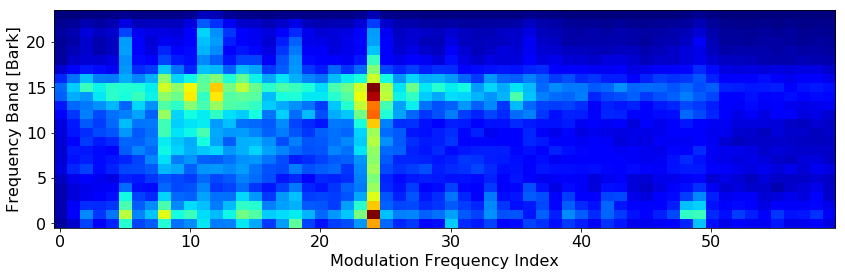
A more state-of-the-art feature is the so called rhythm pattern, also known as fluctuation patterns for instance mentioned by [69]. To extract these features the rp extractor library [70] was made publicly available by the TU Vienna [71]. Figure 3.14 shows the extracted rhythmic patterns of the previously 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



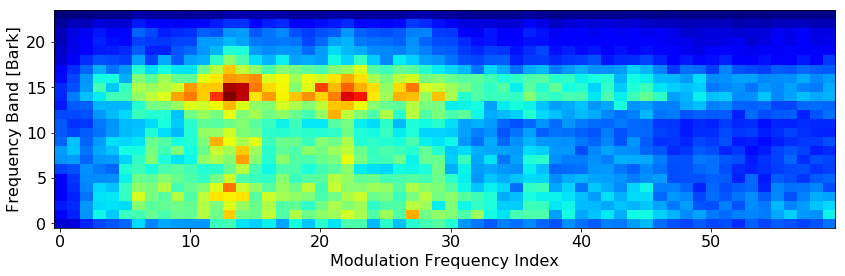
(

a) Scorpions



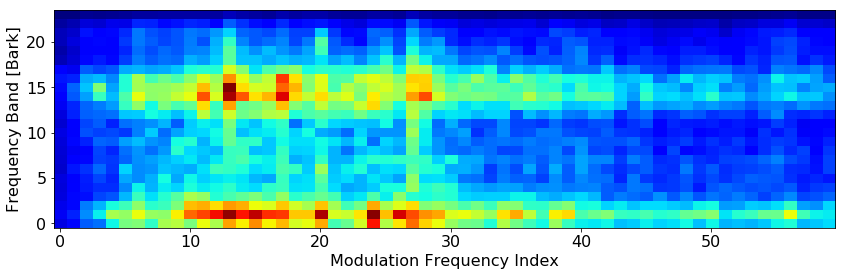
(

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* = 13arctan(0*.*00076*f*) + 3*.*5arctan((*f/*7500)2) (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.

Audio Signal

Pre-Processing

Power Spectrum (STFT)

Critical Bands (Bark scale)

Equal Loudness (Phon)

Specific Loudness Sens. (Sone)

Statistical Spectrum Descriptor

*→*

SSD

Modulation Amplitude (FFT)

Rhythm Histogram

*→*

RH

Fluctuation Strength Weighting

Filtering/ Blurring

Rhythmic Patterns

*→*

RP

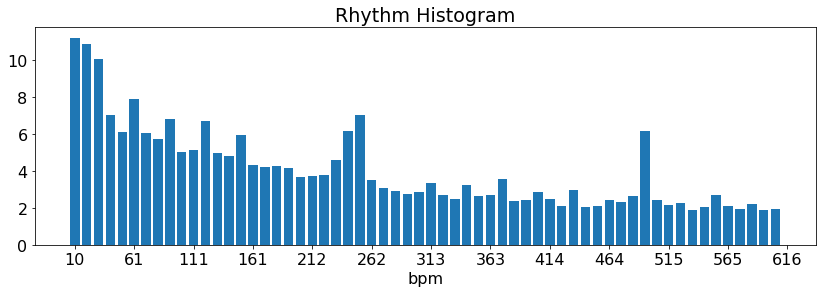
Figure 3.15: Rhythm Pattern extraction [71]

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.

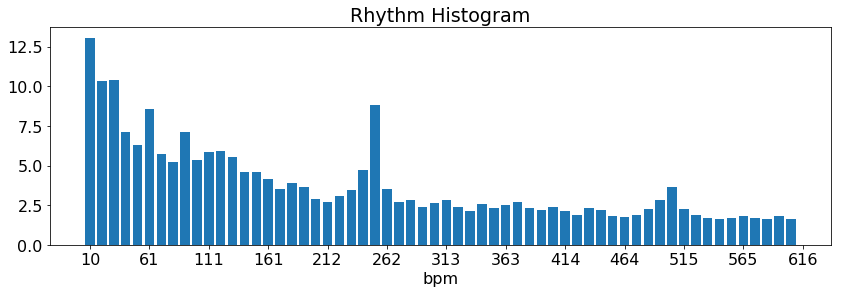
### 3.3.3 Rhythm Histogram

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.



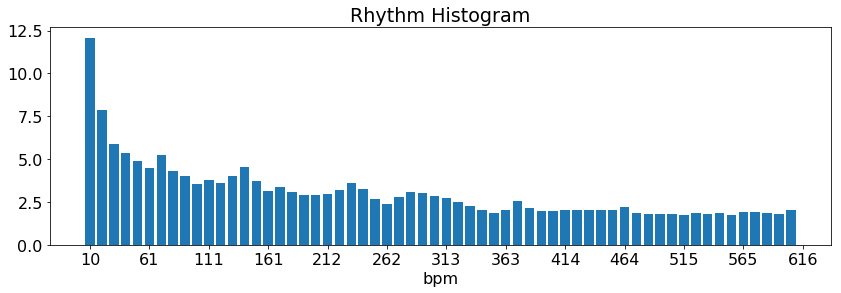
(

a) Scorpions



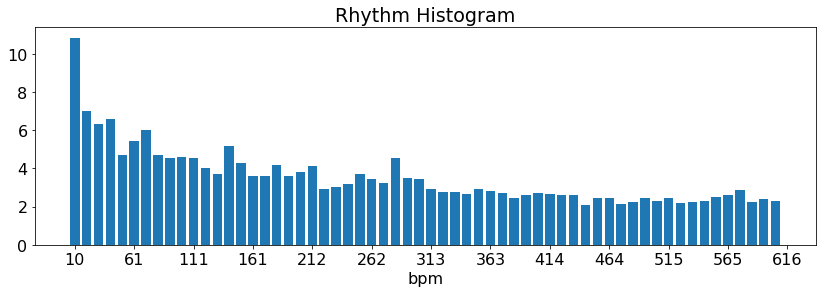
(

b) Knightsbridge



(

c) 94’ Version Stanne



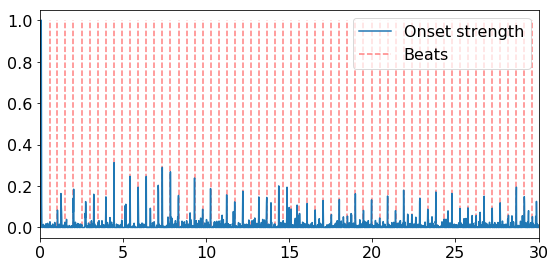
(

d) 99’ Version Stanne

Figure 3.16: Rhythm Histogram

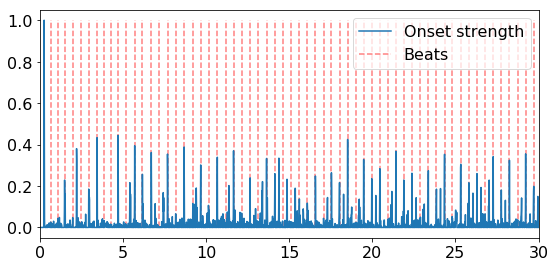
### 3.3.4 cross-correlation

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.



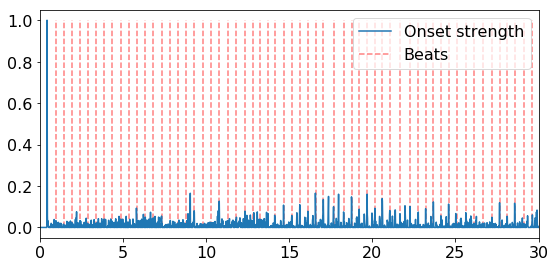
(

a) Scorpions



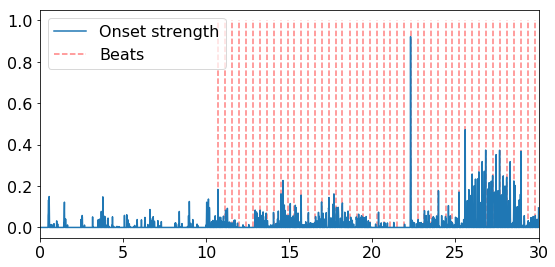
(

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 *b·*(*b*2+1)))
* out.mfcckl (containing mean vector (length *b*) and full covariance matrix (length *b · 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 *b·*(*b*2+1) instead of

*b·*(*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 *× 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) |

1

2

3

4

5

6

7

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) |

1

2

3

4

5

6

7

8

9

10

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 |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

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() |

1

2

3

4

5

6

7

8

9

10

11

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.

#*songs*

*time* =  *·* 15*.*4*s* (4.1)

#*CPUs*

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.

0

20

40

60

80

100

120

0

500

1

*,*

000

1

*,*

500

2

*,*

000

2

*,*

500

3

*,*

000

timeins

essentiaparallel(4cores)

rp

extract(4cores)

essentia

librosa

#songs

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.

|  |
| --- |
| 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]) |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

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).

\*.chroma

\*.rp

\*.bh

\*.mfcckl

\*.mfcc

\*.notes

\*.rh

\*.files

0

2

4

6

8

7

*.*

4

2

*.*

6

0

*.*

32

0

*.*

32

0

*.*

18

0

*.*

17

9

*.*

8

*·*

10

*−*

2

5

*·*

10

*−*

3

filesizeinGB

Size i

n GB

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 × 4) running Spark 2.4.0.

The cluster test were performed on the ARA-cluster, that offers 16 compute-nodes with

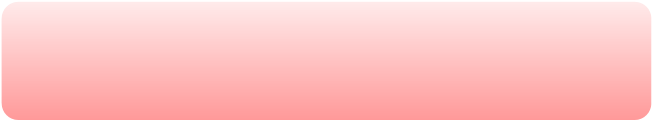
32 CPU-cores (Dual Socket, 2 x Intel Xeon ”Scalable” 6140, 2.30 GHz x 18) per node

(72 with HT) and 192GB of RAM. The cluster was running an older version of Spark

(1.6.0)

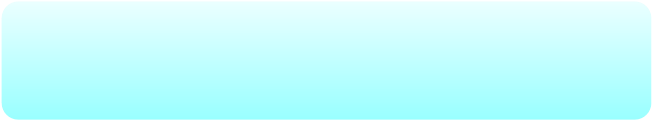
### 4.2.2 Workflow

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.



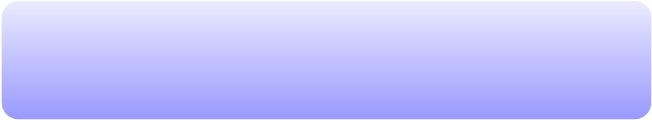
returnresult

5)



4)

joiningresults



3)

distancescaling



2)

distancecomputation



1)

dataprepara

-

tionandcaching

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"]) |

1

2

3

4

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")) |

1

2

3

4

5

6

7

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() |

1

2

3

4

5

6

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"] |

1

2

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)))) |

1

Code Snippet 4.10: euclidean distance RDD

##### Bucketed Random Projection

As an alternative to the euclidean UDF, Spark offers an implementation of a Localitysensitive 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]

*x · v*

*h*(*x*) = (4.2)

*r*

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() |

1

2

3

4

5

6

7

8

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 5.1.2. 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’) |

1

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() |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

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.

|  |
| --- |
| 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() |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

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 5.1.1). As a temporary solution all distances larger than a certain threshold get filtered out.

Secondly when using the fma dataset a few of the songs returned 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() |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

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 5.1.3) 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() |

1

2

3

4

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])) |

1

2

3

4

5

6

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.

|  |
| --- |
| import time *#...* featureDf = preprocess\_features().persist() *#p3* print(featureDf.first()) *#p4* tic1 = int(round(time.time() \* 1000))  neighbors = get\_distances(songname, featureDf).persist() *#p6*  neighbors = neighbors.orderBy(’scaled\_dist’, ascending=True).persist() *#p7* neighbors.show()  neighbors.toPandas().to\_csv("neighbors.csv", encoding=’utf-8’) neighbors.unpersist()  tac1 = int(round(time.time() \* 1000)) time\_dict[’time: ’]= tac1 - tic1  print time\_dict |

1

2

3

4

5

6

7

8

9

10

11

12

13

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.

opt

p6

p7

l4

p3

0

20

40

60

80

100

37

*.*

53

44

*.*

3

1

6

5

*.*

5

7

7

*.*

6

89

*.*

85

timeins

nce

rforma

pe

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.

*0 d − min*(*d*)

*d* = (4.3) *max*(*d*) *− min*(*d*)

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 5.1.1 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 5.1.1. 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 *µf*. Outliers should be detected and removed before calculating the mean distance. A better way to rescale the data could be evaluated in future research.

*0 d*

*d* = (4.4) *µf*

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)"] |

1

2

3

4

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)"] |

1

2

3

4

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.

P*M−*1 *w*

*dist* = *m*=0*M−*1*m · dm* (4.5)

P*m*=0 *wm*

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.

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

Code Snippet 4.21: cluster setup

The cluster configuration in the code example 4.21 turned out to perform well compared to other test configurations. 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.

With the help of the spark.dynamicAllocation parameters the amount of Executors spawned can be determined [54, p. 153]. 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. It should be mentioned that normally spark.shuffle.service.enabled should also be set to true when using the dynamicAllocation and an external Shuffle service should be configured to avoid shuffle data gets lost in case an Executor gets deleted, but during the tests this option was disabled. This shouldn’t be a problem because the dynamicAllocation is mainly used to ensure that a certain minimum amount of Executors is spawned at all and with this configuration no more than 16 Executors can be spawned anyway because of the missing resources on the ARA-cluster, so for this configuration the Executors never actually get killed.

20

40

60

80

100

120

140

160

180

200

220

240

260

12

14

16

18

20

22

24

timeins

songrequestoncachedfeatureDF

#Executor

Figure 4.5: #Executors spawned

The Spark driver programm is executed on the ARA-cluster login-node where also software from clients runs, possibly influencing the results of the performance tests. Fine-tuning the cluster settings is a tricky task. Increasing the number of Executors also increases the additional overhead of managing the Executors and shuffling the data while on the other side more unique tasks can be distributed better over the compute nodes. To get a performant cluster configuration various other cluster settings were tested. Increasing the repartition\_count and the amount of Executors spawned (with fewer ressources each) seemingly increased the overhead and network traffic on the cluster without reducing the overall computation time. Increasing the repartition\_count while keeping the Executors the same size as in the code snippet turned out to be slower as well.

Although each node on the ARA-cluster has 36 CPU cores (without hyper-threading), only 32 cores were assigned to each Executor because this turned out to perform just a little bit better when calculating the similarities for only one song in the first tests. Therefor the cluster configuration was set as described in the code snippet 4.21 for the following tests in this section to keep the tests comparable to each other.

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

Figure 4.5 shows the execution time for one full song request for all features on all

114210 songs, but the features are already cached in one large DataFrame (this approach is explained later on in more detail, see figure 4.9). The x-axis shows the numbers of Executors that are spawned on the cluster. Because there are limited resources on the cluster the amount of CPU cores assigned to each Executor decreases when more Executors get spawned. In total there are 576 cores on 16 nodes, so the number of

CPU cores per Executor can be calculated as #*CPUs* = #*Executors*576 . The available main memory per node (192GB) is split equally. The large DataFrame is cached and split in twice as many parts as Executors are spawned, so every Executor has to handle two data chunks.

##### Differences between the feature types

0

10

20

0

*.*

9

9

0

*.*

5

0

*.*

5

8

1

*.*

1

3

0

*.*

5

6

0

*.*

1

6

3

*.*

1

5

*.*

2

7

23

*.*

38

3

*.*

51

6

*.*

79

9

*.*

53

6

*.*

21

6

*.*

48

10

*.*

85

26

*.*

16

timeins

only

istances

d

scaling

tances +

data

preparat

ion + dis

rp rh bh skl js mfcc notes chroma

Figure 4.6: Performance on different 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 4.6. 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 ARAcluster 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.

0

0

*.*

1

0

*.*

2

0

*.*

3

0

*.*

4

0

*.*

5

0

*.*

6

0

*.*

7

0

*.*

8

0

*.*

9

1

1

*.*

1

1

*.*

2

*·*

10

5

0

20

40

60

timeins

MergedDF

DF

RDD

#songs

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

0

0

*.*

1

0

*.*

2

0

*.*

3

0

*.*

4

0

*.*

5

0

*.*

6

0

*.*

7

0

*.*

8

0

*.*

9

1

1

*.*

1

1

*.*

2

*·*

10

5

0

20

40

60

timeins

MergedDF

DF

RDD

#songs

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

Figure 4.9 shows the adapted workflow (original see figure 4.3) of this approach

data preparation

many DFs (features)

join and repartition

single DF (features)

distance computation

single DF (features) + many DFs (distances)

join results

single DF (features) + single DF (distances)

scale results & combine

single DF (features) + single DF (distances)

sort and return result

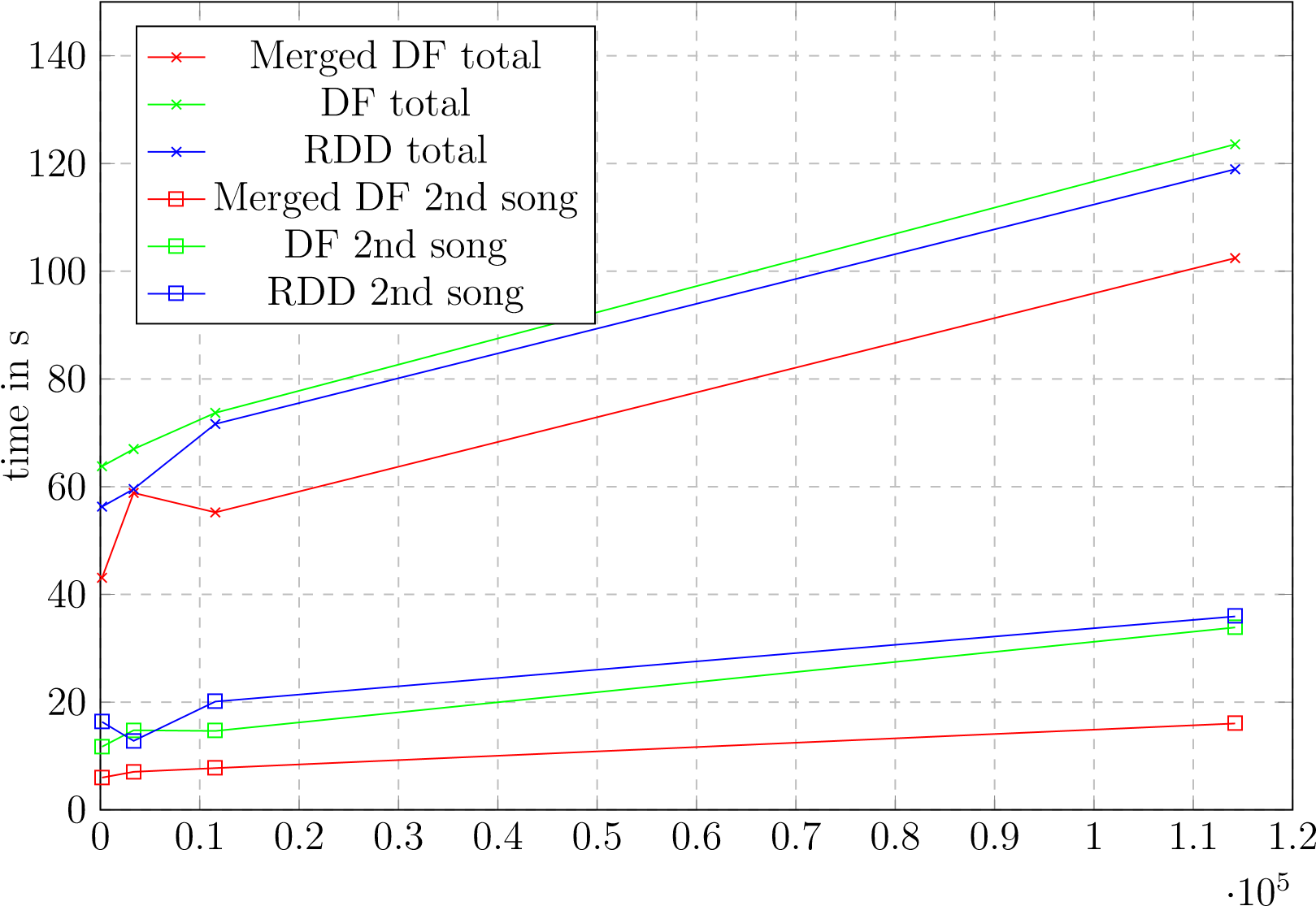
single DF (features)

Figure 4.9: 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.

##### Performance for subsequent song requests

In contrast to the performance analysis from the last section, figure 4.10 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 4.10 shows the according results.



#songs

Figure 4.10: 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 4.9 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.

0

0

*.*

1

0

*.*

2

0

*.*

3

0

*.*

4

0

*.*

5

0

*.*

6

0

*.*

7

0

*.*

8

0

*.*

9

1

1

*.*

1

1

*.*

2

*·*

10

5

0

10

20

30

40

timeins

MergedDF

DF

RDD

Filterchromafirst

Filterbhfirst

#songs

Figure 4.11: Descending importance filter and refine, all features

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 4.11 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. 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 ”Fur 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.

##### Cluster size

The runtime and its dependencies on cluster configuration, size of the input dataset and implementation details was already given in the previous chapter 4.2.7. With about eleven seconds response time for the filter and refine method and 16 seconds for the merged DataFrame approach on 16 compute nodes and for 114000 songs, the result is reasonably fast but not really fast enough for real-time processing.

0

2

4

6

8

10

12

14

16

18

0

50

100

150

200

timeins

Songrequest

#Nodes

Figure 4.12: Performance / Executors (36 CPU cores each)

To simulate the impact of growing cluster sizes in figure 4.12 the cluster configuration was changed from 1 up to 16 Executors spawned, each reserving 36 CPU cores (the available cores on one node) and 64GB (+ 32GB overhead) of main memory. To do this, the parameters of the dynamicAllocation were changed. When setting the minimum Executor count above 16 but there are not enough resources on the cluster, the Spark

Driver only spawns as many as he is able to (16 on the ARA-cluster with 36 CPU cores/

Executor). As the tests algorithm, the merged DataFrame approach (repartitioned in

32 chunks) with two subsequent song requests was chosen and the computation time of the second song request for all feature-sets is shown.

### 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

# 