Music classification using Constant-Q based Features

Lena Sophie Brüder

January 4, 2013

Master Thesis
Ruhr-University Bochum

Written in cooperation with
Research In Motion Deutschland GmbH

Advised by
Prof. Dr. R. Martin and Dr. Wolfgang Theimer

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Bochum, January 4, 2013

Lena Brüder

Abstract

In this thesis an approach to music similarity classification using exclusively the audio signal is derived, which helps users to explore their music collections. It differs from classically used approaches in certain aspects, e.g. that users can create personal music categories aside from the classical genres.

To extract information and derive features, the *Constant Q transform* has been used, which is a transform to the frequency domain with geometrically spaced frequency bins. Timbre, chroma, dynamic range and the length of the piece are used as *complementary* features, which has been rarely used in the past. Gaussian Mixture Models are the main method to represent the signal distribution, ranging from feature extraction to classification. A soft ranking is derived, which is better suited to explore a music collection for good matches than a hard classification.

The system is suited for creation of personal categories and can generate robust categories using only a few (5-15) examples, which can be both positive and negative. Negative examples may be omitted. The implementation is able to run on both embedded devices and regular computers.

Zusammenfassung

In dieser Masterarbeit wird ein System für Musikklassifikation vorgestellt, das ausschließlich mit dem Musiksignal arbeitet. Das System soll Benutzern dabei helfen, ihre Musiksammlung zu durchsuchen. Das System unterscheidet sich in einigen wesentlichen Merkmalen von klassischen Ansätzen, die bisher in der Literatur vorgestellt wurden; zum Beispiel kann der Benutzer persönliche Musikkategorien definieren, die nicht auf die sonst verwendeten Genres beschränkt sind.

Zur Extraktion von Merkmalen und Informationen wurde die *Constant Q Transformation* verwendet. Sie ist eine Frequenztransformation, deren Frequenzbins im geometrischen Abstand angeordnet sind. Klangfarbe, Chroma, Dynamikumfang und die Länge des Stücks werden zugleich als sich ergänzende Merkmale benutzt, was in der Vergangenheit nur selten der Fall war. Gauß-Mix-Modelle stellen einen wesentlichen Baustein sowohl für die Merkmalsextraktion, als auch für die Klassifizierung dar. Statt einer harten Kategorienzuordnung erstellt das System eine Rangfolge der Zugehörigkeit zu einer Kategorie mit allen Stücken der Datenbank.

Das vorgestellte System ist geeignet, persönliche Benutzerkategorien aufgrund von wenigen (5-15) Beispielen zu erzeugen. Beispiele können sowohl positiv, als auch negativ sein; auf negative Beispiele kann dabei verzichtet werden. Die Implementierung ist sowohl auf eingebetteten Geräten, wie auch auf regulären Computern lauffähig.

Contents

1 Introduction

This chapter gives a brief introduction to the goals and the basic structure of the thesis.

1.1 Goals and problem description

Since centuries, humans have created music and share their love for it. In the past century, techniques have been developed to record music on physical media. In the past two decades, the invention of digital audio compression algorithms have led to a change from music libraries on physical media with limited storage capacity to digital libraries on single harddrives. It has never been that easy to have all your music in a small form factor, even in your pocket, and not only since many people are sharing their music with their friends, large music databases are now common. With large databases, problems arise that have been unknown in the past: Music databases get larger than people can handle. It is not easy to keep track of such large databases, and some songs might get lost because people miss the "forest for the trees".

The main goal of this thesis is to develop a software tool that helps users to explore their music databases. The idea is to let a user define categories of music based on a collection of songs they like. The software tool should then be able to give ratings to all other music pieces in the database, such that the user will be able to find music that is similar to the music he used to define the category. That way, users can keep track of the more important parts of their music collections. The system should run on mobile devices.

Many existing music recommendation systems are based on community intelligence and rely on the tags of a piece of music. Popular examples for tag- and community-based systems are `last.fm` and `iTunes Genius`[1] from Apple; other systems use experts to create tags for music, such as `pandora.com`. A goal of this thesis is to perform a similar task exclusively using the audio signal of the music – metadata and tags will not be used.

1.2 Basic structure of a signal-based music classificator

In the literature, many examples for signal-based music classification systems can be found (e.g. [TC02], and a survey of different systems in [FTZ11]). Although they differ

[1]see `http://www.apple.com/legal/itunes/de/genius.html`

in many aspects, a basic structure can be derived that fits to most classification systems (see figure 1.1).

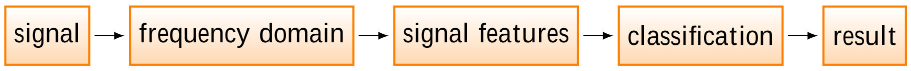


Figure 1.1: Basic structure of a frequency domain-based music classification system

The audio signal is first transformed to the *frequency domain*[2], which means that the signal will be split into its frequency components. This step makes it possible to take a look at the different parts the signal consists of. Secondly, *features* are extracted from the frequency domain representation. These features represent properties of the signal that can be used to describe it. For some of these features, it is not necessary to transform the signal to the frequency domain (e.g. the *zero crossing rate*), but for most features this is the case.

The third step is about *classification*. The signal features are examined for similarities or clusters by machine learning algorithms. This step leads to the results of the classificator, which will be presented to the user, e.g. as a list of best matches.

This thesis will use a *constant Q transform* for the first step and derive all signal features from this intermediate representation.

1.3 Structure of the thesis

The structure of this thesis is derived from the basic structure of the music classification systems found in the literature. In the first part, mathematical prequisites and the transforms, such as the constant Q transform, will be introduced. The second part is about the features and their calculation, the third part outlines the classification algorithms.

In the fourth part, implementation details will be discussed, such as the design of performance-critical modules, the database layout, the interplay of the software modules and some basic software testing strategies. The last part is about the results of the thesis and the application of the resulting software to real-world data.

[2] For some features the time domain or the phase domain are used, but for most features, the frequency domain is used (see [FTZ11]).

2 Terminology, mathematical terms and mathematical prequisites

This chapter gives an introduction to the terminology and the mathematical terms used. It also presents the mathematical prequisites and some algorithms which are needed to understand the concepts which will be developed in the following chapters.

2.1 Mathematical terms

Here, the mathematical terms and symbols will be introduced. The goal is to define an unambiguous mathematical language.

2.1.1 Symbols

Symbol	**Label**	**Description**
x	scalar	a scalar value, as in $x = 2$
$\boldsymbol{v}$	vector	a column vector, as in $\boldsymbol{v} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$
$\boldsymbol{M}$	matrix	a matrix, as in $\boldsymbol{M} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. The coefficients are noted as a_{ij} or $(\boldsymbol{A})_{ij}$.
$\mathbb{R}$	field of real numbers	the field of real numbers, i.e. $\pi \in \mathbb{R}$
$\mathbb{C}$	field of complex numbers	the field of complex numbers, i.e. $3 + 2\mathrm{j} \in \mathbb{C}$
$\mathbb{F}$	field of real or complex numbers	$\mathbb{F}$ means any of $\mathbb{R}$ or $\mathbb{C}$.
j	imaginary unit	the imaginary unit $\mathrm{j} \in \mathbb{C}$ with the property $\sqrt{-1} = \mathrm{j}$.
a^*	complex conjugate	the complex conjugate of $a \in \mathbb{C}$
$\boldsymbol{A}^*$	complex conjugate	the complex conjugate of $\boldsymbol{A} \in \mathbb{F}^{n \times m}$, which is $\boldsymbol{A}^* = \overline{\boldsymbol{A}}^T$
$f \in \mathcal{O}(g)$	algorithmic complexity	f is bounded by g asymptotically (up to a constant factor)

$L^p(\mathbb{F}^n)$	Lebesgue space in $\mathbb{F}^n$ with p-norm	A vector space over $\mathbb{F}^n$ with Lebesgue-integrable functions and a p-norm
$\lfloor \cdot \rfloor$	floor function	rounding towards $-\infty$
$\lceil \cdot \rceil$	ceiling function	rounding towards $+\infty$
$\langle a, b\rangle$	scalar product	The scalar product of a and b, with a and b being objects such as vectors, scalars, functions, ...
$P(x)$, $p(x)$	probability density function	small letters denote probability density functions on continuous sets, large letters denote probability density functions on countable sets

2.1.2 Runtime Analysis

A runtime analysis is an analysis of the theoretical execution time of an algorithm. The runtime analysis is performed on a hypothetical device which has certain properties: It is assumed that we use a fixed number of bits for the calculation, which is usually below 64 bits for integers and floating point numbers. Exceptions will be marked. The important fact here is that there exists an upper bound for the number of bits used in a calculation. Single multiplications and additions have a complexity of $\mathcal{O}(1)$ in this model.

There are numerous mathematical expressions or functions whose runtime depends on the number of bits involved. Examples are $\sin(x)$, $\cos(x)$, $\exp(x)$ and $\log(x)$. All of them have a complexity of at most $\mathcal{O}(\sqrt{n})$ (see [BZ10]), where n stands for the number of bits. The run time does not depend on the argument. Since n is fixed, we can assume that these basic functions run in constant time, $\mathcal{O}(1)$. However, we have to keep in mind that some of these basic functions still can be costly, having a large proportionality factor hidden in the $\mathcal{O}$-notation.

2.1.3 Measurement units

Most units are given in the *International System of Units* (SI). Exceptions are prefixes for sizes in bytes, where the prefixes as proposed by the IEC (IEC 60027-2, revised in year 2000) are used. The SI-prefixes k, M and G intentionally are reserved for powers of 10. In the computer industry, they are more oftenly are used in the context of powers of 2. These prefixes will not be used in this context, in order to avoid inaccuracies.

2.2 Fourier transform

The Fourier transform is widely used in digital signal processing (DSP). The continuous Fourier transform is an integral transform that changes the view on a function: If the

Unit	**Meaning**
1 KiB	1024 bytes
1 MiB	1024 KiB
1 GiB	1024 MiB

Table 2.2: Units for sizes in bytes

original function is a function of time, then the Fourier transform of that function is a function of frequency. The Fourier transform performs a switch from the time to the frequency domain.

Definition 2.1 (*Continuous Fourier transform*):
Let $f \in L^1(\mathbb{R}^n)$ particularly integrable. Then $\mathcal{F}(f)$ with

$$\mathcal{F}(f)(\boldsymbol{t}) = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\mathbb{R}^n} f(\boldsymbol{x}) e^{-\mathrm{j}\boldsymbol{t}^T\boldsymbol{x}} \mathrm{d}\boldsymbol{x} \tag{2.1}$$

is the *continuous Fourier transform* of f. For the special case of $n = 1$, it is

$$\mathcal{F}(f)(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-\mathrm{j}tx} \mathrm{d}x \tag{2.2}$$

The continuous Fourier transform is continuous in time t and in value x.

For audio applications, one is interested in the one-dimensional Fourier transform, but there are other cases, such as image processing, where higher-dimensional Fourier transforms can be useful. The main problem with the continuous Fourier transform is that it is a transform of a *continuous* function. That prevents it from being used in many signal processing applications, since with digital computers, it is only possible to perform calculations on time- and value-discrete series of values. To overcome this shortcoming, the Fourier series will be first defined and later be computed with discrete values.

Definition 2.2 (*p-periodical function*):
Let $f : \mathbb{R} \to \mathbb{R}$ be a function and $p \in \mathbb{R}$. The function f is called *p-periodical* iff $f(x) = f(x + p)$ for all $x \in \mathbb{R}$.

One can show that the set of p-periodical functions over $\mathbb{R}$ is a vector space, and that $\{p | p \in \mathbb{R}, f \text{ is } p\text{-periodical}\}$ is a subgroup of the additive group $(\mathbb{R}, +)$ (see [Beh04, p.347]), so all linear combinations of p-periodical functions are p-periodical functions, too.

Examples for p-periodical functions are $\sin(x), \cos(x), \tan(x)$, which are 2π-periodical, or all constant functions over $\mathbb{R}$. They are p-periodical for all $p \in \mathbb{R}$.

Definition 2.3 (*Fourier series of a p-periodical function*):
Let $f \in L^2([-\frac{p}{2}, \frac{p}{2}])$, $f : \mathbb{R} \to \mathbb{R}$ be an integrable, p-periodical function. Then

$$f(x) = \sum_{n \in \mathbb{Z}} c_n e^{\mathrm{j}\frac{2\pi}{p}nx} \tag{2.3}$$

is the *Fourier series* of f.

It is possible to show that every function can be expressed as a linear combination of $e^{\mathrm{j}\frac{2\pi}{p}nx}$ and that the $e^{\mathrm{j}\frac{2\pi}{p}nx}$ are orthogonal (i.e. $\left\langle e^{\mathrm{j}\frac{2\pi}{p}nx}, e^{\mathrm{j}\frac{2\pi}{p}mx} \right\rangle = 0$ for all $n, m \in \mathbb{Z}, n \neq m$), so they form an orthogonal basis of the space of all p-periodical functions.

Remark 2.4:
This definition is equivalent to

$$f(x) = \sum_{n \in \mathbb{Z}} c_n \left(\cos\left(\frac{2\pi}{p} nx \right) + \mathrm{j} \sin\left(\frac{2\pi}{p} nx \right) \right) \tag{2.4}$$

due to Euler's formula $e^{\mathrm{j}x} = \cos(x) + \mathrm{j}\sin(x)$.

Theorem 2.5:
The coefficients in equation (2.3) are given by

$$c_n = \frac{1}{p} \int_{-\frac{p}{2}}^{\frac{p}{2}} f(x) e^{-\mathrm{j}\frac{2\pi}{p}nx} \, \mathrm{d}\, x \tag{2.5}$$

A proof of theorem 2.5 will not be given. With the Fourier series, it is only possible to approximate periodic functions. Note that in the Fourier series, only integer multiples of a fundamental frequency are being used to approximate the function (i.e. the frequency bins are linearly spaced); the series is frequency-discrete, but not time-discrete. The transform is exact as long as $n \to \infty$. This cannot be fulfilled on computers since we do not have infinite time and memory. Choosing a maximal value for n and discretizing it in the time domain leads to the discrete Fourier transform.

2.2.1 Discrete Fourier transform

From now on, the function $f(x)$ will be replaced by a series $x(n)$. The series will be N-periodical analogous to definition 2.2 with period length $p = N \in \mathbb{N} \setminus \{0\}$. Every occurence of x will be replaced by n. The c_n will be called $X(k)$ and the sums will be finite and limited by N elements. The details can be found in [OS95, p.624]. For a more mathematical explanation take a look at [Sto05, pp.79].

Definition 2.6 (*Fourier series of an N-periodical series*):
Let $x(n)$ be an N-periodical series. Then

$$x(n) = \sum_{k=0}^{N-1} X(k) e^{\mathrm{j}\frac{2\pi}{N}nk} \tag{2.6}$$

is the *Fourier series* of $x(n)$.

Theorem 2.7:
The coefficients in equation (2.6) are given by

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn} \tag{2.7}$$

For a proof see [OS95, p.625]. With this definition, the Fourier transform is used in DSP applications: Both the time and frequency domain are discretized. A naïve algorithm using these equations runs in $\mathcal{O}(N^2)$, but there exists an algorithm for its fast calculation, the Fast Fourier Transform (FFT), which runs in $\Theta(n \lg N)$. It can be used on sequences with a length which is a power of 2, but it is possible to pad a shorter sequence with zeros, so it is not a real shortcoming of the algorithm.

2.2.2 Discrete Cosine transform

The *Discrete Cosine transform* (DCT) is related to the Discrete Fourier transform. Both are basis transformations in the vector space of all p-periodical continuous functions. Both can be seen as time-frequency-transforms. The Fourier transform uses sine and cosine functions as basis functions (see remark 2.4), the Cosine transform uses only cosine functions for this. The derivation of the Discrete Cosine transform is similar to the Discrete Fourier transform, so the derivation is omitted.

Definition 2.8 (*Discrete Cosine transform*):
Let $x(n)$ be a series that is even[1] around -0.5 and even around $N-0.5$. Then the Discrete Cosine transform is defined through the transform

$$X(k) = \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right). \tag{2.8}$$

[1] An function $f: D \to I$ is *even* around $a \in D$ if $\wedge_{x\in D} f(x-a) = f(-x-a)$.

The DCT is often used for its decorrelating properties. For more details on the DCT, see [PTVF07, pp.624].

2.3 Constant Q transform

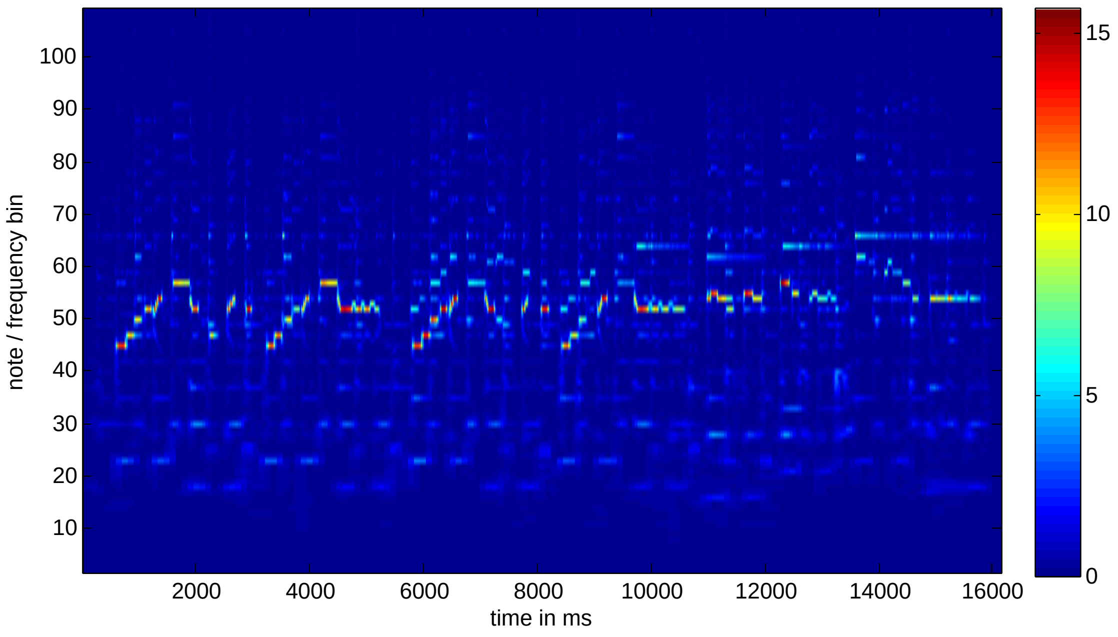


Figure 2.1: The magnitudes of a Constant Q transformed audio sample (Excerpt from *Quand je serai grand* by *David Löhstana* (licensed as `cc-by-3.0`), `test.mp3` from the `testdata/`-folder of `libmusic`). The lowest note 0 is an `f` (21.83 Hz), the highest note 108 is an `e` (10548.08 Hz). The transform shows nine octaves.

The *Constant Q transform* (CQT) refers to a technique that transforms a signal from the time domain $x(n)$ to the frequency domain, but in contrast to the Fourier transform, the center frequencies of the frequency-bins are geometrically spaced and their Q-factors are all equal (see [SK10], figure 2.2). For an example of a Constant Q transformed signal, see figure 2.1.

These properties make it better suited for the analysis of music than the Fourier transform, as it fits to the scale of western music. With the Fourier transform, the frequency bins would be linearly spaced, which is not well-suited for examination of musical data: Low frequencies would have a resolution much lower than needed, and

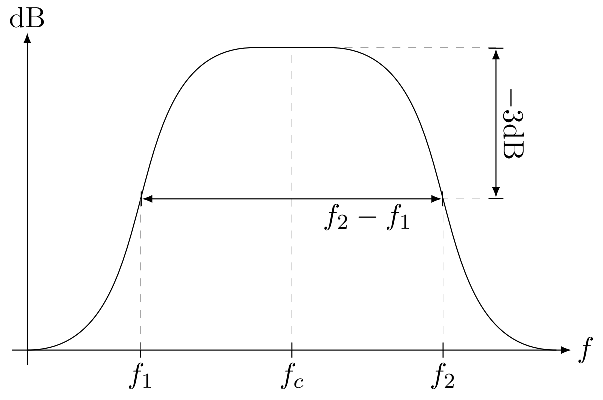


Figure 2.2: The Q-factor of a signal is defined as $Q = \frac{f_c}{f_2 - f_1}$. $f_2 - f_1$ is the bandwidth of the signal.

high frequencies would be overrepresented. Nevertheless, the FFT will be used to speed up the calculation of the Constant Q transform.

Definition 2.9 (*Constant Q transform*):
Let $x(n)$ be a discrete time-domain signal, f_k the center frequency of bin k and B the frequency bin count per octave. $N_k \in \mathbb{R}$ is the window length of bin k and is inversely proportional to f_k to keep the Q factor constant (see [SK10, p.2] for details). f_s is the sampling rate and $w : \mathbb{R} \to \mathbb{R}, t \mapsto w(t)$ a continuous window function with $t = 0$ for $t \notin [0, 1]$.

$$a_k(n) = \frac{1}{N_k} w\left(\frac{n}{N_k}\right) \exp\left(-\mathrm{j}2\pi n \frac{f_k}{f_s}\right) \tag{2.9}$$

are *complex basis functions*, *time frequency atoms* or *temporal kernels*. Then

$$X^{\mathrm{CQ}}(k, n) = \sum_{l=n-\lfloor \frac{N_k}{2} \rfloor}^{n+\lfloor \frac{N_k}{2} \rfloor} x(l) a_k^*\left(l - n + \frac{N_k}{2}\right) \tag{2.10}$$

is the *Constant Q transform* of $x(n)$. The center frequencies f_k are defined as

$$f_k = f_1 2^{\frac{k-1}{B}}. \tag{2.11}$$

In equation 2.10, k can be thought of being a musical note if $B = 12$. $X^{\mathrm{CQ}}(k, n)$ having a large value at time n and note k means that the frequency of note k is prominent near sample n.

Remark 2.10 (*Center frequencies*):

The center frequencies of this Constant Q transform refer to musical notes in *equal temperament*, which means exactly that the notes are defined through equation 2.11 with $B = 12$.

There are other musical temperaments, i.e. *just intonation*, in which the frequencies of the notes are defined through integer ratios between each other with small numbers. These intonations have problems with transposition, therefore most western music uses equal temperament.

Just intonation is better suited for having exact bins for overtones, since with equal temperament, the overtones do not exactly match to one bin. This is useful for the examination of *timbre* of music, where the overtones are the describing elements.

To use this transform and adjust it to musical pieces in just intonation, the center frequencies could be adjusted to not be geometrically spaced.

$\lrcorner$

In [SK10], a Blackman-Harris window is used for $w(t)$, which is defined as

$$a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right) \tag{2.12}$$

with $a_0 = 0.35875$, $a_1 = 0.48829$, $a_2 = 0.14128$ and $a_3 = 0.01168$ (see [Har78, p.65]). This window function is used here, too.

It is possible to construct an algorithm that calculates the Constant Q transform directly through equation 2.10. This algorithm will be called the *naïve approach*. It evaluates equation 2.10 for every k and every n.

$\ulcorner$**Run Time Analysis 2.11 (*naïve approach*):**
We will assume that the data is zero-padded in the beginning and in the end:

$$x(n) = 0 \text{ for all } x \notin [0, N].$$

First, we will have a look at equation 2.9. The exponential can be calculated in $\mathcal{O}(1)$[2], and $w(\frac{n}{N_k})$ as a composition of cosines as well, so the computational complexity of a_k is $\mathcal{O}(1)$. However, keep in mind that calculating $w(n)$ or $\exp(x)$ might nevertheless take a fairly long constant time, compared to other constant-time operations.

Evaluating the sum in equation 2.10 is possible in $\mathcal{O}(N_k)$, since every summand can be calculated in $\mathcal{O}(1)$. Let $N_k^{\max} = \max(N_k)$, then $a_k(n) \in \mathcal{O}(N_k^{\max})$ for all k.

Let K_{oct} be the number of octaves we are interested in, and B the number of bins per octave, as in definition 2.9. Then we altogether have $B \cdot K_{\text{oct}}$ bins over all octaves, so to get all bins of $X^{\text{CQ}}(k, n)$ for one sample n, we have a computational complexity of $\mathcal{O}(B \cdot K_{\text{oct}} \cdot N_k^{\max})$.

[2] see p.4 for details on the computational complexity of some elementary functions

Since all samples are calculated independent from each other, the total computational complexity is $\mathcal{O}(B \cdot K_{\text{oct}} \cdot N_k^{\max} \cdot N)$, with N being the sample count.

$\square$

Normally, one would not calculate the Constant Q transform for every sample, but it still is clear that this approach is not affordable in an embedded environment. In the next section, a more efficient algorithm will be presented.

2.3.1 Efficient calculation of the Constant Q transform

The efficient algorithm is outlined in [BP92] and [SK10].
The main ideas of the papers are:

- It is possible to switch from the time to the frequency domain, and carry out the calculations there.
- In the frequency domain, the transformed temporal kernels (called *spectral kernels*) are zero most of the time, as we will see later on. Near-zero values can be omitted while still achieving numerically good results.
- It is possible to apply the transform to a single octave, and get all the lower octaves via changing the pitch through low-pass-filtering and subsampling
- Applying the transform to one octave can be carried out quite fast, using the sparsity of the spectral kernels and the Fast Fourier transform.

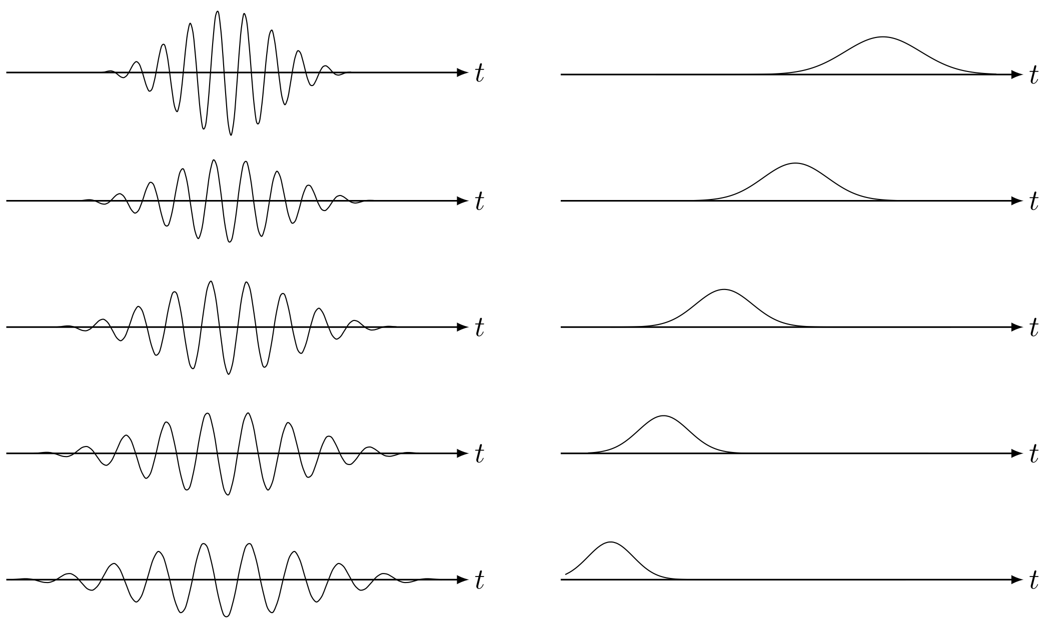
Using these ideas, the calculation can be performed more efficiently. In the following, these ideas will be approached one after another.
Switching from the time to the frequency domain is possible with the identity

$$\sum_{n=0}^{N-1} x(n)a^*(n) = \sum_{i=0}^{N-1} X(i)A^*(i) \tag{2.13}$$

where $X(i)$ is the DFT of $x(n)$ and $A(i)$ is the DFT of $a(n)$. Equation 2.13 holds for any discrete signal and is not restricted to this case. Using equation 2.13, [SK10] rewrite equation 2.10 and get

$$X^{\text{CQ}}(k, \frac{N}{2}) = \sum_{i=0}^{N-1} X(i)A_k^*(i) \tag{2.14}$$

The **spectral kernels $A_k^*(i)$ are sparse**, since they are built as the transformation of modulated sinusoidal functions in the time domain. Most of the values are small compared to the maximum of the function, and there is only one peak in the whole



(a) temporal kernels (b) spectral kernels

Figure 2.3: Some temporal kernels with their corresponding spectral kernels, from highest to lowest frequency. Note that the spectral kernels do all not have the same length.

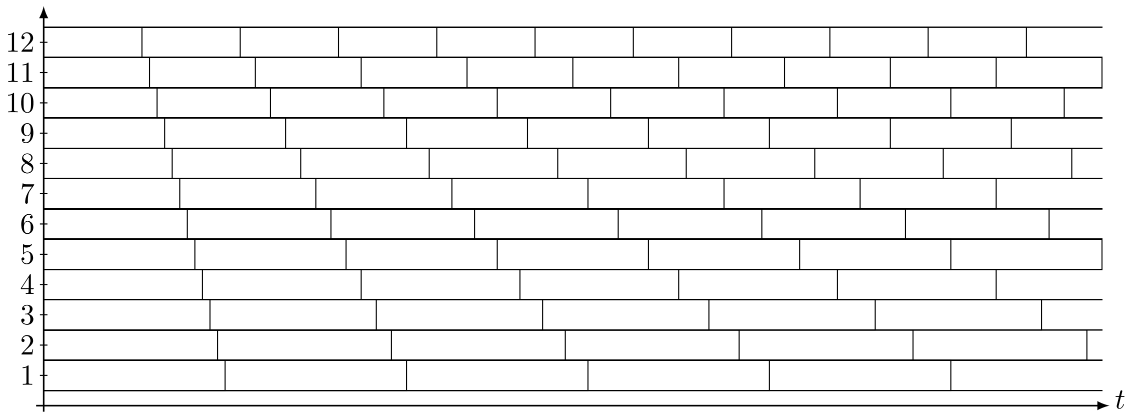


Figure 2.4: The relative lengths of the N_k in the Constant Q transform. The lengths of the blocks differ very much, taking the shortest block for about 9 octaves is a huge waste of computational power. For the lowest octave, it would suffice to calculate the CQT every $9 \cdot N_k^{\min}$ samples.

frequency spectrum (see figure 2.3). The spectral kernels will now be stored in matrix form, thus equation 2.10 becomes

$$\boldsymbol{X}^{\mathrm{CQ}} = \boldsymbol{A}^{*}\boldsymbol{X} \tag{2.15}$$

$\boldsymbol{X}$ is a vector containing the values of $X(i)$. This is mainly for convenience, but also is an indication for how the speed will be gained in the implementation later on: Small values will be canceled out and set to zero. After this step, fast matrix multiplication algorithms for sparse matrices can be used: Multiplications with near-zero values will be omitted.

It is now possible to precalculate the matrix $\boldsymbol{A}$ and thus compute all frequency bins for one sample at a time, because $\boldsymbol{A}$ does not change from frame to frame. However, some problems still persist and make this approach inefficient: As you can see from figure 2.3, the higher the frequency of the temporal kernel, the longer the frequency response of the spectral kernel. The sparsity of the spectral kernel thus decreases with increasing frequency. Additionally, when the Constant Q transform is calculated over a wide range of frequencies, the frame length of the DFT is quite long.

It is not necessary to calculate the CQT for every sample, it is sufficient to calculate it for a range of overlapping frames, since the CQT does not change qualitatively from one sample to the other. One reason for this is the frequency-time-relationship. Normally, an overlap of 50% of the frame length N_k is chosen. Note that the overlap and frame length depend on the frequency and thus are different for every frequency bin k (see figure 2.4). This complicates the application of equation 2.15 if the same overlap is chosen for every bin, since it is not possible to use the matrix form if different frame lengths are used for every frequency bin: The resulting object $\boldsymbol{X}$ would not be a matrix – it would not be in rectangle, but in a trapezodial form. To ease this, the overlap is chosen to be equal (sample count) for all bins. To not loose information, the overlap needs to be at least 50% of $N_k^{\min}$, which is the frame length of the shortest spectral kernel (i.e. the one with the lowest frequency)[3], so $N_k^{\min}/2$ will be chosen as overlap for every bin. This way, redundancy and computational overhead is introduced, but the calculation of the transform and the handling of the results of the transform is easier.

By **processing one octave at a time**, a substantial speedup is gained in [SK10]. First, the CQT is calculated over the whole signal, but only for the highest octave. To obtain the next octave, the kernels $\boldsymbol{A}^*$ are kept, and the input signal $x(n)$ is lowpass-filtered at $\frac{f_s}{4}$. Every second sample of the lowpass signal is dropped, effectively placing the next lower octave in the frequency domain of the old kernels. This process can be applied multiple times to calculate all octaves. Using this approach, it is possible to shorten the length of the DFT block, and $\boldsymbol{A}^*$ remains sparse. Within one octave, the

[3] [SK10] use $N_K/2$ for this purpose (note the large K), with N_K being the frame length of the bin with the highest frequency of one octave. The two approaches are equivalent in most cases, but their approach is limited to the one-octave case.

frame lengths are equal; between the octaves, the frame length doubles: There are half as many values for next lower octave when compared to the next higher one, due to the sample dropping.

Schörkhuber and Klapuri further speed up the process by calculating equation 2.14 with some **temporally translated** versions of the same **temporal kernels**. Using this technique, it is possible to calculate the DFT spectrum $X(i)$ less often. The temporal kernels of every frequency bin need to cover the whole time line, otherwise some part of the input signal would be missed and some information would get lost. This problem can be approached by solving equation 2.15 every $N_k^{\min}/2$ samples apart, but that would also require to calculate $X(i)$ every $N_k^{\min}/2$ samples, which is costly – especially if it is not needed for the low frequencies. Usually, $N_k^{\min} < N$ with N being the FFT window length (e.g. $3N_k^{\min} < N < 4N_k^{\min}$). Assumed that P successive temporal kernels for the highest frequency fit into one FFT window, it is possible to shift these temporal kernels to P different locations within that first window and calculate equation 2.15 with the emerging spectral kernels (see figure 2.5) to achieve the very same effect as if the FFT window would have been moved P times. Using this approach, only one FFT window is needed instead of P, which saves $P-1$ FFT calculations. In practice, P is in the range of $5-10$. Note that the emerging spectral kernels only differ in phase, thus the property of sparseity holds for these kernels, too.

Run Time Analysis 2.12 (***efficient calculation of the CQT***)**:**
Schörkhuber and Klapuri perform a run-time analysis in [SK10, p.5]. Their result is not directly comparable to the run-time analysis performed for the naïve approach, since they did not use the $\mathcal{O}$-notation. Here, their analysis is extended to fit that purpose.

Their main result is that regardless of the number of octaves processed, their algorithm does not need more than $C \leq 2(\lfloor (L - N_{\text{DFT}})/H_{\text{DFT}} \rfloor + 1)$ FFT calculations, where L is the length of the input signal with zero-padding, N_{DFT} is the length of an DFT frame and H_{DFT} is the hop size of the DFT. Since every FFT calculation takes up to $\mathcal{O}(n \log n)$ operations, the total cost for the FFTs is $\mathcal{O}(CN_{\text{DFT}} \log N_{\text{DFT}})$.

Since the length of the input signal halves in every iteration cycle, the total length of all the input signal is bounded by $2L$ (see [SK10]). Thus it is equivalent to calculate the cost of the lowpass-filtering as the cost of one filtering for a signal of length $2L$. The lowpass-filtering is carried out by a sixth-order Butterworth filter, which is an IIR filter. These filters can be implemented in a *direct form II transposed* structure (see figure 2.6 and [OS95, p.375]). This structure is described through

$$y(n) = \sum_{i=0}^{B} b_i x(n-i) - \sum_{j=1}^{A} a_j y(n-j) \tag{2.16}$$

where A and B are the maximum indicies of the coefficients a_j and b_i are appropriate

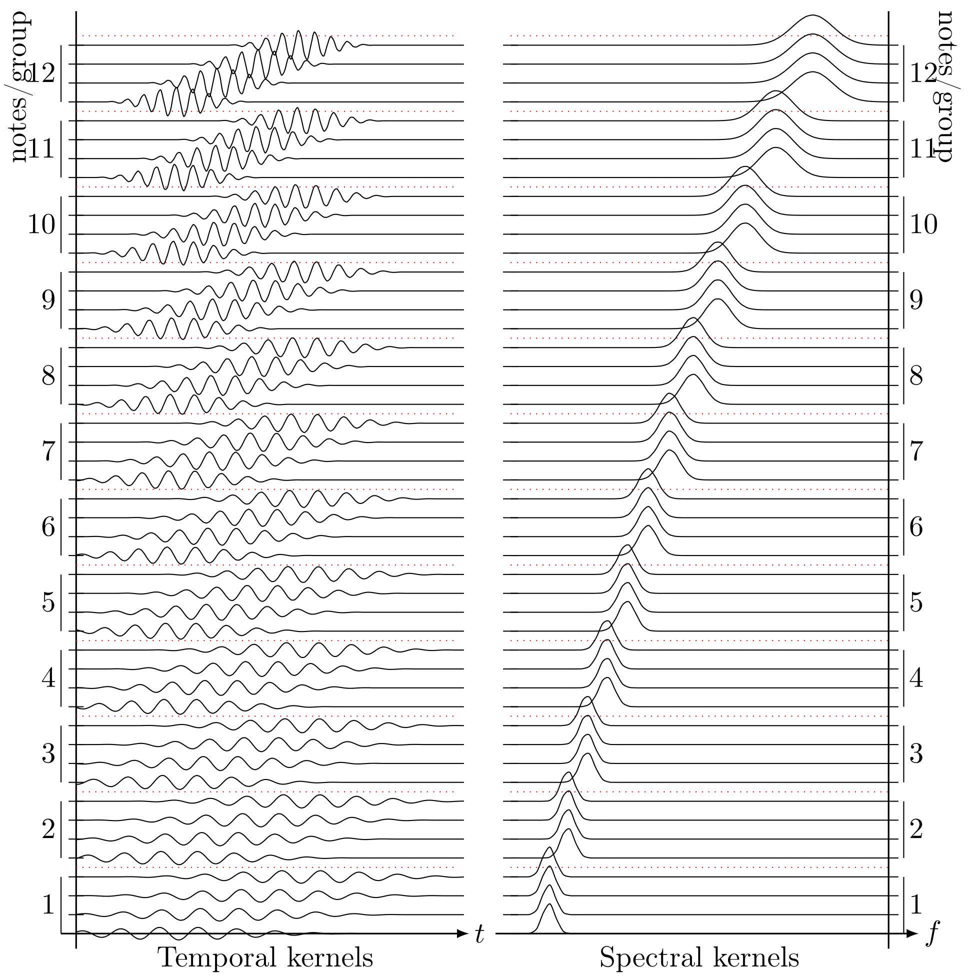
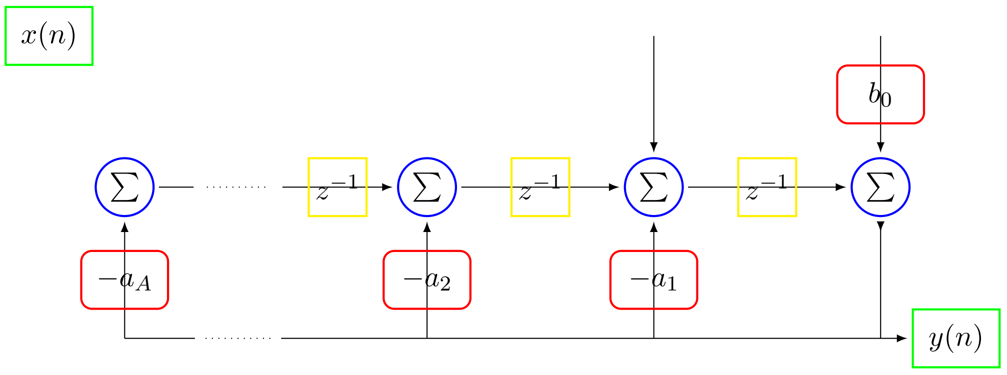


Figure 2.5: Shifted temporal kernels and corresponding spectral kernels. Note that in both graphics only the real part is drawn. The spectral kernels for one frequency at different temporal positions differ only in phase, which cannot be seen from the real-only plot.


$x(n)$
b_0
$\sum$
z^{-1}
$-a_A$
$-a_2$
$-a_1$
$y(n)$

3 Feature extraction

Feature extraction is crucial to be able to compare musical pieces to each other. Features are properties that can be extracted from data and describe an aspect of the data. The extracted features will be fed into a classifier.

In the literature, many different features have been used for music classification. [FTZ11] gives a brief introduction and divides the possible features into classes: low- and mid-level features. Low-level features are further divided into timbre and temporal features. Mid-level features are usually extracted based on the low-level ones. They can be subdivided into pitch, rhythm and harmony features. High-level features are descriptions like the genre etc. Features can describe short- and long-term processes in a musical piece: timbre features usually are short-term features, the other mentioned features are computed on a longer timescale (long-term features).

In this thesis features are extracted that directly refer to components of music that are used by musicians, such as chord structure, dynamic range, tempo and timbre. One of the main advantages of features with a direct musical meaning is that they can be checked for plausibility by humans and thus are easier to handle for signal analysis newbies. These features may also be used later on for tasks other than classification, such as finding recordings with similar tempo. Many of the mentioned features in [FTZ11] and [TVE08] do not have such a direct meaning, e.g. the zero crossing rate of the signal.

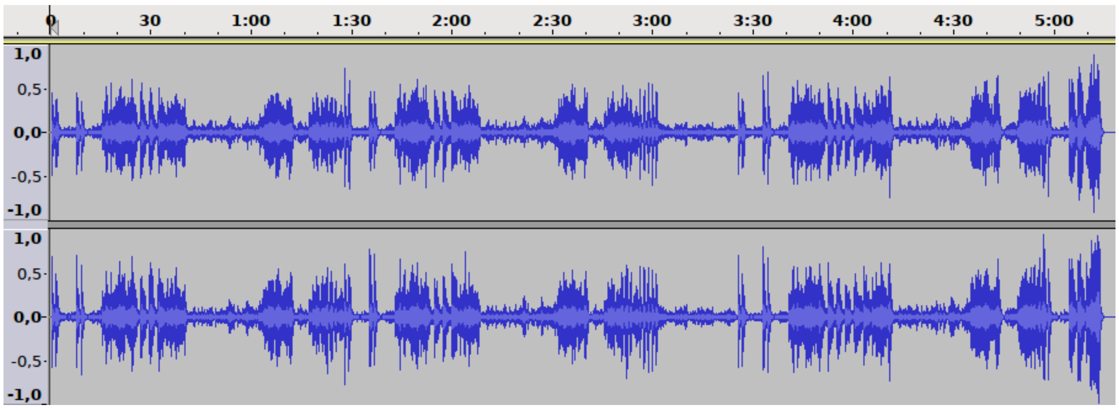
All described features will be extracted ontop of the Constant Q transform (see section 2.3).

3.1 Dynamic range

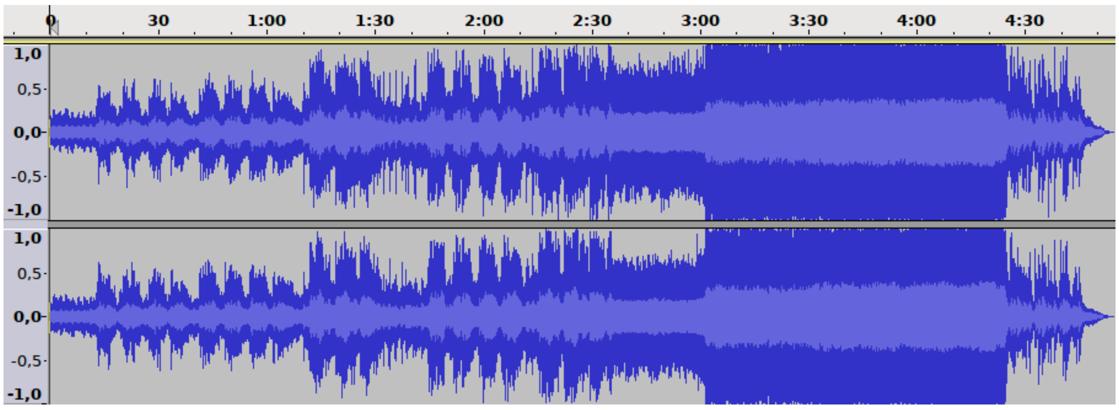
The dynamic range of a musical piece is a measure that describes how the parts with lower volume relate to the parts with higher volume. If a piece is dominated by the louder parts, the dynamic range is low, if there is a mixture between loud parts and quieter parts, then the dynamic range is high. For example, the dynamic range of a classical masterpiece is expected to be higher than the dynamic range of a punk rock song. Usually, modern songs have a low dynamic range when compared to the older ones. This process is sometimes referred to as the *loudness war*[1]. This feature should help to

[1] see http://en.wikipedia.org/w/index.php?title=Loudness_war&oldid=522727592. This is a permanent link to article version of 11/22/2012.

distinguish between these types of recordings. The dynamic range will be a long-term feature, it is represented by one value for the whole piece. It is a low-level feature.



(a) Normalized waveform of *Symphony No. 28 in C major, first movement, allegro spirituoso* by *Wolfgang Amadeus Mozart*



(b) Normalized waveform of *Fix you* by *Coldplay*

Figure 3.1: Different normalized waveforms of two pieces of music. The first one has a higher dynamic range.

Definition 3.1 (*Dynamic range of a continuous signal*):
The *dynamic range* of a signal is defined as the root mean square (RMS) of the continuous input signal $x_c(t)$, which is

$$\mathrm{x_{cRMS}} = \sqrt{\frac{1}{T_c} \int_0^{T_c} x_c^2(t)\,\mathrm{d}\,t} \tag{3.1}$$

with T_c being the duration of the signal.

$\lrcorner$

This definition is derived from the notion that $x_\mathrm{c}(t)$ is directly used as driving voltage $u(t)$ for an amplifier, which in turn behaves like an ideal resistor such that the current $i(t)$ and voltage $u(t)$ are proportional and can be set equal up to a constant factor. The constant factor was assumed to be 1.

This definition can easily be transferred to the case of discrete values. To ease calculation and comparability of different signals, the above definition will be changed slightly to give results in the interval $[0, 1]$ with low values indicating a low dynamic range. Additionally, the result will be derived from the Constant Q transform bins.

$\ulcorner$**Definition 3.2 (*Normalized sum of Constant Q bins*):**
Let $\mathrm{nsum}_\mathrm{CQ}(\boldsymbol{X}^\mathrm{CQ}, t) : \mathbb{R} \to [0, 1]$ be the normalized sum of the precalculated Constant Q transform bins at a point in time t:

$$\mathrm{nsum}_\mathrm{CQ}(\boldsymbol{X}^\mathrm{CQ}, t_n) = \frac{1}{R}\sum_{b=0}^{B} |X^\mathrm{CQ}(b, t_n)| \tag{3.2}$$

with

$$R = \max_n \left(\sum_{b=0}^{B} |X^\mathrm{CQ}(b, t_n)|\right). \tag{3.3}$$

where the t_n are discrete points in time. Every t_n refers to the continuous time interval $[t_n, t_{n+1}]$.

$\lrcorner$

Choose $t_{n+1} - t_n \approx 5 - 10ms$. $|X^\mathrm{CQ}(b, t_n)|$ is then calculated as the mean of all Constant Q transform bins within this interval. If the interval should be too short for a Constant Q transform bin, the value of the nearest Constant Q bin will be chosen.

Keep in mind that this definition is incomplete since it is undefined for the case of a constant signal, since in that case $R = 0$. In this case, set $\mathrm{nsum}_\mathrm{CQ}(\boldsymbol{X}^\mathrm{CQ}, t_n) := 1$, which makes sense with the following definition (leads to a signal with low dynamic range). In all other cases, $\mathrm{nsum}_\mathrm{CQ}$ is well-defined.

$\ulcorner$**Definition 3.3 (*Dynamic range of a time-discrete signal*):**
The *dynamic range* of a time-discrete signal is defined through

$$\mathrm{dyn}_\mathrm{dRMS}(\boldsymbol{X}^\mathrm{CQ}) = 1 - \sqrt{\frac{1}{T}\sum_{n=0}^{N} {\mathrm{nsum}_\mathrm{CQ}}^2(\boldsymbol{X}^\mathrm{CQ}, t_n)} \tag{3.4}$$

$\lrcorner$

Since $\mathrm{nsum}_\mathrm{CQ}(\boldsymbol{X}^\mathrm{CQ}, t) \in [0, 1]$ by definition, $\mathrm{dyn}_\mathrm{dRMS}(\boldsymbol{X}^\mathrm{CQ}) \in [0, 1]$, too. This definition is derived from equation 3.1; it was discretized and to better reflect the intuition, reversed through the term $1 - x$. Through normalization, the values of different signals are comparable. High values of $\mathrm{dyn}_{\mathrm{dRMS}}$ correspond to the dynamic range being high, thus the definition corresponds to an intuitive definition of the dynamic range. This definition is independent of the actual maximum value of the recording, due to the normalization step. Nevertheless, it might still be susceptible to distortions of the input signal.

It is not known to the author that this measure has been used in the past for music classification.

3.1.1 Problems, drawbacks and improvements

Many recordings do not have a defined end of the song but are faded out. This would end up in a dynamic range that is calculated to be much higher than expected. Typically, recordings are not faded out before the last 20 seconds of the end. Omitting these last few seconds of the signal in calculation of the dynamic range produces more accurate results. In the implementation, the last 20 seconds of a recording are omitted if the recording is longer than 2 minutes.

3.2 Extracing the tempo of a musical piece

Most humans are able to intuitively tap along a piece of music in a way that tapping and the musical piece fit together. It is somewhat harder to define these properties explicitly and derive an algorithm that reliably extracts a measure of tempo from raw audio signals. Nevertheless, tempo is a crucial property of music which most listeners, wether they are musicians or not, are able to recognize. Therefore, a music similarity classifier should also use this feature for classification.

Mainly, the approaches to beat estimation try to find the times where notes are played (called *onset times*) and then try to find recurring patterns (see e.g. [TC02, p.296]). Many algorithms split the raw signal in multiple bands (see e.g. [Got01]), because it eases the detection of different rhythm instruments, such as a snare drum, bass drum, or hi-hat.

Like most of the other algorithms, the algorithm used in this thesis is based on the observation that beats most often occur at recurring positions within one measure, and that a beat drastically changes the energy in the signal. To reflect the possibilities, especially the low computational power of the device, the algorithm is not very complicated when compared to the algorithms, e.g. as presented in [Got01]. It makes use of the sum of the Constant Q bins to determine the dynamic range (see equation 3.2). The sum value does not need to be normalized, but normalization does not harm. In the implementation, the non-normalized sum vector is shared between the algorithm for dynamic range calculation and tempo estimation. This vector can be saved temporarily and only

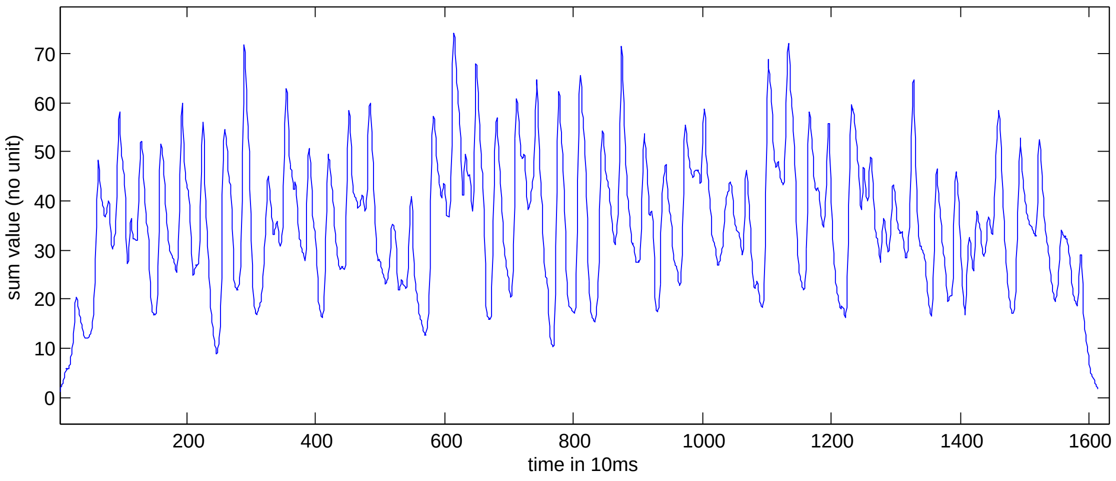


Figure 3.2: Non-normalized sum vector of `test.mp3`. The audio consists of string instruments only, no percussive instruments involved.

needs to be calculated once for both use cases. The algorithm then calculates the series of differences of consecutive elements of $\mathrm{nsum}_{\mathrm{CQ}}$:

$$d(t_n) = \mathrm{nsum}_{\mathrm{CQ}}(\boldsymbol{X}^{\mathrm{CQ}}, t_n) - \mathrm{nsum}_{\mathrm{CQ}}(\boldsymbol{X}^{\mathrm{CQ}}, t_{n+1}) \tag{3.5}$$

The difference series is one element shorter than the sum series.

Now, the signal contains peaks at times with a large change in the energy of the signal (see figure 3.3). The next step is to find recurring parts of the difference vector. One possibility to achieve this is to use autocorrelation as a measure of self-similarity.

Definition 3.4 (*Autocorrelation of a series*):
Let τ be the time shift, $\tau_{\max}$ the maximum time shift. Then, the *autocorrelation* of a series of real values $x(t)$ is calculated as

$$a(x(t), \tau) = \sum_{n=0}^{N} x(t_n)x(t_n - \tau). \tag{3.6}$$

All values that lie outside the bounds of the series shall be treated as zero.

The autocorrelation changes the view on the signal. The sum and difference vectors refer to concrete points in time of the original signal. The autocorrelation series describes the correlation of the signal to itself when shifted about a certain amount of time. The application of the autocorrelation function to the series of differences leads to a signal that contains peaks at multiples of the period length of a periodical signal which are the

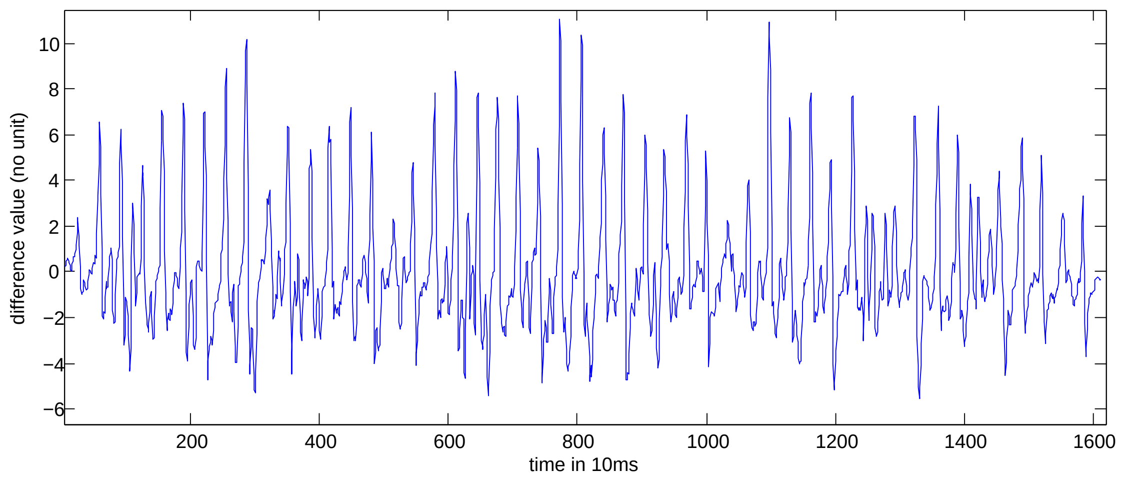


Figure 3.3: Difference vector of the sum vector from figure 3.2 as defined by equation 3.5

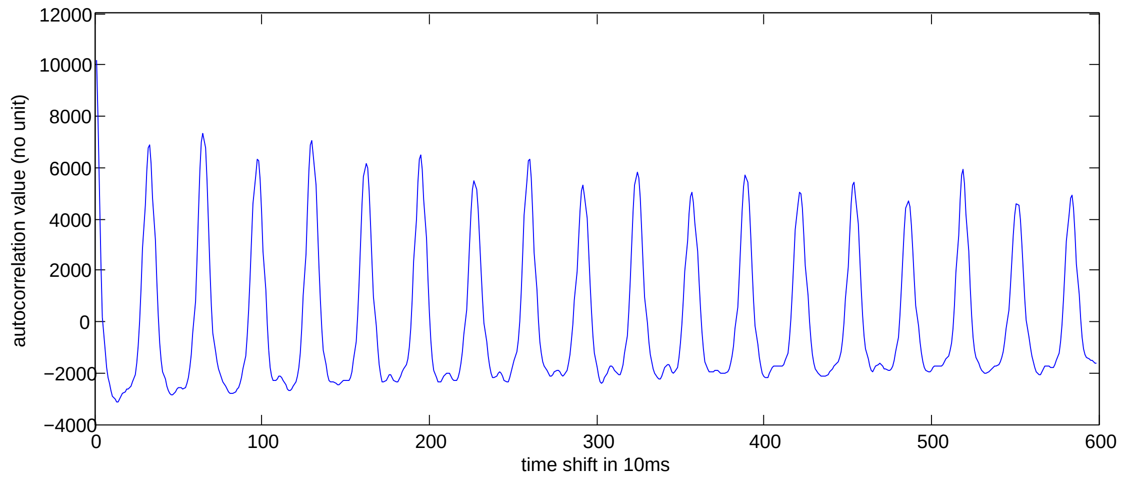


Figure 3.4: Autocorrelation of the data in figure 3.3, maximal shift 6s. The distances of the peaks describe the tempo of the piece, here about one beat every 320ms, which is about 187.5bpm. Measured by hand: 93bpm $\approx \frac{187.5}{2}$bpm

times that are perceived as beats. Thus, the autocorrelation series describes the distances of the beats. In this representation, the mean distance of the peaks of the signal is the tempo of the song in bpm (see figure 3.4). For the complete algorithm, take a look at algorithm 1.

Algorithm 1 Beat detection based on Constant Q transform

```
Precondition: $\boldsymbol{X}^{\mathrm{CQ}}(b,t)$ is the Constant $Q$ transform result of bin $b$ at time $t$ with a
    total bin count of $B$ and a total duration of $T$.
1: Let $\boldsymbol{s}$, $\boldsymbol{d}$, $\boldsymbol{a}$ be vectors.
2: Sum $|\boldsymbol{X}^{\mathrm{CQ}}(b,t)|$ at all bins $b$ at time slices of $\sim$ 5ms and save it to $\boldsymbol{s}$.
3: Build derivative of $s$ and save that to $\boldsymbol{d}$.
4: Calculate autocorrelation of $\boldsymbol{d}$ with an offset of up to $\sim$ 6s[2] and save that to $\boldsymbol{a}$.
5: Identify peaks in the signal and measure their distances
```

A difficult part of the algorithm is the robust recognition of the peaks in the autocorrelation signal. The classical approach to detect peaks in a function is to look for a change of the sign in the deriavtion of the function. However, a close look at figure 3.4 reveals that the signal contains sub-peaks, so this approach would find too many peaks. In general, it is problematic to use derivatives of noisy signals.

One possibility is to use a fixed decision boundary. Peaks that go beyond the boundary are then considered as valid. The length of a peak is measured, the beginning or the mean can be used as reference point for a particular peak. The distances of the reference points can then be measured. The value for the decision boundary can be calculated in dependence of the rest of the signal, e.g. in dependence of the standard derivation and mean of the signal. In this thesis, a $2\sqrt{\sigma}$ distance from the mean of the signal has been used as decision boundary.

3.2.1 Problems, drawbacks and possible improvements of the proposed algorithm

The algorithm is capable of calculating the tempo of music signals up to a difference of integer factors to the powers of two. Anyway, it is not easy to distinguish between these cases for humans, too. It is not always wrong to set a metronome at double or half time of the original piece of music. Nevertheless, this is the major drawback: In the context of music comparison, it is not of great interest if a song has 80bpm or 83bpm (which can be distinguished by the algorithm). A more interesting question is if a recording has 80bpm or 160bpm, which cannot reliably be distinguished with the proposed algorithm.

Additionally, the recognition of the peaks in the autocorrelation signal is not as robust as it should be. Some peaks are slightly lower than the recognition level and therefore are not properly recognized. If the recognition boundary is lowered, other signals that should not be mistaken as beat are taken into account. This is definitely the weakest part of the algorithm and should be adressed first. At the time, the recognition of the peaks is done by a fixed decision boundary. A close look at figure 3.4 reveals that the peaks of the autocorrelation signal get smaller with rising time shift, and a fixed decision boundary does not take this fact into account. A more sophisticated algorithm for peak detection should raise the correct recognition rate significantly.

Another possible improvement of the algorithm refers to the use of multiple frequency bands to directly try to recognize percussive instruments. This has not been adressed up to now, since it naturally requires more processing time and/or memory than the proposed approach, which might be infeasible due to the low memory and processing power requirements of the platform. Details about this approach can be found in [Got01]. A simple implementation would swap the steps of summation and calculation of the difference series, resulting in the difference vector being calculated for every Constant Q bin first, then a step of noise filtering, and the sum calculated afterwards.

3.3 Timbre features

Timbre is for music what color is for images. Humans typically refer to timbre as "what it sounds like". Instruments can be differentiated via their specific timbre. Some instruments are similar, or at least comparable in timbre. Since there are some genre-typical combinations of instruments, the timbre can be used to differentiate between or group them. In the literature, one finds definitions (taken from [Jen99, p.10]) such as (quote)

> "Timbre is defined in [ASA 1960][3] as that which distiguishes two sounds with the same pitch, loudness and duraton. This definition defines what timbre is not, not what it is."

Timbre is a musical feature that still is under debate. It is not clearly fixed to one well-known and accepted mathematical definition, such as it is the case with pitch, loudness and duration of a note. Typically, timbre refers to an instrument, not a whole signal with mixed instruments: Every instrument has its own timbre, thus making it difficult to recognize all timbres when given a polyphonic signal with multiple different instruments.

3.3.1 Timbre for monophonic and monotimbral signals

As mentioned in the above quote, some properties of the signal of a musical instrument have already been described: Pitch, for example, is the fundamental frequency of a musical note. Volume, as the energy of the note, and the duration of it. The same instrument has a similar timbre at different notes, so timbre and pitch can be made independent of each other.

[3] <A/N>: [ASA 1960] in the quote stands for [Ass60] in this thesis.

The timbre of an instrument is considered to be of multiple dimensions [Jen99, p.10]. On the one hand, timbre consists of the interplay of the *fundamental frequency* of a note with its corresponding overtones [Jen99, p.14]. On the other hand, the envelope of the signal also is of importance [Jen99, p.51]: E.g. violin and overdriven/compressed guitar are dissimilar but can nevertheless not be distinguished in every case, namely if the first few milliseconds of the signal (later called *attack* and *decay* phases of the envelope) are missing. Clarinet and guitar can easily be distinguished without these first few milliseconds of the signal.

First, the overtone ratio and envelope of an audio signal is described and its advantages and drawbacks for a definition of the timbre are discussed. Secondly, the *cepstrum* is defined and discussed, too, as a better approach for a definition of timbre.

Overtone ratio

Traditionally, the timbre of an instrument is known to be a combination of the overtone ratio of a signal and its envelope (see [Zie00, p.12]). It seems natural to use this definition for all sorts of signals and to try to directly extract these properties from the signal, especially since a frequency decomposition of the signal already has been calculated.

The final definition of timbre of an arbitrary signal will be approached by first defining timbre for monophonic, monotimbral signals without taking the envelope into account.

Definition 3.5 (*Fundamental frequency and overtones*):
The *fundamental frequency* f_0 of a signal is the perceived pitch of a musical note. In most times, it is the lowest strong frequency in the spectrum. Besides the fundamental frequency, a note played on an instrument also consists of *overtones* f_k, which are frequencies at integer multiples of the fundamental frequency:

$$f_k = kf_0, \qquad k \in \mathbb{N} \setminus \{0\} \tag{3.7}$$

Definition 3.6 (*Timbre of a monophonic, monotimbral signal/overtone ratio*):
Let $\mathrm{vol}(f_k, t)$ be the loudness of a given frequency (i.e. the value of the corresponding constant Q bin $|X^{\mathrm{CQ}}(b,t)|$). Then

$$\boldsymbol{T}_{\mathrm{mp,mt}}(f_0, t) = \frac{1}{R} \begin{pmatrix} \mathrm{vol}(f_0, t) \\ \mathrm{vol}(f_1, t) \\ \vdots \\ \mathrm{vol}(f_k, t) \\ \vdots \end{pmatrix} \tag{3.8}$$

is the *timbre of a monophonic, monotimbral signal* or *overtone ratio*, the normalized vector of the loudnesses of the fundamental frequency and their overtones. $R \in \mathbb{R}$ is chosen such that $\|\boldsymbol{T}_{\text{mp,mt}}(f_0, t)\| = 1$.

Note that in theory, there is no upper bound for the number of overtones. In practice, an upper bound would be chosen that enables most instruments to be correctly recognized.

Envelope of a signal

In addition to the overtone ratio, there are other factors that enable humans to distinguish between musical instruments, most notably the *envelope* of the audio signal.

Definition 3.7 (*Envelope of a signal*):
The *envelope* of a signal is a smooth curve outlining the extremes of the amplitude of the signal.

Typically, an instrument has volume changes when it produces a sound, for instance the signal of a guitar is much louder in the first few moments than later on – also, more noise is produced at that time. For a monophonic and monotimbral signal, the envelope can be derived from the nsum_{CQ} series defined in equation 3.2.

There exist some models for the envelope of a signal. In the context of music production, the ADSR envelope model is used to describe some properties of a signal. It has been used for the creation of sounds with digital and analog synthesizers since decades, and for the examination of these signals, too.

Definition 3.8 (*ADSR envelope model*):
The envelope of a signal is divided into four phases: *attack*, *decay*, *sustain* and *release*. In the attack phase, the signal will rise to a maximum level. In the decay phase, the signal will fall back to a level at which it will stay constant for a while (sustain), until it goes back to noise level/silence. The transition from sustain to noise level is called release.

See figure 3.5 for an intuitive explanation.

The overtone ratio together with the envelope describe an instrument very well, although the overtone ratio during the different phases of the envelope can be different. If the envelope and overtone ratio is known, the signal can be synthesized. Overtone ratio and envelope are well-suited for the task of synthesis. For monophonic signals, it is possible to recognize instruments with these tools. For polyphonic and especially for polytimbral music, this is not necessarily true. For these signals, frequencies may overlap. Their signals will sum up, making it hard to extract their individual contributions. The same applies to the envelopes. Even if the signals were created using pure linear combinations of the overtones, this would be a hard task. Nevertheless, these properties are the most natural ingredients of music signals and are important properties.

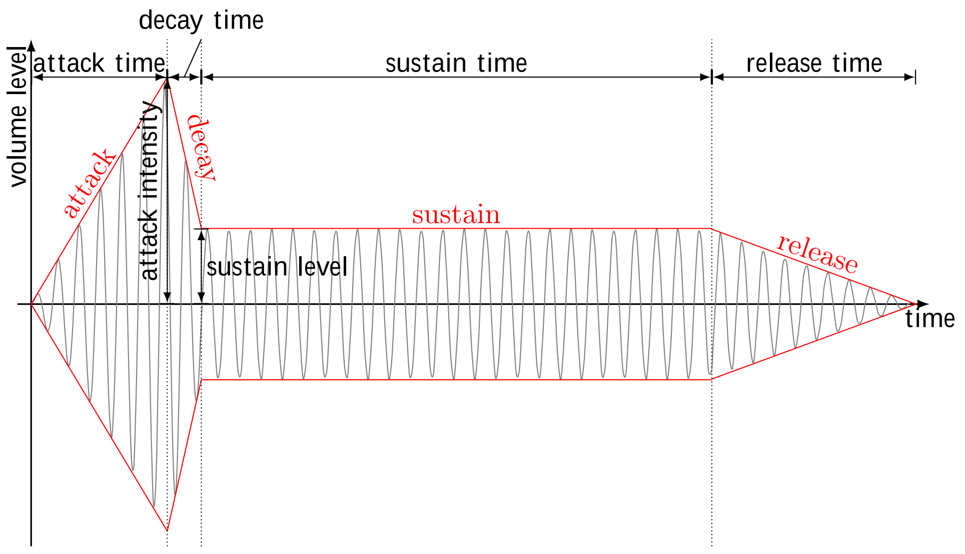


Figure 3.5: ADSR signal envelope model. Note that the different phases do not need to have a linear description. For instance, a flute excited by a Dirac pulse is an exponential (see [YGKM96]). The same applies to guitar and other string instruments (see [KVJ93]).

To overcome these problems, other features have been developed. The *cepstrum* was not developed for timbre recognition, but can be used for this task. It is described in the following. Parts of the cepstrum describe the envelope of a signal, and other parts describe the overtones and the rest of the signal (see [KKF00]).

3.3.2 Cepstral coefficients: Derived from the Mel and Constant Q scale

Mel Frequency Cepstral Coefficients (MFCCs) were first used in automated speech recognition. Their definition is motivated by the way a signal is generated when humans speak. In simplified terms, the vocal folds produce a sound which is shaped by the rest of the vocal tract. Since the words spoken are mainly formed by the vocal tract – the actual pitch of the signal is of lesser interest – one tries to separate these parts of the signal.

MFCCs help to achieve this task. The very first coefficient is related to the energy of the signal and is usually not used. The following few coefficients belong to the shape of the signal, which can be interpreted as the position of the vocal tract, or the envelope of the signal. The last coefficients belong to the pitch. There is no sharp boundary between the different meanings of the coefficients. In practice, "the first few coefficients" means about the first 10-20% of the coefficients, the very first coefficient is excluded unless otherwise noted.

Definition 3.9 (*Mel Frequency Cepstral Coefficients*):
The *Mel Frequency Cepstral Coefficients* are defined through an algorithm:

Algorithm 2 Mel frequency Cepstral Coefficients

```
1: Transform the input signal $x(t)$ to the frequency domain via a discrete Fourier transform, using windows of approximately 20ms
2: Transform the resulting coefficients to the *Mel frequency scale*[a] via overlapping triangular functions.
3: Take the logarithm of the resulting signal
4: Apply a discrete cosine transform
```

[a] The Mel scale is a logarithmic scale for frequencies. Sometimes it is described as being linear for the lower frequencies (i.e. below 1kHz).

Some researchers perform the Mel scaling after taking the logarithm (see e.g. [Log00]); this approach will not be covered here. See figure 3.6 for a graph of the calculation.

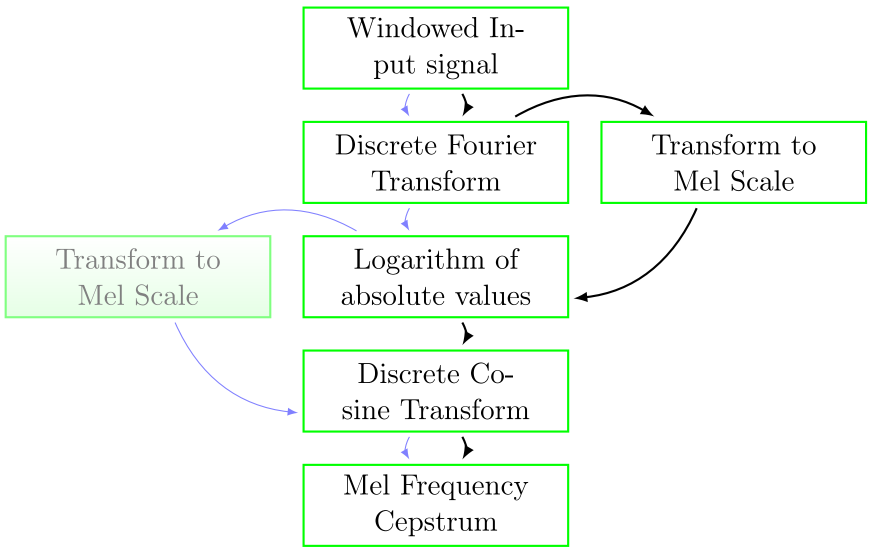


Figure 3.6: Definition MFC. Some researchers perform the Mel scaling after taking the logarithm (see e.g. [Log00]) (shown grayed out). This approach is not covered here and only shown for completeness

The intent of the last step is to decorrelate the values. Since at first sight it seems meaningless to transform the signal back to the time domain, the name *cepstrum* was chosen, which is just a reordering of the letters of *spectrum*. The MFCCs have been successfully used in the past for speech recognition and music similarity analysis (see e.g. [Sch11], [AP02], [Log00]).

Since this thesis focuses on features derived from the Constant Q transform, the MFCCs will not be used. Instead, a concept similar to the MFCCs will be derived. It will be based on the Constant Q transform, and therefore the coefficients will be called *Constant Q Cepstral Coefficients* (CQCC).

Definition 3.10 (*Constant Q Cepstral Coefficients*):
The *Constant Q Cepstral Coefficients* will be defined through an algorithm, too:

Algorithm 3 Constant Q Cepstral Coefficients

```
1: Transform the input signal $x(t)$ to the frequency domain via the constant $Q$ transform, using windows of approximately $20ms$
2: Take the logarithm of the resulting signal
3: Apply a discrete cosine transform
```

See figure 3.7 for a graph of the calculation. The constant Q cepstral coefficients can be

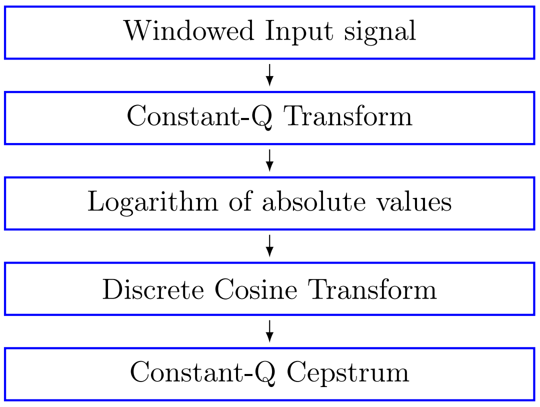


Figure 3.7: Definition CQC

calculated through the equivalent equation

$$c(m, t_n) = \sum_{k=1}^{M} \log(X^{\mathrm{CQ}}(k, t_n)) \cos\left(m\left(k - \frac{1}{2}\right)\frac{\pi}{M}\right) \tag{3.9}$$

The idea behind this definition is that the MFCCs try to summarize frequency bins of the discrete Fourier transform in a way to emulate the perception of frequencies of the human ear. The constant Q transform fits to that purpose, too: It was designed to fit to the scale of western music, so it seems legitimate to use it instead of the Mel frequency scale in this context. Additionally, the constant Q transform has been calculated and its results are available, so using it saves some computations.

The cepstral coefficients are typically calculated over about 10-20 ms of audio. See figure 3.8 for an example of CQCC vectors. As with the MFCCs, the most interesting part of the Constant Q Cepstrum are the first few coefficients. The Constant Q Cepstrum

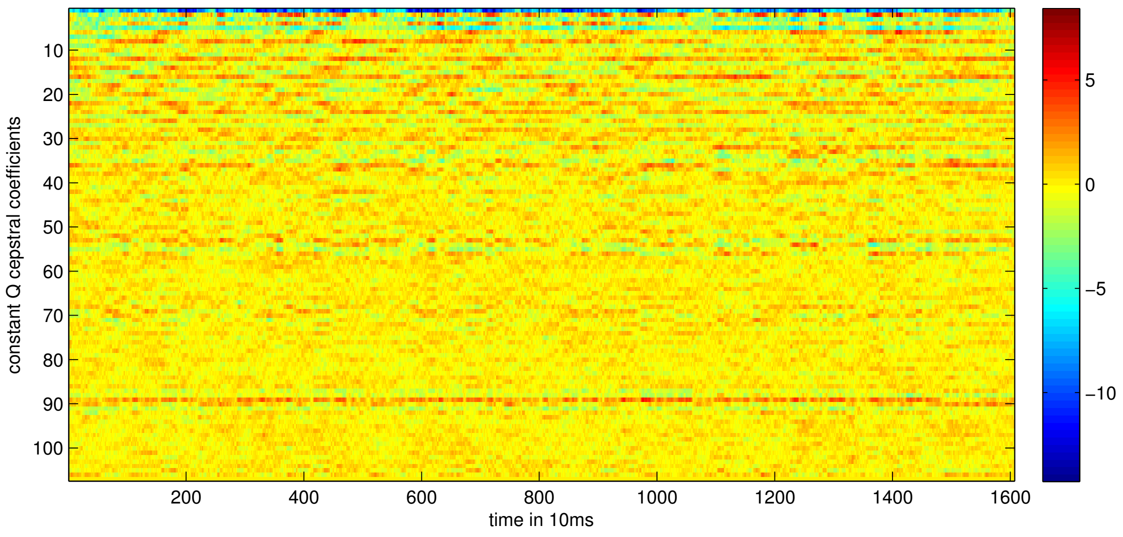


Figure 3.8: The CQCC vectors of file `test.mp3` (omitted the very first coefficient).

has successfully been used for instrument identification by Brown et al (see [Bro99]). They used three bins per octave in the Constant Q transform, in contrast to 12 bins in this thesis.

As we will see in the following, using the Constant Q Cepstrum directly as features has disadvantages in terms of memory usage.

Memory consumption of Constant Q Cepstral Coefficients as timbre feature vectors

Now, a memory consumption calculation will be performed. It will be shown that it is infeasible to use the CQCC vectors directly and save them to a database for later comparison. For the memory consumption calculation, a window of length 10 ms was chosen, because a window of 10 ms works good for the timbre feature, as well as the dynamic range and tempo feature.

This definition would lead to about $\frac{4\text{min}}{10\text{ms}} = 24000$ vectors for a piece of music with a length of 4 minutes, if there is no overlap between the analysis windows. Assuming the first 16 coefficients[4] of the vectors are used as timbre feature vectors this would at least

[4] 16 coe□cients are about 15% of 108 total cepstral coe□cients (12 values for nine octaves).

consume about $24000 \cdot 16 \cdot \text{sizeof(float)Byte} = 1500\text{KiB}$ of memory for single-precision (4Bytes/number) floating point numbers. For a single song in memory, this is affordable even on mobile devices. Since the features should be saved on nonvolatile memory for further processing or classification tasks and a typical song takes about 5MiB, it is not affordable to save the feature vectors in a database; it would take about 30% of the space needed by the song itself. For a database of 1000 songs, this would require about 1.5GiB of additional disk space.

To reduce the memory needs of the timbre features, a model of the timbre vectors will be learned, and that model will be saved instead. In this case, a *Gaussian Mixture Model* will be used, which consists of a mixture of multivariate normal distributions. The main purpose for using this approach is (lossy) data compression, but there are other useful properties arising from their usage. Using this approach, the memory requirements for the timbre of one musical piece can be lowered to 3-12KiB (see page 34 for details), which is only 0.2-0.8% of the size needed by the vectors themselves.

3.3.3 Gaussian Mixture Models

Definition 3.11 (*Gaussian Mixture Model*):
A *Gaussian Mixture Model* (GMM) is a probabilistic model. Its probability density function is a sum of I weighted normal distributions:

$$p(\boldsymbol{x}) = \sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{3.10}$$

$$\text{with} \quad p_i(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{k/2}\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) \tag{3.11}$$

$$\text{and} \quad \sum_{i=0}^{I-1} w_i = 1, \quad w_i >= 0 \quad \text{for all } i \tag{3.12}$$

with k being the dimensionality of the surrounding vector space.

Equation (3.12) is necessary to fulfill the normalization property of probability density functions (PDF), which is

$$\int_X p(\boldsymbol{x})\mathrm{d}\boldsymbol{x} = 1 \tag{3.13}$$

Apart from this, PDFs need to be non-negative (with the exception of values in a null set[5]).

[5] A null set is a set with *Lebesgue measure* 0, which effectively means that this set is negligible. All countable sets are null sets. In this sense, null sets are comparable to the exceptions in the term *almost all* when speaking of sequences.

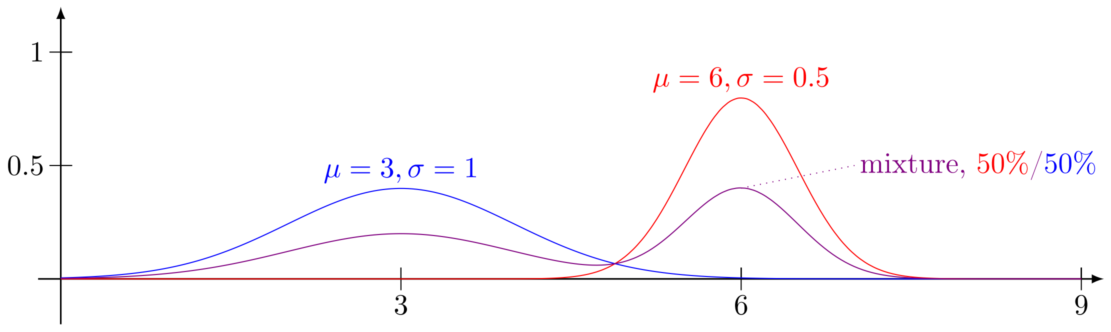


Figure 3.9: Equally weighted mixture of two one-dimensional gaussian distributions. The mixture has lower values due to the normalization property (equation 3.12).

It is not only possible to use normal distributions as an ingredient for a mixture model; every weighted (finite) sum of PDFs itself is a PDF:

Proof. PDFs need to be normalized and non-negative. Weighted sums of PDFs are normalized, since

$$\int_X \sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \sum_{i=0}^{I-1} \int_X w_i p_i(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \sum_{i=0}^{I-1} w_i \underbrace{\int_X p_i(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}}_{=1} = \sum_{i=0}^{I-1} w_i = 1.$$

Integral and sum may be swapped since the sum is finite. The property of being non-negative follows directly from the pdf being a finite sum of non-negative functions (note that countable unions of null sets always are null sets, and thus finite unions are null sets, too). □

The above is not necessarily true for infinite sums, as integral and sum may not be swapped in this case without further investigation.

Nevertheless, normal distributions are used as building blocks for the mixture models by many researchers due to a) their simple description (parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$) and b) the existance of simple algorithms to fit a GMM to a given set of data points. In many cases, the *Expectation-Maximization Algorithm* (see below) is used for that purpose.

Building a model for the data eliminates the need to save the timbre vectors directly and thus reduces the data count. It is a form of lossy data compression. For example, in figure 3.10, it suffices to save the parameters of the two distributions to get a good representation of the data points.

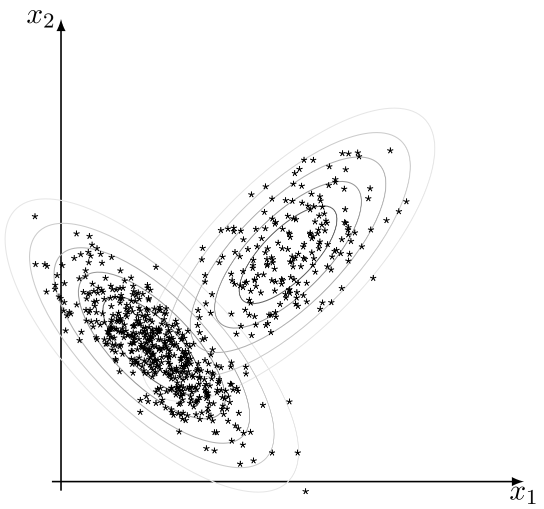


Figure 3.10: Scatterplot of 2D example data produced by two gaussian distributions. 75% of the data are produced by the left distribution, 25% are produced by the right distribution. Some contour lines have been drawn per distribution, these are equipotential lines of the probability density of the underlying normal distribution.

Memory consumption

The usage of GMMs comes with the drawback that the number of gaussian distributions needs to be set in advance. [PA05] show that a number of 50 normal distributions is ideal to describe the timbre feature vectors of one musical piece, but that a lower number (e.g. down to 20) does not hurt the performance that much. As in the example beforehand, a CQCC dimension of 16 is assumed.

For each Gaussian distribution, the weight w, mean $\boldsymbol{\mu}$, and covariance matrix $\boldsymbol{\Sigma}$ need to be saved on the disk. For the weight and mean, $1 + 16 = 17$ values are needed per distribution. Since covariance matrices are positive semidefinite and thus symmetric, only the upper (or lower) triangular matrix, including its diagonal, needs to be saved. The memory requirements for the covariance matrices are thus lowered to $\frac{n^2+n}{2}$, which is 136 for $n = 16$.

If full covariance matrices were used, such a model would require about $20 \cdot (17 + 136) \cdot \text{sizeof(float)} = 12240$ Bytes, which is only 0.8% of the memory all the feature vectors would require. The memory requirements could be further reduced if diagonal covariance matrices were used; in this case the model for one song would only take $20 \cdot (17 + 16) \cdot \text{sizeof(float)} = 2640$ Bytes.

Additionally, the model size is fixed; it will not change with the length of the piece, where the number of timbre feature vectors would.

3.3.4 Calculation of a GMM for a given set of data points

The *Expectation-Maximization algorithm* (EM) is used to train a GMM. It is an *unsupervised learning algorithm* (for details see section 4.2, page 62) that is used to find data clusters. Other variants of the same algorithm exist for applications such as missing data reconstruction; these applications of EM will be omitted. The algorithm is described only for the case of GMMs. This section uses terms usual in probability theory. For a good book on this, refer to [Kre05].

The EM algorithm performs *soft partitional clustering*, which means that it assigns probabilities for cluster memberships to every data vector. A simpler approach assigns every data point to one cluster and is called *hard partitional clustering*. An example for a hard partitional clustering algorithm is k-means, which is a special case of the presented EM algorithm.

The EM algorithm is a *maximum likelihood* method. It is presented in short, for more details see [PTVF07, pp.842] and [Bis06, pp.430]. The name for the EM algorithm stems from the fact that it consists mainly of two steps, which are repeated until convergence is reached: The *expectation* step and the *maximization* step. These steps will now be motivated through a gradient descent approach.

The desired output of the algorithm are w_i, $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$ as the parameters of the GMM, as well as $P(i|\boldsymbol{x}_n)$, which is the probability of cluster i being responsible for creating the data point $\boldsymbol{x}_n$. For many applications of GMMs, this is an important output parameter – for us, it will only be of importance inside the algorithm. In the following, equations will be given that hold if these parameters were known, and an algorithm will be derived how to iteratively compute them.

The PDF of cluster i is $p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, which will be written as

$$p(\boldsymbol{x}|i) := p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i), \tag{3.14}$$

the *probability distribution of* $\boldsymbol{x}$ *if cluster* i *is chosen*. The joint probability distribution for all clusters $p(\boldsymbol{x})$ is then given as

$$p(\boldsymbol{x}) = \sum_{i=0}^{I-1} P(i)p(\boldsymbol{x}|i) \tag{3.15}$$

with $P(i)$ being the *prior probability* of a data point to belong to cluster i. This prior probability was defined as the weight w_i, so rewriting equation 3.15 reveals:

$$\sum_{i=0}^{I-1} P(i)p(\boldsymbol{x}|i) = \sum_{i=0}^{I-1} w_i p(\boldsymbol{x}|i) = \sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i), \tag{3.16}$$

which is the definition of the Gaussian Mixture Model (see definition 3.11).

Using *Bayes theorem*[6], one gets the *conditional probability of i given $\boldsymbol{x}$*:

$$P(i|\boldsymbol{x}) = \frac{P(i)p(\boldsymbol{x}|i)}{p(\boldsymbol{x})} = \frac{P(i)p(\boldsymbol{x}|i)}{\sum_{i=0}^{I-1} P(i)p(\boldsymbol{x}|i)} \tag{3.17}$$

This can be seen as the *responsibility* of cluster i for data point $\boldsymbol{x}$. It is one of the desired outputs of the algorithm (see above).

So, how can we estimate the desired output parameters? The idea is to use a *maximum-likelihood estimation*. The ansatz of these methods is to choose the parameters such that the data becomes most probable. So, what is the probability of the data given the model? It is

$$p(\bigcup_{n=0}^{N-1} \{\boldsymbol{x}_n\}) = \prod_{n=0}^{N-1} p(\boldsymbol{x}_n) = \prod_{n=0}^{N-1} \sum_{i=0}^{I-1} P(i)p(\boldsymbol{x}_n|i) = \prod_{n=0}^{N-1} \sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}_n, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{3.18}$$

So, this equation should be maximized (as a function of the desired output parameters w_i, $\boldsymbol{\mu}_i$, $\boldsymbol{\Sigma}_i$)). Since it is hard to maximize the product, one usually maximizes the logarithm of these functions. Since the logarithm increases monotonically, the maximum of the logarithmized function is at the same argument as the non-logarithmized function. So, one can equivalently maximize

$$\ln\left(\prod_{n=0}^{N-1} \sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}_n, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right) = \sum_{n=0}^{N-1} \left(\sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}_n, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right). \tag{3.19}$$

Finding the maximum of a differentiable function is usually done through setting the derivative of the function to zero. Minimizing the negative of the function has some advantages in calculating the partial derivatives (see below):

$$-\sum_{n=0}^{N-1} \ln\left(\sum_{i=0}^{I-1} w_i p_i(\boldsymbol{x}_n, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right) \overset{!}{=} 0 \tag{3.20}$$

[6] $P(i|\boldsymbol{x})p(\boldsymbol{x}) = p(\boldsymbol{x}|i)P(i)$

Calculating the partial derivatives of the corresponding variables leads to the equations

$$\boldsymbol{\mu}_i = \frac{\sum_{n=0}^{N-1} P(i|\boldsymbol{x}_n)\boldsymbol{x}_n}{\sum_{m=0}^{N-1} P(i|\boldsymbol{x}_m)} \tag{3.21}$$

$$\boldsymbol{\Sigma}_i = \frac{\sum_{n=0}^{N-1} P(i|\boldsymbol{x}_n)(\boldsymbol{x}_n - \boldsymbol{\mu}_i)(\boldsymbol{x}_n - \boldsymbol{\mu}_i)^T}{\sum_{m=0}^{N-1} P(i|\boldsymbol{x}_m)} \tag{3.22}$$

$$w_i = P(i) = \frac{1}{N}\sum_{n=0}^{N-1} P(i|\boldsymbol{x}_n). \tag{3.23}$$

Both [PTVF07] and [Bis06] do not show the derivation of these equations, but they can be derived on about three sheets of paper with one glass of wine using the chain rule and Bayes theorem. These equations hold in a local optimum. Unfortunately, the right sides of the equations are indirectly connected to the left sides (through $P(i|\boldsymbol{x})$ in equation 3.17 and 3.16), so a direct calculation is not possible. It is however possible to apply the idea of a gradient descent approach: One starts with some initial guesses and iteratively applies equations 3.17 (called the *E-step*, *expectation*) and 3.21 – 3.23 (called the *M-step*, *maximization*). This is done until a convergence criterion is fulfilled (e.g. that the values do not change much, or the log-likelihood does not raise much) or a maximal number of iterations has been run (e.g. 50 iterations). The algorithm is then guaranteed to find a local optimum.

There are some numerical problems with the standard implementation of the algorithm. One problem of the algorithm is that the values of $p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ can get smaller than can reliably be saved in an `IEEE-754 double` (underflow problem). To overcome this, instead of calculating $p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ directly, $\ln(p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i))$ is calculated:

$$\ln\left(p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right) = \ln\left(\frac{1}{(2\pi)^{k/2}\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)\right) \tag{3.24}$$

$$= -\frac{k}{2}\ln(2\pi) - \frac{1}{2}\ln\left(\det(\boldsymbol{\Sigma})\right) - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) \tag{3.25}$$

A problem arises in equation 3.17 where the value of the GMM in a particular point needs to be calculated. To overcome this, [PTVF07, p.844] suggest a trick called the *log-sum-exp* formula (quote):

$$\ln\left(\sum_i \exp(z_i)\right) = z_{\max} + \ln\left(\sum_i \exp(z_i - z_{\max})\right) \tag{3.26}$$

"where the z_i's are the logarithms that we are using to represent small quantities and $z_{\max}$ is their maximum. Equation (16.1.3) [A/N: here, equation

> 3.26] guarantees that at least one exponentiation won't underflow, and that any that do could have been neglected anyway."

Using this trick, underflows in the algorithm do not happen anymore and equation 3.25 can be used to calculate $p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ whenever needed.

Another problem is numeric stability. This problem is hidden inside the calculation of the value of the PDF $p_i(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ (see equation 3.25). While being popular in linear algebra, matrix inverse operations are not popular in numerical linear algebra, since operations involving inverse matrices are numerically unstable if the *condition value of the matrix*[7] is large. This value can be seen as a "distance to singularity" of a matrix. A matrix with large condition number can be made singular with only small pertubations of its values. A singular matrix has a condition value of $\kappa(\boldsymbol{A}) := \infty$, the identity has a condition value of $\kappa(\boldsymbol{I}) = 1$. It is not unusual for a covariance matrix to be near singularity, it is sufficient if one dimension of the input data is is underrepresented for this to happen.

To solve this problem, the inverse matrix $\boldsymbol{\Sigma}^{-1}$ is replaced by an equivalent operation with better numeric stability. The term $\boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) =: \boldsymbol{y}$ is equivalent to solving the equation $\boldsymbol{\Sigma y} = (\boldsymbol{x} - \boldsymbol{\mu})$. One possible, numerically stable solution is to use a *Cholesky decomposition* for this purpose.

Definition 3.12 (*Cholesky decomposition, $\boldsymbol{LL}^T$ variant*):
Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ symmetric and positive definite. Then the $\boldsymbol{LL}^T$ *variant of the Cholesky decomposition* of $\boldsymbol{A}$ is defined as

$$\boldsymbol{LL}^T = \boldsymbol{A} \tag{3.27}$$

with $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ being a lower triangular matrix.

Numerically even more stable than the $\boldsymbol{LL}^T$ variant of the Cholesky decomposition is the $\boldsymbol{LDL}^T$ variant[8].

Definition 3.13 (*Cholesky decomposition, $\boldsymbol{LDL}^T$ variant*):
Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ symmetric and positive definite. Then the $\boldsymbol{LDL}^T$ *variant of the Cholesky decomposition* of $\boldsymbol{A}$ is defined as

$$\boldsymbol{LDL}^T = \boldsymbol{A} \tag{3.28}$$

with $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ being a lower triangular matrix with $l_{ii} = 1$ for all i and $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ a diagonal matrix with positive entries and $\boldsymbol{D} = \boldsymbol{D}^{\frac{1}{2}}\boldsymbol{D}^{\frac{1}{2}}$. For diagonal matrices, $\boldsymbol{D}^{\frac{1}{2}}$ means taking the roots of the diagonal elements of $\boldsymbol{D}$.

[7] $\kappa(\boldsymbol{A}) := \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|$ and $\kappa(\boldsymbol{A}) = \left|\frac{\lambda_{\max}}{\lambda_{\min}}\right|$ with $\lambda_{\max}$ being the largest, and $\lambda_{\min}$ being the smallest eigenvalue of $\boldsymbol{A}$. The variant with the eigenvalues is used to compute the condition of a matrix on computers.

[8] The $\boldsymbol{LL}^T$ variant is needed later on for drawing samples from a GMM, so both variants are introduced at the same time.

The $\boldsymbol{LDL}^T$ variant is more stable since the roots of $\boldsymbol{D}$ are not necessary to be calculated while $\boldsymbol{L}$ is calculated. Solving a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ with the Cholesky decomposition[9] works via forward substitution of $\boldsymbol{LD}^{\frac{1}{2}}\boldsymbol{y} = \boldsymbol{b}$ and following back substitution of $\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{y}$.

To speed the algorithm up, it is possible to use diagonal covariance matrices instead of full covariance matrices. Using diagonal covariance matrices helps with noisy or sparse data, too (see [PTVF07, p.847]). If it is known that the data dimensions are decorrelated, it is sufficient to use diagonal covariance matrices: Full covariance matrices contain information about the correlation of the different dimensions, which is zero or at least very small if the data is decorrelated. This is particularly the case for data that was decorrelated using a DCT due to the decorrelating properties of the DCT.

3.3.5 Comparison of GMMs

If a model has been built for the timbre of a song, the timbre representations of songs can be compared by directly comparing the models, instead of consulting the timbre vectors. There exist several approaches to measure the similarity of probability density functions. Let $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$ be two probability density functions that should be compared to each other.

Kullback-Leibler divergence is not a metric, but has nevertheless been used in the past to measure the difference of two probability density functions.

Definition 3.14 (*Kullback-Leibler divergence*):
The *Kullback-Leibler divergence* is defined as

$$\mathrm{KL}(p, q) = \int_{-\infty}^{\infty} p(\boldsymbol{x}) \log\left(\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\right) \mathrm{d}\,\boldsymbol{x}. \tag{3.29}$$

Small values indicate similar PDFs. The Kullback-Leibler divergence is always non-negative (see figure 3.12), with $\mathrm{KL}(p, q) = 0$ iff $p = q$.

It is easy to see the the KL divergence is not symmetric; if $p \neq q$ then $\mathrm{KL}(p, q) \neq \mathrm{KL}(q, p)$; but it can be made symmetric with

$$\mathrm{sKL}(p, q) = \frac{\mathrm{KL}(p, q) + \mathrm{KL}(q, p)}{2}. \tag{3.30}$$

Many researchers use this symmetrized Kullback-Leibler divergence when using it as a measure for similarity. The symmetrized version has all properties of a metric. It is almost never questioned that a “good” similarity measure needs to be symmetric.

[9] The used matrix library, `Eigen`, has a `LDLT` class coming with a `solve()` function for this purpose.

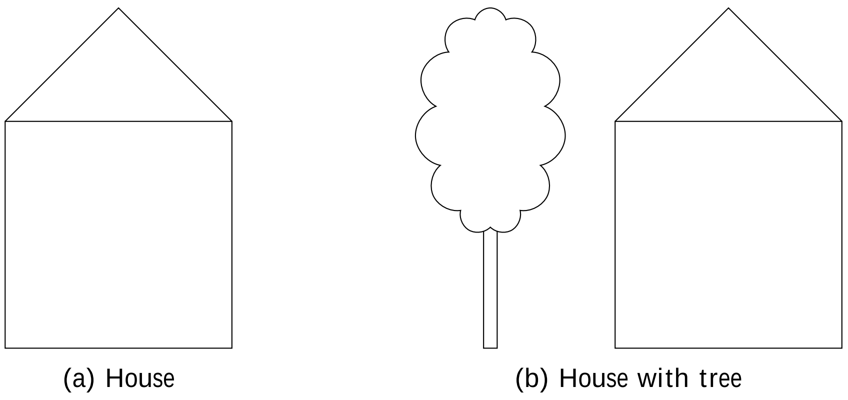


Figure 3.11: Why a similarity measure does not need to be symmetric. Image inspired by [Sch05, p.217]

However, this does not need to be true: similarity does not need to be symmetric (see [Sch05]). Consider the gedanken experiment shown on figure 3.11: The house on the left is more similar to the house on the right with the tree than the other way round; in a database search for the house, one wants to be able to find the house on the right with the tree. The opposite might not be true.

L^p distance is a measure from analysis. It tries to measure "the difference" between functions.

Definition 3.15 (*L^p-norm and L^p-distance*):
Let f, g be integrable functions with $f, g : [a, b] \to \mathbb{R}$ and let $p \in \mathbb{R}$. Then the L^p-norm is defined through

$$\|f\|_p = \left(\int_a^b |f(x)|^p \,\mathrm{d}\,x \right)^{1/p} \tag{3.31}$$

The L^p-distance is defined as the induced metric of the L^p-norm:

$$d_{L^p}(f, g) = \|f - g\|_p \tag{3.32}$$

As it is the case with the Kullback-Leibler divergence, an L^p distance is not a metric in a strict sense, because the L^p-norm is not a norm in a strict mathematical manner. It does not fulfill the needed property $\|f\|_p = 0 \Leftrightarrow f = 0$: If f is not continuous, then the integral in the norm might be zero even if the function is not zero over the domain: It might have a countable number of removable discontinuities[10]. It is nevertheless most

[10] in german: *hebbare Unstetigkeitsstelle*, a discontinuity that can be removed because $f(x_0) \neq \lim_{x \to x_0+} f(x) = \lim_{x \to x_0-} f(x)$, and both limits are finite.

often called a norm, because it is a norm for all continuous functions, and the only caveat is that the implication $\|f\|_p = 0 \Rightarrow f = 0$ is not true. In most cases, $p = 2$ is chosen, as it is equivalent to the common 2-norm for vectors.

Earth Mover's distance (EMD) or *Wasserstein metric* is the most intuitive similarity measure, but at the same time the one where the most implementation effort is needed to calculate it. Consider a bunch of dirt piles A in a region D, and another bunch of piles B. The value of the EMD is the minimum cost of reshaping the dirt piles A to the shape defined by the piles B, where the cost is defined as the amount of dirt moved times the distance. Moving of dirt is allowed; creation and destruction is allowed at high costs, if the amount of dirt in A and B is not equal. This metric has been used in image similarity analysis (see [RTG98]). The EMD can be seen as a continuous version of the *edit distance*.

This very intuitive view can easily be identified with GMMs A and B being transformed to each other. While being very easy to understand, it is nevertheless hard to implement. To calculate the value of the EMD, a transportation problem needs to be solved (see [RTG98]); the buckets are infinitesimal small. In most cases, this would be implemented via a histogram of the input space, thus having a minimal bucket size. While being efficient in low dimensional spaces, this is not the case for high dimensions: The number of needed buckets rises exponential with the number of dimensions. An efficient implementation capable of calculating a value for the EMD for arbitrarily shaped earth piles in high dimensions is nontrivial.

In this thesis, the non-symmetrized version of the Kullback-Leibler divergence has been used to compare two GMMs. [JECJ07] found that the performance of the KL divergence is not much behind the two other possible solutions. Additionally, the KL divergence was designed for comparison of probability densities, in contrast to the other approaches.

The central problem with calculating the Kullback-Leibler divergence is that the GMMs are defined in a multi-dimensional space, thus the integral is multi-dimensional, too. Calculating multi-dimensional integrals is considered to be a hard problem in the general case. For the calculation of the KL divergence, the locality of the normal distributions helps in finding a more efficient implementation. Here, a technique called *Monte-Carlo importance sampling* is used to evaluate the integral. The technique can be adopted to use an L^p-distance.

3.3.6 Monte-Carlo integration with importance sampling

Monte-Carlo integration is a method to numerically integrate a function, and is especially suited for functions in higher dimensions.

There exist many methods for numerical integration of a definite integral. Most methods for functions of dimension 1 are based on a piecewise interpolation of the integrand with polynomials of a low degree, as it is the case with the *Newton-Cotes-formulas*. Probably the best-known Newton-Cotes-formulas are the *rectangle*, *trapezoidal*, or *Simpson rule*, which make use of polynomials of degree 0, 1 and 2 for piecewise interpolation. Their antiderivatives are known, and can thus easily be calculated.

The integration intervals of Newton-Cotes-formulas have equal lengths; more sophisticated methods use an adaptive length of the intervals. These formulas are called *Gaussian quadrature formulas*. Both types of rules can be generalized to the multi-dimensional case via *Fubini's theorem*. It states that, meeting certain prequisites[11],

$$\int_{A\times B} f(x,y)\,\mathrm{d}(x,y) = \int_A \int_B f(x,y)\,\mathrm{d}\,x\,\mathrm{d}\,y \tag{3.33}$$

which means that it is possible to calculate a multi-dimensional integral dimension by dimension. Recursive application of this theorem makes it possible to use the well-studied methods of calculating one-dimensional integrals for higher-dimensional integrals, too. Note that the computational cost grows exponentially with the number of dimensions. This is affordable for low dimensions such as 2 or 3, but not for dimensions from higher orders such as 8 to about 20, as used for the timbre vectors.

Another group of algorithms which can also be used for numerical integration are called *Monte-Carlo* methods. Their use is not limited to numerical integration, but they play an important rule for high-dimensional integrals and often are the method of choice when other methods fail due to computational costs. Monte-Carlo methods are randomized algorithms, which means that they use probabilistic methods to estimate the value of the integral. They can be applied to both one- and multidimensional definite integrals. This part is mainly based on [Gen03, p.230ff], details and proofs can be found there. Consider the one-dimensional integral

$$\int_a^b f(x)\,\mathrm{d}\,x =: \theta \tag{3.34}$$

and a uniformly distributed random variable Y on $[a,b]$. The value of the integral then can be estimated via

$$\hat{\theta} = \frac{(b-a)}{M}\sum_{i=1}^{M} f(y_i) \tag{3.35}$$

with M being the sample count and y_i being numbers distributed according to Y. It can be shown that

$$E(\hat{\theta}) = \int_a^b f(x)\,\mathrm{d}\,x. \tag{3.36}$$

[11] Prequisites are: A, B complete measure spaces, $A\times B$ measurable, $\int_{A\times B} |f(x,y)|\,\mathrm{d}(x,y) < \infty$

This way, it is possible to calculate the integral of a one-dimensional function on the interval $[a, b]$.

This can be generalized to an arbitrary random variable Y with density $p(y)$ over an arbitrary region D. The integral can be rewritten as

$$\theta := \int_D f(x)\,\mathrm{d}\,x = \int_D \frac{f(x)}{p(x)} p(x)\,\mathrm{d}\,x = \int_D g(x)p(x)\,\mathrm{d}\,x \tag{3.37}$$

In this case, equation 3.35 can be generalized to

$$\hat{\theta} = \frac{1}{M}\sum_{i=1}^{M}\frac{f(y_i)}{p(y_i)} = \frac{1}{M}\sum_{i=1}^{M} g(y_i) \tag{3.38}$$

This method now is not restricted to the one-dimensional case anymore; if f is multi-dimensional, the deviates y_i may be drawn from a multi-dimensional random variable. p and g need to be chosen accordingly. Even more, the bounds of the integral may now be infinite, as long as $p(x)$ is a probability density function! This property is crucial for the calculation of the value of the KL divergence, since its bounds are infinite.

The performance of the method can be estimated via the variance of the estimate $\hat{\theta}$, which is

$$\operatorname{Var}(\hat{\theta}) = \frac{1}{M}\operatorname{Var}\left(\frac{f(Y)}{p(Y)}\right). \tag{3.39}$$

Without going into detail, one can see that the derived error margin is a statistical measure and no hard margin, and that it depends on the function itself, as well as on the chosen probability density $p(x)$. Additionally, the error decreases with $\mathcal{O}(\frac{1}{\sqrt{M}})$, since it will be based on the standard derivation instead of the variance. [Gen03, p.242] proves that the error is minimal when

$$p(x) = \frac{|f(x)|}{\int_D |f(x)|\,\mathrm{d}\,x} \tag{3.40}$$

is achieved, so it can be expected that $p(x)$ is of good quality as long as it follows the rule of thumb "$p(x)$ is roughly proportional to $f(x)$". That is basically the idea behind Monte-Carlo importance sampling: To integrate a function, a Monte-Carlo method is applied. Where $f(x)$ is large and therefore the impact on the value of the integral is large, $p(x)$ should be large, thus more samples are taken into account. That way, the variance of the estimate is reduced when compared to the case of a uniformly distributed random variable Y, resulting in a smaller error margin. Within this context, $p(x)$ will be called the *importance function* because it indicates which regions of f are important.

Now, the Monte-Carlo importance sampling technique will be applied to calculate the Kullback-Leibler divergence. One important parameter of the Monte-Carlo method is the importance function $p(x)$. Since it is possible to mix up the importance function with the first parameter of the Kullback-Leibler divergence, the importance function will be called $c(x)$ from now on.

If equation 3.29 and 3.38 are combined, one gets

$$\mathrm{KL}(p, q) \approx \hat{\theta} \tag{3.41}$$

$$= \frac{1}{M} \sum_{i=1}^{M} \frac{p(y_i) \log \left(\frac{p(y_i)}{q(y_i)} \right)}{c(y_i)}, \tag{3.42}$$

and some terms vanish if $c(x) = p(x)$ is chosen:

$$\hat{\theta} = \frac{1}{M} \sum_{i=1}^{M} \frac{p(y_i) \log \left(\frac{p(y_i)}{q(y_i)} \right)}{p(y_i)} \tag{3.43}$$

$$= \frac{1}{M} \sum_{i=1}^{M} \log \left(\frac{p(y_i)}{q(y_i)} \right) \tag{3.44}$$

It is arguable that $c(x) = p(x)$ is not optimal with respect to the error margin because $q(x)$ does not have any influence on $c(x)$. Nevertheless, the KL-divergence is dominated by $p(x)$: The area to be integrated is large where $p(x)$ is large, and is small where $p(x)$ is small, so it follows the previously stated rule of thumb. Since the KL divergence cannot be negative, the influence of the negative log-values cannot be large; and the log-values are negative where the influence of $q(x)$ is greater than the influence of $p(x)$. Therefore, choosing $c(x) = p(x)$ is reasonable. See figure 3.12 for a visualization.

Thus, $\hat{\theta} = \frac{1}{m} \sum_{i=1}^{M} \log \left(\frac{p(y_i)}{q(y_i)} \right)$ is a good estimate for $\mathrm{KL}(p, q)$. Since the error decreases with $\mathcal{O}(\frac{1}{\sqrt{M}})$, a choice of $M = 100$ leads to an accuracy of about 10%, which should be a good compromise between computational speed and accuracy.

3.3.7 Drawing samples from a GMM

Drawing samples from a GMM involves the need to draw samples from a normal distribution, which is then expanded to the multidimensional case. Normal pseudorandom number generation on standard PCs is carried out using a *Linear Congruency Generator* (LCG). For instance, the C/C++ `rand()` function is implemented with this technique. These pseudorandom numbers are uniformly distributed[12].

The step from unifromly distributed numbers to deviates from a normal distribution is done with the *Box-Muller transform*. The algorithm and details are presented in

[12]Pseudorandom numbers generated with an LCG are weak in a cryptographic sense and should not be used for applications where security is of importance.

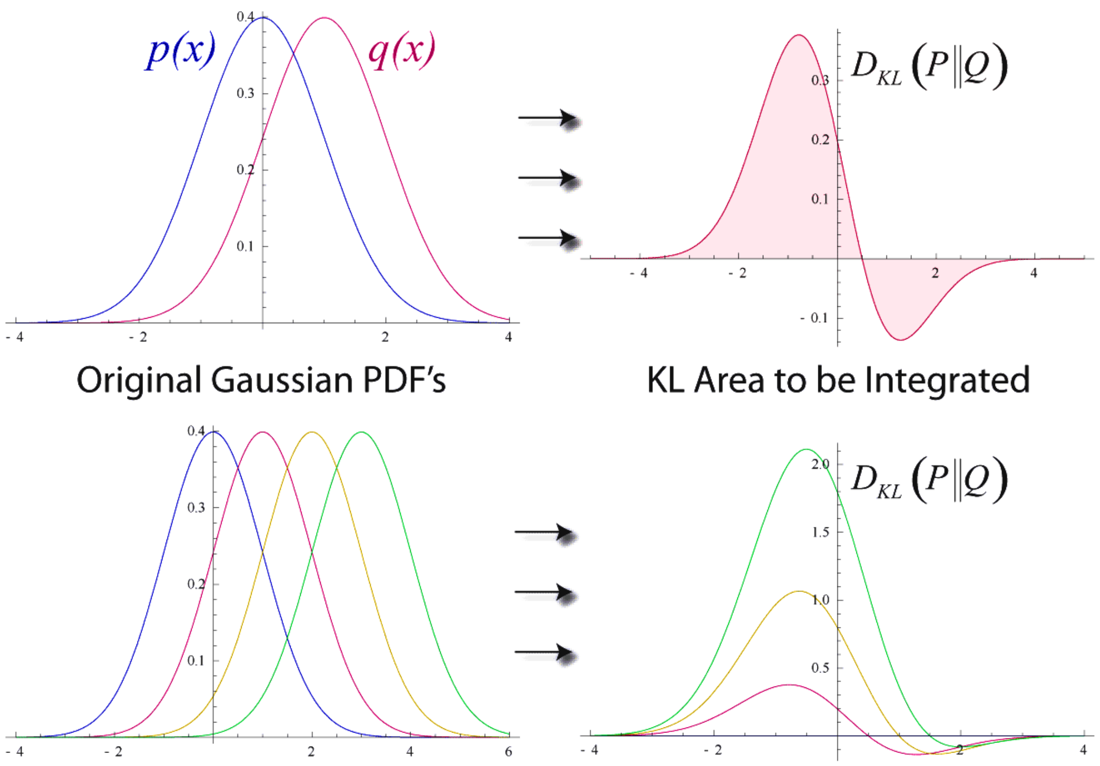


Figure 3.12: Example of the Kullback-Leibler divergence applied to some Gaussian distributions. Note that the largest contribution to the value of the divergence comes from the part where p is large, especially when p and q are dissimilar.
Picture taken from `http://en.wikipedia.org/wiki/File:KL-Gauss-Example.png` and slightly modified (color filters), the picture is licensed under `CC-by-sa-3.0` and was originally created by T. Nathan Mundhenk.

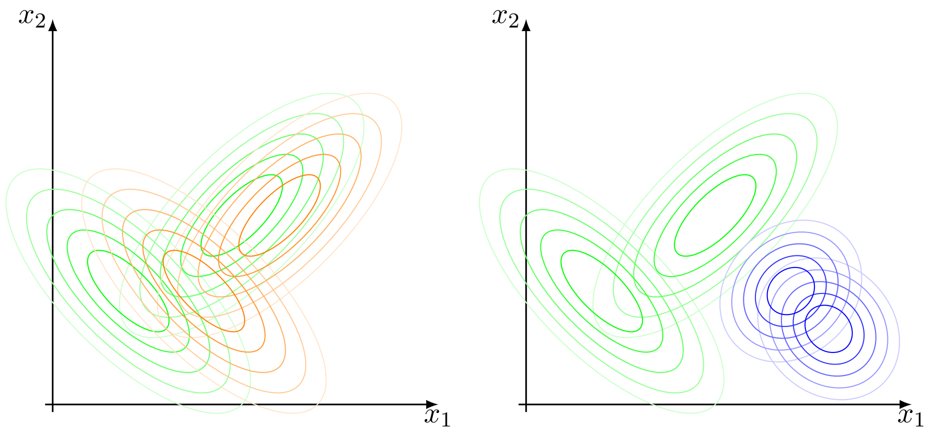


Figure 3.13: Comparing two Gaussian Mixture Models: finding a similarity measure. The green and orange mixtures should be more similar than the green and blue mixtures.

[PTVF07, pp.362] and will not be repeated here. The main idea is to use the inverse function of $F(y) = \int_0^y p(x)\,\mathrm{d}\,x$. $F(y) \in [0,1]$ for all y due to the normalization property of a PDF (see equation 3.13). $F(y)$ increases monotonically since $p(x) \geq 0$ for all x. So, $F^{-1}(y)$ is distributed according to $p(x)$ if y is distributed according to a uniform distribution on $[0,1]$.

To create samples from a multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, we make use of a Cholesky decomposition of $\boldsymbol{\Sigma}$ as defined in definition 3.12 on page 38, namely the $\boldsymbol{L}\boldsymbol{L}^T$ variant of the Cholseky decomposition.

Let $\boldsymbol{x}$ be a vector of normally distributed numbers, calculated through the *Box-Muller transform*. Let $\boldsymbol{L}$ be as defined in definition 3.12 (through $\boldsymbol{L}\boldsymbol{L}^T = \boldsymbol{\Sigma}$). Then

$$\boldsymbol{y} = \boldsymbol{L}\boldsymbol{x} + \boldsymbol{\mu} \tag{3.45}$$

is distributed according to the normal distribution with parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. For details and a proof, see [PTVF07, pp.378].

The last step is to draw from a GMM instead of a multivariate normal distribution. This is achieved through first drawing one normal distribution out of the GMM, according to the weights w_i. A distribution with weight w_i is drawn with probability w_i. To do so, a sample s from a uniform distribution in $[0,1]$ is used. Since $\sum_{i=0}^{I} w_i = 1$, $\min_{J \leq I}(\sum_{i=0}^{J} w_i \geq s)$ is the index of the normal distribution that should be drawn. Once the normal distribution is chosen, a deviate as described before is generated.

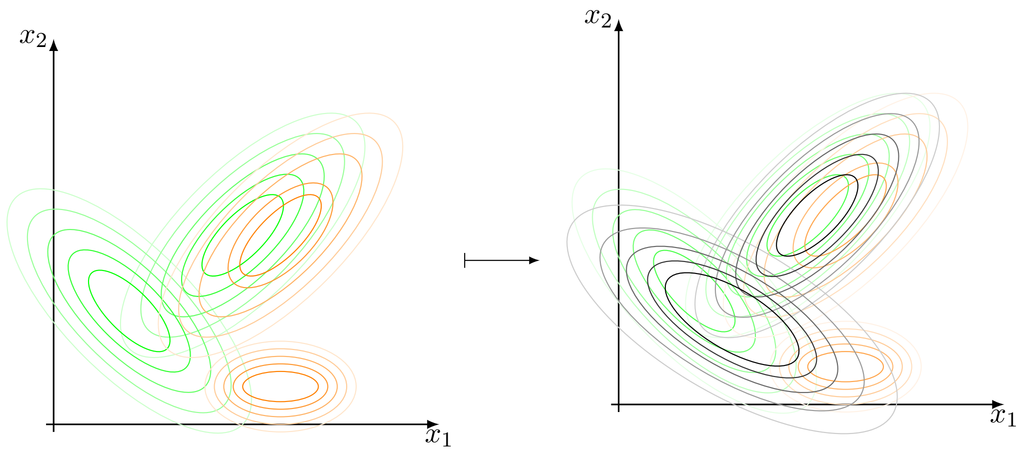


Figure 3.14: Building a new model out of two others; the number of Gaussian distributions for the category model is the same as the number of Gaussian distributions for the song model. This does not need to be true in the general case.

3.3.8 Building a model for multiple recordings

Up to now, the similarity of two recordings can be compared by comparing their models. Since the goal of this thesis is to build personal music categories and compare a recording with a category, a technique needs to be developed to build a model for a category and compare a recording with that category instead of a single song. To achieve this, the approaches for song-level models are reused: The category-level model is built from the song-level models through sampling. Here, the same technique is used as it is to compare two models: Monte-Carlo importance sampling. Many samples (e.g. 20.000 in total) will be drawn from the timbre models of the given recordings. The resulting vectors will be used to learn a new model with the EM algorithm. The new model is a representation of all recordings. The approach is roughly equivalent to re-creating the timbre vectors of all recordings from their timbre models (see [AP02, p.2]) and then building a new model from all timbre vectors. This model represents the timbre of all recordings. See figure 3.14 for a visualization.

It is possible to choose a different number of M Gaussian distributions for the category model than was used for the song model (N). Usually, M is chosen with $M \geq N$, since it does not make sense to have less information in the category-model than in the song-model. With $M = N$, the category-model is forced to generalize from the recordings if the category model is learned from more than one song. $M = 3N$ was found to be a reasonable good choice, since it allows (in theory) 3 songs to be exactly represented, and for more songs, generalization is not restricted to the same complexity level as the song model.

3.4 Extracting chords

Extracting notes from the signal of a musical piece is a challenging task due to overtones and noise that are also part of the signal. Although the Constant Q transform extracts the signal strength of geometrically spaced frequency bins that fit to musical notes, it does not eliminate the need for noise-cancelling techniques to extract the notes and chords, especially if the task was to transcribe the whole piece and especially if percussive instruments are present in the recording. For the simpler task of basic (i.e. only major/minor) chord extraction, it is possible to use some simpler techniques.

First, an introduction to some basic music theory will be given, to the extent that is needed to understand the algorithms presented. After that, some algorithms will be shown to address the problem of chord extraction of an audio signal.

3.4.1 Basic music theory: notes, intervals, chords

This section is mostly based on [Zie00].

Music consists of *notes*. They are the smallest building blocks of music. For non-percussive instruments, notes are associated with a pitch. All instruments associate time intervals (durations) with notes. The durations are relative to one (e.g. an eighth note). Notes are grouped in *measures*, whose lengths and beat positions define their names (e.g. a measure of 4/4, 3/4 or 6/8; 3/4 and 6/8 are differentiated through their beat positions).

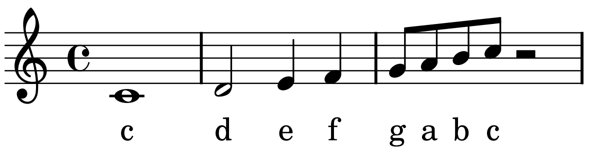


Figure 3.15: A `C` (C major) note scale with measure 4/4, different note lengths and a half rest.

An *interval* is the tone height difference between two musical notes. The difference will be measured in half steps, although the names of the intervals stem from the steps in the musical scale; e.g. an interval of 4 half steps will be called a *major third*, 3 half steps will be called a *minor third*, both are called *third*s in short.

If multiple notes with different pitches sound at the same time, it is called a *chord*. Every chord has a *root* after which it is named, and a *mode*. There are mainly two modes of chords: *Major* and *minor*. Both consist of at least 3 notes, but the intervals differ. Major chords are composed of the root with first a major third (4 half steps from the root), followed by a fifth from the root (7 half steps from the root). Note that major

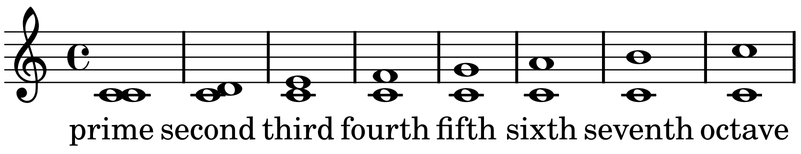


Figure 3.16: Intervals of C up to an octave with notes of the scale. The intervals are, in order of occurrence, (n/a), major, major, minor, major, major, major, minor.

chords sometimes are described as a stack of a major third and a minor third from the second note; this is just a change of perspective ($7 = 4 + 3$).

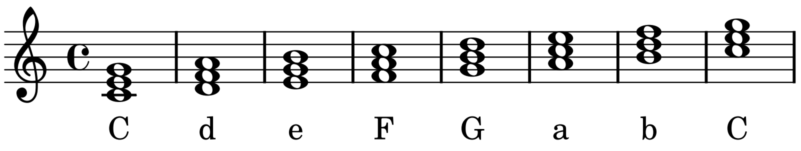


Figure 3.17: Chords of the C scale.

Minor chords exchange the major third by a minor third (3 half steps from the root), the fifth will not be changed; this results in a stack of a minor third and a major third from the second note. Minor and major chords are called *triads* because they can be seen as stacks of thirds. Chords with 3 notes but any combination of intervals are called *trichords*.

A *scale* is a defined series of ascending notes. There are many different scales, most of the western music scales origiate from the so-called *church modes*. The by far most popular church modes are *major*, *natural minor* and *harmonic minor*. A major scale is described in figure 3.15. Natural minor a consists of the same notes as major C, but the scale starts and ends in a instead of c. Every major scale has a *parallel minor* with its root located a minor third below the root of the major scale and the same notes as the major scale. The chords used for major and parallel natural minor scales are the same.

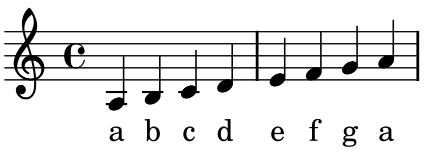


Figure 3.18: The a natural minor scale

In a *harmonic minor* scale, the seventh note of the scale is raised by a half tone. This leads to the fifth chord of the scale being a *major* chord instead of a minor. The other chords are not affected! All other chords originate from the parallel major scale. It is

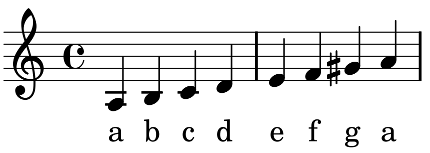


Figure 3.19: The `a` harmonic minor scale

possible that the chords of a harmonic minor scale are used together with the notes of the natural minor scale in one musical piece.

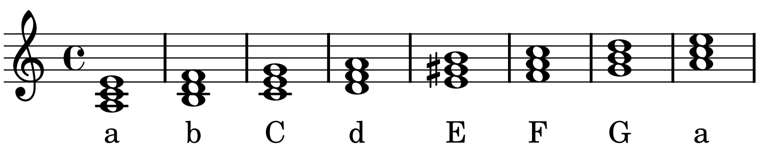


Figure 3.20: Chords from the `a` harmonic minor scale

There are different systems to write the mode of a chord, if musical notation with notes is not used or not available. In this thesis, chords will be written with typewriter letters: The root will be written in lowercase if the mode of a chord is minor (e.g. `d` for d minor), and with uppercase if it is a major chord (e.g. `F` for F major). Notes will be noted bold (e.g. **f**). If it is necessary to name the correct pitch of a note, the *scientific pitch notation* will be used (see [You39]). Within this pitch notation, the chamber pitch of 440Hz is called **a4**.

Triads may be extended with additional notes, thus having 4 or 5 notes building the chord. These chords are named after the the interval of the additional note(s) with the root. Examples: `a` with an additional seventh will be called `a7`. `C` with an additional nineth will be called `C9`.

It is possible that the root of a chord is not the lowest note. Such a chord is called an *inversion.*

3.4.2 Chord estimation

Chords are not always played by one instrument. Sometimes, instruments are used in conjunction to create a chord, and sometimes a chord spans multiple octaves. An algorithm for chord estimation should cover all these cases.

In the literature, many approaches to chord estimation make use of *chroma* features (see [TVM$^+$11, p.28]). It is a short-time low-level feature which describes the harmonies of a musical piece at a point in time.

Definition 3.16 (*Chroma vector*):
The *chroma bin* is defined as

$$c(b,t) = \sum_{p=0}^{P-1} |\boldsymbol{X}^{\mathrm{CQ}}(b+12p,t)| \tag{3.46}$$

where P is the number of octaves in the constant Q transform, b is one bin in an octave, and t is a point in time. The *chroma vector* is the vector of chroma bins

$$\boldsymbol{c}(t) = \begin{pmatrix} c(1,t) \\ c(2,t) \\ \vdots \\ c(12,t) \end{pmatrix}. \tag{3.47}$$

Effectively, the chroma feature is a sum of one note over all octaves, repeated for all notes in an octave: It is a mapping of all octaves to one octave. It can be defined without relying on the Constant Q transform (see [TVE08, p.21, 3.3.3]). Sometimes, it is called the *pitch class vector* because it summarizes notes with the same pitch. See figure 3.21 for a graphical example of some chroma vectors.

The chroma vector contains all necessary information to extract the root and mode of a chord. Some information is lost: it is not possible to extract the inversion of a chord from the chroma. This information however is of minor importance and thus can be dropped without having large influence.

Since the Constant Q bins also contain noise and overtones, a decision for one chord cannot be made using fixed thresholds (e.g. by comparing with a fixed chord example vector). Instead, a scalar product is calculated with a chord template vector t_i which marks the characteristic notes of a chord with 1 and all others with 0. There are 24 chord template vectors; one each for major and minor chords and for each of the 12 possible root notes. Examples (assumed the first note is **f**[13]):

$$\boldsymbol{t}_1 = \mathtt{F} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

$$\boldsymbol{t}_{13} = \mathtt{f} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

The vectors for the remaining 11 roots can be calculated via circular rotation of these two examples. The 24 scalars arising from the multiplication of $\boldsymbol{c}(t)^T \boldsymbol{t}_i$ can be compared directly, the vector with the maximum value is considered the most likely chord template. See algorithm 4 for details.

[13] The vectors start with **f** naturally as the highest note with frequency lower than 11025Hz is **e9** with a pitch of 10548.1Hz; **f9** has a pitch of 11175.3Hz. The highest octave thus is the one from **f8** to **e9**.

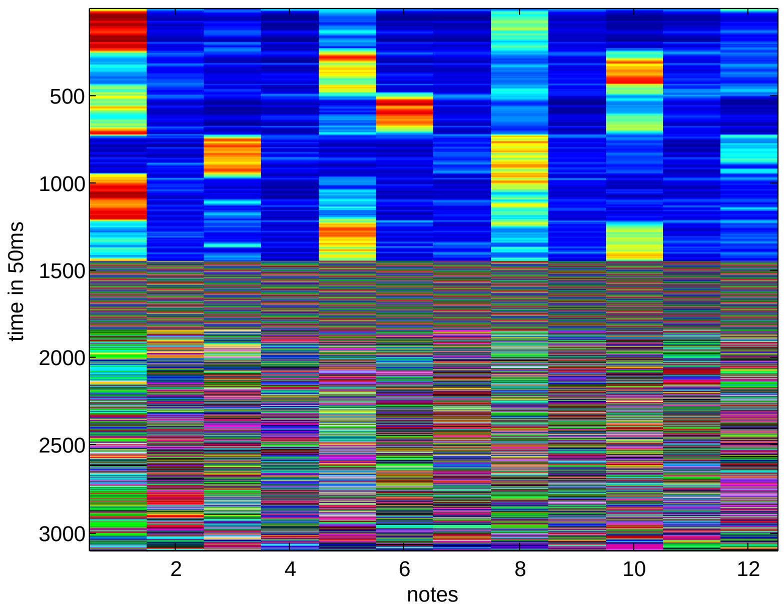


Figure 3.21: Plot of the chroma vectors of the file `mixture-all.mp3`, found in the `testdata` directory of `libmusic`. Note 1 is **g**, note 12 is **f**$\#$. The chord progression is `G`, `e`, `C`, `D`, the key of the recording is `G`.

Algorithm 4 works in many cases, but the correct chord is not always the one with the largest peak in the chroma vector, especially for real-world examples. With the chord test data in the folder `testdata/chords/` of `libmusic`, this algorithm works correctly (e.g. identifies the correct chord as the one with the largest peak). This is not the case with more noisy real-world examples, where you can find e.g. percussive instruments. The main problem is that the chroma vectors are not qualitatively stable between two subsequent calls to the chord estimation function when noisy instruments are used. Intuitively, the chroma vectors should not change qualitatively in very short amounts of time, as it does not make sense to make chord progressions faster than e.g. every 200ms.

A refined algorithm should make use of a noise-cancelling technique for the calculation of the chroma vectors. There are some simple techniques available that both fit to the problem and are easy to calculate, one of them being *exponential smoothing*. In communications engineering, this technique is called a *recursive filter of order 1*. It is a low-pass filter that can be described as a weighted moving average with infinite history and exponentially decreasing weights. In contrast to a moving average, it is not necessary to save past samples to calculate it. Let $x(t)$ be the input signal, $\alpha \in [0, 1[$, $t > 0$ and $s(t)$ the smoothed output, which is defined by

$$\boldsymbol{s}(t) = (1 - \alpha)\boldsymbol{s}(t-1) + \alpha\boldsymbol{x}(t) \tag{3.48}$$

Algorithm 4 Chord detection from the Constant Q transform result at a given time t, first version

```
Precondition: $\boldsymbol{X}^{\mathrm{CQ}}(b,t)$ is the Constant $Q$ transform result of bin $b$ at time $t$ with a
    total bin count of $B$ and octave count of $\bar{O}$.
1: Let $\boldsymbol{c}$ be a vector of dimension $B$                ▷ $\boldsymbol{c}$ is the chroma vector
2: for bin $b$ from 0 to $B-1$ do                               ▷ calculate chroma vector $\boldsymbol{c}$
3:     $c[b] \leftarrow 0$
4:     for octave $o$ from 0 to $\bar{O}-1$ do
5:         $c[b] \leftarrow c[b] + \boldsymbol{X}^{\mathrm{CQ}}(b + o \cdot B, t)$
6: Let $\boldsymbol{d}$ be a vector of dimension $2B$              ▷ $\boldsymbol{d}$ is the chord likelihood vector
7: Let $\boldsymbol{t}_i$ be the chord template vectors as described beforehand.
8: for $i$ from 0 to $2B-1$ do                                  ▷ calculate chord likelihood vector
9:     $\boldsymbol{d}[i] \leftarrow \boldsymbol{c}^T \boldsymbol{t}_i$
   return $\arg\max_{i \in [0,2B-1]}(\boldsymbol{d}[i])$
```

with

$$\boldsymbol{s}(0) = \boldsymbol{x}(0). \tag{3.49}$$

It can be used for both scalars and vectors. It can be used to stabilize the chroma vectors. To achieve this, algorithm 4 is changed to first calculate all smoothed chroma vectors and then calculate all *chord likelihood vector*s. Additionally, lists L_c and L_d are returned. They are used later on for building chroma models and musical key estimation. The resulting algorithm is better suited for real-world examples with percussive instruments. Its performance depends on the value of α and on the recording: If it is too large, noisy instruments can destroy the vectors. If it is too small, fast chord progressions cannot be captured. The algorithm introduces a delay, but since the vectors are not used for real-time transcription, this is acceptable. The delay is smaller with large α. Algorithm 5 is more stable than algorithm 4 in real-world applications, but the factor α needs to be tuned. It seems natural to choose $\frac{\alpha}{t_s}$ constant, since this would lead to a behaviour that is (mostly) independent of t_s, which seems to be desirable.

There are some chord progressions that cannot be found with this algorithm, such as *arpeggio*, which is a chord broken into its component notes which are then played in fast sequence[14] (see figure 3.22). On a piano, arpeggio chords might be recognized correctly due to the relatively long duration of the notes; arpeggios of an instrument like a harp can be impossible to recognize with this algorithm – it depends on the setting of α and on the piece of music.

Additionally, there are chords with the risk of misclassification which are not composed of three, but e.g. 5 notes, stacked as thirds: For example `Cmaj9`, a C major chord with

[14] Listening examples: *Beethovens Moonlight sonata*, *Bliss* by *Muse* (first 15 seconds), or *Arpeggio* by *Friska viljor* (chorus, e.g. seconds 41-90)

Algorithm 5 Chord detection from the Constant Q transform result at a given time t, second version

```
Precondition: $\boldsymbol{X}^{\mathrm{CQ}}(b,t)$ is the Constant $Q$ transform result of bin $b$ at time $t$ with a
    total bin count of $B$ and octave count of $\bar{O}$.
 1: Let $\boldsymbol{c}$ be a vector of dimension $B$                    ▷ $\boldsymbol{c}$ is the chroma vector
 2: Let $L_c$ an empty list data structure which preserves ordering
 3: $\boldsymbol{c} \leftarrow 0$
 4: for time $t$ from 0 to $T$ in steps $t_s$ do
 5:     $\boldsymbol{c} \leftarrow \boldsymbol{c} \cdot \alpha$
 6:     for bin $b$ from 0 to $B-1$ do               ▷ calculate smoothed chroma vector $\boldsymbol{c}$
 7:         for octave $o$ from 0 to $\bar{O}-1$ do
 8:             $c[b] \leftarrow c[b] + (1-\alpha)\boldsymbol{X}^{\mathrm{CQ}}(b + o \cdot B, t)$
 9:     add $\boldsymbol{c}$ to $L$
10: Let $\boldsymbol{d}$ be a vector of dimension $2B$             ▷ $\boldsymbol{d}$ is the chord likelihood vector
11: Let $L_d$ an empty list data structure which preserves ordering
12: for every element $\boldsymbol{c}$ in $L_c$ in order of insertion do
13:     for $i$ from 0 to $2B-1$ do                     ▷ calculate chord likelihood vector
14:         $\boldsymbol{d}[i] \leftarrow \boldsymbol{c}^T \boldsymbol{t}_i$
15:     add $\arg\max_{i \in [0,2B-1]}(\boldsymbol{d}[i])$ to $L_d$
    return $L_c$, $L_d$
```

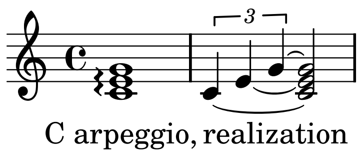


Figure 3.22: Arpeggio: notation and possible realization

major seventh and additional nineth, can at least be misinterpreted as `Cmaj7`, `C`, `e` or `G` (see figure 3.23). `Cmaj7` can itself be interpreted as an inversion of `e6`; this chord is composed of the same notes. The chords are not different; their different name stems from the surrounding chords and the key of the piece. To be sure to not misclassify these chords, a strong knowledge about the global musical structure of the piece is needed, the decision cannot be made solely on the local chord structure. Even if the piece was transcribed, these misclassifications are possible.

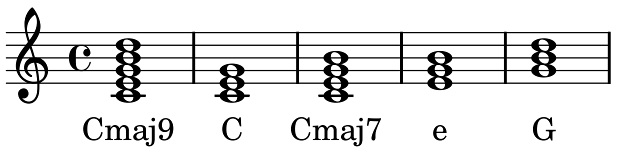


Figure 3.23: Chords of the `C` scale that can be misinterpreted by the algorithm.

The chords themselves do not help that much when trying to compare music pieces[15], but it is possible to derive measures for the chord complexity, which can be used to characterize a recording. The *chord complexity similarity* or *key-invariant chroma model* will be derived in the next section. The idea is to capture the generic structure of the chords used in a recording.

3.5 Key-invariant chroma models

Chords and their progressions are not easy to compare if the key of a musical piece is unknown, because the resulting chroma vectors or chords are different even if they have the same structural meaning (e.g. `a` has the same structural meaning in key `C` as `e` has in key `G`). Given the key, chords can be shifted to a key-invariant scale which avoids this problem. This process is called *roman numeral analysis*[16] because it uses roman numbers for the notation of the chords.

Given the key of the piece, the roman numbers identify the root note relative to the key and the mode of the chord that should be played, e.g. if the key of a musical piece is

[15]Some researchers use chord progression schemes for cover song identification, see [FTZ11, p.307]. This usage is beyond the scope of this thesis.
[16]dt. *Stufenanalyse*/ *Stufentheorie*

C, the chord C is noted as I, whereas G is noted as V (see figure 3.17 and 3.24). The chords I and V are called *tonic* and *dominant* and play an important role in the recognition of the key of a recording.

If the key was G, then I would be identified with G and V with D. The mode of the chord depends on the key: for major keys, chords I, IV and V are major, II, III and VI are minor; VII is called *diminished* because the fifth is diminished (it consists of 6 half tones instead of 7). For minor keys, chords I and IV are minor, III, VI and VII are major; II is diminished and V may be either minor or major, that depends on the type of the minor key (*natural* or *harmonic* minor).

For major and natural minor scales, the chords are the ones that can be built from the notes of the scale; knowing the root fixes the other notes and the mode of the chords. Harmonic minor keys rise the seventh note about one half tone[17].

Transposition and even a mode change of a piece that is noted in the roman numbers notation is achieved by just changing the key. This scale makes it possible to directly compare the chord progressions of two musical pieces with a different key. This is the

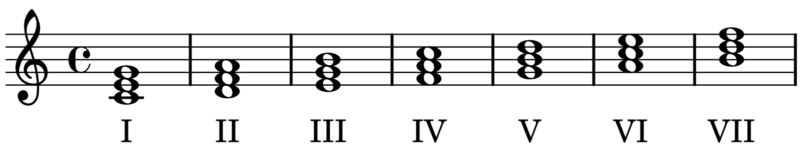


Figure 3.24: Notation of all chords from key C with roman numbers. Compare this figure with figure 3.17 for the names of the Chords.

main idea of key-invariant chroma models: The algorithm tries to eliminate the key from the chroma vectors, thus making them comparable in a better way. This reflects some basic properties of music theory: There exist chord progressions that are used more oftenly than others (e.g. IV–V–I), and some progressions are typical for some kind of music. If progressions can be recognized, or at least the chords used, this can be used as one input for the music classification algorithm.

To achieve this, the key of a recording needs to be estimated, which will be done in the next section.

3.5.1 Estimation of the key of a recording

In the literature, many researchers use the chroma together with a *hidden Markov model* to estimate the key (see e.g. [Pee06] or [NS06]). Their results are promising (recognition rates of 84-93% in [Pee06]). Nevertheless, in this thesis an approach with Gaussian mixture models is used. The main reason for this decision is that the implementation

[17] This has several implications that go beyond the scope of this thesis. The most prominent and relevant effect is the chord V being major instead of minor.

of the timbre estimation with the Gaussian mixture model approach was finished and tested, and the idea arose to simply apply the same idea to the chroma vectors.

The idea for the algorithm stems from the observation that in most recordings[18], chords `I` (tonic) and `V` (dominant) are present. All other chords of the scale may be used or not, there is no need for them to be present. A missing dominant is unusual, and it is quite impossible to find a musical piece where the tonic is missing. It is unusual for pieces to use chords that are not composed of notes from the scale[19]. Merging all these observations with the ideas from the chord recognition algorithm leads to algorithm 6.

Algorithm 6 Key estimation of a musical piece based on the detected chords

```
Precondition: $L_d$ is the result of algorithm 5    ▷ $L_d$ is the the list of maximum
    likelihood chords (list of scalars, e.g. $L_d = \{0, 5, 2, 22, \ldots\}$)
1: Let $\tilde{\boldsymbol{d}}$ be a vector of dimension $2B$.    ▷ $\tilde{\boldsymbol{d}}$ will count the detected chords.
2: $\tilde{\boldsymbol{d}} \leftarrow \mathbf{0}$
3: for every element $d$ in $L_d$ do
4:     $\tilde{\boldsymbol{d}}[d] \leftarrow \tilde{\boldsymbol{d}}[d] + 1$
5: Let $\tilde{\boldsymbol{k}}$ be a vector of dimension $3B$    ▷ $\tilde{\boldsymbol{k}}$ is the key likelihood vector
6: Let $\boldsymbol{k}_i$ be the key template vectors as defined in the surrounding text
7: for $i$ from $0$ to $3B-1$ do $\tilde{\boldsymbol{k}}[i] \leftarrow \boldsymbol{k}_i^T \tilde{\boldsymbol{d}}$
   return $\arg\max_{i \in [0, 3B-1]}(\tilde{\boldsymbol{d}}[i])$    ▷ return the key with maximum likelihood
```

First, the counts of the minor and major chords will be calculated. This leads to a vector $\tilde{\boldsymbol{d}}$ of dimension $2B$. Then, similarly to the $\boldsymbol{t}_i$ in algorithm 5, vectors $\boldsymbol{k}_i$ are defined that will be used to calculate a *key likelihood vector*. In difference to the $\boldsymbol{t}_i$, the vectors $\boldsymbol{k}_i$ will not hold binary, but weighted data. The tonic and dominant will get values larger than the values of the other possible chords to ensure their presence in the original chord

[18] The statements in this part relate to western tonal music; atonal music does not follow these rules.

[19] special case: change of key during the piece, which is not that unusual and cannot be recognized by the proposed algorithm without changes. Application of the algorithm to parts of a recording has been tried, but did not work well.

count vector $\tilde{\boldsymbol{d}}$. The tonic will get the largest value due to its relative importance.

$$\begin{array}{rcccccccccccccl}
\text{note name} & & \text{f} & \text{f}\# & \text{g} & \text{g}\# & \text{a} & \text{a}\# & \text{b} & \text{c} & \text{c}\# & \text{d} & \text{d}\# & \text{e} & & \text{mode} \\
\boldsymbol{k}_1 = & (& I_t & 0 & 0 & 0 & 0 & 1 & 0 & I_d & 0 & 0 & 0 & 0 & & \text{major} \\
 & & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 &)^T & \text{minor} \\
\boldsymbol{k}_{13} = & (& 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & & \text{major} \\
 & & I_t & 0 & 0 & 0 & 0 & 1 & 0 & I_d & 0 & 0 & 0 & 0 &)^T & \text{minor} \\
\boldsymbol{k}_{25} = & (& 0 & 0 & 0 & 1 & 0 & 0 & 0 & I_d & 1 & 0 & 1 & 0 & & \text{major} \\
 & & I_t & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 &)^T & \text{minor}
\end{array}$$

with I_t being the importance value of the tonic and I_d being the importance value of the dominant. Values $I_s = 3.0$ and $I_d = 1.5$ produced comparably good results. $\boldsymbol{k}_1$ is a vector for major keys, $\boldsymbol{k}_{13}$ for natural, and $\boldsymbol{k}_{25}$ for harmonic minor keys. With the given vectors, it is possible to recognize `F` and `f` keys. For the other keys, row-wise circular shifting of the appropriate vectors leads to the needed vectors; e.g. for recognition of `F#`, we need to circularly shift the row of the major chords of $\boldsymbol{k_1}$ one step to the right (feeding the value of old major-row-**b** to the position of old major-row-**c**). The row of the minor chords of $\boldsymbol{k_1}$ needs to be treated in the same way.

The key likelihood vector $\tilde{\boldsymbol{k}}$ will thus be $3B$-dimensional with its entries calculated as the scalar product $\boldsymbol{k}_i^T \tilde{\boldsymbol{d}}$. The algorithm then returns the index of the maximum value, which identifies the key of the recording. Values in $[0, 11]$ indicate a major key, values in $[12, 35]$ indicate minor keys. The root of the key can be calculated as the return value modulo 12.

There are some circumstances where the algorithm will fail:

- Atonal music (music with no key at all)
- Music that makes use of more than one key. If the change of key only lasts for a few moments, the original key might be correctly identified.
- Music that uses harmony differently (e.g. trying to not use the tonic, for whatever reason)
- Highly percussive music

At the moment, it is not possible to distinguish between these cases, and it is not possible to decide wether the algorithm was successful or not. Probably, there are more cases where the algorithm fails. The case of atonal or percussive music can be neglected, because the algorithms in this thesis were not designed for these cases and at least atonal music is not commonly listened to. A suggestion for an algorithm that should not fail for songs with key changes would be to estimate the key locally instead of globally.

3.5.2 Making the chroma keyinvariant

By making the chroma keyinvariant, one tries to make chords from different keys comparable. Both minor and major keys use (nearly) the same chord progressions, so using the roman numbers, one would be able to compare all major with major, minor with minor and major with minor keys. But using the roman numbers directly leads to other problems:

- Using pure roman numbers, one is not able to capture chords that are out-of-place, e.g. `D` in a `C` key.
- It is possible to note chords such as `C7` with roman numbers (e.g. `C7` is $\texttt{I}^{\texttt{7}}$). To recognize these chords directly, the recognition of triads and more complex chords like `Cmaj7` needs to be comparable, and one needs to distinguish between `C` and `C7`, which is not an easy task because they use a different number of notes.
- There might be naming issues with the chords. Example: `Cmaj7` and `e6` use the same notes, which would be noted as $\texttt{I}^{\texttt{maj7}}$ and $\texttt{III}^{\texttt{6}}$ in key `C`. The name depends on the musical context.
- There are many different possible chords and chord variants, and for every variant, there would arise the need for a special rule. To get to these rules, special and deep knowledge of music theory is needed.

In order to avoid these problems, the roman numbers are not used directly. Instead, the idea of key invariance is applied to the chroma vectors: The recognized root note of the key will be used to shift all chroma vectors of all recordings to the same scale (using a circular shift of the elements), regardless of the mode (major or minor). This leads to chroma vectors that cover the structure of the musical piece relative to the key.

The resulting chroma vectors can directly be compared if the mode (major or minor) of the key is identical. If the mode of two chords is different, one note is shifted about one half step. The same applies to the chroma vectors of the chords: two bins are exchanged; the differing bins are neighbours. This is a problem for comparison because with naïve approaches to comparision, it cannot be distinguished between the case of a fifth with additional random note and a minor if compared with a same-root major chord chroma vector. Developing a feasible solution to this issue is beyond the scope of this thesis.

In this thesis, recordings with major and minor keys are comparable up to the uncertainty of the third.

3.5.3 Creating a model for chroma

The creation of the chroma models is equivalent to the creation of the timbre models from the point where the timbre vectors are known: A Gaussian mixture model is trained with the keyinvariant chroma vectors. Two mixture models will be compared through the non-symmetric Kullback-Leibler divergence. A model for a category will be trained with samples calculated from the mixture model of the recording to reduce the persistent memory requirements.

4 Classification

In the classification step, the extracted features will be used to train classifiers – one for every category defined. The classifiers will then be used to classify recordings. A music category is not restricted to e.g. a musical genre. It can be defined through

- a genre
- different instrumentation
- the context of the user
- personal selection
- ...

Ideally, there should be no restriction on how category is formed. Once a category is defined (e.g. through a set of examples and counter-examples), the classifier should give a list of recommendations based on the category.

4.1 Requirements for the classifier

The goal is to design a software that helps users to explore their music databases. A user should be able to define categories of music. The procedure for "defining a category" is crucial for the choice of a classifier. Ideally, the usage of the software should feel natural to the user, he should not be constrained by the underlying classifier in the way he defines the categories. For instance, there should not be the need to first choose hundreds of songs that fit to a category before output is generated. Additionally, it should be possible to choose some counter-examples: It is always possible that the classifier incorrectly labels some songs, and there should be a possibility to remove these files from the list of recommendations. The need for some files to be absent in the list of recommendations should ideally have an influence on other songs to be absent, too; otherwise it would be sufficient to create a blacklist of files for every category. Nevertheless, it should not be *needed* to supply the classifier with negative examples before output is generated, since that might feel unnatural. Counter-examples should be a feature that can be used additionally, and omitted if not needed.

So the definition of a category will be done through a set of examples and counter-examples for the category. One of the sets may be empty. Good classification results should be achieved with a relatively low number of examples (e.g. 5-15 songs).

Another property that feels natural to the author is the ability to create a ranking of the pieces of music, how well they fit to a certain category. Having continuous score values e.g. in the interval $[-1, 1]$ helps the user in finding "the good songs first" instead of presenting all songs in a category with approximately equal probability.

4.2 Classification algorithms used in the literature

There is a huge variety of different algorithms used for machine learning tasks. Every algorithm is designed for a special purpose. The algorithms and their properties will be presented in this section.

In general, there are two types of machine learning algorithms: *supervised* and *unsupervised* algorithms. Both types of algorithms get training data to learn from. Supervised algorithms receive, in addition to the input vectors $\boldsymbol{x}$, *required output data* $\boldsymbol{f}(\boldsymbol{x})$. Both input and required output data may be vectors or scalars. The target of supervised algorithms is to learn an appropriate *input-output function* $\boldsymbol{y}(\boldsymbol{x})$ that minimizes the error on the training data. The required output is sometimes called the *label* of the training data. Unsupervised algorithms do not have labels for the training data. They try to extract structure of the data, e.g. clustering algorithms like *k-means* or the presented EM algorithm for GMMs (see page 35).

4.2.1 Linear Discriminant Analysis

Linear Discriminant Analysis is a supervised machine learning algorithm used to seperate two classes of objects by hyperplanes. It is possible to expand this approach to multiple classes (see [Bis06, p.182]). The name stems from hyperplanes being descibed by linear functions. Classes of objects that can be seperated by a linear function are called *linearly separable*. The hyperplane is defined through a linear function f:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0. \tag{4.1}$$

All $\boldsymbol{x}$ with $f(\boldsymbol{x}) \geq 0$ belong to $\mathcal{C}_1$, and all $\boldsymbol{x}$ with $f(\boldsymbol{x}) < 0$ belong to $\mathcal{C}_2$. During the training step, values for $\boldsymbol{w}$ and w_0 have to be found which fulfill these properties for the training data. The training vectors have labels $\mathcal{C}_1$ or $\mathcal{C}_2$. Good results can be expected from *Fisher's linear discriminant*, which tries to find a good ratio of the within-class variance and the between-class variance (see [Bis06, pp.186]).

In general, the decision of an LDA classifier is binary. It is possible to use the value of the function f as a measure of how good the result is (e.g. if $f(\boldsymbol{x}) \gg 0$ or $f(\boldsymbol{x}) \ll 0$,

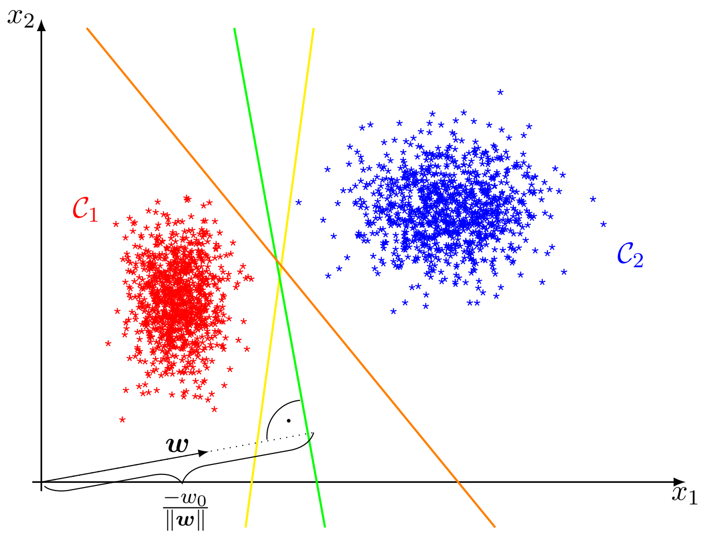


Figure 4.1: Linear Discriminant Analysis: Possible linear discriminant functions for two classes of data. All three lines are valid linear discriminants, the green one is close to Fisher's discriminant.

the decision is considered to be safer). Effectively, $f(\boldsymbol{x})$ describes a directed distance of $\boldsymbol{x}$ from the decision boundary $\{\boldsymbol{x}|f(\boldsymbol{x}) = 0\}$ with respect to $\|\boldsymbol{w}\|$.

The main problem of LDA is its linearity. Consider the tempo feature: It should be possible to find recordings of medium speed. With LDA, it is not possible to cancel out both recordings that are too fast and those who are too slow. It is however possible to extend the approach to more classes or apply a kernel function (see below about the "kernel trick").

4.2.2 Support Vector Machines

A *Support Vector Machine* is a supervised classification algorithm that seperates two classes of objects by finding a maximal margin hyperplane seperating these classes. The name stems from the *support vectors* used to define the hyperplane inside the algorithm. The pure Support Vector Machine is a linear algorithm. It has become popular through the invention of the *kernel trick* that is described later on and makes it possible to use the algorithm with nonlinear classifiers. The kernel trick is very efficient for SVMs (see [Bis06, p.325]). See figure 4.2 for an example of the margins, decision boundaries and support vectors.

Support Vector Machines provide, like LDA classifiers, binary decisions. Their decision boundaries are hard, they do not give posterior probabilities or a distance from the decision boundary in the "standard implementation" (see [Bis06, p.326]). However there

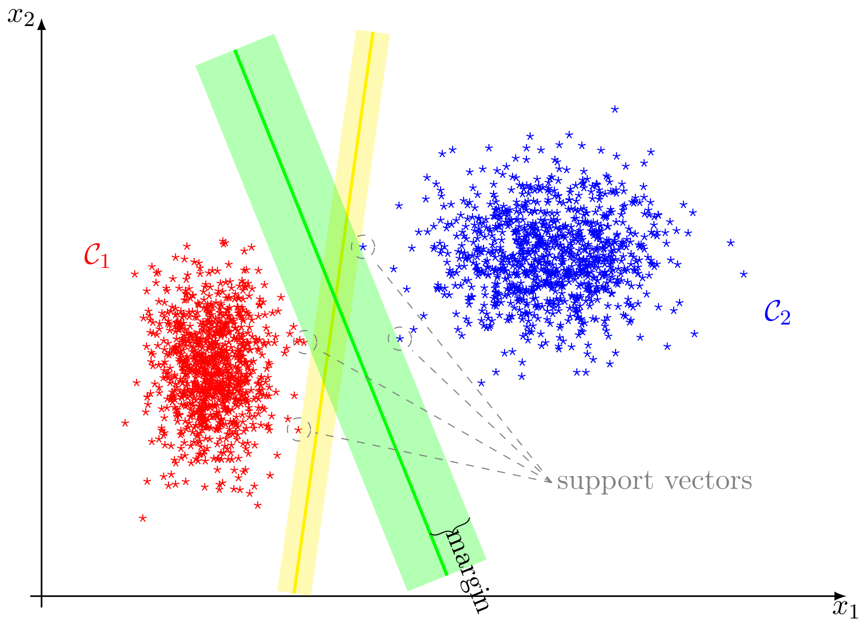


Figure 4.2: Example of a linear SVM and two possible classification boundaries with margins drawn. The green one is better than the yellow one because the margin is larger. Note the highlighted support vectors.

exist modifications of the SVM approach, e.g. the *Relevance Vector Machine* (RVM), that can give posterior probabilities. The RVM is an interesting algorithm and could have been used in this thesis, but has been found too late to be applied in the thesis.

The main advantages of SVMs in comparison to LDA classifiers are

- nonlinear decision boundaries can be created, through application of the kernel trick, which is implemented in all SVM libraries known to the author
- SVMs are computationally efficient compared to other approaches using a kernel on the feature space (it uses dot products in the algorithm, see below about the kernel trick)

A drawback is that the SVM is not able to learn the kernel function (at least the shape has to be designed). It is not suited for the goals of this thesis as it is a binary classifier.

4.2.3 Artificial Neural Networks

As the name states, *Artificial Neural Networks* are networks. They are used to create supervised machine learning algorithms. The networks are built from nodes (called *neurons*) and directed edges with *weights*. The most common network type are *Feed-Forward Networks*, they are formed by *directed acyclic graphs*. They typically have one *input neuron layer*, one *output neuron layer* and a *hidden neuron layer* in between. In many cases, there is only one hidden layer, but there may be an arbitrary number of hidden layers. There are edges from all input neurons to all hidden neurons and from all hidden neurons to all output neurons, with the exception of the *bias neurons*: they do not have incoming connections. Bias neurons have a fixed value of $x_0 = z_0 = 1$ and are not counted as input neurons. They are used to simplify the learning rules.

If there is more than one hidden neuron layer, they are connected the same way (see figure 4.3). If one edge should not be present, its weight is set to 0 and the edge is omitted in the picture.

The number of input neurons equals the dimensionality of the input vector. Each component of an input vector is the value of one input neuron. The values are fed to the hidden layer according to the following equation (definition of the variables follows from figure 4.3):

$$z_m = \sum_{n=0}^{N} w_{nm}^{(1)} \varphi_n(x_n) \tag{4.2}$$

where the φ_n are *activation functions* making the relation nonlinear. In many cases, all φ_n are chosen equal with

$$\varphi(x) = \frac{1}{1 + \exp(-x)}. \tag{4.3}$$

The function is called *sigmodial activation function*. The relation between the hidden layer and the output layer is defined in a similar manner.

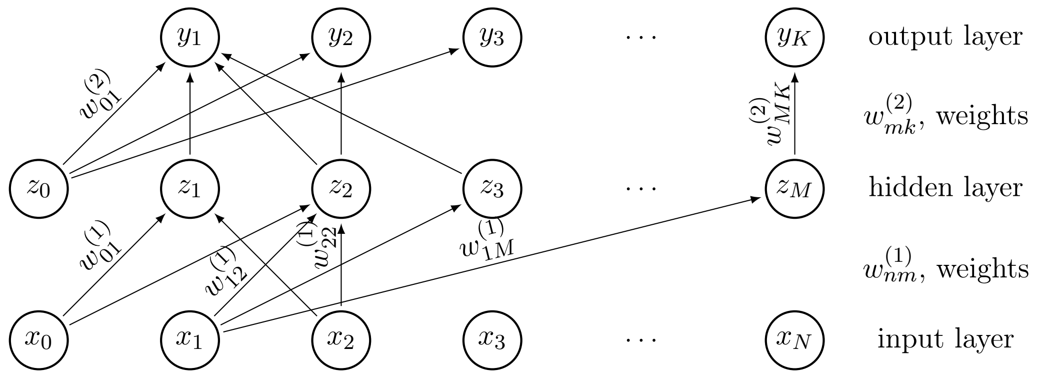


Figure 4.3: Feed-Forward Neural Network. Only some edges are shown or are labeled for clarity. Normally the layers are fully connected, with the exception of the bias neurons x_0 and z_0: they do not have incoming connections. Without loss of generality there are no direct connections bypassing one layer.

The reason why the number of hidden layers often is limited to one is that it can be shown that every continuous function defined on a compact subset of $\mathbb{R}^n$ can be approximated by a Feed-Forward Network with one hidden layer (see [Cyb89]). This theorem does not state that these functions can be learned and does not state anything about the speed with which the network could be learned. So, networks with more than one layer are used to speed up the learning process or increase stability. For a theoretical explanation, one hidden layer suffices. To learn a function, Feed-Forward Networks use a technique called *error backpropagation*, which essentially is a gradient descent approach (see [Bis95, p.140]).

The important implication of the theorem is that Feed-Forward Neural Networks can learn nearly all functions. Artificial Neural Networks can theoretically give scores for the pieces of music instead of a binary classification. A major drawback is that in practice, the learning step needs hundreds to thousands of learning examples for the error backpropagation algorithm to work. In case of using the same samples over and over again (to reduce the needed sample count), the risk of overfitting rises significantly – so this algorithm is not suited for the goals of this thesis.

4.2.4 The kernel trick

The *kernel trick* is a technique that can be applied to most linear classification algorithms to make them suited for nonlinear classification. The idea is to nonlinearly transform the feature space to a higher dimension and solve the classification problem in the higher-order space linearly. Projected to original space, it looks like as if the classifier is nonlinear. For an example, see figure 4.4.

Instead of transforming all samples to the higher-dimensional feature space, which

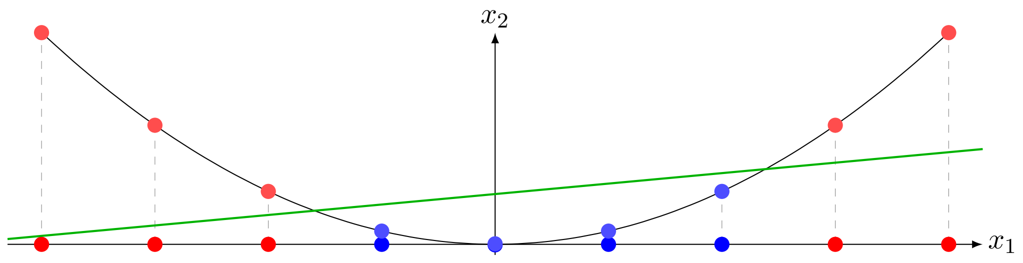


Figure 4.4: Nonlinear expansion to two dimensions (x_1, x_2) of the one-dimensional input feature space (x_1 only) to gain a nonlinear classification algorithm with a linear classifier. The decision boundary is the green line. The expansion function used is $\boldsymbol{\phi} : \mathbb{R} \to \mathbb{R}^2, \boldsymbol{\phi}(x) \mapsto (x, x^2)^T$.

can be very costly both in terms of memory and processing power, it is possible to replace all occurences of dot products $\boldsymbol{x}^T\boldsymbol{y}$ with a *kernel function* $k(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{y})$ (see [MMR+01a, p.184]). The kernel function calculates the dot product directly in the high-dimensional feature space, without the need to transform all points to the high-dimensional space. This operation can be efficiently implemented with fewer computations than first transforming the points and then apply the dot product (see [MMR+01a]). This efficient way of calculation is called the kernel trick. Nonlinear expansion can be applied to all linear classification algorithms, the kernel trick can be applied to all algorithms that rely on dot products (e.g. SVMs). For a detailed explanation see [Bis06, pp.291].

4.3 General problems with classification and multiple different features, and a solution to this

All algorithms use feature vectors for classification, and all feature vectors need to be of the same dimension. This complicates the use of features that are not comparable, like dynamic range and timbre vectors: the timbre vectors are high-dimensional and there are thousands of timbre vectors per recording, in contrast to one scalar value for the dynamic range.

One possibility to approach this problem is to simply create a new feature vector with e.g. one whole timbre vector and additionally the value for the dynamic range, and in subsequent feature vectors repeat the same value for the dynamic range. This approach however could potentially lead to overgeneralization, especially if artificial neural networks were used.

In another approach one feature vector is created out of all features for one song, and all feature vectors are merged to one large, very high-dimensional vector. Problems with this approach are on the one hand the high dimensionality of the feature vector and on the other hand the low number of feature vectors per recording; the presented classification algorithms rely on having many feature vectors.

Up to now, the main reason for using GMMs for timbre and chroma vector modeling was data compression (see section 3.3.3). Another reason is to overcome the problem of non-comparable features. The ansatz chosen in this thesis uses GMMs to reduce both the data count *and* dimensionality of the timbre and chroma vectors. Instead of using the GMMs directly as features, or draw samples and then use the samples as feature vectors, a different approach has been chosen: First, a model for the timbre of a category will be created, as described in section 3.3.8. Then, the timbre model of one recording will be compared to the model of the category, resulting in one one-dimensional value representing the similarity of the recording to the model. The same approach is used for the chroma model.

This one-dimensional timbre similarity value will be used in conjunction with the one-dimensional values for chroma, dynamic range and length of the musical piece to form a new feature vector[1]. Every dimension now represents one property of the recording, and there is one vector per recording. This vector can be used as feature vector for classification with all of the presented classification algorithms. A feature vector looks like this:

$$\text{feature vector} = \begin{pmatrix} \text{timbre similarity to category model} \\ \text{chroma similarity to category model} \\ \text{dynamic range} \\ \text{length of the recording} \end{pmatrix}$$

A drawback is the relatively low number of feature vectors resulting from this process (one per piece of music), and the need to recalculate the feature vectors for every category, since the timbre and chroma models are calculated per category. The feature vectors need to be recalculated when the category description changes, too, since the timbre and chroma similarity values depend on the definition of the category. Anyway, the algorithms used in the literature have properties (e.g. binary classification) that do not coincide with the goals of the thesis, so a different approach has been chosen. It is described in the following.

4.4 Proposed classification approach

The general idea of the proposed algorithm follows from the use of GMMs in the rest of the thesis, but for the classification, they are used in a slightly different way. Instead of a complete mixture model, only one Gaussian distribution with full covariance matrix is used. The vectors as presented in the section prior to this are used as feature vectors. Their distribution is captured by the Gaussian distribution.

[1] The tempo feature has been omitted since it was not possible to get a reliable tempo measure which made it possible to compare two recordings and decide which of both is slower or faster.

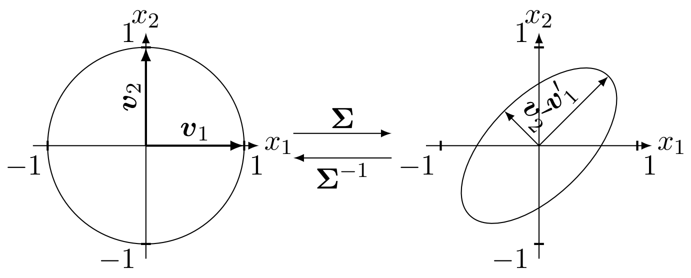


Figure 4.5: Application of a linear transformation with regular matrix $\mathbf{\Sigma}$ and the effects on the unit hypersphere. No information is lost on the transformation back to the original space via $\mathbf{\Sigma}^{-1}$.

The *Mahalanobis distance* of a feature vector for a piece of music to the Gaussian distribution is used to calculate the score, which is later forced to the interval $[-1, 1]$. The approach is first described for positive examples only and is then expanded to the case of positive and negative examples.

4.4.1 Mahalanobis distance and the Moore-Penrose pseudoinverse

The Mahalanobis distance and the Moore-Penrose pseudoinverse are used below. They are described first to allow a more fluent reading experience.

Definition 4.1 (*Mahalanobis distance*):
The *Mahalanobis distance* is defined as

$$d_{\mathbf{\Sigma}}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \mathbf{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{y})}. \tag{4.4}$$

with $\mathbf{\Sigma}$ being a positive definite Matrix (e.g. a regular covariance matrix).

Remark 4.2:
The Mahalanobis distance is a generalization of the Euclidian distance for elliptically distorted spaces. The eigenvectors of $\mathbf{\Sigma}$ are the semi-principal axes of the ellipsoid, and the eigenvalues are the lengths of the semi-principal axes. Iff $\mathbf{\Sigma}$ is the identity matrix, the term $\mathbf{\Sigma}^{-1}$ in equation 4.4 vanishes and equation 4.4 reduces to the Euclidian distance.

The application of $\mathbf{\Sigma}$ on a vector $\boldsymbol{x}$ is the transform from the undistorted space to the elliptically distorted space (see figure 4.5). The term $\mathbf{\Sigma}^{-1}$ is the transform back to the undistorted space. The Mahalanobis distance thus is a transform from the distorted space back to the undistorted space with subsequent calculation of the Euclidian distance.

The Mahalanobis distance can be seen as a "distance in the coordinate system spanned by the PDF of a normal distribution". The axes of this coordinate system are directed according to the eigenvectors of $\mathbf{\Sigma}$, their lengths are scaled by the magnitudes of the eigenvalues of $\mathbf{\Sigma}$. In this case, $\mathbf{\Sigma}$ is the covariance matrix of the normal distribution, and $\boldsymbol{y}$ will be set to $\boldsymbol{\mu}$, the mean of the distribution. This way, it is possible to calculate the distance of a vector to the center of a normal distribution in its own coordinate system: $d_{\mathbf{\Sigma}}(\boldsymbol{x}, \boldsymbol{\mu})$. For a picture of the Mahalanobis distance see e.g. figure 3.13 on page 46. The level lines are coordinates with the same Mahalanobis distance from the center of the normal distributions.

The Moore-Penrose pseudoinverse is a generalization of the concept of an inverse matrix for singular matrices. It is used to allow the calculation of the Mahalanobis distance even if $\mathbf{\Sigma}$ is singular.

Definition 4.3 (*Moore-Penrose pseudoinverse*):
The *Moore-Penrose pseudoinverse* $\boldsymbol{A}^+ \in \mathbb{F}^{n\times m}$ is a matrix with the properties

$$\boldsymbol{A}\boldsymbol{A}^+\boldsymbol{A} = \boldsymbol{A} \tag{4.5}$$
$$\boldsymbol{A}^+\boldsymbol{A}\boldsymbol{A}^+ = \boldsymbol{A}^+ \tag{4.6}$$
$$(\boldsymbol{A}\boldsymbol{A}^+)^* = \boldsymbol{A}\boldsymbol{A}^+ \tag{4.7}$$
$$(\boldsymbol{A}^+\boldsymbol{A})^* = \boldsymbol{A}^+\boldsymbol{A} \tag{4.8}$$

with $\boldsymbol{A} \in \mathbb{F}^{m\times n}$.

Note: Compare this definition to the properties of a regular matrix (e.g. assume $\boldsymbol{A} \in \mathbb{F}^{n\times n}$ regular and $\boldsymbol{A}^+ := \boldsymbol{A}^{-1}$).

Theorem 4.4 (*Properties of the Moore-penrose pseudoinverse*):
The Moore-Penrose pseudoinverse has the following properties:

1. The Moore-Penrose pseudoinverse exists and is unique
2. If $\boldsymbol{A}$ is invertible, then $\boldsymbol{A}^{-1} = \boldsymbol{A}^+$
3. $(\boldsymbol{A}^+)^+ = \boldsymbol{A}$

A proof of the statements (together with the definition and additional properties) can be found in [Sto05, pp.255]. As one can see, the Moore-Penrose pseudoinverse behaves like an inverse would in many cases. Nevertheless, *it may be singular*! Thus it is a generalization of the concept of inverse matrices, which is only defined for regular and quadratic matrices, for general rectangular matrices. In the case used here, the matrix will be quadratic, but singular.

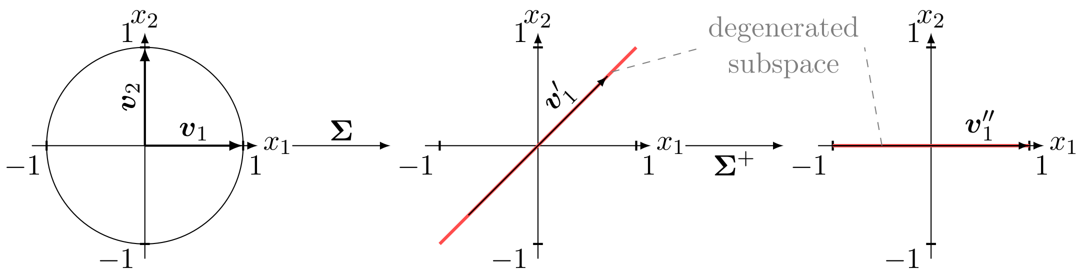


Figure 4.6: Application of a singular matrix $\mathbf{\Sigma}$ and the effects on the unit hypersphere. $\boldsymbol{v}_2$ is mapped to the zero vector, the ellipsoid is degenerated (compare to figure 4.5). All vectors from the original space are mapped to a one-simensional subspace (marked red). Application of $\mathbf{\Sigma}\mathbf{\Sigma}^+$ transforms the unit hypersphere $\{(x_1, x_2)|x_1^2 + x_2^2 = 1\}$ to $\{(x_1, 0)| -1 \leq x_1 \leq 1\}$.

Theorem 4.5 (*Construction of the Moore-Penrose pseudoinverse*):
The Moore-Penrose pseudoinverse can be constructed through the *Singular Value Decomposition* (SVD) of a matrix. This construction method is numerically stable.

The SVD is a matrix decomposition: Let $\boldsymbol{A} \in \mathbb{F}^{m\times n}$; $\boldsymbol{U} \in \mathbb{F}^{m\times m}$, $\boldsymbol{V}^* \in \mathbb{F}^{n\times n}$ both unitary[2], $\boldsymbol{S} \in \mathbb{F}^{m\times n}$ diagonal[3] with $(\boldsymbol{S})_{ii} \geq (\boldsymbol{S})_{jj}$ for all $i < j$. $(\boldsymbol{S})_{ii}$ are called *singular values*. The SVD exists for all matrices.

The Moore-Penrose pseudoinverse can be constructed with the SVD via

$$\boldsymbol{A}^+ = \boldsymbol{U}\boldsymbol{S}^+\boldsymbol{V}^* \tag{4.9}$$

with

$$(\boldsymbol{S})^+_{ij} = \begin{cases} \frac{1}{s_{ij}}, & \text{if } i = j \text{ and } s_{ij} \neq 0 \\ 0, & \text{else.} \end{cases} \tag{4.10}$$

For details on the calculation of an SVD and proofs for the theorem, see [SB05, pp.21].

The Moore-Penrose pseudoinverse is used for the calculation of the Mahalanobis distance instead of $\mathbf{\Sigma}^{-1}$. This is a generalization of equation 4.4, since $\mathbf{\Sigma}^{-1} = \mathbf{\Sigma}^+$ if $\mathbf{\Sigma}$ is invertible.

Remark 4.6:
If the covariance matrix $\mathbf{\Sigma}$ is singular, then the transform to the distorted space goes hand in hand with a loss of information (see figure 4.6). The transform back to the

[2](dt. *unitär*) is the complex analogon for orthogonal matrices. Properties: $\boldsymbol{U} \in \mathbb{F}^{n\times n}$ is unitary, i $\boldsymbol{U}^*\boldsymbol{U} = \boldsymbol{I}$ and $\boldsymbol{U}^* = \overline{\boldsymbol{U}}^T$

[3]For all (not only quadratic) matrices, the term *diagonal* denotes matrices with nonzero values only at entries where row and column number are equal.

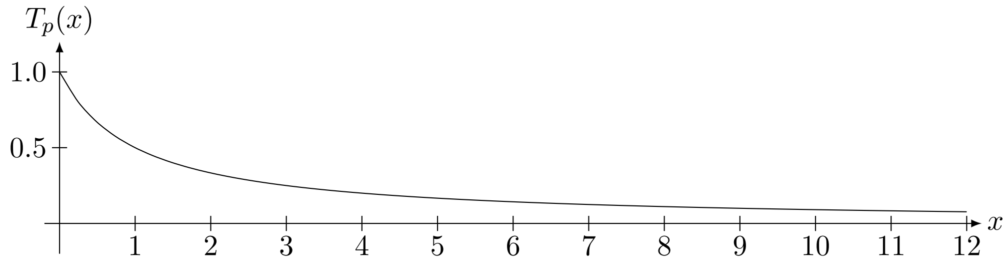


Figure 4.7: Function $T_p(x) = \frac{1}{1+x}$ in the domain $[0, 10]$.

original space via $\mathbf{\Sigma}^+$ does not bring the information back. The Mahalanobis distance in this case is the Mahalanobis distance in the degenerated and retransformed subspace (see figure 4.6, on the right), which is as near as possible to the definition of the Mahalanobis distance for regular covariance matrices.

$\lrcorner$

Thus, the generalized Mahalanobis distance can be used for all covariance matrices $\mathbf{\Sigma}$, even if they are singular.

4.4.2 Approach without negative examples

First, the covariance matrix and the mean of all feature vectors for the recordings that are positive examples for one category are calculated. This normal distribution is called *positive classification model*, because it describes the distribution of the feature vectors for one category. With this model, the Mahalanobis distances of the feature vectors for all recordings (not only the positive examples) are calculated, the values are from the interval $[0, \infty[$. To force the distance values to the interval $[0, 1]$, a transformation function is applied: $T_p(x) = \frac{1}{1+x}$. This creates a ranking for the recordings: Values near 1 indicate good matches, values near 0 indicate pieces of music that do not match (distance very large).

One tweak is applied to the calculation of the covariance matrix: The values for timbre and chroma similarity in the feature vectors for positive examples are usually very small[4], because the timbre and chroma model was built using these songs as examples. Instead of calculating their mean, it is set to zero (hardcoded). This is done to avoid situations where a song gets a lower rating just because it is more similar to the timbre and chroma models than the average of the positive examples. Example: The mean of the timbre similarity of positive examples is calculated as 1.0. Recordings with timbre similarity value 0.5 and 1.5 would get the same score, which is counter-intuitive since “better similarity” should get “better scores”, and the timbre similarity model itself is a good

[4] Ideally, these values should be zero, in practice they are from the interval $[0, 1.5]$ with all similarity values being from the range $[0, 500]$.

measure for similarity. Anyway, the distribution of the timbre similarity should have an influence on the scores. Hardcoding the mean of the timbre similarity to 0 and using this mean value in the calculation of the covariance matrix allows to have both properties. Note: This step increases the variances and covariances in the covariance matrix.

A problem with the usage of full covariance matrices is that they are singular if the number of samples is too low[5]. In this case the covariance matrix is not invertible, and the classical Mahalanobis distance as defined in definition 4.1 cannot be calculated. This happens if four or less feature vectors are used to calculate the covariance matrix[6]. To be able to use the same approach even with four or less positive examples, the generalized Mahalanobis distance is used, which uses the Moore-Penrose pseudoinverse. Without using the generalized Mahalanobis distance, the chosen approach would be unusuable with less than four examples.

With this approach, it is possible to calculate scores for all recordings from an arbitrary number of positive examples. See figure 4.8 for a visualization of the four-dimensional classification model together with a few feature vectors. However, comparatively many recordings that do not fit well to a category get high scores. This effect is comparable to the effect of *false positives* for binary classification. Even if this would not be the case, it is desirable to have the possibility to sharpen the classifier by giving it some negative examples. A solution for this problem, which helps increasing the general performance of the algorithm, is presented in the next section.

Remark 4.7 (*Why a single Gaussian distribution is sufficient for classification purposes*):
A single Gaussian distribution is sufficient to capture the distribution of the feature vectors since the features are good-natured. For the timbre and chroma features, this follows from their definition: Positive examples for a category have a low timbre and chroma similarity value since the category model is built from them, so the songs matching the category build clusters in these feature dimensions by definition.

There are two reasons why one Gaussian distribution is sufficient for the other features, too. On the one hand, it can be expected that the user groups songs of a similar dynamic range or length, just because some values for the dynamic range or length of a musical piece are typical. On the other hand, if the user groups songs with different dynamic ranges or lengths, this specific feature is rendered unimportant through the use of the Mahalanbis distance: The variance of that feature will be large, thus the distances within this feature will get smaller and thus loose importance, indicating a good match for that feature in every case. This is congruent with the intuitive observation that a feature

[5] The covariance matrix cannot be regular if the number of samples is lower than the number of dimensions of the covariance matrix.

[6] It is possible for the covariance matrix to be singular even if the number of feature vectors is higher than four, in case of the feature vectors being linearly dependent.

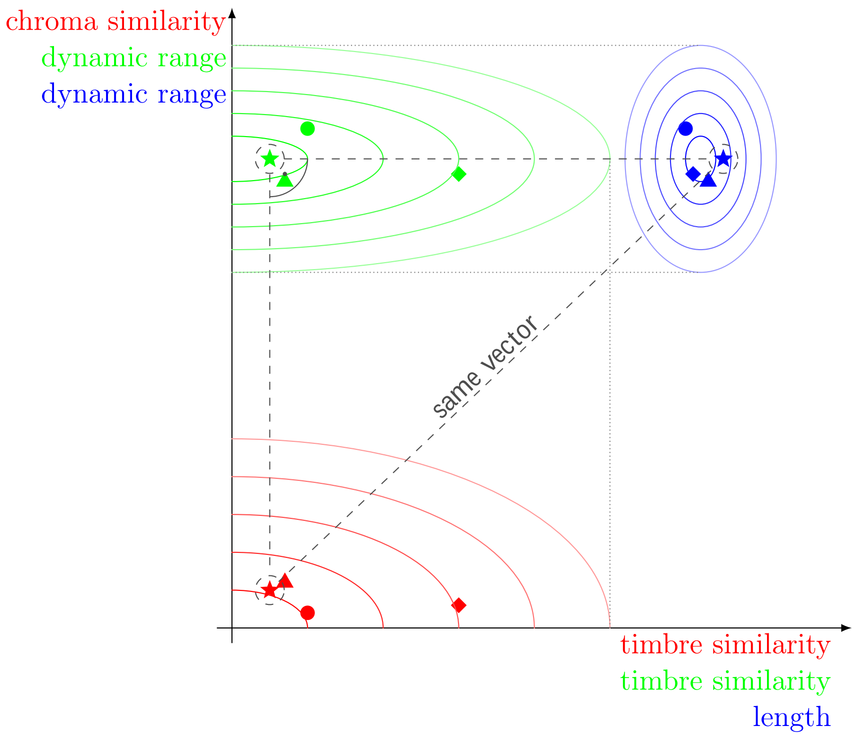


Figure 4.8: Sectional drawing of the four-dimensional positive classification model in two dimensions with some song examples. Points with the same symbol denote the same vector/song, different colors are different views. The ellipses denote the equidistant level lines of the model in the specific view in the Mahalanobis distance. The distance of ★ would be small (score near 1), the distance of ◆ would be larger (score smaller than for ★).

cannot be important if for instance there are very short and very long songs, or songs with a very high and very low dynamic range.

If there were other features, e.g. the tempo feature, this might not be true anymore.

$\lrcorner$

4.4.3 Approach with both positive and negative examples

From the previous section we have a score value for how well a recording matches the positive model, and the score is in the interval $[0, 1]$. The idea for a model with positive *and* negative examples is to reuse the approach for positive examples. A second model is built, but for a set of negative examples. This results in a *negative classification model*, a *negative timbre model* and a *negative chroma model*. These models are independent of the models for the positive example set. The models for the positive example set are renamed to *positive timbre model* and *positive chroma model*. The negative models are calculated exactly the same way as the positive models, with the exception of the transformation function $T_p(x)$, which is replaced by $T_n(x) = -T_p(x)$. The change forces the negative model scores to be in the interval $[-1, 0]$.

The overall score is then calculated as the sum of the positive model score and the negative model score. These combined score values are from the interval $[-1, 1]$. The effect is as follows: For recordings where the score of the positive model is large (near 1) and the negative score is small in magnitude (near 0), nothing changes qualitatively. If the negative score raises in magnitude, the overall score of the recording lowers. So, recordings that are near to both the positive and negative model will get a score near 0. Recordings that are near to the positive, but far from the negative model get scores near 1; recordings that are near to the negative, but far from the positive model get scores near -1. If no negative model is built and no negative examples are present, the negative model score shall be treated as zero.

4.4.4 Advantages and drawbacks of this approach

This approach has several advantages, compared to the more classical ones with binary decision boundaries:

- It is possible to use both positive and negative examples for classification.
- It works with few examples.
- *Negative examples can be omitted.* They are used to raise the quality of the classification, but they are not needed.
- The algorithm can create a ranking instead of a binary classification.

However, it is rather hard to measure the performance of this algorithm. These problems and a solution are described in the chapter about performance tests (see page 97). It can be seen as a drawback that the algorithm does not perform a binary classification.

Note that *classification* does not seem to be the right word for the chosen approach; *ranking* seems to be better suited.

5 Design and Implementation

In this chapter, the software design is presented and some implementation details are discussed. First, the hardware and software environment are described in short, thereafter an overview about the software modules and their dependencies is given. Then, the software modules are described in detail.

5.1 Hardware and software environment

The software runs on PCs and mobile devices as well, from standard x86 processor systems with lots of memory, processing power and processing units, to mobile devices with a lower amount of memory and processing power. For example, the BlackBerry Playbook is equipped with two 1GHz ARM cores and 1GiB RAM. Even though this does not seem like a limitation in processing power, it has to be considered that mobile devices are not only limited in processing power, but also in the total number of computations with one battery charge. Keeping the computation count low is not only a matter of time, but a matter of battery efficiency, too.

The operating systems used are all Unix variants: For the PC, Linux is used. On the mobile devices, a QNX system is installed. Both systems comply to the POSIX standard[1], so a common code base for both systems has been realized. The programming language used is C++.

Although the systems seem to be similar, there are some differencies in the details. For example, the QNX system uses a different C and C++ standard library: Instead of a GNU library, a Dinkumware library can be used. The GNU C++ library is available and can be chosen, but it is not possible to mix both libraries. During development of the software, there were some situations where the standard libraries were mixed through external library dependencies and produced crashes and undefined behaviour that was not easy to keep track of. Additionally, it is state-of-the-art to use procecessors with a 64 bit architecture on PCs. On mobile devices, a 32 bit architecture is common, although this is subject to change[2]. Programs can behave differently if the sizes of data types are different than expected. For code written in a native language like C++, this can make a difference if the standard data types like `int` or `long` are used as indices or in other

[1] *Portable Operating System Interface*, a standardized interface for operating systems, see `http://pubs.opengroup.org/onlinepubs/9699919799/` (12/6/2012)

[2] see `http://www.arm.com/products/processors/instruction-set-architectures/armv8-architecture.php` (12/6/2012)

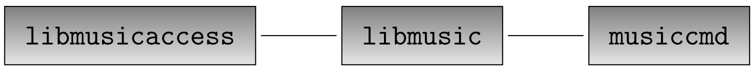


Figure 5.1: Overview of the program structure. `libmusicaccess` is used to read media files. `libmusic` contains the main algorithms for feature extraction and classification. `musiccmd` is a commandline interface for `libmusic`.

applications. Using preprocessor-defined types like `size_t` produces code that is more portable.

There may be other minor differences that can potentially lead to problems, but they did not show up during development.

5.2 Structure

The program is designed as a shared library. This has some advantages: The algorithms can easily be reused with another frontend, other than the command line version that has been implemented. It is possible to integrate the algorithms into a readily available program, such as a music player, without too much effort. Additionally, it is possible to give the algorithms to other developers without the need to open the source code, which can be an advantage in commercial applications. In case of multiple programs using the library at the same time, only one instance of the library needs to be loaded into memory. This is espesially useful on mobile devices. The name of the library, `music`, is a recursive acronym for **mu**sic **si**milarity **c**lassifier.

The code is subdivided into two libraries: `libmusicaccess` contains all code that is related to reading music files and resampling them to the format `libmusic` uses for further processing. On the one hand, it is possible to recompile `libmusicaccess` without the need to recompile `libmusic` (and vice versa), so the compilation step takes less time: in most cases, only one of the libraries will be changed. On the other hand, decoders for new file formats can be added to `libmusicaccess`, too – even by external persons who do not know the source of `libmusic` – as long as the interfaces of the classes do not change. Last but not least, it adds legal certainty. The libraries used for decoding the audio files are licensed as `GNU LGPL`[3], which might force the user of such a library to open his source code under certain conditions. It is better to be safe than sorry, having all code touching these libraries in a shared library reduces that risk: The worst-case-scenario would be to need to open the source of `libmusicaccess`.

The three parts of the software will now be described in the following.

[3]see http://www.gnu.org/licenses/lgpl-2.1.html

File format	Library	Platform availability		Tag extraction
		Linux	QNX	
mp3	libmpg123	y	y	y
wav	libsndfile	y	n	n
ogg	libsndfile	y	n	y (Linux)
flac	libsndfle	y	n	n

Table 5.1: Availability of file formats in `libmusicaccess`. Some formats are not available on QNX due to `libsndfile` depending on other libraries that were not ported up to now.

5.2.1 libmusicaccess

This library is able to decode audio from various media file formats and transform the audio samples to the format `libmusic` uses (22kHz, `float`, mono). It uses `libsndfile`[4] and `libmpg123`[5] as decoders. An overview of supported formats can be found in table 5.1.

The most important classes of this library are `SoundFile`, `Resampler22KhzMono` and `IIRFilter`. `Soundfile` is capable of decoding audio and reading file metadata. The audio data will be read as either 16-bit `signed integer` or 32-bit `float` values, depending on the user's choice.

`Resampler22KhzMono` can be used to resample the data read by `SoundFile` from any sample rate to 22Khz and mix the stereo stream down to a mono stream. To achieve this, `Resampler22KhzMono` makes use of `libsamplerate`[6] for resampling. An internal resampling algorithm was implemented, but it is dysfunctional and needs to be reworked. The resampling part does not work, but the lowpass-filtering part (`IIRFilter`) works and is used within the Constant Q Transform. `IIRFilter` is a general IIR Filter implementation for single precision `float` values. It comes with an initialization function for a lowpass filter which takes a relative cutoff frequency (e.g. `0.5`) as parameter. Since the function is not capable of calculating the internal parameter settings for the filter, they have been precalculated by Matlab and hardcoded[7] into the library. Values for some standard relative cutoff frequencies (e.g. `0.5`, `0.25`, `0.125`, ...) are available. If no matching parameter set can be found, the one with the next-lower relative cutoff will be chosen. The IIR Filter is implemented via the direct II transposed structure (see figure 2.6).

[4] see `http://www.mega-nerd.com/libsndfile/`
[5] see `http://www.mpg123.de/api/`
[6] see `http://www.mega-nerd.com/SRC/`
[7] Butterworth filter of order 6

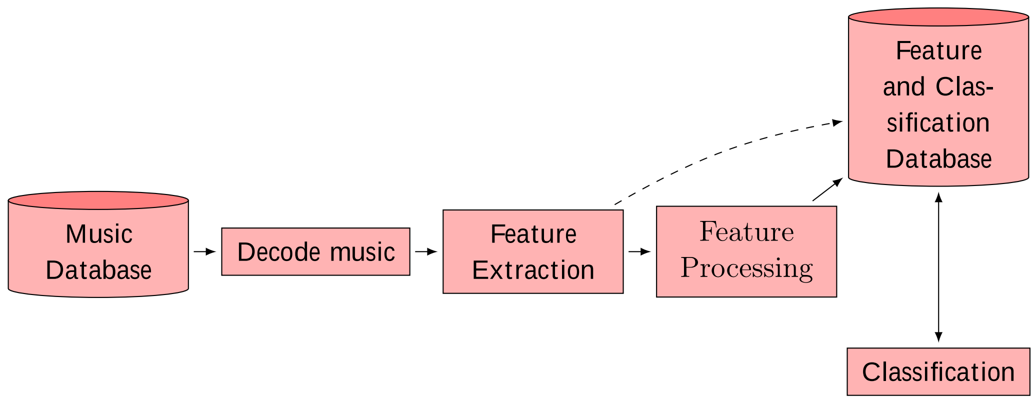


Figure 5.2: Overview of the internal structure and internal data flow of `libmusic`.

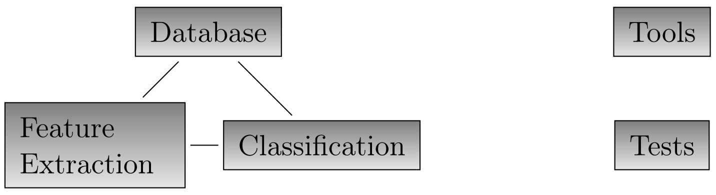


Figure 5.3: Overview of the software modules of `libmusic`. Feature extraction, classification and the database are strongly connected. Tools and Tests are used by all modules, but their connection is loose.

5.2.2 libmusic

This library contains the main algorithms. As a black box, it takes audio data and returns scores of that data for categories. An overview of the internal structure and data flow can be found in figure 5.2.

The library has several software modules as described in figure 5.3. The modules and their interaction is described in the following. `libmusic` uses the `Eigen` library[8] for matrix and vector arithmetic, the `Eigen` type `Eigen::Matrix` is used for both matrices and vectors. `Eigen` also has modules for sparse matrices and matrix decompositions such as the SVD and Cholesky decomposition.

Feature extraction

In this module, the feature extraction process takes place. The classes `FilePreprocessor` and `MultithreadedFilePreprocessor` perform all needed actions to extract the features

[8]see `http://eigen.tuxfamily.org`

and save them to the SQL database[9]. This concept has been chosen for usability reasons: The user only needs to call a single function (`preprocessFile()`) to add files to the database, including feature extraction.

The two classes are mainly remote control classes for the feature extraction algorithm classes. It is possible to not use these two classes at all and reimplement the feature extraction process using the feature extraction classes directly (see below). There may be some rare cases in which re-implementing the functionality of the classes better integrates with the rest of the application.

The classes first decode the given audio files using `libmusicaccess` and use the resulting floating-point samples to extract tempo (using class `BPMEstimator`), dynamic range (using class `DynamicRangeCalculator`), length, timbre (using classes `TimbreEstimator` and `TimbreModel`) and the keyinvariant chroma (using classes `ChromaEstimator` and `ChromaModel`). The algorithms implemented in these classes are described in chapter 3. Where appropriate, these extraction classes either directly take the results of the Constant Q Transform, or they take an intermediate result from another class as input for their calculations. In some cases (e.g. tempo calculation and dynamic range) this saves extra computation cycles, as intermediate results of computations can be shared between classes.

The process of feature extraction has been parallelized to make use of all processing units available, since it is the most time-consuming part of the implementation.

Multithreading concept There exist several paradigms for efficient multithreading. Some algorithms can directly be parallelized, taking advantage of multiple processors for every single call of the algorithm. The task will be split up for multiple processors. Some types of algorithms, such as divide-and-conquer-algorithms, are especially suited for these parallelization tasks: Divide-and-conquer is a technique to recursively split a problem into smaller subproblems, later combining the subproblems to solve the original problem. A popular example for divide-and-conquer is the parallel implementation of a FFT. The smaller subproblems can be solved independently on independent processors, speeding up the whole process. These algorithms are used without parallelization, too.

However, parallel algorithms can be hard to implement because using threads leads to nondeterministic behaviour, since timings are not always exactly the same in different threads. Synchronization problems can lead to undefined behaviour: Sometimes the processes need to share memory. This memory can be changed by one process during the read access of another process, since read accesses do not need to be *atomic operations*[10]. To overcome this problem, locking mechanisms have been developed: Memory can be

[9] The database schema and the database abstraction layer will be described in the database module later on.

[10] *Atomic operations* are operations that cannot be interrupted during execution.

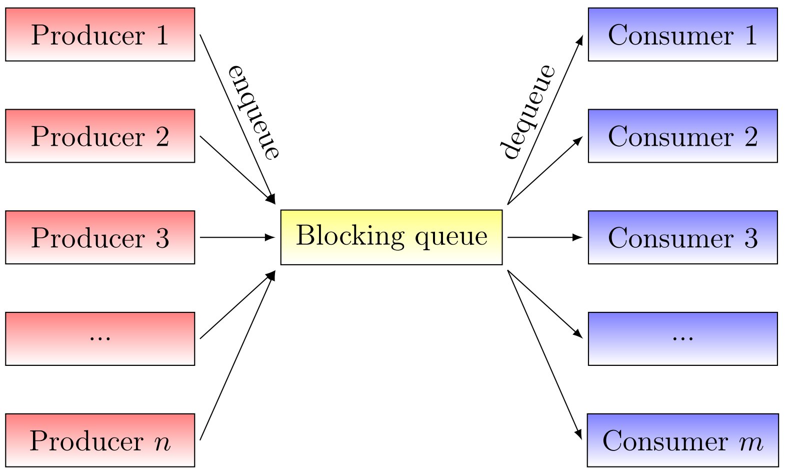


Figure 5.4: Producer-consumer pattern for loose coupling. The blocking queue contains the concurrency logic and has limited capacity.

locked for exclusive access by one process. The access may be only write-exclusive, allowing multiple readers while no process is writing.

Nevertheless, writing code with concurrent access to resources is hard to debug and at the same time, due to the complication through parallelism, potentially error-prone (see [Lee06]). Well-known problems with concurrency include deadlocks, livelocks, and starving of processes. Writing programs exploiting the possibilities of parallel execution remains complicated in comparision to single-threaded software.

A comparatively easy parallelism paradigm is the *producer-consumer pattern* (see figure 5.4). In the simplest form, one *producer* produces goods and puts them into a queue with limited capacity. One *consumer* takes the goods in order of insertion from the queue. In case of the queue being full, the producer waits until there is enough space to place his goods in the queue. In case of the queue being empty, the consumer waits for new goods. The queue is called a *blocking queue* due to its blocking behaviour (producer and consumer need to wait in certain cases). The size is limited to prevent the producer to eat up all the machine's resources in case he is faster producing than the consumer consuming the goods. The producer-consumer pattern can be generalized to multiple producers and multiple consumers. Here, it will be used with a single producer and multiple consumers.

While consumers and producers do not need to worry about concurrency, the blocking queue implementation does. The queue needs to supply two operations: `enqueue()` needs to block iff the queue is full, and `dequeue()` needs to block iff the queue is empty. Additionally, it is useful to have the possibility to tell the consumers that the producer finished producing goods, so that he can stop waiting for new goods to process (here: `destroyQueue()`). The resulting queue has been implemented as a template class to support different data types: BlockingQueue<T>. It is used as a job queue.

The producer enqueues the names of the files to the blocking queue (template data type is std::string) that should be added to the database. Consumers, at least as many as processing units available[11], dequeue these file names and extract the features from the files. This process can be done independently for every file, resulting in a speedup of approximately[12] the number of processing units available. Since it is not possible to write to an SQLite database connection from a different thread than it was created by[13], the resulting features will be written to the database by a separate thread. This thread will get its objects to write to the database with the same technique, using the producer-consumer pattern with multiple producers and a single consumer (template data type of the blocking queue is databaseentities::Recording*, see database module description for details). This way, the cpu-bound feature extraction threads do not have to wait for the I/O-bound database operation and the problem of writing to the same database abstraction object from multiple threads is solved.

Classification

In this module, all classes related to the classification process are implemented. Since the EM algorithm is used as an unsupervised soft partitional clustering algorithm, and because it is used in the classification step, the GMMs are saved in this module, too. See figure 5.5 for an UML class diagram of the most important functions of all classes related to the calculation of GMMs.

In ClassificationCategory, all properties related to the definition of a category are saved. This class is capable of calculating the classification model as described in section 4.4, too. The chroma and timbre models of the category are set from the outside of the class and calculated beforehand in ClassificationProcessor.

Following the concept derived for the feature extraction process, a class was written that manages the classification step: ClassificationProcessor. Since the classification process relies heavily on database accesses and these accesses need to be serial, parallelization of the classification step cannot be done with the same efficiency as for the feature extraction. For this reason, no multithreaded version of ClassificationProcessor has been implemented. ClassificationProcessor has three functions: addRecording() performs the initial classification of a newly added recording; it calculates the score values of the recording for all existing categories. recalculateCategory() recalculates a category if its properties have been changed or a recalculation has been scheduled.

[11]Using more threads than processors available can lead to a speedup in case the processes are both cpu- and I/O-bound or cache misses are likely to appear. This is especially true for the case of hyper-threading seen on Intel processors, which can make use of the processors features more e□ciently.

[12]Because of locking mechanisms and thread context switches that take some time, the speedup will be lower.

[13]see http://www.sqlite.org/faq.html, Question 6 (downloaded 12/3/2012)

`recalculateCategoryMembership()` recalculates the score values for all songs of a category without recalculating the category itself. This function is called upon request by `recalculateCategory()`, right after the recalculation of the category. The results are saved to the database.

For testing purposes, a Fisher LDA classifier was implemented in `FisherLDAClassifier`, which is an implementation of `TwoClassClassifier`. These classes are not used in the project, but some tests have been performed with them. The LDA approach proved not applicable for the goals of the thesis, see section 4.2.1.

Database

The database holds data for extracted features, classification results, and file metadata all in one place. The database access has been encapsulated in an access layer (class `DatabaseConnection`) instead of giving direct database backend access to all other modules. Encapsulation has several advantages in comparison to direct access:

- The database layout can be changed without the need to change every bit of code touching the database. To reflect minor changes, only the database abstraction layer needs to be adjusted.
- The database backend can be replaced, if needed. Some backends might not be available on all platforms, so encapsulation makes sense with portability in mind. Additionally, it is possible to change to NoSQL databases with only minor effort. This might be interesting since NoSQL databases are considered to have speed advantages over SQL databases in cases where the strength of SQL is not needed, because there is no need to parse an intermediate language.
- Some features can be added later on, e.g. network transparency (the database can be saved on another computer without the program noticing it), or caching mechanisms. To implement a caching mechanism, a proxy class implementing a caching technique can be added between the database backend and the rest of the program. The proxy class should inherit from `DatabaseConnection` and take a pointer to another `DatabaseConnection`. Accesses to the pointed class will be redirected through the proxy class, which in turn implements the caching mechanism.

To make use of these advantages, the user needs to implement a new class that inherits from `DatabaseConnection` and implements all the functions defined (see source code documentation for details). In the submodule `databaseentities`, one finds database value objects used to transfer data from all other modules to the database layer (see figure 5.8). The functions from `DatabaseConnection` take these classes as parameters. The classes do not reflect the database layout in detail. Through normalization, many of the properties get their own tables (e.g. artist, album and genre). All models

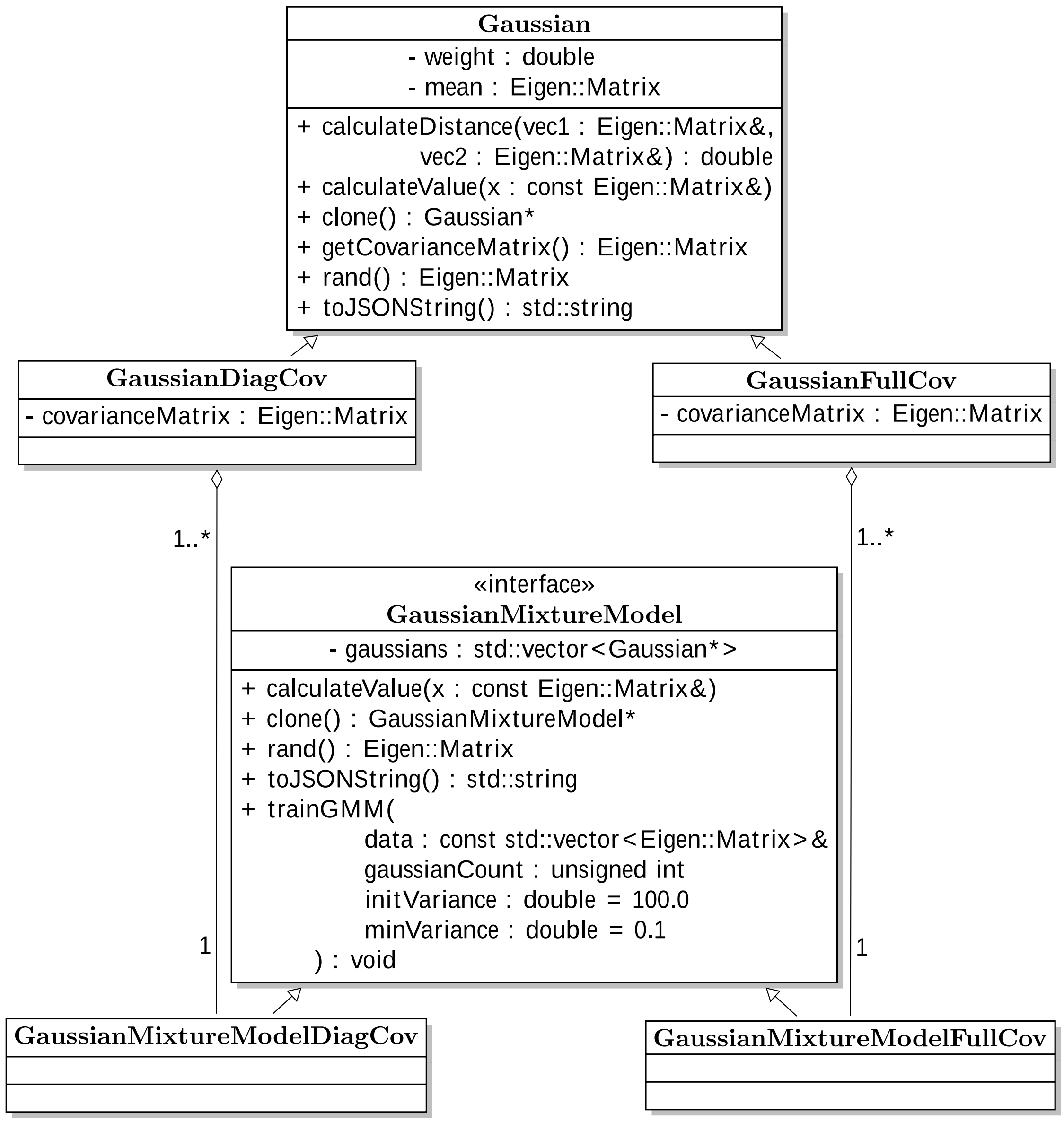
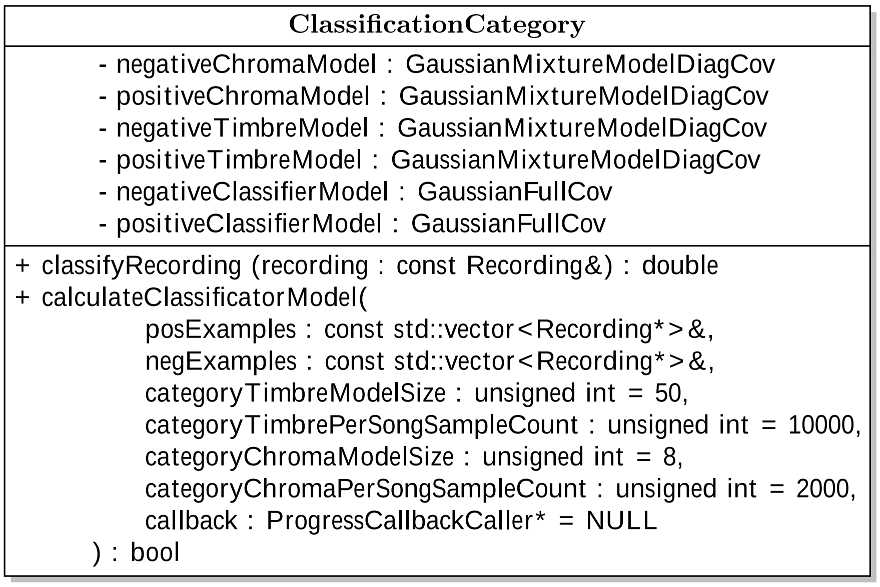

Gaussian
- weight : double
- mean : Eigen::Matrix
+ calculateDistance(vec1 : Eigen::Matrix&, vec2 : Eigen::Matrix&) : double
+ calculateValue(x : const Eigen::Matrix&)
+ clone() : Gaussian*
+ getCovarianceMatrix() : Eigen::Matrix
+ rand() : Eigen::Matrix
+ toJSONString() : std::string
GaussianDiagCov
- covarianceMatrix : Eigen::Matrix
GaussianFullCov
- covarianceMatrix : Eigen::Matrix
1..*
1..*
«interface»
GaussianMixtureModel
- gaussians : std::vector<Gaussian*>
+ calculateValue(x : const Eigen::Matrix&)
+ clone() : GaussianMixtureModel*
+ rand() : Eigen::Matrix
+ toJSONString() : std::string
+ trainGMM(
data : const std::vector<Eigen::Matrix>&
gaussianCount : unsigned int
initVariance : double = 100.0
minVariance : double = 0.1
) : void
1
1
GaussianMixtureModelDiagCov
GaussianMixtureModelFullCov

Figure 5.5: Important properties and functions of `Gaussian`, `GaussianMixtureModel`, and their children. Getters and setters are not shown.
DiagCov and FullCov determine if full or diagonal covariance matrices are used.



```
ClassificationCategory
- negativeChromaModel : GaussianMixtureModelDiagCov
- positiveChromaModel : GaussianMixtureModelDiagCov
- negativeTimbreModel : GaussianMixtureModelDiagCov
- positiveTimbreModel : GaussianMixtureModelDiagCov
- negativeClassifierModel : GaussianFullCov
- positiveClassifierModel : GaussianFullCov
+ classifyRecording (recording : const Recording&) : double
+ calculateClassificatorModel(
        posExamples : const std::vector<Recording*>&,
        negExamples : const std::vector<Recording*>&,
        categoryTimbreModelSize : unsigned int = 50,
        categoryTimbrePerSongSampleCount : unsigned int = 10000,
        categoryChromaModelSize : unsigned int = 8,
        categoryChromaPerSongSampleCount : unsigned int = 2000,
        callback : ProgressCallbackCaller* = NULL
    ) : bool
```

Figure 5.6: Properties and important functions of `ClassificationCategory`. Getters and setters are not shown. `ProgressCallback` is described in the Tools module.

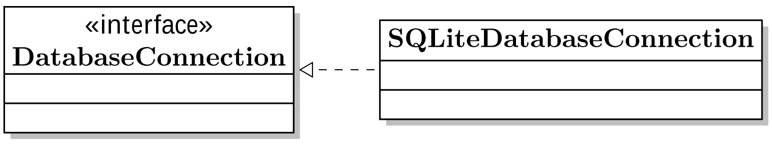


Figure 5.7: Relationship of `DatabaseConnection` and `SQLiteDatabaseConnection`. Functions and variables are not shown, refer to the source code documentation for details.

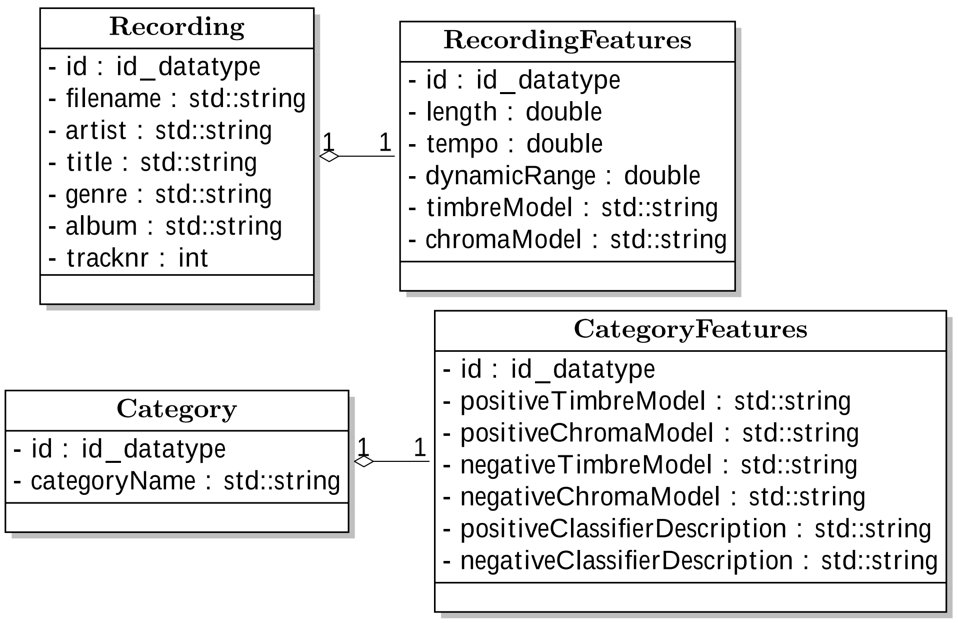

Recording
- id : id_datatype
- filename : std::string
- artist : std::string
- title : std::string
- genre : std::string
- album : std::string
- tracknr : int
1
1
RecordingFeatures
- id : id_datatype
- length : double
- tempo : double
- dynamicRange : double
- timbreModel : std::string
- chromaModel : std::string
Category
- id : id_datatype
- categoryName : std::string
1
1
CategoryFeatures
- id : id_datatype
- positiveTimbreModel : std::string
- positiveChromaModel : std::string
- negativeTimbreModel : std::string
- negativeChromaModel : std::string
- positiveClassifierDescription : std::string
- negativeClassifierDescription : std::string

Figure 5.8: Classes from the `databaseentities` submodule. Getters and setter are not displayed. id_datatype is defined as `long`, but can be changed to another type if needed.

```json
[
  {
    "covariance": [2.44,  1.09],
    "mean":       [-3.22, 3.23],
    "weight":     0.4
  },
  {
    "covariance": [1.23, 5.67, 4.56],
    "mean":       [2.54, 0.97],
    "weight":     0.6
  }
]
```

Figure 5.9: Example of a description of a GMM in JSON notation. The GMM has 2 components and is defined in a 2-dimensional field. The covariance matrix of the first component is saved as diagonal matrix, the second component has a full covariance matrix, saved in the condensed format described in section 3.3.3. The different types of covariance matrices are displayed in one GMM for explanation purposes only, the different types will not be mixed normally.

and descriptions in `RecordingFeatures` and `CategoryDescription` are GMMs saved as JSON strings. JSON is a compact string representation of objects – similar to XML, but shorter and less complicated. Arrays are displayed in `[]` with their elements seperated by `,`, objects are displayed in `{}`. Elements are key-value-pairs, their name is enclosed in `""`, key and value are seperated through `:`. Keys can be any of type object, array, string or numeric. For an example of a description of a GMM in JSON notation, see figure 5.9.

For now, `sqlite` is used as a backend (class `SQLiteDatabaseConnection`), which is compatible with the SQL-92 standard in most parts. Some features of SQL-92 are omitted[14], and some others, most notably dynamic typing[15], are non-standard extensions of SQL-92. Due to this, it is not possible to change the backend to another SQL database without – besides changing the API access functions – changing some of the SQL commands or parts of the database layout. This is not a property of `sqlite` in particular, most SQL databases speak their own SQL dialect.

`sqlite` was used since it is available on many platforms (including QNX, Linux, Windows and many others) and because it was designed as a *fast embedded* SQL database. To reduce the time needed to parse the SQL language and to reduce security risks through SQL injection, prepared statements have been used. With this technique, the SQL statements will be precompiled, without having the data fields set to a particular value. The

[14]see http://www.sqlite.org/omitted.html, 12/3/2012
[15]see http://www.sqlite.org/datatype3.html, 12/3/2012

data will be added afterwards, when the statement has been parsed, examined, and compiled to a format the database can easily handle. This precompiled statement can be reused with other data, and it does not need to be recompiled: the execution plan already has been created through the precompilation step, and execution only needs minimal resources.

An entity-relationship-diagram of the database layout can be seen in figure 5.10.

Tools

In this module, helper functions are collected. Most functions only are snippets and do not have a proper parent module, like `tolower()`, which converts a string to lowercase. Other helper functions include additional string manipulation functions, random number generation, testing if matrices are diagonal or not, POSIX thread wrapper classes, and console output coloring helpers.

`ProgressCallback` and `ProgressCallbackCaller` are special classes in this module. They are used by many other classes helping to indicate how far an operation has gotten if it is foreseeable that an operation will take a long time. These classes implement a simple version of the *observer* pattern with just one observer. Another name describing their use is *callback object*, referring to the callback functions used in imperative programming languages. Usage of these classes is simple: The class which wants to be called by another function upon progress simply needs to inherit from `ProgressCallback` and implement the function `progress()` with parameters `id : const std::string&`, `percent : double` and `progressMessage : const std::string&`. The `id` is intended to tell the function `progress()` which other function is calling back, since it is possible to tell many functions to call back on the same `ProgressCallback` object. The `id` can be compared to the phone number you see on the display of your phone when someone is calling. The parameter `percent` indicates the progress (values from the interval $[0, 1]$) and the string `progressMessage` is a textual representation of what is being done in the calling function. This string may be empty and is for informational purposes only.

`ProgressCallbackCaller` is a proxy class which allows to set the `id` to a fixed value upon construction. Instead of passing an object of type `ProgressCallback` to a function, it will be of type `ProgressCallbackCaller`, eliminating the need for this function to know the `id` with which it should call the `progress()` function of the object passed. The `id` will be automatically added to the `progress()` call by the `ProgressCallbackCaller` object. Example:

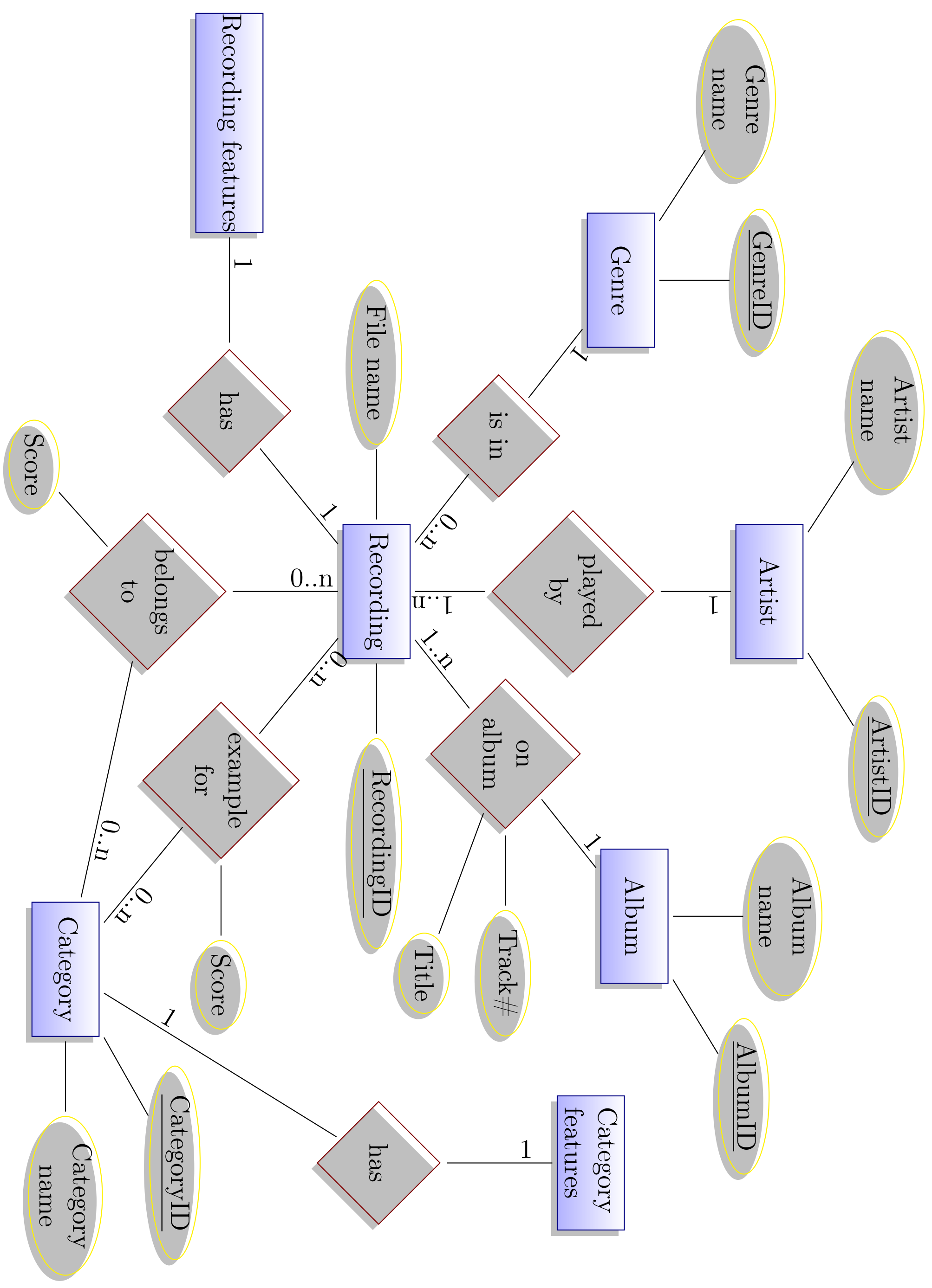


Figure 5.10: ER diagram of the database layout. The details of the recording features and category features entities are shown in figures 5.11 and 5.12.

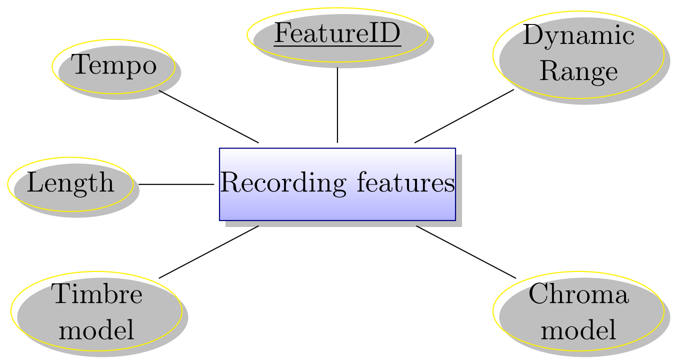


Figure 5.11: ER diagram of the database layout: Details of the recording features entity. All attributes that are called *model*s are saved as JSON in the database.

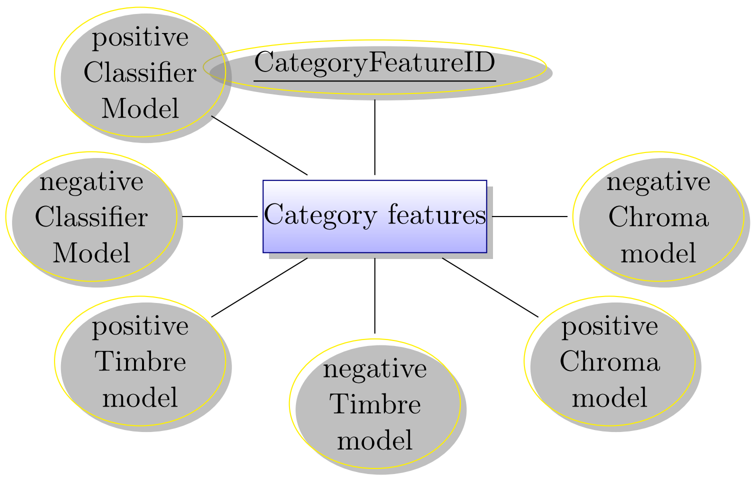


Figure 5.12: ER diagram of the database layout: Details of the category features entity. All attributes that are called *model*s are saved as JSON in the database.

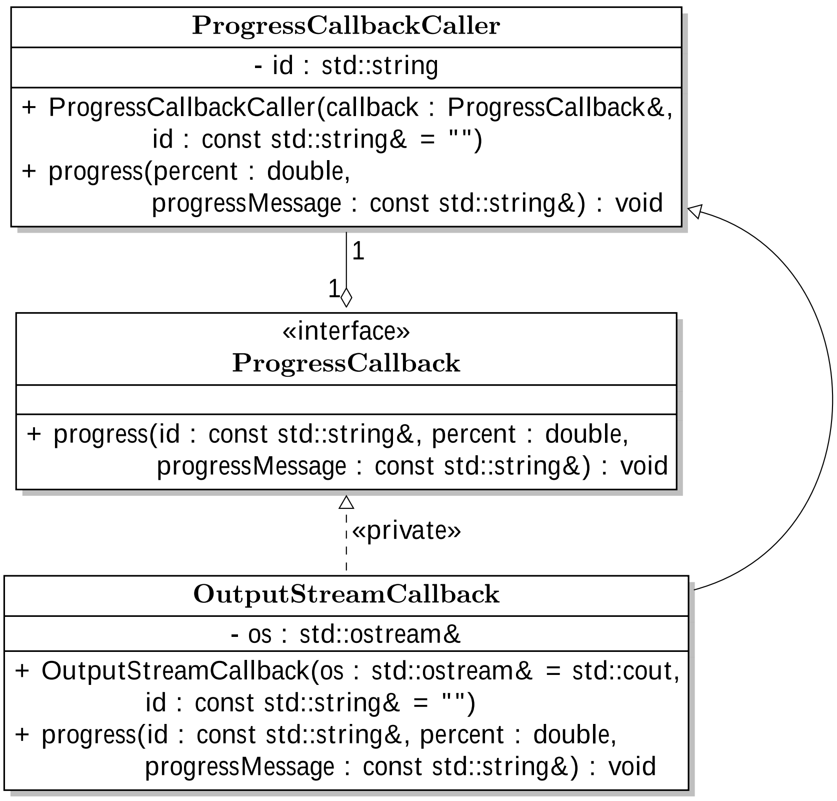

ProgressCallbackCaller
- id : std::string
+ ProgressCallbackCaller(callback : ProgressCallback&, id : const std::string& = "")
+ progress(percent : double, progressMessage : const std::string&) : void
1
1
«interface»
ProgressCallback
+ progress(id : const std::string&, percent : double, progressMessage : const std::string&) : void
«private»
OutputStreamCallback
- os : std::ostream&
+ OutputStreamCallback(os : std::ostream& = std::cout, id : const std::string& = "")
+ progress(id : const std::string&, percent : double, progressMessage : const std::string&) : void

Figure 5.13: UML class diagram of `ProgressCallback` and other classes simplifying its usage.

```
//assume that this object is a ProgressCallback object.
ProgressCallbackCaller* callback =
        new ProgressCallbackCaller(*this, "myName");
someOtherObject.doSomething(callback);
//someOtherObject will now call callback->progress(percent, message)
//from time to time. callback will call
//this->progress("myName", percent, message).
delete callback;
```

`ProgressCallbackCaller` is intended to be used by every function that is known to have a long runtime, e.g. `FilePreprocessor::preprocessFile()`. The function has a parameter `ProgressCallbackCaller* callback = NULL`. If the parameter will not be given, it is set `NULL` and the feature will not be used. If an appropriate object is given to the function, it works as described before. `ProgressCallback`s are used by all functions that are known to have a long runtime. A drawback of the chosen approach is the inability to inherit callbacks: If a function using a callback object itself calls a function with a long runtime, the `id` of the callback cannot be changed. For now, the old callback object will be passed to the new function, resulting in the `progress` parameter suddenly starting at `0.0`. This should be adressed in a future release of the software. There are several approaches possible:

- The `id` could be changed. Appending another string to the given `id` would be an option.
- The progress display could be changed, giving the minimal and maximal values of a specific call at construction time of a `ProgressCallbackCaller` (e.g. telling the object it should call `progress` with values from the interval [0.2, 0.3] instead of [0,1]).

`OutputStreamCallback` is an implementation of a `ProgressCallback` which outputs the progress and messages to a given output stream, normally `std::cout`.

Tests

In this module, all automatic software tests are bundled. Where applicable, a unit test for software submodules has been written[16]. Unit tests are automatic software tests testing a specific submodule (called *unit*) to meet an expected behaviour. In most cases, this will directly be the expected output of a function or class, but it may additionally be the timing or other things that can be tested to meet some requirements. The important part is that a unit test always tests a small unit and the interplay of different (previously tested) units.

[16] There are some parts where an automatic software test is hard to implement, especially where random numbers are part of the algorithm. These functions do not have a unit test for now. This should be adressed in a future release, a good testing strategy needs to be developed for these cases.

The build system `CMake` has an integrated unit test software part called `CTest`. It can be configured to call the binary of the software with some parameters and record its output, as well as the exit state or return value of the program. This tool is used in conjunction with a small unit test framework that has been developed for another program. It provides three macros: `CHECK(expr)`, `CHECK_EQ(expr, expr)` and `CHECK_EQ_TYPE(expr, expr, type)` with `expr` being a valid C++ expression returning a value of arbitrary type and `type` being a valid type the expression needs to be able to be casted to. For the return type of the arbitrary type, `operator==`[17] and `operator«`[18] need to be implemented. The source code of `expr` as well as its return value, line number and source code file name will be written to the standard error stream `std::cerr`. This way, it is possible to see from the logs which line of code failed to execute and why.

`CHECK(expr)` checks if the expression returns `true` or can be interpreted as `true`. If this is the case, execution goes on. If this is not the case, execution aborts and a nonzero value will be returned (`return EXIT_FAILURE;`). `CHECK_EQ(expr, expr)` does the same with checking both expressions for equality; `CHECK_EQ(expr, expr, type)` is an expansion of `CHECK_EQ` which first casts to `type` and then performs the comparision. In some cases, this raises the readability of the logs. A usage example for the macros is:

```cpp
#include "testframework.hpp"
#include <inttypes.h>
int testInteger() {
    int var=0;
    CHECK(!var);
    CHECK_EQ(var, 0);
    var++;
    CHECK(var);
    CHECK_EQ(var, 1);

    uint64_t var2 = 0;
    var2++;
    CHECK_EQ_TYPE(var, var2, uint64_t);

    CHECK(false);

    return EXIT_SUCCESS;
}
```

This short example gives the following output:

```
example.cpp:5:  !var == true?     - passed!
example.cpp:6:  var == 0?         - passed!
example.cpp:8:  var == true?      - passed!
example.cpp:9:  var == 1?         - passed!
example.cpp:13:  var == var2?     - passed!
```

[17] used for comparision of two values
[18] used for writing a value to the output stream

```
example.cpp:15:  false == true?  - failed!
           Value A: 0
           Value B: 1
```

These tools are sufficient to write unit tests. A failing test case indicates a problem in the code, if the test is not erroneous itself. Especially when code is used on various platforms, unit tests help in finding problems more efficiently: The same code can behave differently on different systems. The test cases help in finding problems with refactored[19] code, too, since the external behaviour of a module should be described through the test cases. A way to ensure that the test case covers the whole spectrum of behaviour is to first write the unit test and implement the functionality afterwards (this process is called *test-driven development*). The test cases written for this thesis have been written afterwards, but in the future, the author would prefer to follow the other approach, since it enforces to review the planned architecture of the software in an early stage of development and can help to avoid changes of a larger scale at a later point in time.

5.2.3 musiccmd

`musiccmd` is a command line interface for `libmusic`. It has been implemented to demonstrate the features of `libmusic`. The command line interface takes parameters and executes the commands encoded in the parameters. It can be used from shell scripts. User interaction after starting a command is not needed. A detailed description of the possibilities and all parameters available is given in the source code documentation or can be displayed through `musiccmd -help=all` and will be omitted here. A short example should suffice to understand the concept.

```
#Recursively add all files in /path/to/files to database, show progress.
#Feature extraction takes place here.
musiccmd -v -i -r --add-folder /path/to/files
#add new category "test"
musiccmd --add-category test
#Add all files containing "myart" in the artist field (e.g. myartist)
#as positive example, all files containing "gold" in the title field
#as negative examples, recalculate category memberships, all for
#category "test". "%" can be used as wildcard character.
#Classification takes place here.
musiccmd -v -i --edit-category test add-positive artist %myart% add-
    negative title %gold%
#Export the first 100 matches of category "test" to file "test.m3u".
#m3u files can be read by most music players.
musiccmd --export-category test test.m3u 100
```

[19] *Code refactoring* is the process of changing the code base in a way that internal functionality of modules changes without aﬀecting their external behaviour.

6 Testing

This section should give some examples for how well the proposed algorithm performs on a given test music collection.

6.1 The Dortmund music dataset

The Dortmund music dataset contains 1555 songs and is divided into the groups[1] *classical*, *electronica*, *jazz*, *pop_rock*, *rap* and *RnB*. The categories are built per album, all files of an album are tied to one category regardless wether one song of the album could have been assigned to another category.

Creating a database using `musiccmd` and extract all features of all songs of this database took approximately three hours on an AMD Athlon II X2 240e processor with 2x2.8GHz, which is about six hours of CPU time. This results in 14s per song in CPU time, or 7s in real time. These timings include decoding a file, applying the Constant Q transform, and extraction of all features (including the tempo feature, which was not used in the end for classification). It does not include any classification.

6.2 Testing `libmusic`s classification algorithm

Measuring the performance of the proposed classification algorithm is not straightforward, since it is not a classical supervised classification algorithm. For supervised learning algorithms, the performance measures are the *false positive rate*, which indicates the relative number of samples that have been mistakenly put in the positive group, and the *false negative rate* which is analog for the negative group.

The proposed algorithm gives a ranking instead of a binary classification. Its results are ordered lists of all recordings from the database. To get a measure for the performance of the algorithm, these lists are compared for several different inputs with labeled input data from the Dortmund music dataset.

[1] A complete list of the albums in the dataset with the attached category can be found at http://ls11-www.cs.tu-dortmund.de/rudolph/mi/albumlist.

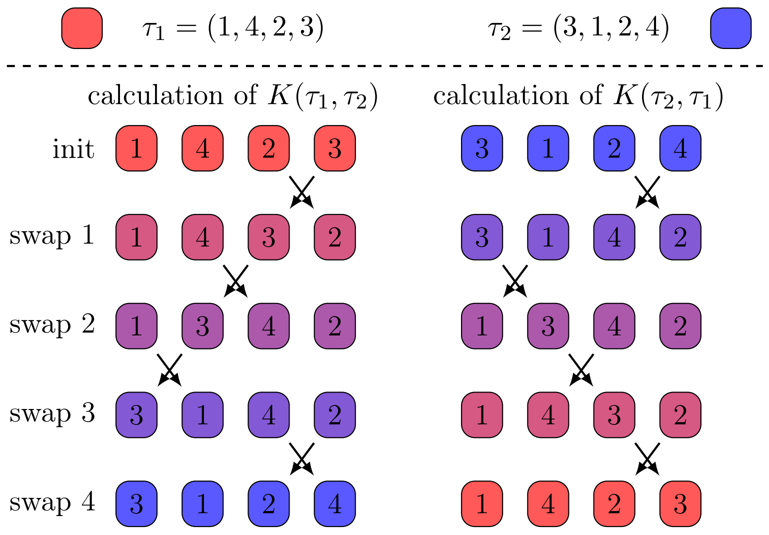


Figure 6.1: Example of the calculation of the Kendall τ distance using a Bubblesort approach. $\tau_1 = (1, 4, 2, 3)$, $\tau_2 = (3, 1, 2, 4)$; $K(\tau_1, \tau_2) = 4$ and $K(\tau_2, \tau_1) = 4$.

6.2.1 The Kendall τ/Bubblesort distance

The idea is that if a category is built of songs from within one label (e.g. *classical* or *jazz*), the distance to a category built of songs that have another label should be significantly larger than if the labels are identical, but the examples from equally-labeled songs differ.

This way, it can be measured how stable the rankings are when the input is changed. This is fundamentally different to a false positive/negative rate, but better reflects the properties of the learning algorithm.

A distance measure for lists is used for that purpose: the *Kendall τ distance*, which is also known as the *Bubblesort distance* (see e.g. [LM06] for a more formal definition).

Definition 6.1 (*Kendall τ distance*):
Let τ_1 and τ_2 be ordered lists with the same elements, their ordering may be different. The *Kendall τ distance* or *Bubblesort distance* $K(\tau_1, \tau_2)$ is defined as the minimal number of neighbouring swaps that is needed to transform τ_1 to τ_2.

The Bubblesort distance can be seen as the number of swaps a *Bubblesort* algorithm would take to transform τ_1 to τ_2. The Kendall τ distance is zero if the lists are identical, and it is commutative: $K(\tau_1, \tau_2) = K(\tau_2, \tau_1)$ holds.

The Kendall τ distance is used on the playlist files generated by `musiccmd` for a series of test cases. For this purpose, a program `bubblesort_distance` has been implemented, which takes two file names as arguments and calculates the row-wise bubblesort distance of these two files. The program additionally outputs a relative bubblesort distance, which is normalized by $n(n-1)/2$ with $n = |\tau_1| = |\tau_2|$.

For the test, six lists have been created: three for the group "classical", and three for the group "jazz". Each category has been trained with a single example from the group. The resulting lists are exported to playlist files, these files are compared.

The lists are not perfectly trained, but clearly contain many examples of the group in the first examples. The files can be found on the CD in the folder `playlists/` and manually be inspected. The file used to train the category is always the first one of the playlist, the group assignments are part of the file name. See table 6.1 for the results of the test. These results show: the Kendall τ distance as approach to testing this algorithm

File 1	**File 2**	**abs. distance**	**rel. distance**
`classical_1a.m3u`	`classical_1b.m3u`	574802	0.4757
`classical_1a.m3u`	`classical_1c.m3u`	598014	0.4949
`classical_1b.m3u`	`classical_1c.m3u`	607014	0.5023
`jazz_1a.m3u`	`jazz_1b.m3u`	580759	0.4806
`jazz_1a.m3u`	`jazz_1c.m3u`	600206	0.4967
`jazz_1b.m3u`	`jazz_1c.m3u`	602423	0.4985
`classical_1a.m3u`	`jazz_1a.m3u`	623572	0.5161
`classical_1a.m3u`	`jazz_1b.m3u`	626027	0.5181
`classical_1a.m3u`	`jazz_1c.m3u`	623598	0.5161
`classical_1b.m3u`	`jazz_1a.m3u`	623010	0.5156
`classical_1b.m3u`	`jazz_1b.m3u`	624633	0.5169
`classical_1b.m3u`	`jazz_1c.m3u`	616382	0.5101
`classical_1c.m3u`	`jazz_1a.m3u`	627480	0.5193
`classical_1c.m3u`	`jazz_1b.m3u`	611047	0.5057
`classical_1c.m3u`	`jazz_1c.m3u`	606734	0.5021

Table 6.1: Results of the Bubblesort distance test

has problems. At first sight, the numbers are less impressive than it was expected by the author. A closer look reveals that there nevertheless are regularities. The absolute bubblesort distance of same-group playlists is lower (mean $\approx$ 600000 swaps) than the distance of different-group playlists (mean $\approx$ 620000 swaps). So, this can be taken as a hint of that the algorithm does what it is expected to do.

The problem with the bubblesort distance approach is that the recordings at the end of the lists count the same as the recordings at the beginning of the list, which is not what is wanted. The recordings at the beginning of the list are more important and thus should have a higher influence on the list distance than the recordings at the end of the list. This leads to the number of swaps for comparing two lists of the same class being very high, even if the first elements are nearly in the same order.

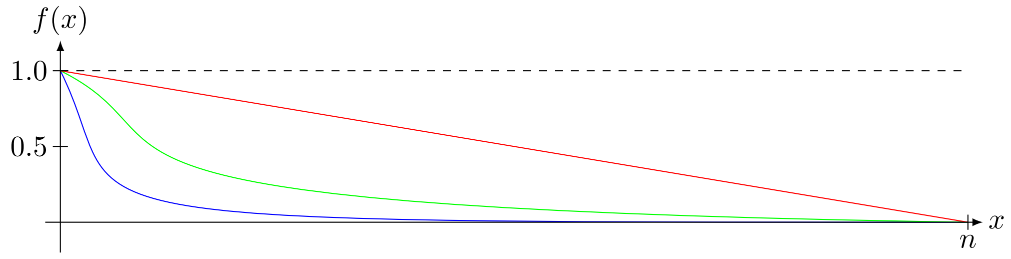


Figure 6.2: Possible weighting functions for the Kendall τ distance. Using the dashed line as weighting function is equivalent to the approach used at the time. The red function is easy to implement, but the green and blue functions might be better suited because they weight the first elements of the list overproportionally.

An idea to solve this problem is to weight the swaps of the algorithm and take the elements in front of the list more into account than the elements at the end of the list. For possible weighting functions, see figure 6.2. Using a weighting functions will most probably remove the commutativity. These ideas have not been implemented in this thesis.

After all, the tests show that although this measure has major shortcomings, the lists generated by examples of the same genre have a higher correlation than lists generated by different genres. To better show the performance of the algorithm, some other tests have been added.

6.2.2 Performance for 100 best matches of different categories

In this section, different examples for the performance of the algorithm are given. In every case, a category is defined through a set of positive examples originating from one group of the Dortmund dataset, and a (possibly empty) set of negative examples originating from another group of the Dortmund dataset. It is then measured how many of the 100 best matches of the algorithm originate from the group through which the category was defined.

Group "classical" For a model of classical music trained with three positive examples (*Beethoven – Sonata quasi una Fantasia No.14 in C# minor, Op 27 No.2 "Moonlight" - Adagio sostenuto*, *Mozart – Sinfonie A-Dur,KV201/1Allegro moderato* and *Mozart – Sinfonie A-Dur,KV201/4Allegro con spirito*) and three negative examples (*John Coltrane – Naima*, *Hancock – Naima*, *Sonny Rollins – Friday the 13th*), 94% of the 100 best matches are from the "classical" group of the Dortmund dataset, with the first match not being from the classical group at rank 57.

Groups “Jazz/RnB” For a model of jazz/rhythmn and blues music trained with two positive (*Hancock – Misstery* and *Charlie Parker – Overtime*) and two negative examples (*Jeff Wayne – Brave New World* and *Robert Schumann – Symphony No.1, op 38 ”Spring” - IV. Allegro animato e grazioso*), 89% of the 100 best matches are from the “jazz” or “rnb” group, with the first match not being from these groups at rank 41.

Group “electronica” It was not possible to construct a model for this group with a performance better than about 30%. This performance can be achieved using nearly any combination of three positive examples and three negative examples. A reason could be that many recordings in this group are highly percussive and thus are similar to many recordings of the other groups.

Group “pop_rock” For a model of pop/rock music trained with two positive (*Madsen – Goodbye Logik* and *Coldplay – Talk*) and one negative example (*2Pac – F*** the world*), 87% of the 100 best matches are from the “pop_rock” group, with the first match not being from this group at rank 13.

6.2.3 Defining a subcategory: Metal out of “pop_rock”

Using five positive (*Nightwish – Bless the child*, *Nightwish – Feel for you*, *Therion – Ginnungagap*, *Dream Theater – Take the time* and *Disturbed – Stricken*) and three negative examples (*Scooter – Well done, Peter*, *Leni Stern – Blue cloud* and *Phil Collins – We wait and we wonder*), it is possible to define a category for *Metal* music. This category is not a group within the Dortmund dataset. However, the author was able to find many metal songs in the “pop_rock” group by using the following approach:

1. An initial song must be known. Add this song to a new category. Recalculate the category scores.
2. Inspect the list created by the set of (positive/negative) examples, beginning on the top.
   a) If a new song matching the category is found and that song does not feel like a direct neighbour of a song in the positive group, add that song as positive example to the category. Proceed at 4.
   b) If a song is found that heavily mismatches the category, add it to the negative example set in case the positive example set has two or more members. Proceed at 4.
3. If more than about 40 list items were inspected, end.
4. Recalculate the category scores. Repeat from 2.

Of course, this approachs works well for other new categories, too.

The metal music category was created using this approach. The 20 best matches of the category were all considered to be metal music, out of 55 songs that were considered to belong to that group (all albums from the artists of the positive example set, plus an album from the group *In Flames*). 45 of the metal songs were in the first 100 matches.

6.2.4 Summary of the test results

So, what should be read from these results? On the one hand, it should be seen that the algorithm is capable of performing standard music classification tasks. The algorithm is able to seperate classical music from pop/rock, as well as jazz/rhytmn and blues from classical music and other pop/rock songs.

On the other hand, it should be seen that there are tasks that the algorithm cannot solve. For instance, the group "electronica" could not be defined using the proposed algorithm. The strengths of the algorithm lie in being a tool to help exploring music collections. It makes suggestions based on examples, and the suggestions should help users in finding out about recordings they forgot or never did hear.

7 Conclusion

In this master's thesis, an algorithm for music similarity analysis was developed. It is capable of computing rankings for music with regard to personal music categories. The algorithm uses the dynamic range of a recording, as well as its length and a model for both chroma and timbre as features. The timbre feature is derived using a Constant Q Cepstrum, and the chroma feature is made key-invariant through estimation of the key of the recording. A tempo feature is extracted but not used, since its performance is not good enough to help in classification. The definition of the personal categories is done through a set of positive and negative examples, where negative examples can be omitted if necessary. The number of songs needed to define a category is very low, good results can be achieved using only three to ten examples. Tests show that the algorithm indeed is capable of performing the tasks it is designed for, although it has some problems with specific types of music (e.g. electronic music).

Suggested future work Some of the algorithms need to be refined. This is the case for the tempo estimation. This algorithm does not work reliably and is only able to estimate the tempo up to a power of two factor. A more robust peak detection algorithm would raise the recognition rate significantly. The use of multiple frequency bands for recognition of different percussive instruments would help, too.

Additionally, the key estimation algorithm and the key-invariant chroma models should be addressed: At the time, the key-invariant chroma models ignore a change of key within a song. This shortcoming should be removed. If the key estimation could give times at which the key changes, the key-invariant chroma could cover cases of key changes, too.

For the GMMs and the EM algorithm, a better initialization technique is recommended. Currently, the EM algorithm is run multiple times to get a better result, each time initialized with a random set of data points. For the initialization step, a run of the k-means algorithm could be used, e.g. with a fixed and small number of iterations. The k-means algorithm itself is already implemented (see `kmeans.hpp`, `kmeans.cpp` in the `classification` module), so this should only be a minor effort. These steps are expected to allow to speed up the feature extraction process by a factor of 1.5-2, as well as the recalculation of the categories.

The features introduced in this thesis are used with the same importance. It might be interesting to weight the importance of the features. At the moment, the importance of the features is scaled by the use of the Mahalanobis distance: Features with large variance in the example set are rendered unimportant due to the distances getting smaller in these cases. It would however be interesting to additionally set for instance the length of a recording less important than the timbre and chroma similarity. This could be achieved by changing the definition of the Mahalanobis distance to e.g.

$$d_{\boldsymbol{\Sigma},\boldsymbol{D}}(\boldsymbol{x},\boldsymbol{y}) = \sqrt{(\boldsymbol{x}-\boldsymbol{y})^T \boldsymbol{D}^{-1} \boldsymbol{\Sigma}^{+} (\boldsymbol{x}-\boldsymbol{y})} \tag{7.1}$$

with $\boldsymbol{D}$ being a diagonal matrix with *importance factors* on the diagonal. The importance factor $(\boldsymbol{D})_{ii}$ is related to the feature dimension i and should be proportional to the importance of the feature.

But that is another story, to be told another time.

List of Figures

Bibliography

[ANP11] Fabrizio Argenti, Paolo Nesi, and Gianni Pantaleo. Automatic Transcription of Polyphonic Music based on the Constant-Q Bispectral Analysis. *IEEE Transactions on Audio, Speech and Language Processing*, 19(6):1610–1630, August 2011.

[AP02] J.J. Aucouturier and F. Pachet. Music similarity measures: What's the use. In *Proceedings of the 3rd International Symposium on Music Information Retrieval*, page 157–163, 2002.

[Ass60] American Standards Association. Acoustical Terminology. 1960.

[BCR91] G. Beylkin, R. Coifman, and V. Rohlkin. Fast Wavelet Transforms and Numerical Algorithms I. *Communications on Pure and Applied Mathematics*, (44):141–183, 1991.

[Beh04] Ehrhard Behrends. *Analysis*, volume 2. Vieweg, April 2004.

[Bis95] Christopher M. Bishop. Clarendon Press, Oxford, 1995.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[BP92] Judith C. Brown and Miller S. Puckette. An efficient algorithm for the calculation of a constant Q transform. *J. Acoust Soc. Am.*, 92(5):2698–2701, 1992.

[BP05] Juan P. Bello and Jeremy Pickens. A Robust Mid-level Representation for Harmonic Content in Music Signals. *Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR-05)*, pages 304–311, September 2005.

[Bro91] Judith C. Brown. Calculation of a constant Q spectral transformation. *J. Acoust Soc. Am.*, 89(1):425–434, January 1991.

[Bro99] Judith C. Brown. Computer identification of musical instruments using pattern recognition with cepstral coefficients as features. *Acoustical Society of America*, 105(3):1933–1941, March 1999.

[BZ10] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic.* Cambridge University Press, 2010.

[CLRS10] Thomas H. Cormen, Charles E. Leierson, Ronald Rivest, and Clifford Stein. *Algorithmen - Eine Einführung.* Oldenbourg Verlag, München, 3. edition, 2010.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[FTZ11] Zhouyu Fu, Kai Ming Ting, and Dengsheng Zhang. A Survey of Audio-Based Music Classification and Annotation. *IEEE Transactions on Multimedia*, 13(2):303–319, April 2011.

[Gen03] James E. Gentle. *Random Number Generation and Monte Carlo Methods.* Springer, 2. edition, 2003.

[Got01] Masataka Goto. An Audio-based Real-time Beat Tracking System for Music With or Without Drum-sounds. *Journal of New Music Research*, 30(2):159–171, 2001.

[Har78] Frederic J. Harris. On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proceedings of the IEEE*, 66:51–83, January 1978.

[JECJ07] Jesper H. Jensen, Daniel P.W. Ellis, Mads G. Christensen, and Søren H. Jensen. Evaluation of Distance Measures between Gaussian Mixture Models of MFCCs. In *ISMIR 2007: Proceedings of the 8th International Conference on Music Information Retrieval*, pages 107–108, Vienna, September 2007.

[Jen99] Kristoffer Jensen. *Timbre Models of Musical Sounds.* PhD thesis, University of Copenhagen, 1999.

[KKF00] T. Kinnunen, T. Kilpeläinen, and P. Fränti. Comparison of clustering algorithms in speaker identification. *Proceedings IASTED Int. Conf. Signal Processing and Communications*, 1:222–227, 2000.

[Kre05] Ulrich Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik.* vieweg studium, Wiesbaden, 8. edition, 2005.

[KVJ93] M. Karjalainen, V. Välimäki, and Z. Jánosy. Towards high-quality sound synthesis of the guitar and string instruments. In *Proc. ICMC*, pages 56–63, 1993.

[Lee06] Edward A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[LM06] N. Lesh and M. Mitzenmacher. BubbleSearch: A simple heuristic for improving priority-based greedy algorithms. *Information Processing Letters*, 97(4):161–169, 2006.

[Log00] Beth Logan. Mel Frequency Cepstral Coefficients for Music Modeling. In *International Symposium on Music Information Retrieval*, volume 28, page 5, 2000.

[MMR+01a] K.R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *IEEE transactions on neural networks*, 12(2):181–201, 2001.

[MMR+01b] Klaus-Robert Müller, Sebastian Mika, Gunnar Rätsch, Koji Tsuda, and Bernhard Schölkopf. An Introduction to Kernel-Based Learning Algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201, March 2001.

[NS06] K. Noland and M. Sandler. Key estimation using a hidden Markov model. In *Proceedings of the 7th International Conference on Music Information Retrieval (ISMIR)*, page 121–126, 2006.

[OS95] Alan V. Oppenheim and Ronald W. Schafer. *Zeitdiskrete Signalverarbeitung*. R. Oldenbourg Verlag, München, 2. edition, 1995.

[PA05] F. Pachet and J.J. Aucouturier. "The way it Sounds": timbre models for analysis and retrieval of music signals. *IEEE Transactions on Multimedia*, 7(6):1028–1035, 2005.

[Pee06] G. Peeters. Chroma-based estimation of musical key from audio-signal analysis. In *Proc. of the 7th International Conference on Music Information Retrieval (ISMIR)*, page 115–120, 2006.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes. 2007.

[RTG98] Y. Rubner, C. Tomasi, and L.J. Guibas. A metric for distributions with applications to image databases. In *Computer Vision, 1998. Sixth International Conference on*, page 59–66. IEEE, 1998.

[SB05] Josef Stoer and Roland Burlisch. *Numerische Mathematik 2.* Springer, Berlin, Heidelberg, New York, 5 edition, 2005.

[Sch05] Ingo Schnitt. *Ähnlichkeitssuche in Multimedia-Datenbanken: Retrieval, Suchalgorithmen und Anfragebehandlung.* Oldenbourg Wissenschaftsverlag, 2005.

[Sch11] Dominik Schnitzer. *Indexing Content-Based Music Similarity Models for Fast Retrieval in Massive Databases.* PhD thesis, Johannes Kepler Universität Linz, October 2011.

[SK10] Christian Schörkhuber and Anssi Klapuri. Constant-Q transform toolbox for music processing. In *7th Sound and Music Computing Conference*, Barcelona, Spain, 2010.

[Sto05] Josef Stoer. *Numerische Mathematik 1.* Springer, Berlin, Heidelberg, New York, 9 edition, 2005.

[TC02] George Tzanetakis and Perry Cook. Musical Genre Classification of Audio Signals. *IEEE Transactions on Speech and Audio Processing*, 10(5):293–302, July 2002.

[TVE08] Wolfgang Theimer, Igor Vatolkin, and Antti Eronen. Definitions of Audio Features for Music Content Description. Technical report, February 2008.

[TVM$^+$11] Wolfgang Theimer, Igor Vatolkin, Rainer Martin, Christian Igel, Holger Blume, Bernd Bischl, Martin Botteck, Günther Roetter, Günther Rudolph, and Claus Weihs. Huge Music Archives on Mobile Devices. *IEEE Signal Processing Magazine*, page 24–39, July 2011.

[YGKM96] S. Ystad, P. Guillemain, and R. Kronland-Martinet. Estimation of parameters corresponding to a propagative synthesis model through the analysis of real sounds. In *Proc. ICMC*, 1996.

[You39] R. W. Young. Terminology for Logarithmic Frequency Units. *Acoustical Society of America Journal*, 11:134, 1939.

[Zie00] Wieland Ziegenrücker. *ABC Musik.* Breitkopf & Härtel, Wiesbaden, 3. edition, 2000.

Glossary

CQCC Constant Q Cepstral Coefficients, see 3.3.2 for an explanation.. 30, 31, 34

CQT Constant Q transform, refers to a technique that transforms a signal from the time-domain $x(n)$ to the frequency domain, but in contrast to the Fourier transform, the center frequencies of the frequency-bins are geometrically spaced and their Q-factors are all equal (see [SK10]).. 8, 12–14

DCT Discrete Cosine transform. 7, 8, 39

DFT discrete fourier transform, refers to the usage of Fourier series to approximate a function.. 11, 13, 14

dominant The dominant is the chord `V` of a scale. In major scales, it has the same mode as chord `I`, in minor scales it can occur in both modes major and minor.. 56–58

DSP digital signal processing, signal processing with digital systems such as computers. 4, 7

EM Expectation-Maximization algorithm. 35, 47, 62, 83, 102

EMD Earth Mover's distance. 41

FFT fast fourier transform, refers to an algorithm that is capable of calculating the discrete Fourier transform with an algorithmic complexity of $\Theta(n \lg n)$ ([CLRS10, p. 930]).. 7, 9, 14, 81

GiB gibibyte, refers to 2^{30} bytes. $1\text{GiB} = 1024\text{MiB}$.. 5

GMM Gaussian Mixture Model. 32–35, 37, 38, 41, 46, 62, 68, 83, 88, 102, 105

IEC International Electrotechnical Commission, a standards organization.. 4

JSON JavaScript Object Notation. 88, 105

KiB kibibyte, refers to 2^{10} bytes. $1\text{KiB} = 1024\text{B}$.. 5

KL Kullback-Leibler divergence. 41, 44

LCG Linear Congruency Generator. 44

LDA Linear Discriminant Analysis. 63, 65, 84

MFCC Mel Frequency Cepstral Coefficients, see 3.3.2 for an explanation.. 28–31

MiB mebibyte, refers to 2^{20} bytes. $1\text{MiB} = 1024\text{KiB}$.. 5

PDF probability density function. 32, 33, 35, 38, 39, 46, 70

RVM Relevance Vector Machine. 65

SI International System of Units (from french *Système international d'unités*), a metric system for physical units.. 4

SVD Singular Value Decomposition. 71, 80

SVM Support Vector Machine. 63, 65, 67

tonic The tonic is the chord `I` of a scale and defines its name. If the chord is minor, the scale will be called minor. Example: If the tonic is `C`, the scale is called `C`. The tonic is the most important chord of a scale.. 56–58

XML eXtensible Markup Language. 88