### 2.4.2 Alternatives

##### Spotify API/ Echonest

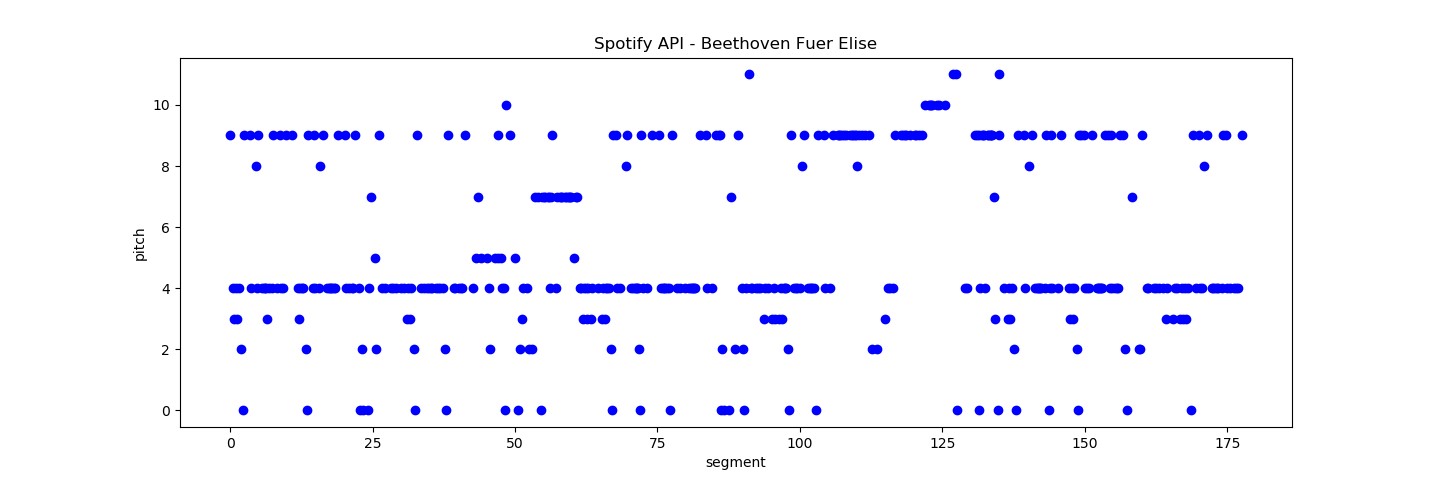
Another way of getting music information, audio analysis and metadata is by using the Spotify API[41] Part of the available audio features comes from the Echo Nest[42].

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 7.2 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 Fur 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.

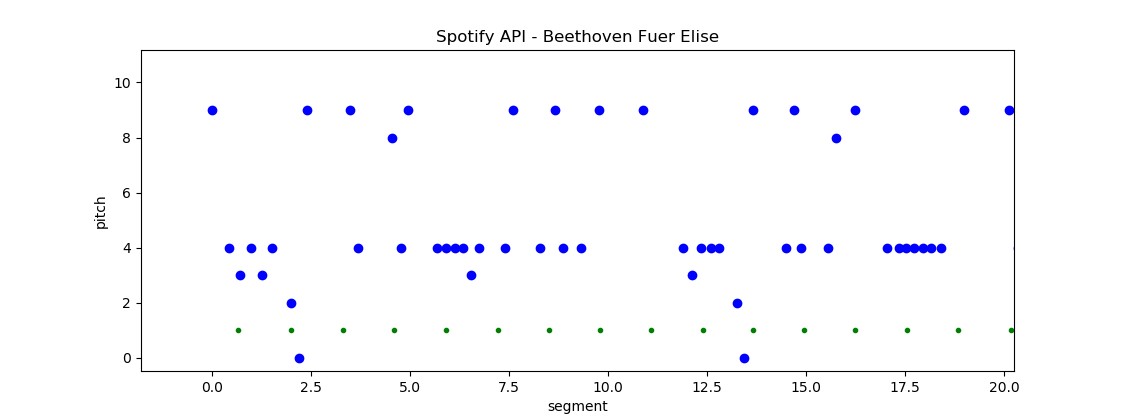


(

a) F

¨

ur Elise Spotify pitch



(

b) F

¨

ur 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]

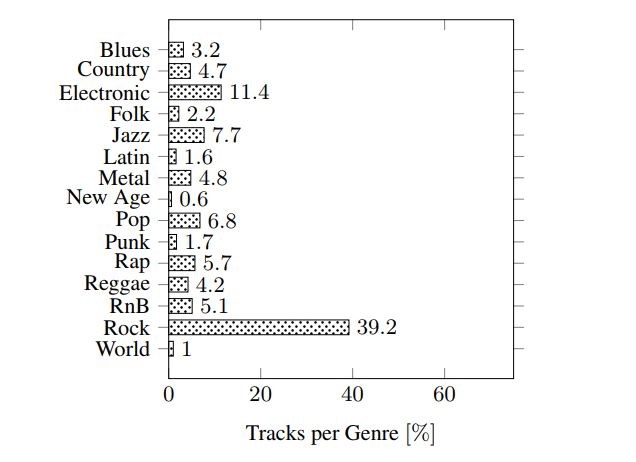


Figure 2.15: million song dataset genre distribution

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.

**Inputdata**

Input

Map

Input

Map

Input

Map

Input

Map

**Results**

Tuples

*h*

*k,v*

*i*

Reduce

Tuples

*h*

*k,v*

*i*

Reduce

Tuples

*h*

*k,v*

*i*

Reduce

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.

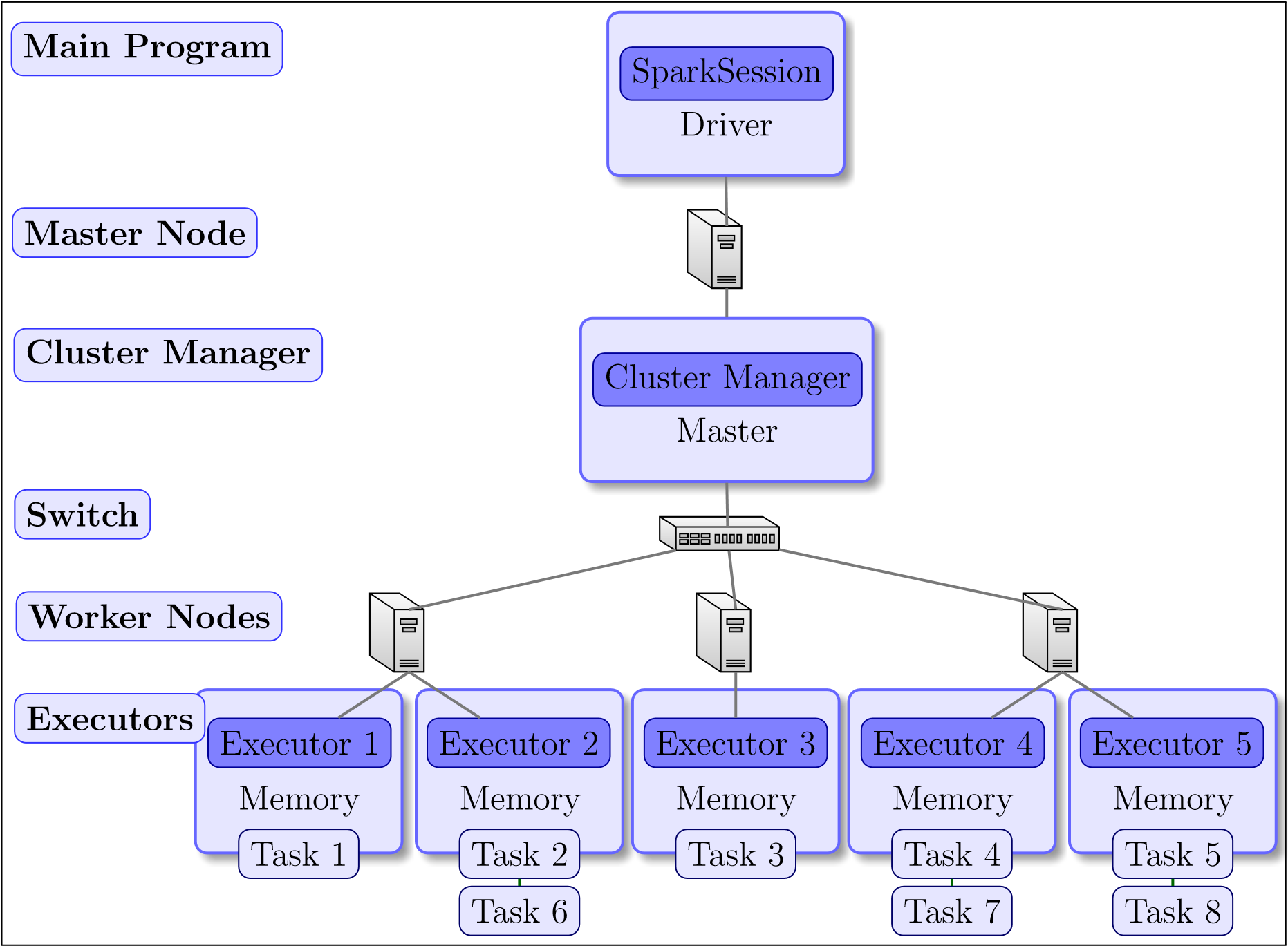
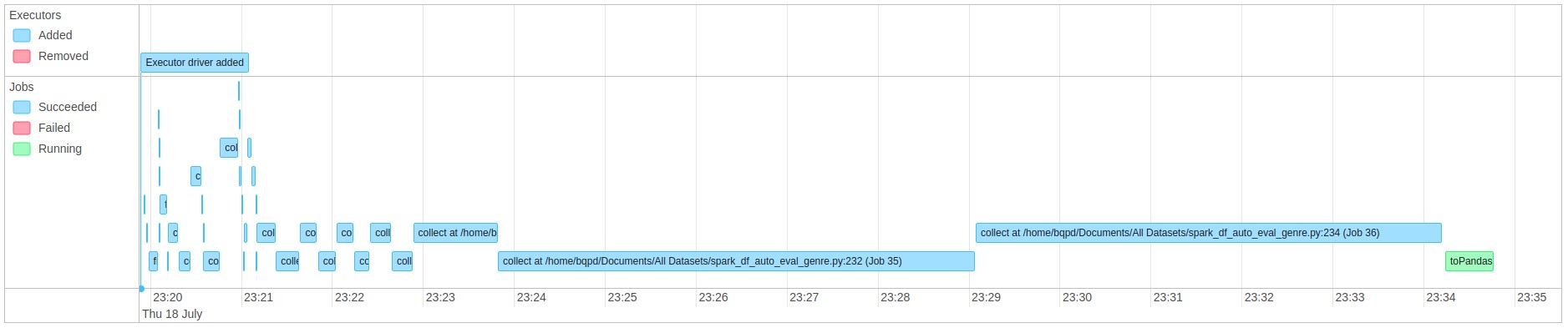


Figure 2.17: Spark Cluster

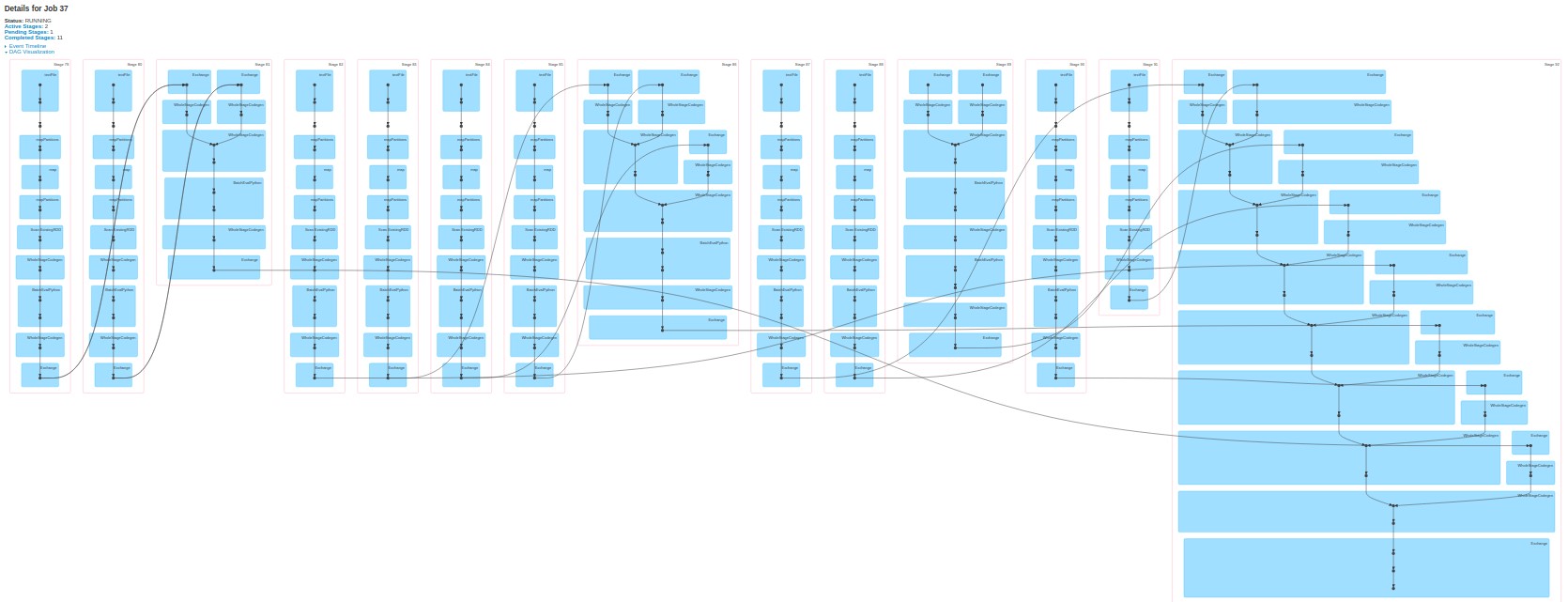
The core components of a Spark application are the Driver, the Master, the Cluster Manager and the Executors. The Driver is the process where the client submit their applications to. It is responsible for the panning and execution of the Spark program and returns status logs and results to the client. 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)

|  |
| --- |
| 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.shuffle.service.enabled", "True") 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() |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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() |

1

2

3

4

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, batchand 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-tomany-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 consecutive song 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.

The similarities are calculated as ”one item to many items” similarities. That means that for only one song at a time the similarities have to be calculated. This is the approach investigated in this thesis. The other option would be to pre-calculate a full similarity matrix (All-pairs similarity) but looking at large-scale datasets with 30 million songs this would take a considerable amount of time. A combination of both approaches would be to calculate the similarities for one song request at a time but store these similarities, to speed up subsequent requests of the same songs, but this wont be the topic of this thesis.

To clarify the usage of a few terms further throughout this thesis (especially later in chapter 4.2), the term ”song request” describes the song title passed to the recommendation engine to estimate the similarities. The terms similarities and distances are used synonymous in this thesis because all the similarity estimations are based on distances between feature vectors of different feature types (see chapter 3). The smaller the distance between two songs gets the higher are the similarities.

# 

# 3. Similarity Analysis

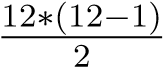
*This needs an introduction, the smaller the distance the more similar the songs are etc. -> smilarity = 1 / distance*

## 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 ( 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:

1 *n* !*p*

*d*(*x,y*) = *||x − y||p* = X*|xi − yi|p* (3.1)

*i*=1

also known as the *Lp* 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 (*µP* and *µQ*) 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*(*·*) being the trace (i.e. the sum of the diagonal of a matrix), *d* being the dimensionality

(number of MFCCs) and *|*Σ*P |* being the determinant of Σ*P*

1 *|*Σ*P | −*1 *T −*1

*KL*(*P||Q*) = [*log* + *tr*(Σ*P* Σ*Q*) + (*µP − µQ*) Σ*P* (*µQ − µP* ) *− d*] (3.2)

##### 2 *|*Σ*Q|*

As a second step the result has to be symmetrized.

*dKL*(*P,Q*) = (*KL*(*P||Q*) + *KL*(*Q||P*)) (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]:

*dSKL*(*P,Q*) = (*tr*(Σ*P* Σ*−Q*1) + *tr*(Σ*Q*Σ*−P*1) + *tr*((Σ*−Q*1Σ*−P*1)(*µP − µ*2)2) *−* 2*d*) (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 *Xm* of the two distributions is defined”[22, p. 43]. ”To use the Jensen-Shannon divergence [...] to estimate similarities between Gaussians, an approximation of *Xm* as a single multivariate Gaussian can be used [...] This approximation of *Xm* is exactly the same as the left-type Kullback-Leibler centroid of the two Gaussian distributions [...]” [22, p. 44]

*µm* = *µP* + *µQ* (3.5)

Σ*m* = (Σ*P* + *µP µTP* ) + (Σ*Q* + *µQµQT* ) *− µmµTm* (3.6)

*JS*(*P,Q*) = *log|*Σ*m| −* *log|*Σ*P | −* *log|*Σ*Q|* (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 *Dx,i*=1*..N* from an object *x* to all other objects in the data set follow a certain probability distribution; thus, any discance *Dx,y* can be reinterpreted as the probability of *y* being the nearest neighbor of *x*, given the distance *Dx,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 > Dx,y*) = 1 *− P*(*X ≤ Dx,y*) = 1 *− F*(*Dx,y*) | (3.8) |
| *MP*(*Dx,y*) = *P*(*X > Dx,y ∩ Y > Dx,y*) | (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 GMMbased 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 good measure for the efficiency of a timbre similarity algorithm in general is the ability to recommend songs of the same genre.

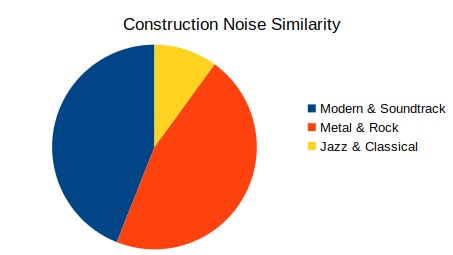
##### Genre recall rate/ construction noise

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 were returned in descending order:

* Ziegenmuhlen 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 (Orph´ee aux enfers) -

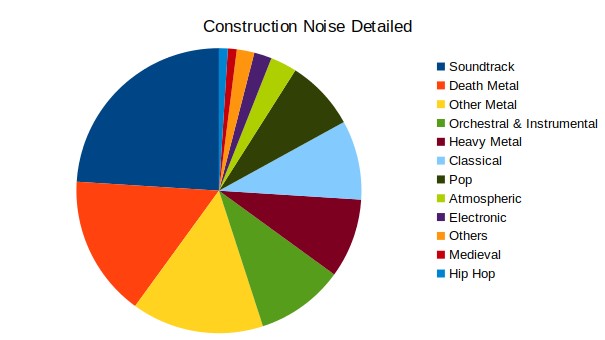
Can-Can (Live At Grosser Saal, Musikverein) (Klassik)

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 - Evocacio´n
* Mozart Sonate Nr. 11 in A-Dur, K. 33 - Mozart - Alla Turca Allegretto (3:28)
* 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
* Mozart Sonate Nr. 11 in A-Dur, K. 33 - III. Tuerkischer Marsch (3:30)
* 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 recommendations are however all from the same genre of classical music. The inability to detect cover versions was also observable for other songs in the dataset like Serj Tankians song ”Lie Lie Lie” from the CD ”Harakiri” 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.

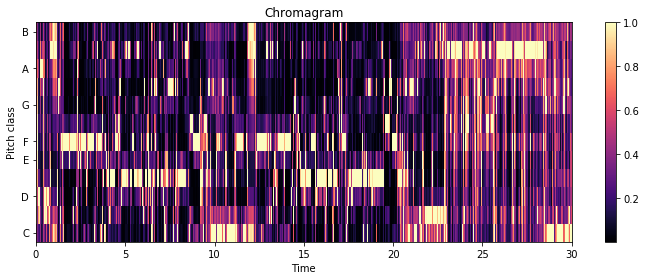
## 3.2 Melodic Similarity

### 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.

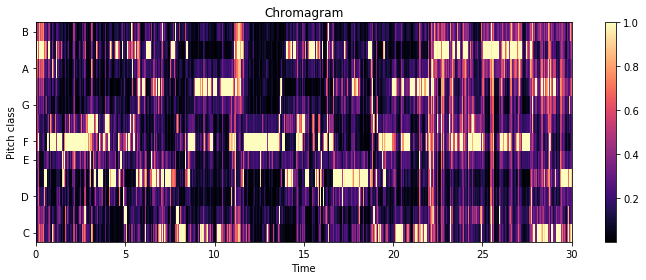
### 3.2.2 Chroma Features pre-processing

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

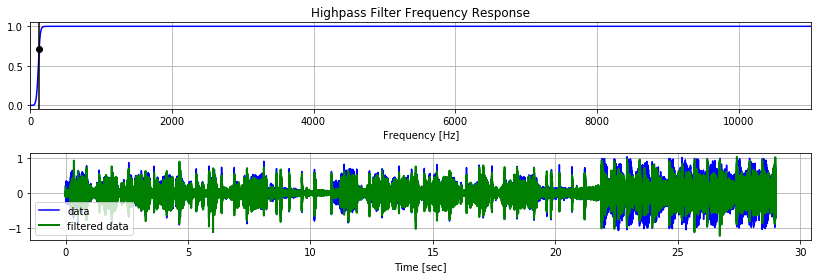


(

b) Chroma Features Pvris - Chandelier

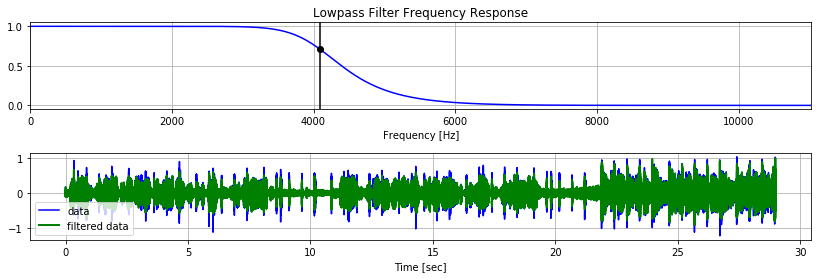
Figure 3.2: Chroma Features

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



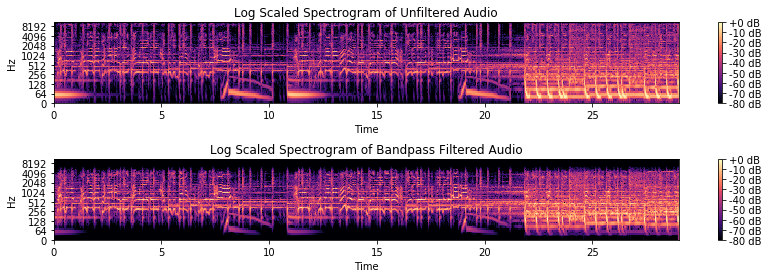
(

a) Highpassfilter



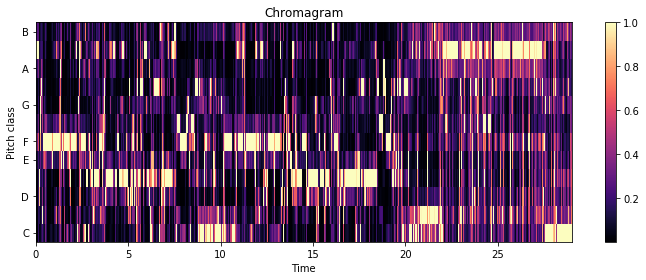
(

b) Lowpassfilter



(

c) FFT Bandpassfilter Sia



(

d) Bandpass Filtered Chromagram

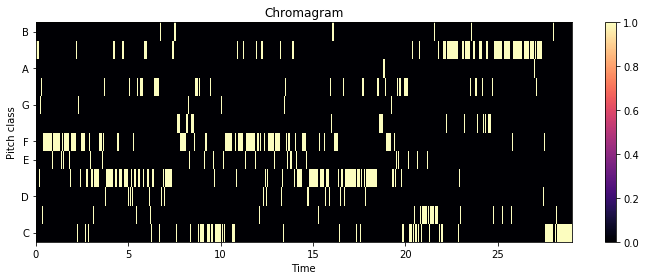
Figure 3.3: Bandpass - Sia

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

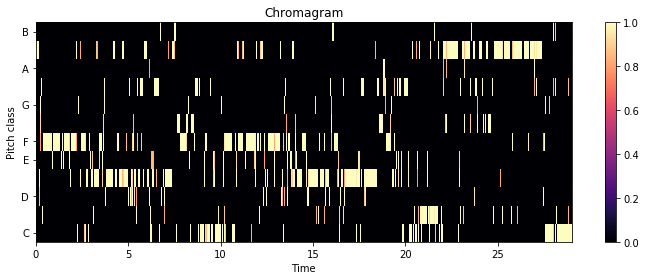
0.8.

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

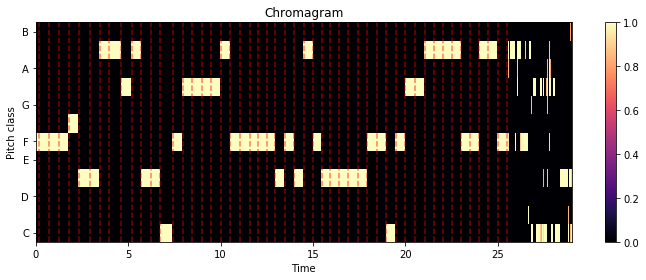


(

b) First two most dominant notes

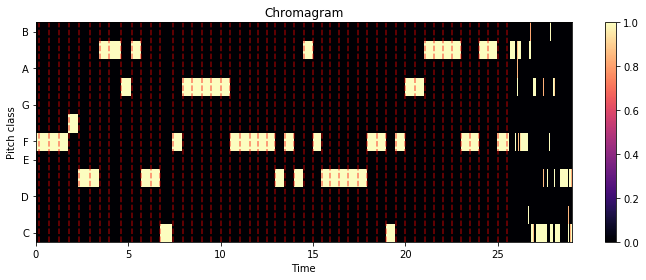
Figure 3.4: Thresholded Chroma Features - Sia

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



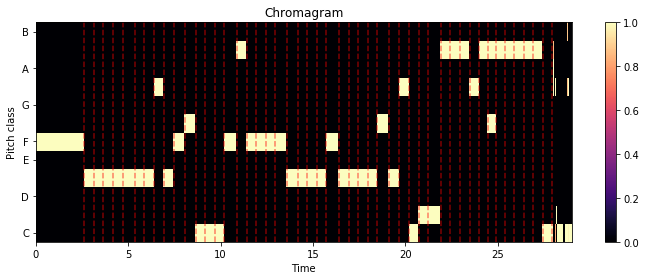
(

a) Beat aligned of unfiltered audio - Pvris



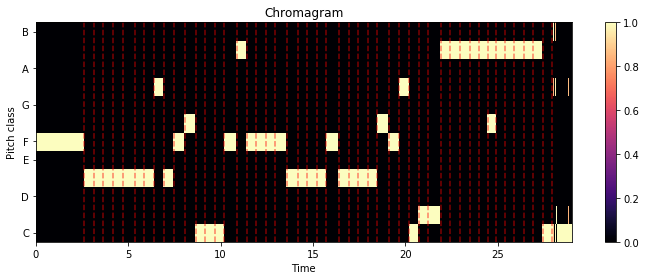
(

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.

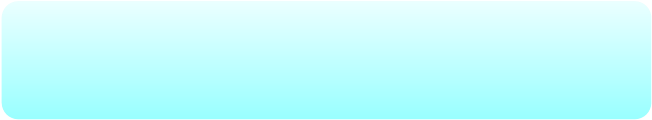
The whole workflow to extract the chroma features for this thesis is shown in figure

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



keyshifting

6)



beatalignment

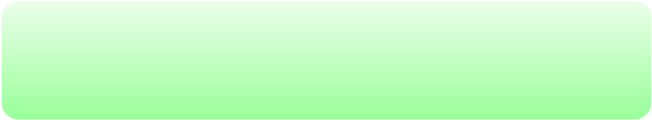
5)



4)

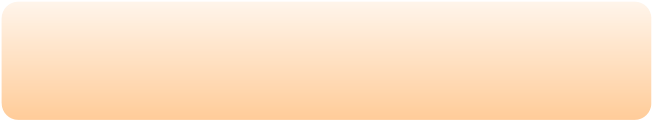
extractmost

dominantpitches



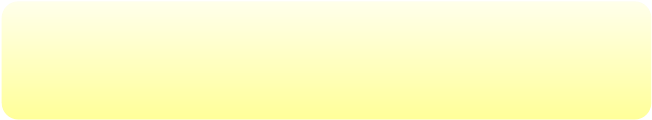
3)

detectsilence



calculatechromagram

2)



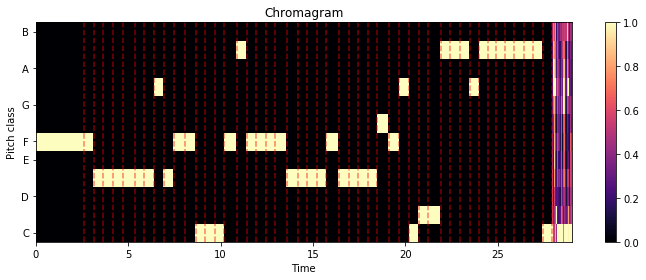
1)

filteraudio(bandpass

)

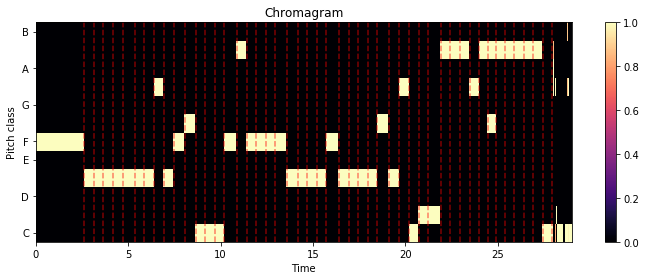
Figure 3.6: Workflow chroma feature extraction

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]



*max*(*i,jlev*)*,S,T* (*i −* 1*,j*) + 1 if *min*(*i,j*) = 0

 

*levS,T* (*i,j*) = *levS,T* (*i,j −* 1) + 1 (3.10)

*min*+*levcostS,T* ([*Si i−6*=1*,jTj*]*−* 1) else

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.