Utrecht University

Master's Thesis in
Computing Science

The Descriptive Power of Chords: Music or Noise?

Submitted by

Dirk van Wijk

ICA-3647196

Supervisors:
Dr. Frans Wiering
Prof. Dr. Remco Veltkamp
Dr. Bas de Haas

May 3, 2016

Abstract

In the field of music information retrieval (MIR), several features can be extracted from audio, which can then be used for tasks such as query-by-humming or chord extraction. However, chord extraction can also be applied on non-harmonic (non-musical) audio, although this generally results in meaningless chord sequences.

We are interested in distinguishing harmonic from non-harmonic audio, based on the extracted chords only. This may benefit services, such as Chordify, that automatically extract chords from musical audio, so musicians can play along with a song. As these services would only like to show meaningful chords to their users, being able to filter out chords for non-harmonic audio is useful for them. We divide the audio in two groups: H-music, which is music that follows the rules of western harmony, and non-h-music, which is everything that is not h-music (non-musical audio, but also atonal and percussive music).

In this thesis, we study three novel tasks that are all applied on chord sequences only: 1) We classify an extracted chord sequence as either h- or non-h-music, 2) we segment an extracted chord sequence into parts of h- and non-h-music with a novel segmentation algorithm, and 3) we assign a quality score to an extracted chord sequence; this score indicates to which degree a Chordify user finds the chord sequence acceptable. A framework is constructed that is able to perform the tasks mentioned above, by making use of two different models that can describe (a set of) chord sequences: a language model, which we use to create a probability distribution over words, and a chord histogram, which stores the relative distribution of the chords in a song.

With our framework, we are able to accurately distinguish h-music from non-h-music. The chord extraction algorithm also affects the performance, as the CHORDINO algorithm appears to give us better results than Chordify's own chord extraction algorithm. We are able to predict relatively well, within a margin of tens of seconds, at which point in time in an audio file there is a switch from h-music to non-h-music and vice versa (the segmentation task). Predicting the quality of a chord sequence proves to be more difficult, as our predictor requires more data than we currently have.

Additionally, we have constructed our own data set that consists of several hundred creative commons non-h-music audio files and radio podcasts, which we have used for our experiments. This data set is made publicly available.

CONTENTS

1 INTRODUCTION 5

2 BACKGROUND AND STATE OF THE ART 7

2.1 Musical Terminology 7

2.2 Chord Extraction 8

2.2.1 Template Pattern Matching 8

2.2.2 Hidden Markov Models 9

2.2.3 Chordify Harmony Model 11

2.2.4 Performance of Chord Extraction Algorithms 11

2.3 Simulated Annealing 12

2.4 Methods for Classification 12

2.4.1 Feature Classification 13

2.4.2 N-Gram Language Models 16

2.5 Segmentation Methods 20

2.6 Regression Analysis 21

2.7 Evaluation Methods 22

2.7.1 Classification Evaluation Methods 22

2.7.2 Segmentation Evaluation Methods 23

2.7.3 Quality Assessment Evaluation Methods 24

2.7.4 Testing the Statistical Significance 24

3 PIPELINES AND ALGORITHMS 26

3.1 H- and Non-H-Music Classification Pipeline 26

3.1.1 Classification on Chord Features 26

3.1.2 Chord Language Model Classification 29

3.2 H- and Non-H-Music Segmentation Pipeline . . . 31
3.2.1 Segmentation Algorithms . . . 34
3.3 Assigning Quality Scores to Chords . . . 38
3.3.1 Rating System . . . 38
3.3.2 Score Prediction Pipeline . . . 43
4 DATA 45
5 CLASSIFICATION OF H-MUSIC AND NON-H-MUSIC 49
5.1 Experiments and Results . . . 49
5.1.1 Feature Classification Experiments . . . 49
5.1.2 Language Model Experiments . . . 56
5.2 Discussion . . . 62
6 SEGMENTATION OF H-MUSIC AND NON-H-MUSIC 63
6.1 Experiments and Results . . . 63
6.1.1 Variable Length Classification Experiments . . . 63
6.1.2 Segmentation Experiments . . . 66
6.2 Discussion . . . 73
7 ASSESSING THE QUALITY OF CHORDS 75
7.1 Case Study on Quality Assessment . . . 75
7.1.1 Experiments . . . 75
7.2 Discussion . . . 76
8 CONCLUSIONS AND FUTURE WORK 77
8.1 Conclusions . . . 77
8.2 Future Work . . . 78
Appendices 80
A IMPLEMENTATIONS 81
A.1 Feature Classification . . . 81

A.2 Language Model Classification 82
A.3 Regression methods 82
B EXTRA RESULTS 83
B.1 Multiple LMs 84
B.2 Additional Results 85

1

INTRODUCTION

In the field of music information retrieval (MIR), which is the science of extraction information from music, there are several tasks, such as query-by-humming or chord extraction. Chord extraction, which is a very active research topic in (MIR) at the moment, is the task of acquiring a chord sequence from a piece of audio (music). There are many applications of chord extraction such as finding similarities between songs, classifying music into genres or allowing players of a musical instrument to acquire music tablature automatically.

Although chords have harmonic properties, which can only be found in western tonal music, it is possible to extract chords from non-harmonic audio such as movies or percussive music. If we extract chords from this kind of audio, we obtain chord sequences that do not make much sense. We are interested in filtering out chord sequences of audio that do not contain a harmony, as these chord sequences are meaningless.

A company that makes use of chord extraction and may benefit from filtering out non-harmonic audio is Chordify[1]. In their web application people can submit audio (YouTube, SoundCloud or an audio file) and Chordify will show the (automatically generated) chords alongside the audio being played. Users can also browse a large library of songs that have already been ”chordified” by other users and play along with those songs. Chordify wants that users have a very easy to use method to play the chords alongside a song. Being able to filter out non-harmonic audio (segments) would benefit companies such as Chordify, since the user experience will improve when users are given only meaningful chords.

We will now define the two types of audio more precisely:

H-MUSIC This is music that contains a chord sequence that is meaningful in western tonal harmony. Harmony is the theory and practice of how chords are used. Even though chords could be extracted from any kind of audio, that doesn't necessarily mean that these chords follow the rules of harmony. So atonal music for instance, is not considered h-music, because the chords in atonal music do not follow the rules of harmony.

NON-H-MUSIC This is any kind of audio that is not h-music. So this includes all kinds of sounds such as people talking, random noise and environmental sounds, but also music without a harmony such as atonal or percussive music.

Our initial goal of this research is to create a method that can determine whether a chord sequence corresponds to h-music or to non-h-music. We will take a computational approach to reach this goal and make use of classification techniques to classify chord sequences as h-music or non-h-music.

[1] `https://chordify.net`

We can do such a classification for the entire chord sequence, but we can also look at segments of the sequence. This could for example be interesting if we have a chord sequence for a movie or a long radio recording. A person is only interested in the parts that contain music. We could then determine what segments contain h-music and which do not. With the use of this segmentation we could choose to only display chord information for the parts that actually contain h-music.

As the chord extraction task is far from being solved, many chord extraction algorithms also produces chord sequences for h-music that contain errors, meaning that they could still be improved. This leads to another interesting problem, namely, how can we assess or rate the quality of an h-music chord sequence. When we are given a chord sequence that corresponds to h-music (assuming classification has already taken place), can we give some quality measure to the chord sequence? This measure should reflect to which degree the user thinks the chords match the audio. When, for example, Chordify has two chord sequences of the same song (but one recording could be a of a poorer quality, resulting in a worse extraction of chords), we can use the chord sequence that has a better score. If we are able to find recurring chord sequences for low or high quality chords, we might be able to learn why users prefer certain chord sequences over others.

The main research question is as follows:
For a given chord sequence, can we determine if the sequence corresponds to h-music and if it does, can we determine to which degree a user finds the chord sequence acceptable? Can we also segment a chord sequence into parts that contain h-music and parts that do not?

It remains to be determined what exactly a high quality chord sequence is in the eyes of a Chordify user.

There are some concrete problems that we will be looking at:

1. **Classification**: Given a chord sequence, can we determine whether the chord sequence corresponds to h-music or non-h-music?
2. **Segmentation**: Can we make a segmentation of the chord sequence such that we obtain h-music and non-h-music segments?
3. **Quality Assessment**: When we deal with a chord sequence that corresponds to h-music, can we give a score to this sequence that reflects to which degree a Chordify user finds the chord sequence acceptable?

Apart from presenting a solution for these three novel research problems, we provide another contribution in the form of a data set. For our research we require a data set with h- and non-h-music, which we will carefully construct ourselves. Since the non-h-music portion of this data set only contains creative commons audio, it can be made publicly available. The data set, together with its download link, is further discussed in Chapter 4 .

This thesis is outlined as follows. Chapter 2 will provide background information that is needed in order to understand the remaining chapters. We will start with explaining some musical terminology and concepts that are used throughout this thesis and then follow it up with some state of the art computational approaches that we will use in this research. In Chapter 3 we will introduce some ideas and algorithms that we have come up with to tackle the research problems, and we discuss the pipelines that will be used for our experiments. In Chapter 4 we will discuss which data is needed for our experiments. The experiments and evaluation of the experiments follow in Chapters 5, 6 and 7 for classification, segmentation and quality assessment respectively. We will draw our conclusions and discuss potential future work in Chapter 8.

2

BACKGROUND AND STATE OF THE ART

This chapter starts with explaining some musical terminology and concepts that are used throughout this thesis. After that, we will explain the basics of chord extraction, since we are working with chord sequences that are obtained from chord extraction. Following that, we will discuss some promising computational approaches that we will use in this research. We start with feature classification techniques and language models in Section 2.4. Then discuss the state of the art in musical segmentation methods in Section 2.5. Following that, in Section 2.6 we will discuss a few methods that can help us with obtaining quality scores for chord sequences. Lastly, we will discuss how we can evaluate the performance of the techniques of the previous sections.

2.1 MUSICAL TERMINOLOGY

Music at the most basic level consists of *notes*. A note has a perceived *pitch*, which is referred to with a letter ranging from A to G, with possibly a flat (♭) or a sharp (#) (called accidental) attached to it. It has an *onset*, which is the time at with the note starts, and a duration. Apart from the perceived pitch of a note, which is the fundamental frequency of the sound that you hear, often overtones are also produced. These are sounds of a different, but weaker, frequency.

The distance between two notes is called an *interval* and is measured in semitones. An interval with a size of 12 semitones is called an octave. Two notes that are an octave apart have the same perceived pitch, and the higher note has double the frequency of the lower note. An octave contains the following notes: A, A#/B♭, C, C#/D♭, D, D#/E♭, E, F, F#/G♭, G. A sharp raises a note by half a semitone and a flat lowers the note by half a semitone. So for example, C# and D♭ have the same pitch and are sometimes considered to be the same note.

Two or more notes that sound together (either simultaneously or shortly after each other) result in a *chord*. The specific intervals that are being used determine what kind of chord is played. For example, starting with a C note followed by an interval of length four and then an interval of length seven (from the root note) results in the notes C-E-G and is a C major triad chord (often denoted as just the C chord or Cmaj chord). On the other hand, starting with C and then an interval of length three and seven respectively, results in C-E♭-G, which is a C minor triad chord (often denoted as Cm or Cmin).

The number of chords that are used in a single music piece isn't unlimited and depends on several factors. A musical scale defines the notes (and chords) that can be used and that fit well together, which can of course be subjective. In western music, the minor and major scale are most popular, which consist of seven root notes (per octave). In Asian music the pentatonic scale with only five root notes is often used.

In experiments involving chords, such as chord extraction or genre classification, a specific vocabulary of chords is used. Harte et al. [1] have proposed a chord grammar in which millions of chords can be defined, whereas generally only hundreds of chords appear in a data set. It has become common in the MIR field to use a simple vocabulary that uses the 12 root notes of the octave (where sharp and flat counterparts are considered to be the same note) of only the major and minor chord type, resulting in 24 different chords. Often a special N chord is used as well for silence or an unrecognizable chord.

2.2 CHORD EXTRACTION

Chord extraction is a prerequisite of our research. Even though this research is not about chord extraction itself, it is important to understand how the chord sequences that we use are obtained. Since we are trying to classify and assess the quality of chord sequences, it could prove useful to gain some insight in the extraction process and learn what could affect the quality. We will first explain two of the most common methods for chord extraction and after that describe how Chordify extracts its chords.

All chord extraction algorithms have one thing in common: A so-called *chromagram* or *Pitch Class Profile (PCP)*, which is introduced by both Fujishima [2] and Wakefield [3]. This is a vector that represents the relative strength of all 12 semitones (regardless of octave) at a certain time frame in an audio signal. Often, 12 dimensions are used, where each dimension corresponds to one semitone. Sometimes more than 12 dimensions are used, to compensate for mistuning. The amount of frames (and PCPs) depends on the resolution that is worked with. A higher resolution results in having more frames, thus more PCPs.

A PCP is obtained by first applying the Fast Fourier Transform (FFT) or Constant Q Transform (CQT) on a time frame of the audio signal. This transformation divides all the frequencies of the signal over several bins. In the most simple case we have 12 bins; one bin for each pitch. From these bins we can read the strength of every pitch, resulting in a PCP. When more than 12 bins are used, we check whether bins that are directly below or above the bin of a pitch contain more strength. If this is the case, we know there is a small down- or up-tuning.

A problem that often occurs is that the fundamental frequency that we want to capture in a bin also has several overtones, frequencies that are multiples of the fundamental frequency. These overtones end up in the bins as well, which is undesired. Section 2.2.4 mentions a solution for this problem.

Next, we will explain two paradigms of chord extraction; one that uses a template matching method and one that makes use of statistical learning with Hidden Markov Models (HMMs).

2.2.1 *Template Pattern Matching*

In template pattern matching the previously mentioned PCP is compared with a set of predefined chord templates (CTs), which are represented as a binary mask. This mask uses the template of T = $[Z_C, Z_{C\#}, Z_D, Z_{D\#}, ..., Z_{A\#}, Z_B]$. This is a vector of zeros and ones that represents which notes are present in the chord and which are not.

We can define such a mask for every chord. For example, the masks of C-major and D-minor are:

$T_{C:maj} = [1,0,0,0,1,0,0,1,0,0,0,0]$ and $T_{D:min} = [0,0,1,0,0,1,0,0,0,1,0,0]$

The next step is to calculate the distance between the PCP and the CTs of every chord. The CT that has the shortest distance (thus highest similarity) with the PCP will be the chord that the PCP represents. The most often used distance measures as proposed by Fujishima [2] are the nearest neighbor and weighted sum measures:

NEAREST NEIGHBOR This calculates the euclidean distance between the PCP and the CT. As more pitch values of the PCP match the binary mask, the result of the sum becomes smaller. Chord c with the smallest score will be the chord that the PCP represents.

$$Score_{nearest,c} = \sqrt{\sum_{p=0}^{11} (T_c(p) - PCP(p))^2} \tag{1}$$

WEIGHTED SUM The dot product is calculated between the PCP and a weighted variant of the CT, which is not binary anymore. The weights are user defined and result in some pitches having a bigger impact in the CT than others. The higher the resulting dot product, the more the PCP matches the (weighted) CT.

$$Score_{weighted,c} = \sum_{p=0}^{11} W_c(p) PCP(p) \tag{2}$$

Apart from these simple measures, some other measures have been proposed such as the Kullback-Leibler and Itakura-Saito divergences [4]. These measures react more strongly towards missing or extra notes in the PCP.

After the chords have been obtained from the signal, a smoothing step can also be applied. Generally, chords don't change very often (i.e. less than every second), therefore chords that appear in only one or two frames can be smoothed out with a median filter, for example.

2.2.2 *Hidden Markov Models*

HMMs have their origins in speech recognition, but have been introduced for chord extraction by Sheh and Ellis [5]. We will first explain how a regular Markov model works by means of an example and then extend the example to an HMM.

This example will be about the weather forecast. Once a day at a specific time, the weather is observed. There are four different weather conditions: Rain, snow, sun, clouds. These are the different states of the Markov model. When we have certain weather i on day t ($s_t = i$), we know what the probability is of every weather state for the next day. This is the transition probability a_{ij} that tells us what the odds are of going from state i to state j. Formally $a_{ij} = P(s_t = j | s_{t-1} = i)$ (the probability of being in state j on day t given that on day $t-1$ we were in state i).

Note that we only know the transition probability between two subsequent days. This is because of the so-called *Markov property* that tells us that the future (day $t+1$) is independent of the past (day $t-1$) given the present (day t).

In this model we can always observe in which state we are (we can always observe the weather). So it is easy to predict the probability of the weather of four consecutive days to be rain-rain-snow-sun given that the current day has sun, for example. We can just multiple the state transition probabilities with each other: $a_{31}a_{11}a_{12}a_{23}$.

Next, assume we live in a world where the seasons do not follow a natural cycle but follow a random sequence (e.g. spring-autumn-spring-summer would be completely viable) and have a random length (e.g. ranging from one day to one year). Now we do not want to predict the weather, but determine for a sequence of days which season it is every day. We can still observe what weather it is; an observation on day t is denoted as $O_t = o_t$ (O_t being the variable and o_t being the specific value). The seasons cannot be observed and are the hidden states. We are now dealing with a Hidden Markov model.

The HMM consist of three parameters:

$\pi = P(s_i)$ The initial (a priori) state distributions. In other words, the probability that we are in a certain season, given that we have no other knowledge.

$A = a_{ij} = P(s_t = j|s_{t-1} = i)$ The state transition probabilities of the hidden states (i.e. the probability of going from one season to another).

$B = b_i(o_t) = P(O_t = o_t|s_t = i)$ The emission probabilities. This is the probability of observing weather o_t on day t given that the season on that day is i.

When we have training data, we can estimate the aforementioned parameters with the expectation maximization (EM) algorithm [6] (called Baum-Welch, for HMMs). The algorithm only needs a set of observations (the training data) and will adjust the parameters (π, A and B) in such a way that these observations have the highest likelihood of occurring. This is also known as finding the maximum likelihood estimates.

Now, when we are given a set of n new observations of the weather for each day $\{O_1 = o_1, ..., O_n = o_n\}$ together with our HMM $\{\pi, A, B\}$ we are able to determine the most likely sequence of seasons. We could look at each day separately and maximize the probability of being in a certain season by just looking at the weather. In this case we only use the emission probabilities from B and we are maximizing $P(s_t = i|O_t = o_t)$ (the probability it is season i given the weather o_t on day t). But this could result in infeasible sequences; the state transition matrix may contain zero probabilities between certain states, which are not taken into account when we only use the emission probability matrix.

To solve this, we use our transition probabilities from A as well; $P(s_t = i|O_t = o_t)$ (emission probability from B) is multiplied with $P(s_t = j|s_{t-1} = i)$ (transition probability from A). For calculating the probability of the first state of the entire sequence we use π instead of A, because the first season has not undergone a state transition. Maximizing this probability product for a long sequence of days is not that straightforward, because for a sequence of n days, $n^{|S|}$ combinations must be checked, $|S|$ being the number of seasons. Fortunately, finding the optimal sequence can be done efficiently with the Viterbi algorithm [7] by making use of dynamic programming.

We will now return to the problem of chord extraction. When the HMM is used for chord extraction the chord labels are the hidden states (each chord being a separate state), and the PCPs are the observations. We first train the model with observed sequences of PCPs (without chord labels) to learn the parameters $\{\pi, A, B\}$. After the model is trained, we can use the HMM to predict a chord sequence out of a sequence of PCPs.

For a more thorough explanation of HMMs we refer the reader to the work of Rabiner [8].

2.2.3 *Chordify Harmony Model*

This section will explain how Chordify extracts chords with their system called MPTREE as explained by the paper of De Haas et al. [9]. As with most chord extraction systems, MPTREE also starts with acquiring a PCP. A small difference is that MPTREE obtains two PCPs: *Bass*, which emphasizes the lower frequencies and *treble*, which emphasizes the higher frequencies. The bass PCP could then be used to find the root note of the chord and the treble PCP for the specific chord type).

After the PCPs are obtained, they are aligned to the beat of the song, which is done with a beat extraction algorithm. This is done by averaging all the PCPs that are between two beats. Next, the template pattern matching method (as explained in Section 2.2.1) with euclidean distance measure is used to assess the probability that a certain chord belongs to a PCP. If there is only one chord that has a clearly higher probability than the rest, then this chord is chosen to be the only candidate. Else, a larger candidate list is used for that beat.

The next step consists of using a harmony model, called HarmTrace to create the optimal chord sequence. For a thorough explanation of this model we refer the reader to [10]. The input is a sequence of candidate list, and the output is a chord sequence. The steps of the algorithm are as follows:

1. Find the local keys of the song, as the song does not always stay in the same key for its entire duration.
2. Merge some adjacent candidate lists that contain similar chords together, as a chord change does not occur on every beat.
3. Create smaller segments based on the key. The segment borders are located at places where the key of the song changes.
4. Per segment, parse every possible chord sequence by making use of the rules of the harmony model. Select the sequence that contains the least amount of parsing errors.

2.2.4 *Performance of Chord Extraction Algorithms*

One of the obvious reasons why chord extraction may perform poorly is the audio signal itself. If the signal is very noisy or when there are multiple melody lines sounding simultaneously then the obtained PCP is not very helpful. Since almost all algorithms rely on the PCP, every algorithm suffers from this problem. This problem is more occurrent in the template matching method, because that method solely relies on the PCP.

A serious problem that arises with template matching is that some chords share multiple notes. For example, a C major and A minor share two out of three notes in their chord template. Thus it is easy to confuse these chords with each other. In some cases there are overtones in the signal as well which could have a higher amplitude than the fundamental frequency. Lee [11] proposed an Enhanced Pitch Class Profile (EPCP), which is a PCP that does not contain these overtones. This is done by computing a Harmonic Product Spectrum (HPS) from the Discrete Fourier Transform (DFT) and then computing the PCP from the HPS instead of the DFT. In the HPS, only the fundamental frequencies are left intact. Gomez [12] also proposes a similar modified PCP, one that does not contain any overtones, called a Harmonic Pitch Class Profile (HPCP).

For the HMMs we have our training data to rely on as well. As much as it is a strength, it is also a weakness. If the chord labels used for training contain mistakes, the chord extraction will also contain mistakes. A large amount of different training data is needed in order for the model to deal with all kinds of music. If the model is only trained on perfectly clear MIDI data, it may perform badly on real orchestra music, because it hasn't encountered that kind of audio signal before.

Every year, MIREX (Music Information Retrieval Evaluation eXchange) hold several tasks, of which chord extraction is one of them. The latest results of the chord extraction task from 2015 show accuracies between 75% and 82%[1].

2.3 SIMULATED ANNEALING

Simulated annealing is a (combinatorial) optimization algorithm introduced by Kirkpatrick et al. [13] that is applied on problems that have a high amount of different solutions. The simulated annealing algorithm does not try every possible solution, but instead tries to slowly walk towards an optimal solution in a somewhat random manner. The steps of the algorithm are as follows:

1. Start with a random solution and compute the score of this solution
2. Move to a neighboring solution by slightly modifying the current solution and compute the score of this solution
3. If the new solution has a better score, accept it immediately (in this case, better means higher). Else, accept the new solution with the following probability:

   $$e^{(S_{new}-S_{old})/T} \tag{3}$$

   where S_{new} and S_{old} are the scores of the new and old solution respectively. T (>1) is an integer variable that controls how likely it is that worse solutions get accepted and decreases over time.
4. Decrease the value of T and go back to step 2 if T is still above some constant value.

The reason why worse solutions also get accepted with a certain probability is to prevent the algorithm from getting stuck in a local optimum. This is illustrated in Figure 2.1: The x-axis represents the solution space and the y-axis the score of a solution. If the algorithm would start with a solution all the way to the right and would only climb up (accept better solutions), it would get stuck in the local optimum and never reach the global optimum more to the left.

2.4 METHODS FOR CLASSIFICATION

Since one of our research problems is trying to classify a chord sequence as h-music or non-h-music, we will now discuss how the classification process works and discuss some classifiers in more detail. The classification process is a form of supervised learning, which means that we provide labeled data from which a model (or classifier) can be constructed. We will first discuss feature classification, which works on data points with specific features and follow up with an explanation of language models.

[1] http://www.music-ir.org/mirex/wiki/2015:Audio_Chord_Estimation_Results

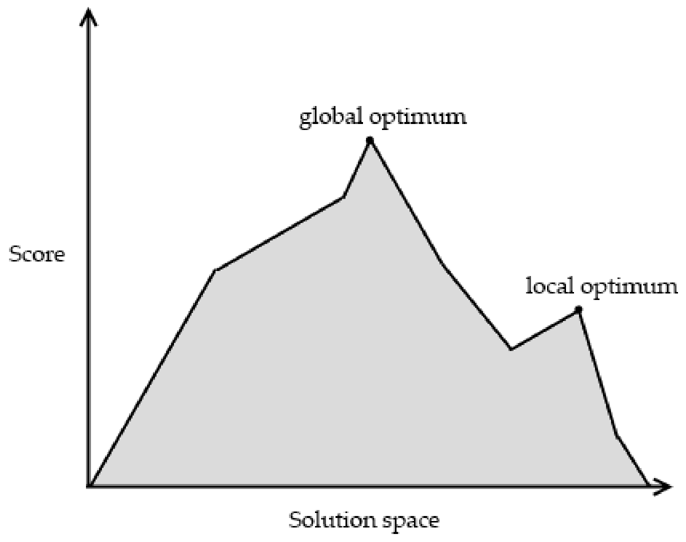


Figure 2.1: The search space of simulated annealing.

2.4.1 *Feature Classification*

In feature classification, each data point consists of some features together with a class label. An example of a data point is the physical condition of human being. E.g. body temperature, headache and appetite. With these features we can try to determine whether someone has a fever or not. If enough data is used to create this classifier, the classifier may then have learned how body temperature, headache and appetite correspond to a fever. Having good features plays a key role in the performance of a classifier.

With features and class labels, the classifier can learn which (combinations of) feature values belong to a class. The difference between each classification method is the manner in which the model is constructed. For some classifiers, such as a classification tree (see Figure 2.2 for an example), the model of the classifier is clearly visible; we can see which rules the classifier follows to determine the class label. However, for other types of classifiers it is hard or impossible to determine which concrete rules have been created.

It is possible to combine several classifiers into a new classifier. One way is by making use of ensemble methods [14] of which the most popular are bootstrap aggregating (bagging) [15] and boosting [16]. These are methods where multiple classifiers are trained. Classifying new points is done by taking a (sometimes weighted) majority vote on these classifiers.

In bagging, we take m samples, which could contain some overlap, from the training data on which m classifiers are trained. These classifiers are trained on different parts of the data and are allowed to be rather complex, overfitted, models.

The idea of boosting is to turn several weak classifiers (that are slightly better than random guessing) into strong classifiers. Iteratively, these classifiers are fed training points that have a weight. Points that are classified incorrectly will receive larger weights and will be fed to the classifiers more often. After a number of iterations the classifiers will reach acceptable accuracies.

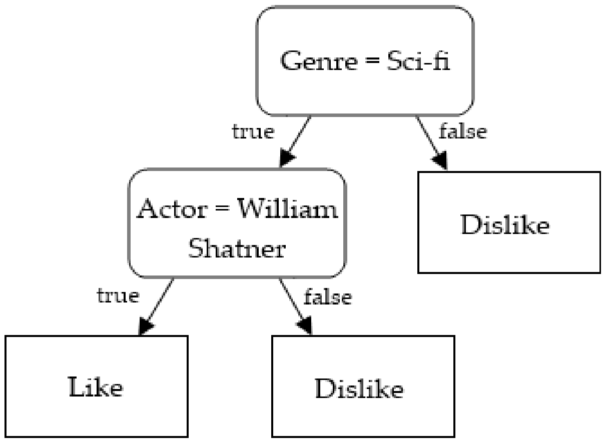


Figure 2.2: A decision tree that determines whether someone will like a movie or not based on the genre and actor of the movie.

Next, a list is shown with descriptions of the most popular classifiers that we will be using. These classifiers are known to work well, without the need for much parameter tweaking. For a more in-depth explanation of these (and other) classifiers, we refer the reader to the book of Hastie et al. [17]

K-NEAREST NEIGHBORS (KNN) KNN has one of the most simple models. The model consists of all training points, which are represented as vectors in a d-dimensional space (d being the number of features). Classification of point p takes place by looking at the k nearest neighbors of p. Point p gets assigned to the majority class of the k nearest neighbors. Generally the euclidean distance is used, but other measures can be used as well. A variety of KNN is Weighted KNN where the distance from p is used as a weight, resulting in closer points having more influence.

DECISION TREE A decision tree is easy to understand as well (Figure 2.2). The tree is built by creating rules over features. A tree gives very good insight into the classification process, because the decision rules can be accessed. The tree is built by choosing rules such that both branches of the tree contain a strong majority of a class (i.e. the distribution is far away from even). A node is not split any further if the distribution of two classes (in binary classification) is above a certain threshold (e.g. if 99% of the samples are of one class and only 1% are of the other class).

NAIVE BAYES (NB) The idea of NB is that a feature vector (data point) gets the class assigned that obtains the highest probability. This probability is calculated with Bayes' theorem (Equation 4) and assumes that each feature is independent from one another.

$$P(y|x_1,\dots,x_n) = \frac{P(y)P(x_1,\dots,x_n|y)}{P(x_1,\dots,x_n)} \tag{4}$$

where y is the class label and $x_1,\dots,x_n$ is the feature vector.

Since the feature vector is constant and thus the same for every class, it can be removed from the denominator. We can then use Equation 5 to compute the class with highest probability:

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(y) \prod_{i=1}^{n} P(x_i|y) \tag{5}$$

The independence assumption that NB makes results in incorrect probabilities, but for classification this does not really matter as long as the correct class gets assigned the highest probability. Other versions of NB differ in the way $P(x_i|y)$ is estimated.

SUPPORT VECTOR MACHINE (SVM) In an SVM classifier, as with KNN, points are represented as feature vectors in a d-dimensional space. The SVM tries to create a linear boundary that separates two classes in the best way possible; the distance between all the points and the boundary should be maximized. It is allowed to have points that are on the wrong side of the boundary, but they are penalized. The points that are closest to the boundary (and define the boundary) are called *support vectors*. Only these support vectors are part of the model of the SVM, no other points have to be stored.

It is not always possible to create a linear boundary, even when allowing some points to be on the wrong side of the boundary. This can be solved by applying a so-called *kernel trick*, which consists of mapping the points to a higher dimension. In this higher dimension, it is possible to linearly separate the classes. An example of this can be seen in Figure 2.3. The kernel trick allows the SVM to work well on data with many dimensions.

New points, of which the class is to be predicted, are mapped to the vector space as well and are assigned to a class based on which side of the boundary they fall on.

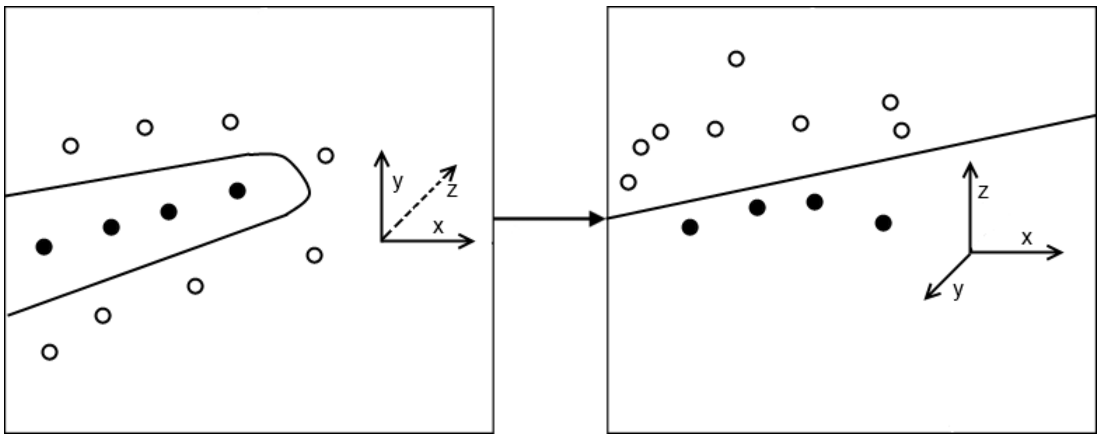


Figure 2.3: The data that can't be linearly separated (left) is mapped to a higher dimension with the use of a kernel trick (right). In this higher dimension the data can be linearly separated.

RANDOM FOREST A random forest is a specific type of bagging classifier. It uses decision trees as its classifiers. The random aspect of this classifier is that the set of features for training trees are picked entirely random.

ADAPTIVE BOOSTING (ADABOOST) AdaBoost is a popular boosting implementation. It is similar to the random forest classifier in that it also trains a set of classifiers, which are generally classification trees.

Obviously, there are other promising classifiers, but they are rather complex and need a lot of tuning before they start to perform well. Some classifiers that are rather complex and try to match the training data in the best way possible have the danger of *overfitting* on the training data. Such a classifier then performs really well on the training data, but does not generalize well on a larger data set. A general rule of thumb is to start off with simpler models, and if they perform poorly, switch to more complex ones.

2.4.1.1 *Music Classification*

Much research on distinguishing music from other sounds (speech, noise etc.) has been done already[18, 19, 20]. These methods generally consist of extracting several features directly from the audio such as the silence ratio, harmonicity and pitch. Classifiers such as SVMs, Naïve Bayes and neural networks are then trained on these audio features. An accuracy of 96% has been obtained for the binary problem of classifying audio signals between music and noise ([20]). Our research differs from previous research, since we only use chord sequences from which we extract features as opposed to using audio. This means the classification process is done with less information, making it slightly more difficult.

A small amount of classification on chord features has been done before [21, 22]. Pérez-Sancho et al. [21] have tested both language models (explained in Section 2.4.2) and an NB classifier. They obtain accuracies between 65% and 85% when doing classification with three distinct genres of music, and obtain accuracies around 50% when using nine subgenres. They train the NB classifier is trained on a *chord histogram*. This is a vector where each dimension represents a chord and one extra dimension for a No chord (N-chord). In every dimension, either the exact count of every chord is stored, or only a 1 or 0 based on whether the chord appears in the song or not. Since the length of a song should not directly influence this feature, the chord histogram can be normalized by dividing every dimension by the total number of chords in the song.

As an example: A file contains the sequence A-A-A-A-Cm-Cm-N. There are a total of 7 chords (including the N-chord). The A dimension will get a value of 4/7, the Cmin dimension a value of 2/7, N will get 1/7 and the other dimensions get a value of 0.

2.4.2 *N-Gram Language Models*

A different kind of approach for classification is to make use of n-gram language models (LMs). We will now give a basic introduction to n-gram LMs. For a more in-depth explanation we refer the reader to the book of Jurafsky and Martin [23]. An LM can be explained as a probability distribution over text of a language. This text is stored in the form of an n-gram, which is a sequence of n words. Every n-gram has a certain probability of appearing in the language. Let us denote x as a sentence and w_i as word i of that sentence, then an LM estimates the probability x appearing in the data, which is denoted as $P(x)$ or $P(w_1, w_2, ..., w_n)$.

An LM can either have a closed or an open vocabulary. A closed vocabulary means that all the possible words that are part of the vocabulary appear in the constructed LM. So the LM is constructed with the entire vocabulary, and words that are not part of this vocabulary will not be queried to the LM. With an open vocabulary, the constructed LM does not contain all of the vocabulary, meaning that unknown words could be queried to the LM. In our case, the vocabulary is closed because there exists a fixed list of of 25 chords (when we use the simple model explained in Section 2.1).

When LMs are used for classification, each class has its own LM. Meaning that each LM has a different probability distribution over words. Pérez-Sancho et al. [21] have used LMs for genre classification in music, where each genre has its own LM. In their LMs the words corresponded to chords, so a sentence of words corresponds to a sequence of chords. In order to classify a chord sequence as a specific genre, all LMs are queried and each LM returns a probability of the chord sequence. This value explains what the probability is that the given chord sequence appears in that model (genre of music). The chord sequence gets the class assigned of the LM that returns the highest probability.

When genres also have rather specific subgenres, the LMs could be further divided into multiple classes (e.g. instead of only having a jazz LM, one could have one for Latin jazz, soul jazz and rock jazz). In this way each model can capture the characteristics of a specific type of music more precisely, instead of trying to generalize all kinds of sounds into a single model. During the classification phase, if one of the jazz subclasses obtains the highest accuracy, then the chord sequences gets classified as jazz. Pérez-Sancho et al. [21] have also split up their main genres into subgenres, but they have not mixed them; they have either only classified the subgenres, ignoring the main genres, or only tried to classify the main genres, not looking at the subgenre information. Using the subgenre information to classify the main genres may improve the classification performance.

The first step in creating an LM is to store the counts of all the n-grams of a training corpus. The value of n is the order of the model, which controls what the highest length n-grams are. For example, in a 3-gram model 1-grams (unigrams), 2-grams (bigrams) and 3-grams (trigrams) are stored, where 3-grams have the highest length. In an example corpus with one sentence: **"The trees are tall"**, the resulting 1-grams would be { $<$ s $>$, the, trees, are, tall, $<$ /s $>$}, the resulting 2-grams would be { $<$ s $>$ the, the trees, trees are, are tall, tall $<$ /s $>$} and 3-grams would be { $<$ s $>$ the trees, the trees are, trees are tall, are tall $<$ /s $>$}. Note that $<$ s $>$ and $<$ /s $>$ are begin- and end-of-sentence tokens, which are generally automatically added to the vocabulary.

After storing n-gram counts, probabilities for these n-grams can be calculated. Using probability theory, we can define the probability of a sentence by making use of the chain rule (Equation 6). This leaves us with a large product that contains many factors. A simple way to calculate these factors is by using the Maximum Likelihood Estimate (MLE) as shown in Equation 7. $C(w_1, ..., w_i)$ stands for the number of times that sentence $w_1, ..., w_i$ appears in the training data. If we have two sentences **trees are tall** and **trees are big** (both having a count of one), then **trees are**, which is $C(w_1, ..., w_{i-1})$, has a count of two. Both sentences then have a probability of $\frac{1}{2}$ according to the MLE calculation.

$$P(w_1, w_2, ..., w_i) = P(w_1)P(w_2|w_1)...P(w_i|w_1, ..., w_{i-1}) \tag{6}$$

$$P_{MLE}(w_i|w_1, ..., w_{i-1}) = \frac{C(w_1, ..., w_i)}{C(w_1, ..., w_{i-1})} \tag{7}$$

To calculate the probability of the sentence **"I often eat Japanese food"**, the following equation holds: P(I often eat Japanese food) = P(I)P(often|I)P(eat|I often)P(Japanese|I often eat)P(food|I often eat Japanese), begin- and end-of-sentence tokens are left out for simplicity's sake. It can be noticed that the factors more the the right have a much larger context.

In an n-gram model the factors of the chain rule product are replaced by n-grams that all have a length of n. The n-gram model is based on the Markov assumption: The future is independent of the past, given the present. This assumption leads to not using the entire context $w_1, ..., w_{i-1}$, but only a subset of that. Thus we approximate with: $P(w_i|w_1, w_2, ..., w_{i-1}) = P(w_i|w_{i-n+1}, ..., w_{i-1})$. The smaller the value of n (the order of the model), the smaller the used context. In the case of an 1-gram model the context is of length zero. For a 2-gram model the context is of length one. Generally, results get worse after an n-value of four, as was the case in the n-gram classifier by Pérez-Sancho et al. [21].

Unfortunately, it is possible that an n-gram factor of the product is zero, which causes the entire product to become zero. E.g. a sentence **"I often eat Japanese food"** is queried, but the training corpus only contains **"I often eat Chinese food"**. Then (in case of a 2-gram model) the 2-grams **"eat Japanese"** and **"Japanese food"** have a zero probability, because their counts are zero.

There is a large chance that a certain n-gram does not occur in the training corpus, resulting in a factor of zero. Even with the Markov assumption and working with a closed vocabulary, there are still n-grams with a zero count. Fortunately, there exist several smoothing techniques to solve this problem. The concept of smoothing is also thoroughly explained in [23], but we will at least explain the basics in the next section.

2.4.2.1 *Smoothing and Backoff*

The idea of smoothing (also called discounting) is to steal probability mass (or word counts) from seen words and distribute them over unseen words. We first introduce a simple formula that holds for every bigram. Equation 8 basically states that the probability that word y is followed by one of the other words from our vocabulary is 1, assuming we have a closed vocabulary.

$$\sum_{x \in V} P(x|y) = 1 \quad \forall y \quad \text{where } x \text{ and } y \text{ are words in our vocabulary } V \tag{8}$$

We will now explain the concept of smoothing with an ongoing example which makes use of chords instead of words:

EXAMPLE Assume we are working with a vocabulary of major and minor chords, together with the N-chord, resulting in a vocabulary of 25 chords (once again leaving out the begin- and end-tokens). If we had infinite training data we would be able to create a model with 25 unigrams and 625 bigrams (25^2). But generally not all n-grams appear in the training data, resultantly some bigrams could be missing.

Now say we have an LM that is trained with only the chord sequences (C, A) and (C, G), then in our LM we have three different unigram probabilities: $P(C)$, $P(A)$ and $P(G)$ that sum up to one and two different bigram probabilities $P(A|C)$ and $P(G|C)$ that sum up to one (by following Equation 8). This means that C can transition into either A or G where both transitions have a probability of $\frac{1}{2}$ using the MLE calculation.

In an unsmoothed LM the probability that C transitions into any other chord than A or G is equal to zero. Although, we know that there are exactly 23 other possible bigrams that contain C as their first chord; they just don't appear in our training data. To solve this, we could take some probability mass from the two bigrams in the training data and spread it over the 23 remaining bigrams. This is the concept of smoothing.

The most simple variant of smoothing is **additive smoothing**, which works by adding a small value δ (generally $0 < \delta \leq 1$) to every n-gram count. When computing the MLE on these new counts, normalization is applied by adding δN to the denominator, where N is the size of the vocabulary. The formula for additive smoothing is shown in Equation 9.

$$P^*_{add}(w_i|w_{i-n+1}, ..., w_{i-1}) = \frac{C(w_{i-n+1}, ..., w_i) + \delta}{C(w_{i-n+1}, ..., w_{i-1}) + \delta N} \tag{9}$$

EXAMPLE If we go back to our example again and apply additive smoothing where $\delta = 1$ then the bigrams (C, A) and (C, G) will get one extra count. Thus they will now both have two counts instead of one. The remaining 23 bigrams will have one count now.

First we had only two bigrams (that start in C) in total, this has now become 27 because of 25 additions. We now have to recalculate the probability of our two bigrams: this used to be $\frac{1}{2}$ for each bigram, but this is now $\frac{2}{27}$. For the other 23 bigrams $\frac{23}{27}$ probability mass is reserved, resulting in each separate bigram gaining a probability of $\frac{1}{27}$.

Chen and Goodman have evaluated several smoothing methods which are known to produce better results than additive smoothing [24], but are also more complex. We will briefly explain some other smoothing methods that we plan to test. The main concept that all smoothing methods have in common is that probably mass is taken from existing n-grams in some way and distributed over unseen n-grams. For a more thorough explanation on these methods, we refer the reader to the work of Chen and Goodman [24].

- **Good-Turing Smoothing** In Good-Turing smoothing the count of every n-gram gets recomputed. Part of the counts of n-grams that occur $r + 1$ times get distributed over the n-grams that appear r times. Generally for the n-grams with $r \geq 3$ the amount of discounting does not vary much. This is where constant discounting improves on.
- **Constant Discounting** Unlike Good-Turing that computes the discounting value for every value of r, in constant discounting every n-gram gets discounted by a constant value. Constant discounting also interpolates higher order n-grams with lower order n-grams. Because some constant value is subtracted from the higher order n-gram, there is room to use some probability mass for a lower order n-gram. Just one value could be used to discount all the different order of n-grams, or a distinct discount value for every n-gram order could be used. The suggested discount factor D can be calculated with $D = \frac{n_1}{n_1+2n_2}$ where n_1 and n_2 stand for the n-grams that have exactly one and two counts respectively.
- **Witten-Bell Smoothing** Witten-Bell smoothing uses interpolation as well, but the weights are calculated differently from absolute discounting; no n-grams are directly discounted, but instead a linear interpolation is applied.

Apart from smoothing, there also is a technique called backoff that can be combined with smoothing techniques. The general idea behind backoff is that if the n-gram has at least k (>0) occurrences in the LM, the (discounted) MLE probability is used; otherwise, use the probability of a lower order n-gram multiplied by a backoff weight α. The formula can be seen in Equation 10.

$$P^*_{backoff}(w_i|w_1,...,w_{i-1}) = \begin{cases} \frac{C(w_{i-n+1},...,w_i)}{C(w_{i-n+1},...,w_{i-1})}, & \text{if } C(w_{i-n+1},...,w_i) > k \\ \alpha(w_{i-n+1},...,w_i)P^*_{backoff}(w_i|w_{i-n+2},...,w_{i-1}), & \text{otherwise} \end{cases} \tag{10}$$

When there is a backoff to a lower order n-gram, this probability cannot be used immediately, but must be multiplied by α; otherwise there won't be a true probability distribution. This will be further explained in the ongoing example.

EXAMPLE By following Equation 8 we know that $P(A|C) + P(G|C) + \sum_{x \in V/A,G} P(x|C) = 1$. Since none of the $P(x|C)$ have real counts, it is replaced by $\alpha(C)P(x)$ (the backoff step). α is now calculated in such a way that $P(A|C) + P(G|C) + \sum_{x \in V/A,G} \alpha(C)P(x) = 1$.

Remember the 23/27 of probability mass that we took away from $P(A|C)$ and $P(G|C)$ during the additive smoothing step? This mass is now distributed over $\sum_{x \in V/A,G} \alpha(C)P(x)$. In the original additive smoothing (without backoff) every n-gram received $\frac{1}{27}$ evenly, but with backoff we use $P(x)$ in the computation, which differs for every x, giving us more accurate probabilities.

2.4.2.2 *Factored Language Model*

A model that can take more information into account than just the chord itself is the Factored Language Model (FLM) introduced by Bilmes and Kirchhoff [25]. In this model a word (or chord) is composed of several factors. Khadkevich and Omologo have used FLMs for chord extraction [26] by splitting a chord into a label (e.g. E-min) and a duration. They obtained a slight increase in accuracy of 0.25% compared to normal smoothed LMs. FLMs also make use of backoff, which is now done over factors instead of words. This is explained in the example below.

EXAMPLE We may use the factors: Note, chord type and duration. We could then query for an n-gram ″C-maj F-maj-0.45 A-min-0.6″. This is a C-maj of any length followed by an F-maj of 0.45 seconds, followed by an A-min of 0.6 seconds. If this n-gram does not exist, there are multiple backoff paths that can be taken. For example, the first chord type could be omitted, resulting in ″C F-maj-0.45 A-min-0.6″, or the second duration could be omitted, resulting in ″C-maj F-maj A-min-0.6″.

2.5 SEGMENTATION METHODS

This section describes some related work in the area of music segmentation. The task of segmentation according to López and Volk [27] is ″the process of automatically determining segments given a symbolic or sub-symbolic representation of a musical piece/melody/part″. Generally these segments are found by detecting boundaries first. There are multiple approaches in finding these boundaries, which are all explained in the overview paper by López and Volk [27].

Most work in [27] focuses on the symbolic representation of music on the note level. These methods generally find two notes that greatly differ from each other (e.g. in pitch or duration). This work cannot be directly applied to chords because of the following reasons: Firstly, the difference between a high note and a low note is very different from the difference between a chord with a high root bass note and a chord with a low root bass note; mainly because notes are distributed over multiple octaves and chords are not. Secondly, timing differences between notes, such as long rests, are more common than timing differences between chords of a chord sequence. Chords are much more beat aligned, as a result the duration of chords will deviate less from one another compared to the duration of notes. Apart from the fact that the methods in [27] mainly work on notes, instead of chords, these algorithms also work on (h-)music only and find segments within this music, thus they cannot directly be used to segment h- from non-h-music.

A different group of algorithms works in the audio domain. In 1996, Saunders [28] was able to segment music from noise with a 98% accuracy by using a simple Naive Bayes classifier trained on audio features. Other audio-based classifiers that followed, gained similar high accuracies [29, 30].

We have to look for an approach that is more suited to symbolic chord sequences. Ferrand et al. [31] have constructed a model that makes use of probabilities to find segment borders. The model stores the probability of several chord transitions. If a chord transition with a low probability occurs in a chord sequence, the likelihood of a border is high. The model that they use is based on notes, but it is straightforward to apply the model to chords as well.

2.6 REGRESSION ANALYSIS

In this section we will give a brief introduction to regression analysis. In regression analysis a model is estimated that consists of several independent variables and (generally) one dependent parameter which is influenced by the independent parameters. The goal of regression analysis is to find the exact influence of the independent variables on the dependent variable.

We will now give a simple regression example in which we want to determine the price of a house. The price of a house is determined by several factors, for example, the size of the house (in square meters), number of rooms and whether the house is located in the city or not. The price of the house is the dependent variable (which we denote as Y), and the factors that determine the price are the independent variables (X_1 is the size of the house, X_2 is the number of rooms, and X_3 is a 1 for being in the city and a 0 otherwise). From these variables a model (or function) can be approximated:

$$price = c_0 + c_1X_1 + c_2X_2 + c_3X_3 \tag{11}$$

The goal of the regression analysis is to find the correct values for the coefficients (c_1, c_2 and c_3). With a lot of training data available, the coefficients are fitted such that they match the data. For example: If it appears from the data that there is no price difference between houses in the city and houses in rural areas, then the c_3 coefficient is set to 0.

The most simple form of regression is the **least squares** method. In the least squares method, the coefficients are fitted in such a way that the error of the predictions is minimal. This error is calculated with the following function:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{12}$$

where n is the size of the data set, y_i is the ground truth value of point i and $\hat{y}_i$ is the prediction for point i.

A slightly more complex model is obtained with **KNN regression**. This model works similar to the classification variant of KNN: The points of the training set are put into a d-dimensional space, where d is the number of independent variables. The value of a new point is calculated by taking the weighted mean of the k closest points; points that are closer to the queried point (e.g. by using the euclidean distance), have a larger weight.

For a more thorough explanation of regression analysis we refer the reader to the book of Faraway [32].

2.7 EVALUATION METHODS

This section describes how we can evaluate the performance of the different (classification, segmentation, and quality assessment) algorithms that we produce.

2.7.1 *Classification Evaluation Methods*

To evaluate the performance of a classifier, generally both a training and test set is required. For all the data points in the training and test set the class labels (ground truth) are also known. A classifier is created using the training data and then evaluated with the test data. After a classifier has made predictions on a test set, we have the predicted classes and the ground truth classes, from which we can compute various evaluation measures. The most basic measures are true positives (TPs), false positives (FPs), true negatives (TNs) and false negatives (FNs). Their meanings are made clear in the so called confusion matrix in Table 2.1

		ground truth class	
		1	0
predicted class	1	TP	FP
	0	FN	TN

Table 2.1: A confusion matrix showing true positives, false positives, true negatives and false negatives.

The most straightforward measure is the *accuracy* measure which calculates the fraction of correct predictions:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{13}$$

When the class distribution is skewed and one class appears a lot more in the training/test data than the other class, then the accuracy measure becomes very high when a classifier simply always classifies everything as the most occurring class. For such circumstances, something called the F_1 measure is a more appropriate measure:

$$F_1 = \frac{2TP}{2TP + FP + FN} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{14}$$

where

$$precision = \frac{TP}{TP + FP} \tag{15}$$

and

$$recall = \frac{TP}{TP + FN} \tag{16}$$

Recall tells us of all the points that are of class c, how many are classified as class c. *Precision* tells us what fraction of points that are classified as class c are really of class c. The F_1 measure combines precision and recall with a similar weight, but other F measures exist that give different weights to precision and recall respectively.

Instead of using a training and test set, **cross validation** could also be used as an alternative. In k-fold cross validation the data is split into k equal parts (folds). The first $k - 1$ folds are used for training the classifier and the k-th fold for testing it. We keep rotating one step, as a result that after *k* iterations every fold has been part of the test set once. The *k* obtained classification scores (which can be the evaluation measures mentioned above) can then be averaged. When a limited amount of data is available, cross validation becomes useful, as it allows one to use all data for both training and testing.

2.7.2 *Segmentation Evaluation Methods*

We will now discuss how we can properly evaluate the performance of a segmentation algorithm. According to López and Volk [27], the most common method is to, per song, store both the algorithm's output and the ground truth in its own vector. The song is divided in multiple small slices of similar size (generally less than a second). Every value in the vector corresponds to such a slice, which is given a label (in our case this is either a 0 for noise, or a 1 for music. A -1 could be used if it's unclear whether there is music or noise, which can then be ignored in the evaluation).

The algorithm's segmentation is compared with a ground truth segmentation that is obtained from human annotators. Generally, one would want multiple annotators, to counter subjectivity and human errors. But for something as unambiguous as deciding at what time the music has stopped playing, having just one annotator is enough.

By comparing the algorithm's segmentation and the ground truth segmentation, the performance of the segmentation algorithm can be computed. This is done by doing a stepwise comparison of the two vectors. For this stepwise comparison we can use regular classification measures that were introduced in the previous section. Every component of the vector basically corresponds to one data point with a class label, thus every value of the algorithm's produced vector can be compared to the ground truth vector. From these comparisons it is straightforward to calculate TPs, TNs, FPs and FNs.

Apart from the stepwise comparison of class labels, the structure of the segmentation can also be evaluated. In Figure 2.4 we can see two different segmentations. They all obtain the same accuracy and recall, but the first segmentation is preferred over the second, because its structure is more consistent. A simple method for determining the segmentation quality is to compare the predicted borders to the ground truth. For this comparison a different vector is used that has a 1 for the timestamps that contain a border and a 0 for timestamps that do not contain a border. From the predicted borders and ground truth borders we can also calculate TPs, FPs, TNs and FNs. Because the majority of a sequence does not contain a border, there will be many TNs.[27] states that in most studies the precision, recall and F1 measure has become a standard, as they do not take TNs into consideration.

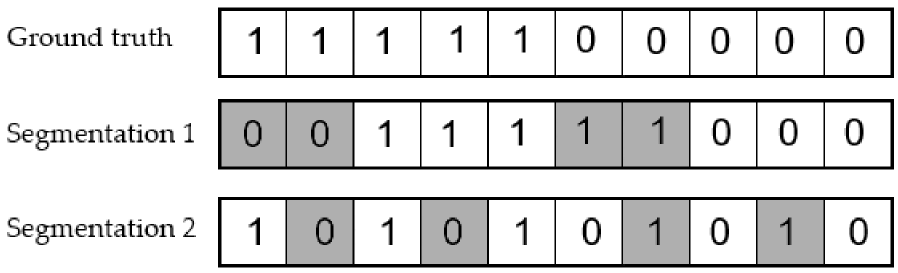


Figure 2.4: Two estimations of a segmentation algorithm can be seen. In grey are the frames that contain an error. Both segmentations obtain the same accuracy and recall, but the first segmentation is preferred since its structure is more consistent.

2.7.3 *Quality Assessment Evaluation Methods*

The performance of a regression method can be evaluated by determining how far the predicted scores are away from the ground truth. Unlike with classification, there is no binary value of 1 and 0 for correct and incorrect predictions. Rather, the distance between the prediction and ground truth is calculated. A common measure is to use the mean squared error, which we also introduced in Section 2.6 for the least squares method. We will repeat this function here:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{17}$$

where y_i is the ground truth value, $\hat{y}_i$ is the prediction of the regression model.

A straightforward way to first evaluate the quality of the regression model is by using the R^2 measure, which is defined in Equation 18:

$$R^2 = 1 - \frac{\sum(\hat{y}_i - y_i)^2}{\sum(y_i - \bar{y}_i)^2} \tag{18}$$

where $\bar{y}_i$ is the average value of the data set.

The R^2 score explains how strong the correlation between the independent and dependent variables are and can be at most 1. The higher the value of R^2, the higher the correlation and the better the regression model is compared to always using the mean value of the data as a prediction. A score of 0 means that using the mean value as a prediction works as well as using the regression model. A negative score means that always using the average value, results in a lower error than using the regression model.

2.7.4 *Testing the Statistical Significance*

In some cases we want to compare the results (performance measures) of two methods or algorithms with one another and determine which of the two performs better. When we compare the mean performance measures of the two algorithms, we could conclude that the algorithm with the highest score performs the best. But how do we know for certain that the winning algorithm wasn't just lucky, and with slightly different data (or more samples of the same kind of data) the other algorithm would perform better? A method for for determining whether the algorithms perform roughly the same or whether one performs significantly better is that of *statistical hypothesis testing*. We will only briefly discuss the basic concept of statistical hypothesis testing. For a more thorough explanation of statistical hypothesis testing (and statistics in general), we refer to reader to the free e-book of Lowry [33][2].

We can generalize the aforementioned algorithms as populations (that contain several values). The so-called null hypothesis states that there is no significant difference between the two populations. If we want to show that there is a significant difference, we have to reject the null hypothesis with a certain confidence (probability) α, which generally is 0.05. One could say that the the null hypothesis is only rejected, when less than 5% of the difference in means can be accounted to luck.

[2] `http://vassarstats.net/textbook/index.html`

There are numerous tests that can be used to determine whether there is a significant difference between two (or more) populations. Simply said, such a test returns a certain value, which reflects the difference between the populations. From this value, we can compute the probability that the null hypothesis is true. If this probability is larger than our confidence value, then we are unable to reject the null hypothesis.

When we want to compare more than two populations with one another, it is advised to not run multiple paired tests that compare two populations every time. Apart from the fact that the number of tests increases substantially as the number of populations increases, for every test we also have a 5% probability of making an error (when 0.05 is our significance level, 5% of the detected difference can be accounted to luck). As the number of tests increases, there is more opportunity for errors. The correct way to compare multiple populations is to first use a test that detects whether there is any significant difference between the populations, and to then use a post-hoc test that determines between which populations there is a significant difference and with which probability.

3

PIPELINES AND ALGORITHMS

In this chapter we describe our experimental pipelines, which differ for each research problem. We also introduce several novel algorithms, which are partly inspired by the related work, and explain how they function within our pipelines.

3.1 H- AND NON-H-MUSIC CLASSIFICATION PIPELINE

This section explains how we plan to tackle the problem of classifying chord sequences between h- and non-h-music. We discuss how our pipeline is functioning, what kind of data we have as input, which features are extracted and how classification results are obtained. Since feature classification and LM classification work slightly different, they each have their own pipeline. The details about the implementation for both classification methods can be found in Appendix A.

3.1.1 *Classification on Chord Features*

Our feature classification pipeline is illustrated in Figure 3.1. In our goal to achieve the best classification accuracy possible, we have found four important parameters that significantly affect the results of the classification. These parameters are explained further below. Apart from that, we also experiment with different data sets, resulting in the fifth parameter of our pipeline. These data sets are further explained in Chapter 4 .

In this pipeline, the five different parameters are: data set, chord merge, features, transposition and classifier. Each of these parameters can take on several different values. A certain value assignment of these parameters is called a configuration. Every parameter can be changed independently of one another, and by changing only one parameter in the configuration we can see the impact of that parameter on the classification results. Once we've determined the optimal value assignment for a certain parameter, we will not change it anymore.

1. **Data set:** One of the data sets (explained in Chapter 4) is chosen:
   - chordino
   - ht
   - corner
   - ht-small
   - corner-small

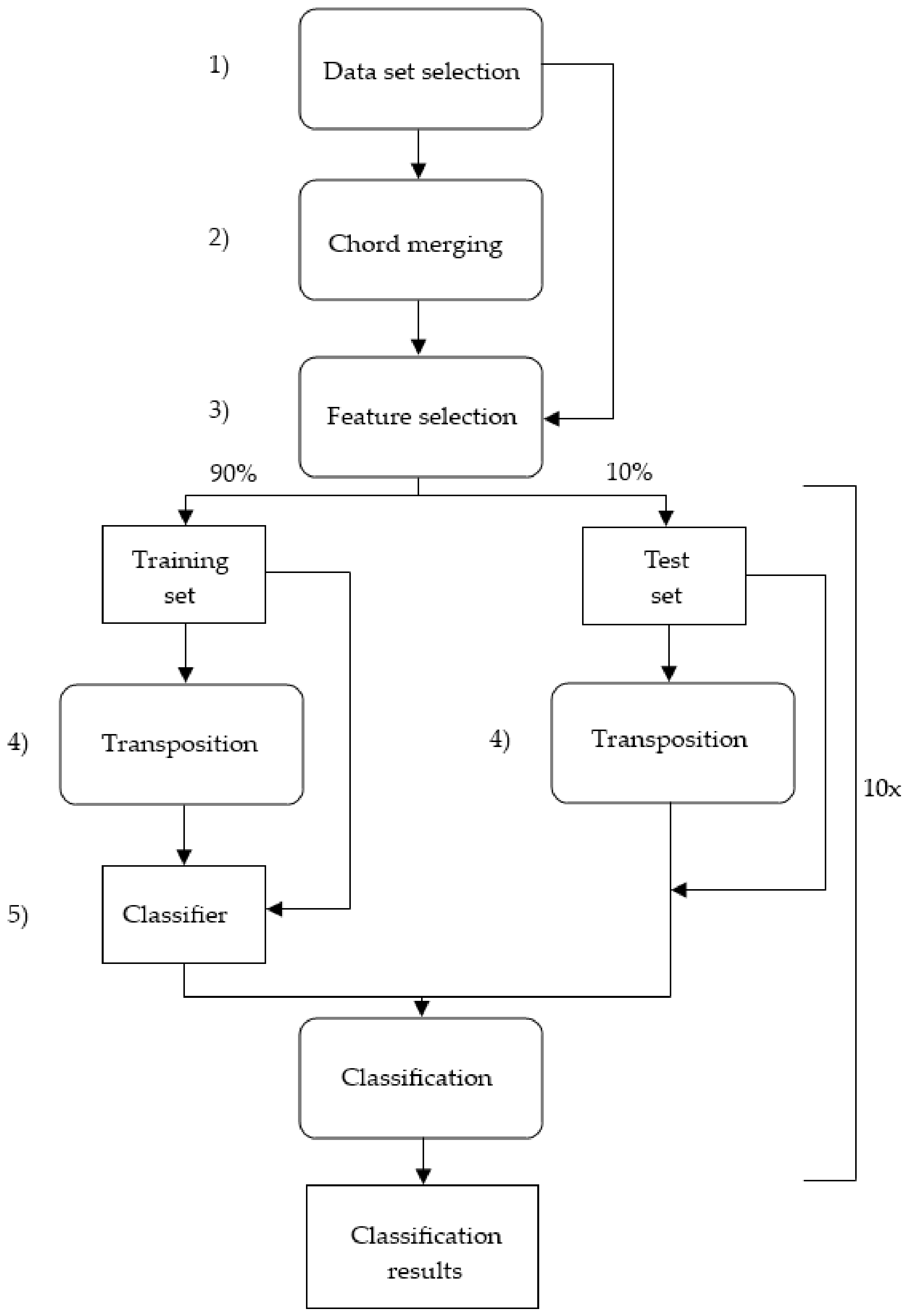


Figure 3.1: The pipeline for feature classification. 1) A **data set** is chosen and parsed. 2) Optionally, adjacent similar chords in the chord sequences are merged in all the data. 3) Specific **features** are extracted and the data is split in a test and training set via 10-fold cross validation. 4) Optionally, **transposition** is applied on the training set or both training and test set. 5) A **classifier** is created from training data. Finally, the test set is classified and scores are obtained.

2. **Chord merge:** During the parsing step we can also **merge** similar adjacent chords together. In a chord sequence it is possible that the same chord is repeated several times after each other. When we merge chords we treat these repetitions of the same chord as one chord (with a longer duration).

3. **Features:** Features are extracted from every chord sequence in the data set. There are four different feature sets that we can choose from, which are explained below:

    - chords
    - chords + beat
    - chords + duration-sd
    - chords + beat + duration-sd

    chords: This is a normalized chord histogram as explained in Section 2.4.1.1

    beat: In many cases, chords sound for longer than just one time stamp. In their paper, Khadkevich and Omologo [26] show that chords often repeat an even number of times (mostly durations of 2, 4 and 8 beats) in music. Using this information, we introduce a feature which we will call a *beat histogram*. We store how many beats a chord repeats itself and create a vector of 1 through n beats. For every beat repetition from 1 till n, we store how many times this beat repetition occurs, disregarding the chord label and type.

    In this example sequence: A-A-Bm-Bm-C-C-C-A-Bm, we have two occurrences of a 1-beat-repetition, two occurrences of 2-beat-repetition and one occurrence of a 3-beat-repetition. These values are then normalized over the number of beats, such that the length of the song does not matter.

    When we use the merge chord option, there are no chord repetitions at all, making this feature useless. As a result, when we use the merge chord settings, we will look at the unmerged chord sequence when constructing the beat histogram.

    duration-sd: In music, the beat is clearly structured, resulting in every chord sounding roughly the same duration. In noise there is no such structure, thus it may be possible that the duration of chords are much more random. We will look at the standard deviation of the chord duration, which is independent of any specific chord. The higher the standard deviation, the more variable the duration of a chord is.

    After the feature extraction, we split the data in a test and training set with 10-fold cross validation. This results in 90% of the data being used for training and 10% for testing. Since we have 10 folds, the remaining steps are repeated 10 times.

4. **Transposition:** In order to increase the training data for our classifier, we can also apply a technique which we call *12-transpose*. We transpose every chord sequence to all of the 12 pitch classes, giving us 12 different versions of the same chord sequence. As an example, a song that consists of 3 chords: $\{C, F, Gm\}$, would result in 11 additional versions: $\{C\#, F\#, G\#m\}, \{D, G, Am\}, \ldots, \{B, E, F\#m\}$.

    This gives us two advantages: 1) It gives us 12 times as much data and 2) it may remove potential biases in our h-music data set; since it could be possible that our chord files often are in a certain key, classifiers would then learn that h-music is always in a certain key, instead of learning relations between chords. On the other hand, songs are, for example, more likely to be in the key of C than C#, which is ignored when using 12-transpose.

    12-transpose is applied after the data is split between training and testing sets. If we would apply 12-transpose before splitting, different versions of the same data point could appear in both the training and test data, resulting in bias. We can either apply 12-transpose to only the training set or to both the training and test set. If we choose to also 12-transpose the test set, then we keep the 12 transposed versions together as a set of data points, let the classifier classify every version separately, and then take a majority vote to decide the class of the data point set.

    The options regarding transposition are:

    - **no transpose:** Do not transpose chord sequences

- **12-train:** Only transpose the training data to all 12 pitches, giving us 12 times as many data points for training
- **12-full:** Transpose both the training and test data, using the voting mechanism

5. **Classifier:** A classifier is chosen and is fit to the training data. The list of classifiers that we test are:

    - KNN
    - NB
    - Decision Tree
    - SVM
    - Random Forest
    - AdaBoost

    After the classifier has been trained, the classes of the test data are predicted with the classifier. A data point from the test set is queried to the classifier, resulting in a prediction for the class of the data point. If we choose to use 12-full, then we also 12-transpose the queried data point and apply a majority voting, such as we explained above.

    From these predictions we can calculate various performance scores that were introduced in Section 2.7.1. Since our class distribution has a ratio of 50:50, the accuracy measure is a good measure to rate the overall performance of a classifier. But for the Chordify use case, it may be interesting to look at a different measure besides accuracy. When a false negative occurs, a Chordify user submits h-music, but the classifier predicts it as non-h-music. This means that chords may possibly be of low quality; is it acceptable that the user will not get chords then? When a false positive occurs, a Chordify user submits non-h-music, but the classifier thinks it's music and accepts it. Then either the audio is real noise, but somehow the chords are acceptable and appeared as music or the user submitted atonal music and got chords back. These chords may not properly match the music, since it is atonal music.

    In general false negatives are worse to have; a user is expecting to receive chords (because h-music is submitted), but (s)he does not receive them. Since we would like to minimize the number of false negatives, we would like to maximize the recall measure on h-music. Thus, besides accuracy, we will also do evaluations based on the recall score. But because the recall could always be optimized by only classifying pieces as h-music, the accuracy measure will still be the main measure to use.

    Each cross validation iteration returns one score. In total we get 10 scores of which we take the mean. For every configuration, we do 100 iterations and take the mean score of those iterations.

3.1.2 *Chord Language Model Classification*

For the LM experiments, we have also found four important parameters (and the fifth parameter being the data set). The parameters for LM classification are slightly different. There are no specific features to extract, since the model is built from pure chord sequences, and instead of having different classifiers, we have one classifier that differs in the n-gram model order and the smoothing method. This gives us the following parameters: data set, chord merge, transposition, n-gram order and smoothing. The pipeline for LM classification is illustrated in Figure 3.2.

1. **Data set:** One of the data sets is chosen and parsed:

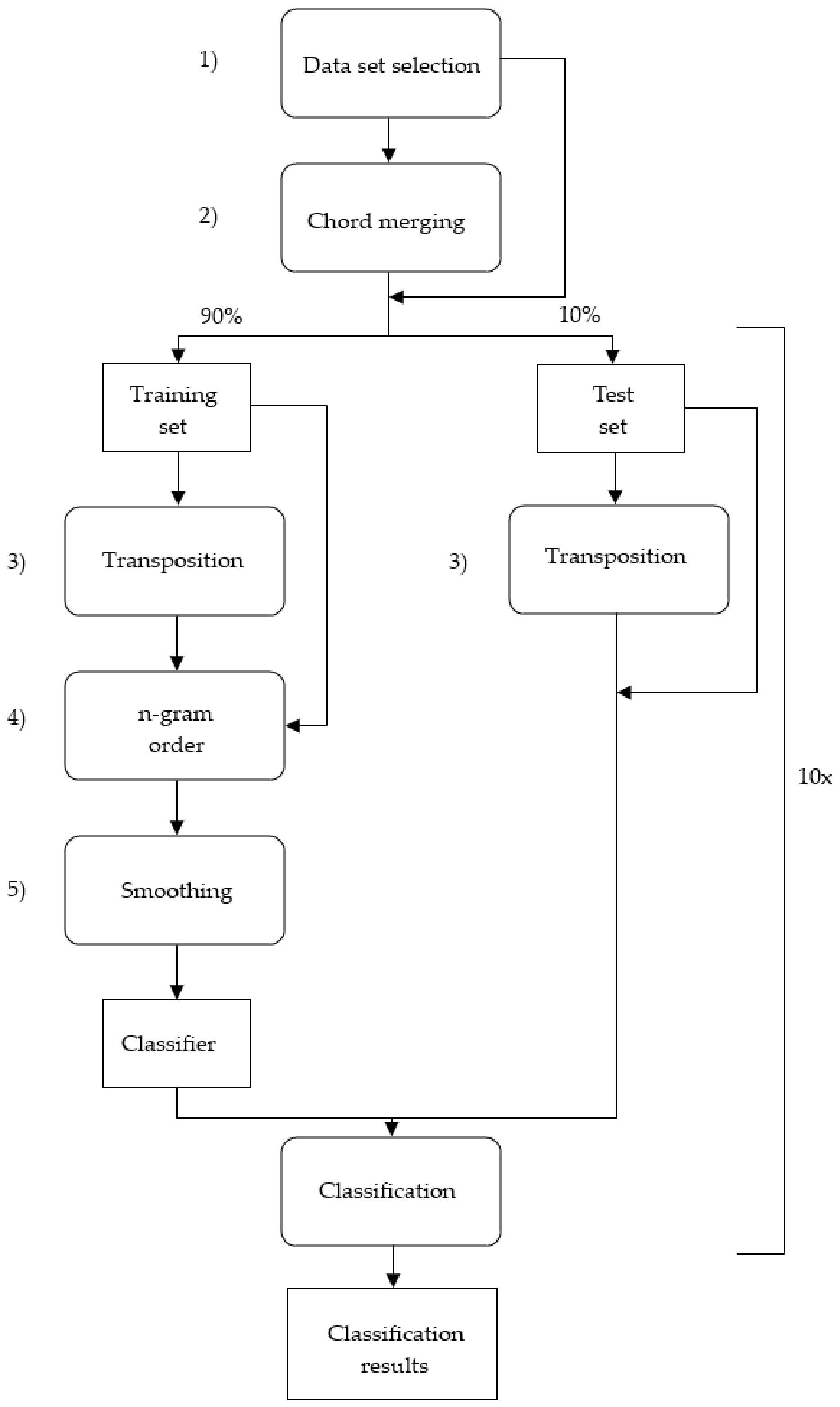


Figure 3.2: The pipeline for LM classification. 1) A **data set** is parsed, chord sequences are put in a simple format. 2) Optionally, adjacent similar chords in the chord sequences are merged in all the data.The data is then split in a test and training set via 10-fold cross validation. 3) Optionally, **transposition** is applied on the training set or both training and test set. 4) A separate LM for h- and non-h-music is build with the chosen **n-gram order** and 5) **smoothing** method. Finally, the test set is classified and scores are obtained.

- chordino
- ht
- corner
- ht-small
- corner-small

2. **Chord merge:** We have the option to **merge** similar adjacent chords together.

   After the possible chord merge, chord sequences are put in a simple format that LM classifiers support: This is one long string where each chord (label and type, e.g. Fmin) is separated by a space. After that, the data is split in a test and training set via 10-fold cross validation.

3. **Transposition:** Once again, the options regarding transposition are:

   - **no transpose:** Do not transpose chord sequences
   - **12-train:** Only apply 12-transpose on the training data.
   - **12-full:** Transpose both the training and test data, using the voting mechanism

4. **n-gram order:** The n-gram order of the LM is chosen between *2 through 8*. The minimum of 2 is used, because we are interested in the relation between chords, thus need more than one chord for that. The maximum of 8 is chosen because models beyond that were not promising anymore; mainly because they became too sparse and did not provide any extra information.

5. **Smoothing:** Choose the smoothing method of the LMs, then estimate the LMs for h- and non-h-music from the training data. The h-music LM is created from only h-music training samples and the non-h-music LM is created from only non-h-music training samples. The smoothing options are:

   - No smoothing
   - Additive smoothing
   - Good-Turing smoothing
   - Constant discounting
   - Witten-Bell smoothing

   For the FLM, **no smoothing** is not an option of the program and additive smoothing is also not available.

   In the final step we query the test data to both models. When we query a data point (a chord sequence), both the h- and non-h-music model return a probability score for that chord sequence. We will classify the data point to the model that returns the highest probability. From these predictions we can calculate the accuracy and recall as we did in the feature classification pipeline. Each cross validation iteration returns one score. In total we get 10 scores of which we take the mean. For every configuration, we do 100 iterations and take the mean score of those iterations.

3.2 H- AND NON-H-MUSIC SEGMENTATION PIPELINE

In this section we discuss how we will try to solve the problem segmenting a chord sequence into parts of h-music and parts of non-h-music. The pipeline that we use to solve this problem can be seen in Figure 3.3.

The first step in our pipeline is to create an h-music classifier. This is the classifier that we have carefully constructed and tested in the pipeline of Section 3.1. This classifier is trained on both the

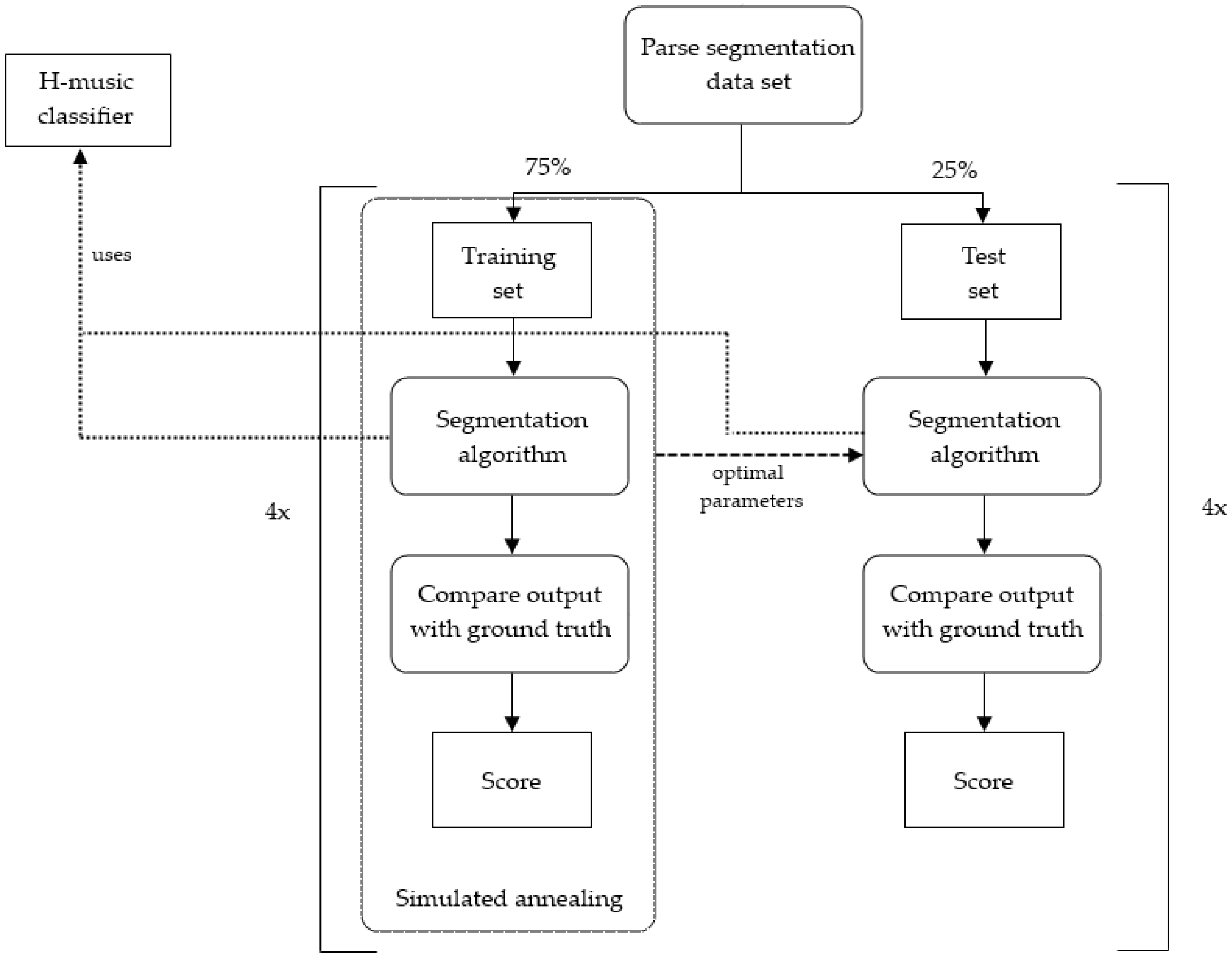


Figure 3.3: The pipeline for segmentation.

training and testing data that was used for the classification experiments. This classifier will be used later on by our segmentation algorithms. During the classifier's construction, we also parse the **mixed** data set that consists of chord sequences generated from audio that we would like to segment. For this data set we have on every time stamp (of roughly a quarter of a second) a chord together with the ground truth (a 0 for noise, a 1 for music or a -1 when both music and noise is sounding at the same time). The data sets are explained further in Chapter 4.

The next step is to split the mixed data set into a training and test set via 4-fold cross validation. Three fourth of the files are used for training our segmentation algorithm and one fourth is used for testing it. We use four folds, because the more folds we use, the smaller the test set becomes. As the audio tracks of the data set can greatly vary in length, we could potentially end up with three tracks of only 15 minutes each in the test set, which is undesirable. If we have more files in the test set, we reduce the chance that the test set consists of only a few short tracks.

The remaining steps are done four different times for each rotation in the cross validation. The training set is used to find a set of optimal parameters for the segmentation algorithm. These algorithms and their parameters are explained in Section 3.2.1. The algorithm outputs a vector of 0s and 1s, which can be compared with the ground truth vector. The -1 values in the ground truth vector (which stand for music and noise sounding simultaneously) are not taken into consideration when calculating the performance of the algorithm. When we compare the ground truth and estimated vectors we can calculate various performance measures as explained in Section 2.7.2. Because not all chords in the input have the same length, we should take this length into account. If we, for example, calculate the number of true positives, we calculate for how many (mili)seconds there was a true positive and not for how many chords there was a true positive. The choice for our performance measure is discussed a bit further below.

For a given training set, we use 1000 iterations of simulated annealing to find the parameters that give us the best results. In every iteration we randomly change the value of one of the parameters and then let the segmentation algorithm segment the training set. From these 1000 iterations we remember the parameters that obtained the highest score. These parameters are then possibly overfitted to the training set, hence we will do the final evaluation on the test set. We do 25 iterations of the entire pipeline, in which we obtain a score for every fold of the cross validation. This leaves us with a total 100 scores, as we have four folds.

If the results from the test set are far lower than those of the training set (i.e. more than 5% difference), then this hints at overfitting on the training set. Unfortunately, it is difficult to avoid overfitting, thus the main goal of this experiment is not necessarily to find the best parameters possible. As for different kinds of data, e.g. audio where 90% consists of h-music compared to audio with 90% noise, different parameters may work well. We are mainly interested in finding out what the effect of these parameters is, and how they are influenced by overfitting.

Now the question that remains is which performance measures to use. An important application of the segmentation algorithm is to filter out the segments that do not correspond to music and to only keep the chords for the segments with music. A false negative indicates that chords are not shown, even though music is actually playing, which is something we strongly want to avoid. Thus, once again we would like to use the recall measure on h-music. Apart from the recall, we also want to keep using the accuracy measure, as it gives us good insight in how correct the overall segmentations are. Unfortunately, we can greatly vary the accuracy and recall of the algorithm. We can tweak the parameters of the algorithm such that the accuracy is high, but the recall mediocre. Similarly, we can obtain a very high recall on h-music, accompanied by a low accuracy. As accuracy and recall can differ considerably from one another in certain parameter configurations, it is better to use the F1 score (explained in Section 2.7.1) as the quality measure for the segmentation algorithm, which is a combination of precision and recall. The precision could be considered a form of accuracy, but only on h-music, as it only takes true and false positives into account.

Additionally, we'd also like to investigate how well the algorithm finds the borders of the segments. There is a border when in the estimated vector we go from a 1 to a 0 or vice versa. We can put the time stamps of the ground truth borders and estimated borders in two separate vectors: *gt* is the vector with all the ground truth border timestamps and *est* is the vector with the timestamps of predicted borders. When the algorithm has detected a border that is within 30 seconds of the ground truth border (either 30 seconds earlier or later), we consider this a true positive (TP). As we are working with recordings of at least 15 minutes, a 30 second offset for a border is acceptable (in Section 6.1.2.1 this is discussed further). We introduce a measure called **border recall**, which is defined as the fraction of correctly predicted borders: $\frac{TP}{|gt|}$, where $|gt|$ is the length, or the number of borders, of *gt*. The **border precision** is then the fraction of predicted borders that are actually a border: $\frac{TP}{|est|}$. If the recall is low, then the algorithm predicts too few borders and/or predicts them at the wrong places. If the precision is low, then the algorithm predicts too many borders, and/or predicts them at the wrong places.

3.2.1 *Segmentation Algorithms*

In this section we will introduce two segmentation algorithms: First a simple baseline algorithm is shown, which is then followed by a more sophisticated algorithm that tries to improve on the baseline. Both algorithms have a set of parameters that greatly influence their performances. These parameters are underlined in both the text and the pseudocode of the algorithms. Both algorithms get as input a list of time stamped chords with ground truth class labels, and give as output a list of class labels which has the same size as the input list.

3.2.1.1 *Separate Segment Algorithm*

We will now explain how the baseline algorithm, which we will call the separate segment algorithm, works. The general idea of the algorithm is as follows: We split up the entire input chord sequence into chunks of similar size (window size). We then classify these chunks separately and give all the chords in the chunk the same class.

The pseudocode can be found in Algorithm 1 and works in the following way: The output vector that will store the class labels is initialized in step 2. The window is defined by the *begin* and *end* indices in step 3 and 4. In step 5 through 10 we will take the chords from the chord vector that are within the window and obtain a class label from our h-music classifier. We then append this obtained class to the class vector as many times as the number of chords in the window. In step 11 through 13 we move the window by window_size steps, such that the current window does not overlap with the previous window and we thus classify an entirely different chunk. We do this until all chords have been classified.

Algorithm 1 Separate Segment Algorithm

```
 1: procedure SEPARATESEGMENT(chord_vector)
 2:     class_vector = list()
 3:     begin = 0
 4:     end = window_size
 5:     while begin < end do
 6:         chords = chords between begin and end of chord_vector
 7:         c = obtain class label of chords from h-music classifier
 8:         for i = 0 to chords.length do
 9:             append c to class_vector
10:         end for
11:         begin += window_size
12:         end += window_size
13:         end = min(end, chord_vector.length)
14:     end while
15:     return class_vector
16: end procedure
```

3.2.1.2 *Sliding Window Algorithm*

The pseudocode of the sliding window segmentation algorithm can be seen in Algorithm 2. We will cover all the steps of the pseudocode in the explanation below. After the description of the algorithm, we will discuss the ideas behind the algorithm, the choices that we have made and the details of the parameters.

Algorithm 2 Sliding Window Algorithm

```
1:  procedure SlidingWindow(chord_vector)
2:      prob_vector = list()
3:      prev_p = null
4:      begin = 0
5:      end = window_size
6:      while end < chord_vector.length do
7:          chords = chords between begin and end of chord_vector
8:          p = obtain class probability of chords from h-music classifier
9:          append step_size values to prob_vector interpolating between prev_p and p
10:         begin += step_size
11:         end += step_size
12:         prev_p = p
13:     end while
14:     add prob_vector[0] window_size/2 number of times in the front of prob_vector
15:     add prob_vector[-1] window_size/2 number of times in the back of prob_vector
16:     for i = 0 to probVector.length do
17:         class_vector.append(max(1, round(prob_vector[i]+music_bias)))
18:     end for
19:     smoothed_class_vector = SegmentSmoothing(class_vector)
20:     return smoothed_class_vector
21: end procedure

22: procedure SegmentSmoothing(class_vector)
23:     prev_class = null
24:     for i = 0 to class_vector.length do
25:         if prev_class != class_vector[i] then
26:             c = 0
27:             for j = i to i + segment_size do
28:                 if class_vector[j]==class_vector[i] then
29:                     c += 1
30:                 end if
31:             end for
32:             if $\frac{c}{segment\_size}$ > segment_purity then
33:                 Real border detected                              ▷ Do nothing
34:             else
35:                 class_vector[i] = prev_class
36:             end if
37:         end if
38:         prev_class = class_vector[i]
39:     end for
40:     return class_vector
41: end procedure
```

The algorithm has three important procedures:

1. Slide a window over the input chord sequence and classify the entire window. This results in an initial vector of class probabilities which has a one-on-one mapping with the input chord vector. Each value in the probability vector is the probability that the chord corresponds to h-music (occurs in SlidingWindow procedure).
2. Add a small constant value to the probabilities, such that h-music obtains a small bias and round the class probabilities to concrete classes of 0 and 1 (occurs in SlidingWindow procedure).
3. Smooth out small miss-classifications, such that only larger segments remain (occurs in SegmentSmoothing procedure).

- SlidingWindow This procedure is the entrance point and core of the algorithm. The procedure receives a vector of time stamped chords as input. On line 2 the vector that will store all the class probabilities is initialized. The class probability is a value between 0 and 1 that reflects the probability of the time stamp corresponding to h-music. A value below 0.5 results in the classification of non-h-music and above 0.5 results in the classification of h-music. In step 4 and 5 the *begin* and *end* indices of the sliding window are initialized. In steps 6 through 13 we slide over the chord vector by moving the window by step_size number of spaces per iteration. In one iteration we take the chords that are within the window_size (step 7), put them in the h-music classifier, obtain a probability (step 8) and append it to the probability vector (step 9). If the step_size is higher than 1, every iteration we skip some chords, thus we need to add some additional probabilities to the vector, in between the previous and current probability. We add the current probability p and some values that are linearly interpolated between p and $prev\_p$ (e.g. if $prev\_p = 1$, $p = 0$, and step_size = 4, then the probabilities we will add in that iteration are 0.75, 0.5, 0.25 and 0).

  Note that the initial chord sequence and the resulting probability and class vectors have the exact same size, so when we classify a chord sequence, the obtained probability is assigned to the chord in the center of the sequence. Since we assign a probability to the middlemost chord of the chord sequence, the sliding window can't classify the first few and last few chords as these chords will never be in the middle of the sliding window. That is why we add additional probabilities to the start and end of the vector in step 14 and 15. The probabilities added in the front use the probability of the first element in the vector, the probabilities added in the back use the probability of the last element in the vector.

  After having obtained a full probability vector, we round these probabilities to class labels of 0 and 1 and then apply the max function to make sure we do not exceed the value of 1 (step 16-18). But before doing the rounding step, we add a small value of size music_bias to each probability. The result of the bias is that a value that would normally be rounded off to 0 might now be rounded off to 1. The reason for adding this bias is explained further below. The output of this step is a class vector of 0s and 1s. In Step 19 the SegmentSmoothing procedure is called to smooth out potential miss-classifications.

- SegmentSmoothing In this procedure we make some final modifications to the class vector. Starting on line 24, we loop over every value in the class vector. The moment we encounter a change in class (line 25), a potential border is detected and we check the next segment_size number of values in the class vector (line 27 through 31). We count how many values in that window are of the same class as the class that initiated the border. When we divide that count by the segment_size we get a ratio. If this ratio is higher than segment_purity then we have detected a border. Else, it was just a miss-classification and we change the value of the class in step 38. Then, in step 40 we return the final vector of class labels.

We will now discuss the ideas and choices behind the algorithm and explain the parameters.

- **Sliding window** The SlidingWindow procedure is the core of the algorithm, it produces the initial vector on which further computations are done. The parameters that are used here are:
  - **Window size:** The window size is notated in seconds. When we retrieve the chords in a certain window, we obtain all the chords that have their time stamps within the window size. The reason that we use seconds as opposed to number of chords is that using the number of chords would result in a variable window size. As some audio has a faster beat than others, chords last less long, which then results in a shorter window size. Although the downside of using seconds is that some window slices contain more chords than others, we think it is the better choice.
  - **Step size:** The step size, unlike the window size, is notated in number of chords, not seconds. Since not all chords have the same length, it is possible we will not always skip the same number of chords if we use a step size in seconds. We would then not know for sure how many interpolated probabilities we should add in step 9 of Algorithm 2.

  In this procedure we also have the option to add a small bias to all the probabilities, before rounding to class labels. The reason for this is that the classifier tends to classify chord sequences as non-h-music more often. This probably happens, because these sequences contain both music and speech, whereas the classifiers were trained on sequences containing pure music or pure noise. By adding a small bias, we give h-music an increased probability and hopefully counter these miss-classifications. The parameter that controls this bias is:
  - **Music bias:** This is a value between 0 and 0.5. If the value is 0, no bias is added at all. If the value is 0.5, every element will have a probability of at least 0.5, thus every element will get classified as h-music.
- **Segment smoothing** The SegmentSmoothing procedure is an optional smoothing step in the algorithm. The procedure basically filters out the few out-of-place class values; a small number of 0s that are surrounded by several 1s or vice versa. The idea behind this smoothing is that when we come across a border, but the majority of values after the border are of the class before the border, then this border is probably a miss-classification. The following parameters explain this more:
  - **Segment size:** This is the number of elements following the detected border that we look at.
  - **Segment purity:** This is the fraction of elements in the segment size window that must be part of the detected border class, in order for the border to be valid. If the border is not valid, then the class of the border gets changed to the class before the border. The segment purity is a value between 0 and 1.

  Initially, we tried using a median filter to perform smoothing on the probabilities. Unfortunately, there were some side effects, making the filter less suitable. When there are multiple border changes close to each other, the median filter does not always filter the small segments out, but sometimes combines them. This can be seen in Figure 3.4, where we try to filter out all segments smaller than 3 by using a window size of 5. Separate segments will get filtered out correctly, but when two small segments are close to each other, we obtain a new small segment after the filter step.

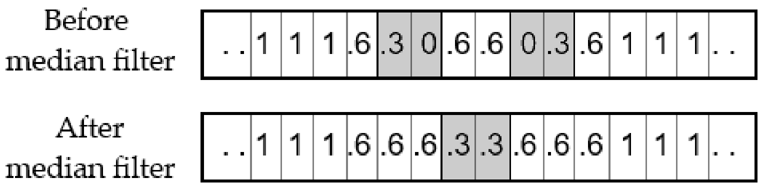


Figure 3.4: A median filter with a window size of 5 is applied on a vector of probabilities. In grey are the segments that are below 0.5 and will get classified as non-h-music.

3.3 ASSIGNING QUALITY SCORES TO CHORDS

We will now describe the process of creating a pipeline that allows us to assign quality scores to chord sequences. This score describes to what degree the Chordify user accepts the chords for a particular song. The most straightforward way to measure this quality is by means of user ratings. Chords with high user ratings are assumed to be of high quality, and chords with low user ratings are assumed to be of low quality. It should be clear that a user rating (thus also a quality score) is all about the user's opinion and not the "real" correctness or ground truth of the chords. These two are not strongly correlated according to Macrae and Dixon [34]. They state that chord sequences that got high ratings from users did not necessarily match the ground truth annotated chords. Thus, the ratings that we obtain from users do not reflect the chord accuracy (compared to some expert ground truth), but rather an acceptance rating of the user.

As was the case for the previous two problems, for this problem we will also need some data set with ground truth scores. A method to gain these ground truth quality scores is to allow users to rate chord sequences with the help of a rating system, which we describe in Section 3.3.1. When there are a sufficient number of ratings available, we can apply several techniques, discussed in Section 3.3.2, to predict the rating of a chord sequence ourselves, without the need for user ratings for that particular song.

3.3.1 *Rating System*

We will first explain which stakeholders are involved in the rating system and how their wishes may conflict with each other. After that we will look into different rating scales and determine which scale would be most suitable for us.

3.3.1.1 *Stakeholders*

There are three stakeholders involved in the rating system:

1. **Researchers**: Our goal is to obtain ratings that properly reflect what users think of the quality of the chord sequence. It is also important to have a sufficient number of songs with a sufficient number of ratings. We expect that at least 50 rated songs are required, and each song must have at least 20 ratings for the average of the ratings to be reliable. There should alse be a variety in good and bad quality chords.

From these ratings, a model can be trained that is able to predict the quality of a song automatically. This is also interesting for Chordify, so that they have information about the quality of the chords of a song for all songs (especially the ones that have not been rated yet).

2. **Chordify as a company**: They want to improve the user experience. When a user searches for a song, an ordering of results can be made based on the chord quality (when there are multiple versions of the same song). A specific "hub" for chord edits could be created, where users could search for songs they want to edit, based on the quality of the chords. (e.g. some users would only like to edit songs that are almost of perfect quality and only contain minor mistakes).
3. **Chordify user**: They would like to know when chords are of low quality, so they do not have to bother using them; they would like to never come across low quality chords. They want the rating process to cost as little time and cognitive load as possible, but don't want to have too few options, such that they cannot properly express their opinion of the quality.

Some of these interests may conflict with each other. The first point is that users may want to express themselves properly, giving a precise score for a chord sequence (just like our goal as researchers is to obtain ratings that reflect the user's opinion properly). But if we want precise information, it may be necessary to give users many options to express themselves. This contradicts with the fact that users may want the rating process to cost as little time and cognitive load as possible. If users need to interact with a rating system with too many options, they may refrain from using it, leaving us with less data. So if the rating system provides fewer options, users may use it more quickly, but the precision may not be as correct.

3.3.1.2 *Play Time Information*

Some user votes may be more reliable than others and the play time of a song may help us to determine the reliability. The longer the user plays the song, the higher the probability is that the user accepts these chords. If a user has played a song for only a short amount of time, but has rated it anything but low, then the rating may not be reliable. On the other hand, if the user gave a low rating after a short listening time, it may be reliable, because one can find out more quickly whether chords are of bad quality than whether they are of good quality. This suggests that an individual user vote should always be combined with the play time, so votes with low play time could possibly be filtered out.

3.3.1.3 *Rating Systems*

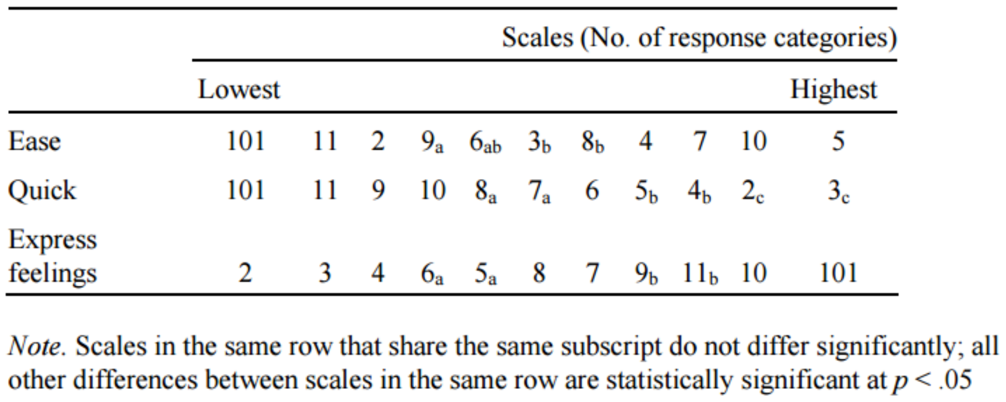
We will now compare different kind of rating systems, looking at their pros and cons. The main difference in these systems is the number of choices that users have, to express the degree of their acceptance of the chords. The least amount of choice would be a "like" or thumbs up such as Facebook does it. A user could either like the chords or do nothing. With one extra option comes a binary scale of like-dislike (or thumbs up/down) such as YouTube does. After that, a dislike-neutral-like scale can be used. This can be expanded further and further, until we reach a scale from 0 to 100 (generally, scales do not become larger than this). Apart from the granularity itself, different meanings could be assigned to every option. Scales exist where there is no neutral option (-3, -2, -1, 1, 2, 3), and some scales might give users more positive than negative options.

If there are fewer options, users may interact with the system more quickly, because it takes less time and effort. On the other hand, users may become frustrated, because they are unable to express their opinion properly. A neutral option gives users more expressiveness. In addition, when users find it difficult to make a choice, the neutral option makes the rating process easier. On the other hand, when users are indifferent, they may not even bother to vote; only when they have a strong opinion (like or dislike strongly), they may want to express that.

There is much literature on determining which rating scale works best [35, 36, 37], but the conclusions are varied so it's difficult to generalize to the context of Chordify. The context in which these rating systems are evaluated are often in surveys or recommender systems. These contexts are different from Chordify in a few ways: A survey is much more confronting than the Chordify rating system; people only interact with the Chordify rating system if they feel like doing so. Another point is that, generally, complex opinions need to be expressed by users in surveys instead of a simple "rate the quality of these chords" for Chordify. Chordify differs from recommender systems, because people gain a direct benefit from rating items in a recommender system, namely receiving better recommendations, whereas users do not directly benefit from rating chords in Chordify (although in the long run they will).

Cosley et al [35] have done a study on the interface of rating systems and whether the user's opinion could be manipulated. They have compared several rating scales for a movie recommender system and evaluated which system users like the most by asking users to give a score between 1 and 5 to several rating scales (the reason for using this 5 point scale to rate other rating scales has not been explained). Of the 26 users that responded, the majority likes to have more choices: the 10 (half star) scale obtained a score of 4.2 (out of 5), the 5-star scale got a 3.8, the no-zero scale (-3 till 3) obtained a 3.2 and the binary scale obtained a score of 2.2. They also state that when users are forced to choose between two options, they generally give a positive rating when they are neutral. In the Chordify rating system people are not forced to make a choice; if they don't have a strong opinion, they do not have to interact with the system. So in that respect, having no neutral option isn't necessarily bad.

A different study, that is done by Preston and Colman [36], provides some more inside in the different scales. They tested scales ranging from 2 till 11 options and also a 101-point scale by letting 149 users rate several aspects of a restaurant or shop. They present the users' preferences of the different scales, which can be seen in Figure 3.5. The conclusions they draw from this is that the scales with few options are the quickest to use, but not necessarily the easiest to use because it does not allow for users to express themselves properly.



	Scales (No. of response categories)										
	Lowest										Highest
Ease	101	11	2	9_a	6_{ab}	3_b	8_b	4	7	10	5
Quick	101	11	9	10	8_a	7_a	6	5_b	4_b	2_c	3_c
Express feelings	2	3	4	6_a	5_a	8	7	9_b	11_b	10	101

Note. Scales in the same row that share the same subscript do not differ significantly; all other differences between scales in the same row are statistically significant at $p < .05$

Figure 3.5: Scales Ranked in order of increasing respondent preference. Taken from [36].

We will now discuss three conventional systems with one another and explain how suitable they are for the context of Chordify.

Unary: Quality Chords

For the unary system, the user has only one button to click (seen in Figure 3.6). The user only interacts with the system (clicks on the button) if it accepts the quality of the song. If the user thinks the quality of the chords is low, they won't use the rating system (will not press the button). This unary measure works OK for gauging the popularity of a song, but the quality is a bit more difficult. For the people that have not voted, we don't know whether they are neutral or negative towards the chords, or if they just don't feel like voting. Even if we use the play time information, the sentiment of users that haven't voted is difficult to determine. The upside is that the interaction is straightforward; the user presses this button if they think the chords are of good quality. This requires minimal user time and effort.

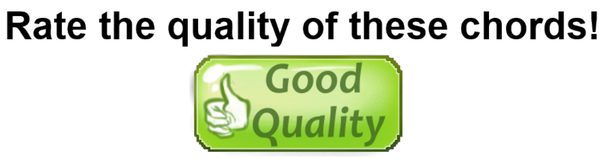


Figure 3.6: A button that Chordify users can click when they think the chords are of good quality.

Binary: Bad and Good Quality

In the binary rating system (Figure 3.7) the user has two choices: either positive (Good quality) or negative (Bad quality). If the user has no strong opinion, he is either forced to pick between one of the options or not to give a rating at all.

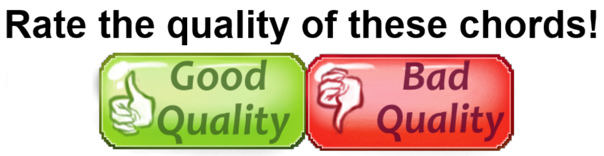


Figure 3.7: Two buttons that Chordify users can click to show their opinion about the chords. The user is forced between a positive and a negative choice

It is possible that there is a skew towards "Good Quality"; when a user likes something a lot, he is more motivated to say something about it. When he dislikes something he just wants to get away from it quickly. This is not necessarily bad, because our guess is that good, bad and average quality chords can still be distinguished from each other. Bad quality chords will most likely hardly have any likes and multiple dislikes. Average quality chords most likely have a low number of dislikes and slightly more likes. We predict that good quality chords will receive many likes and few dislikes.

The cognitive load is increased compared to the unary system; when a user has no strong opinion, it may require effort to make a choice. On the other hand, people know this system very well from for example YouTube (Figure 3.8).

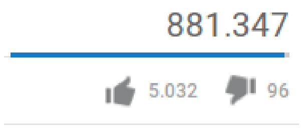


Figure 3.8: User interface of YouTube's rating system.

With this system it is straightforward to gauge the quality of the chords: the fraction of Good Quality chords compared to all the votes. E.g. 80 Good votes and 20 Bad votes result in a score of 80% (100 being the maximum and 0 being the minimum).

5-Stars

A five star rating (Figure 3.9) gives the user the most choice of these three scales. The user can give a song 1 to 5 stars, where 1 star corresponds to a very poor quality and 5 stars corresponds to an (almost) flawless quality. This system also contains a neutral option with 3 stars. An advantage of this system is that users know this system relatively well, as it is used in many different web services such as Amazon or guitar tab websites such as Ultimate Guitar. Another advantage is that it is straightforward to compute a score from these star ratings. We can take the average stars that a song has gotten resulting in a score between 1 and 5.

Both [35] and [37] state that people prefer to have more options (5- or 10-star systems), but other evidence in this blog from YouTube[1] suggests that these different choices are not always used; it can be seen that only very strong opinions (1 and 5 stars) are expressed. Additionally, with the 5-star scale, more information has to be conveyed; it is more difficult to display how the chords have been rated by other people. One could only show the mean rating, or show how many people have given how many stars.

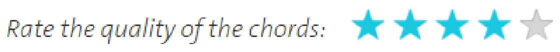


Figure 3.9: 5-star rating system that allows the user to express themselves about the quality of the chords of a song.

3.3.1.4 *Final Choice*

From the proposed options the binary system and 5-star rating system seem to be most appropriate. The reason the unary "like" system is not very suitable is because it only conveys positive information; it is difficult to detect if chords are of low quality.

When we compare the binary scale with the 5-star scale the main difference is in the amount of choice the user has. The main questions are: 1) Are users more inclined to use the binary system than the 5-star system and 2) how is the precision of the 5-star scale used; are all the votes at the far ends or are they spread out evenly? Unfortunately, these questions cannot be answered easily without testing both systems first.

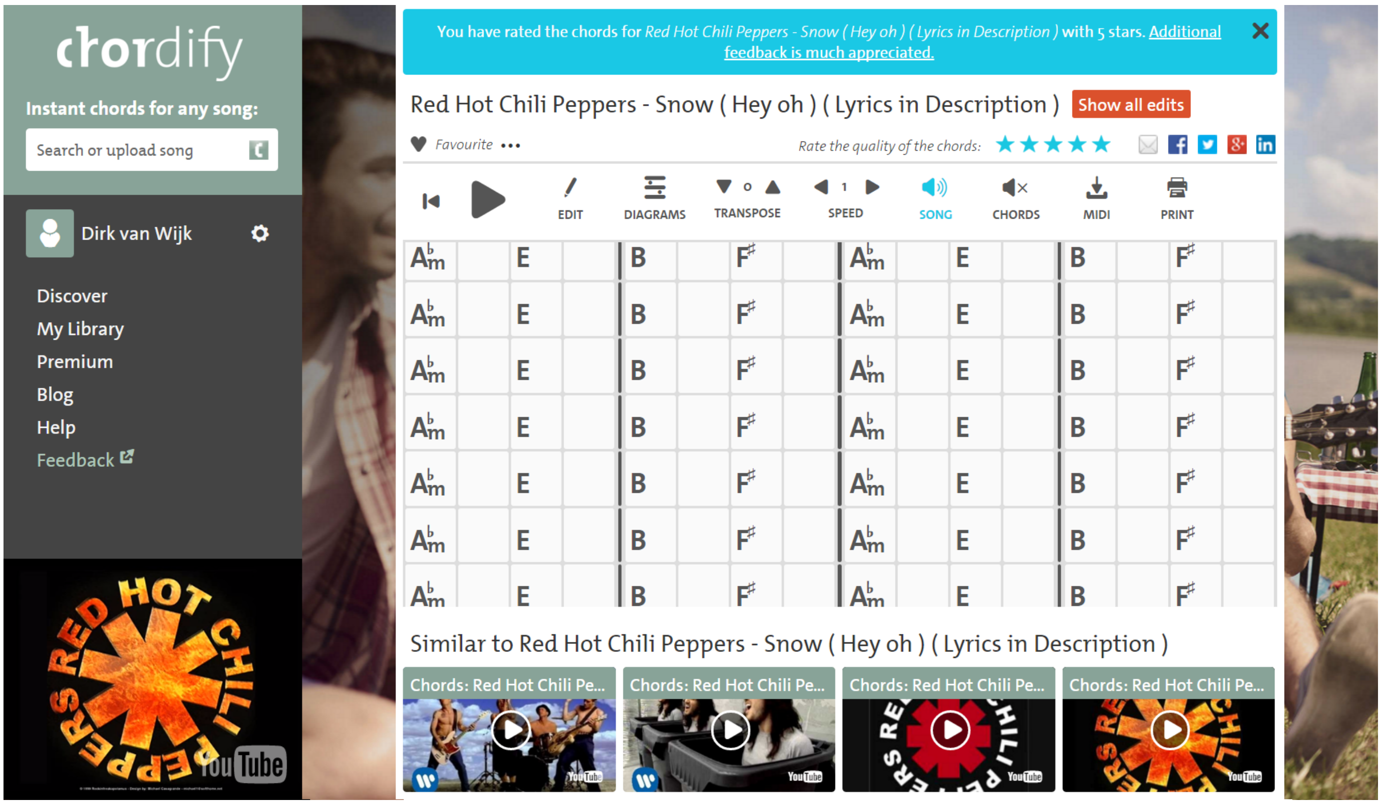
Because we have to make a choice, we choose to implement the **5-star** rating system for the following reasons:

1. People associate the 5-star rating with quality more often than the binary rating. People may associate the binary rating that YouTube has popularized a lot with like and dislike and could then possibly rate whether they like the song instead of the quality of the chords.

2. The 5-star rating gives people more options, so the quality scores may become more precise that way. Even if users will only use 1 or 5 stars, there is not really a negative effect of having these extra options in the rating system.

The implemented rating system can be seen in Figure 3.10. We have chosen to not display any average rating (before the user has given a rating himself), as this could introduce bias.

[1] http://youtube-global.blogspot.com/2009/09/five-stars-dominate-ratings.html


chordify
Instant chords for any song:
Search or upload song
Dirk van Wijk
Discover
My Library
Premium
Blog
Help
Feedback
Favourite
Rate the quality of the chords:
EDIT
DIAGRAMS
TRANSPOSE
SPEED
SONG
CHORDS
MIDI
PRINT
Show all edits
Chords: Red Hot Chili Pe...
Chords: Red Hot Chili Pe...
Chords: Red Hot Chili Pe...
Chords: Red Hot Chili Pe...

You have rated the chords for *Red Hot Chili Peppers - Snow (Hey oh) (Lyrics in Description)* with 5 stars. Additional feedback is much appreciated.

Red Hot Chili Peppers - Snow (Hey oh) (Lyrics in Description)

A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	
A♭m		E		B		F♯		A♭m		E		B		F♯	

Similar to Red Hot Chili Peppers - Snow (Hey oh) (Lyrics in Description)

Figure 3.10: The user interface of Chordify. A user has given this song 5 stars.

3.3.2 *Score Prediction Pipeline*

We will now discuss the pipeline that we use for predicting scores of chord sequences and evaluating the performance of the predictor. We have two separate pipelines: one for regression and one for a sliding window scoring algorithm.

3.3.2.1 *Regression Pipeline*

The regression pipeline is very similar to the classification pipeline of Section 3.1. Instead of a classification model, we use a regression model. The features that we use and the application of 12-transpose remain unchanged, so we refer the reader to Figure 3.1 again for a visualization of our pipeline. We briefly discuss the steps of the pipeline here:

1. **Data set:** The **rating** data set is parsed. This data set contains songs for which we have user ratings.
2. **Chord merge:** During the parsing step we have the option to **merge** similar adjacent chords together.
3. **Features:** Features are extracted from every chord sequence in the data set. There are four different feature sets that we can choose from:
   - chords
   - chords + beat

- chords + duration-sd
- chords + beat + duration-sd

After the feature extraction step we use 10-fold cross validation again to split the data into a training and test set.

4. **Transposition:** We can choose a form of transposition:
   - **no transpose:** Do not transpose chord sequences
   - **12-train:** Only apply 12-transpose on the training data.
   - **12-full:** Transpose both the training and test data, using the voting mechanism
5. **Regression method:** A regression method is chosen and is fit to the training data. The regression methods that we test are:
   - Least squares
   - KNN regression

   After the regression model has been trained, the ratings of the test data are predicted. We can compare these ratings with the ground truth ratings and compute the mean squared error, which is explained in Section 2.7.3.

   Each cross validation iteration returns one error score. In total we get 10 scores of which we take the mean. We repeat this process for 100 iterations and take the mean score of those iterations.

3.3.2.2 *Sliding Window Pipeline*

This pipeline uses the sliding window segmentation algorithm from Section 3.2.1.2 to compute a rating. This method works as follows:

1. Apply the sliding window segmentation algorithm on the queried chord sequence for which we want to compute a rating, but only use the first step of the algorithm, in which we obtain a vector of class probabilities.
2. Take the mean of these probabilities and convert them to a quality rating. The maximum and minimum rating are 5 and 1 respectively and the maximum and minimum probability are 0 and 1 respectively. We can map a probability to a rating with the following equation:

$$rating = 1 + probability \cdot 4 \tag{19}$$

3. Calculate the mean squared error by comparing each predicted rating with the corresponding ground truth rating, and then take the average error of all points in the data set.

Because the segmentation algorithm is deterministic such that it always produces the same probabilities, and it does not have to be trained with the rating data, there is no need to use (multiple iterations of) cross-validation with this method.

4

DATA

In order to train and test a classifier or a segmentation algorithm, a data set is required. This data set should consist audio from which we can extract the chords ourselves. The types of audio that we will use for our experiments are:

1. **H-music:** These are full songs containing only h-music. This is music that follows the rules of harmony.
2. **Non-h-music:** This is audio that does not contain any h-music at all. This is either audio that does not contain any music or is audio containing music without a harmony (atonal and percussive music).
3. **Mixed audio:** This audio is a mixture of h-music and non-h-music (e.g. a song that has a long intro without music or a radio show or podcast where a person is talking in between songs).

Full h- and non-h-music chord sequences are required for the classification and segmentation algorithms. Both algorithms require a classifier that can distinguish h- from non-h-music, hence we need to train the classifier with such audio. The mixed audio is used to train and test the segmentation algorithm.

A good source for h-music is the McGill Billboard [38] data set. This data set contains around 900 songs of a variety of popular music. These songs are representative of the type of music that is queried in Chordify. One issue is that the audio of these songs is not publicly available because of copyright issues. Chord labels of these songs have been annotated by experts, but the format of these annotations are rather different from the output of chord extraction algorithms. Luckily Dan Ellis has written a script that can download the Billboard songs from YouTube, based on audio fingerprints[1]. By using this script, we have access to the audio of the Billboard data set and can then use our own chord extraction algorithms to obtain annotations, instead of using the Billboard annotations.

For non-h-music audio we have gathered a large variety of different sound recordings from the internet. These files have been obtained from the freesound.org sound library [39]. Naturally, it is impossible to get every kind of variation of sound, but it must be sounds that are somewhat common in daily life. The sounds for non-h-music are:

- Radio/TV shows (with only talking people)
- Bar/restaurant noise (crowded places where people talk with each other)

[1] `http://labrosa.ee.columbia.edu/~dpwe/resources/matlab/audfprint/scrape-yt.html`

- City environmental sounds
- Animal sounds
- Construction work
- Weather (rain, thunder or heavy wind)
- Music without a harmony
  - Percussive music
  - Atonal music. These are our corner cases, as they contain an intended melody.

A problem with these audio pieces from the internet is that their length varies from 30 seconds to an hour. We will only pick audio that has a length of at least one minute, otherwise there are too few chords in the audio. For the maximum length we take a look at our h-music data. The average length in that data set is 3m30s and the maximum length is roughly 10 minutes. If an audio piece is below 10 minutes, we will not make any cuts in it. But if it is above 10 minutes, we well take a random slice of around 4 minutes from somewhere in the middle of the audio piece. We pick the middle because the start or end could have weird artifacts (music tune for a radio show, long silences at the end, etc.), so we think that the middle part is the most representative of the audio piece.

For the mixed audio, we have gathered several internet radio podcasts from the archive.org internet library. These podcasts consist of speech (non-h-music) alternated by music (h-music), as this kind of audio is the most logical use case for segmentation of h- and non-h-music. We have a total of 17 files that are all between 15 and 79 minutes long and have a combined playtime of 12 hours. The ratio of h-music vs non-h-music of all files combined is 47.4% h-music and 52.6% non-h-music. Although, for an individual file the ratio is a bit more skewed towards h- or non-h-music. For every file, we have manually annotated the time stamps of the segment borders. With these borders, it is straightforward to calculate a ground truth class vector of 0s, 1s and -1s (mentioned in Section 2.7.2), that we can then use for evaluation later on. The length of the segments differ quite a bit, with the longest segment being 30 minutes and the shortest segment being only 15 seconds. The average length of a segment is 250 seconds with a standard deviation of 267 seconds.

All the audio of the non-h-music and mixed audio are under the creative commons license or can be directly found on YouTube. The name of this audio data set is **Non Harmonic Audio and Music (NHAaM)**. The sources and a download link for the NHAaM data set can be found here: `https://www.projects.science.uu.nl/music/resources/nhaam/`.

We will also make all the extracted chords of the different data sets available and combine them into the **Extracted Chords of Harmonic and Non Harmonic Audio (ECoHaNHA)** data set. This data set can be found at the following link: `https://www.projects.science.uu.nl/music/resources/ecohanha/`.

We also have a data set of 2539 songs that have been given a rating by Chordify users via the rating system. This rating is the average of all given ratings for that song. We will call this collection of chord sequences the **rating** data set. Each song has obtained between one and six ratings, the majority of which only has one or two ratings. The number of ratings per song is lower than our preferred number of 20 ratings (mentioned in Section 3.3.1).

Chord extraction

We let chord extraction algorithms extract the chords from the aforementioned audio files. When the chord sequences are extracted from the audio files, each audio file is converted into a chord file. In this chord file, each line stores a chord together with an onset and duration. These chords can be of one of the twelve pitches and either a major or minor triad. We consider a sharp chord and its flat variant (e.g. C# and D♭) to be the same chord, since they have the same pitch frequency, thus sound the same. Other chord types are not used, because many chord extraction algorithms, including HarmTrace, only work with minor and major types.

There is also the N-chord that indicates that no chord is currently being played. Some chord extraction algorithms only use the N-chord when there is no sound at all, whereas other algorithms also use the N-chord for sound that cannot be properly classified in one of the chords of the vocabulary, such as non-harmonic sounds. We end up with a vocabulary of 25 different chords.

There are two chord extraction algorithms that we will use:

- **HarmTrace**, Chordify's chord extraction algorithm, is the most important one that we will use throughout this research. HarmTrace calculates the beat and generally places four chords in one beat. This means that chords have a length between roughly $\frac{1}{4}$ and $\frac{3}{4}$ of a second, but the same chord can be repeated multiple times. HarmTrace only assigns the N-chord to total silence.
- **Chordino** is a chord extraction algorithm created by Mauch and Dixon [40] that uses a Hidden Markov Model. Chordino assigns N-chords to not only silence, but non-harmonic sound as well. There is a specific parameter (`boostn`) that determines the likelihood of the N-chord appearing. We've set this value slightly lower than the default (from 0.1 to 0.001) such that the chord files in the noise data set do not solely consist of N-chords.

  Chordino also does not put chords on fixed timestamps, thus never shows repetitions of the same chord. Instead it prints the chord with its duration, thus a chord that lasts 10 seconds is the same as a chord that lasts only 1 second in the sense that it gets printed only once. These single chord prints matter when we are using n-grams where the duration is not taken into account or in classification where we only look at the chord and no other information. The chord merge option that we mentioned in Section 3.1.1 is already applied by the Chordino algorithm automatically.

Our data sets are described in Table 4.1. In all data sets, except for mixed and rating, the number of h- and non-h-music files is exactly the same.

[2] All the non-corner data sets don't have any atonal music in them.

Data set	# of total files	# of chords	Description
ht	280	118891	Contains HarmTrace extracted chords for 140 h-music and 140 non-h-music files.
chordino	280	22854	Contains Chordino extracted chords for 140 h-music and 140 non-h-music files.
ht-small	100	44643	A subset of **ht**: contains 50 h-music and 50 non-h-music files (HarmTrace extracted).
corner-small	100	45702	Contains 50 h-music files and 50 atonal non-h-music files[2] (HarmTrace extracted).
corner	380	166019	A combination of **ht**, 50 atonal non-h-music files, and 50 extra h-music files.
mixed	17	63991	Contains HarmTrace extracted chords of 17 mixed audio files that contain both h-music and speech.
mixed-chordino	17	44400	Contains Chordino extracted chords of 17 mixed audio files that contain both h-music and speech.
rating	2539	288125	Contains HarmTrace extracted chords together with an average user rating per file.

Table 4.1: Description of the data sets.

5

CLASSIFICATION OF H-MUSIC AND NON-H-MUSIC

We will now describe the experiments that we have done for the classification problem. Our goal is to classify a chord sequence (of an entire audio file) as either h-music or non-h-music. In these experiments we will investigate the effect of various parameters. We wrap up the chapter by discussing the results of the experiments.

5.1 EXPERIMENTS AND RESULTS

Both the feature and LM classifier experiments are structured similarly. As a starting point we will use a default parameter configuration which we expect will perform well. These parameter configurations are explained prior to the experiments. Then in each experiment, we investigate one parameter by testing every value of that parameter. When a certain parameter assignment obtains better results than our default assignment, it will become the new default.

Occasionally, we will refer to the appendix for results of an experiment that used different parameter values than the default. Generally, these configurations perform worse, but since they have been tested, they are put in the appendix for the sake of completeness. Tables B.1 and B.2 contain all the parameter configurations that we have tested.

5.1.1 *Feature Classification Experiments*

We will now discuss the feature classification experiments, in which we will evaluate the following parameters: classifier, data set, chord merge, features, and transposition. Per parameter we will conduct some experiments and determine what effect the parameter has on the classification results. During an experiment of a specific parameter, we only change that specific parameter and leave the other parameters fixed to a specific value, otherwise an exponential number of configurations must be evaluated. Since each experiment takes several hours to complete, we would need several weeks to exhaustively test every configuration.

The default data set parameter will be fixed to **corner**, since it is the largest data set and contains the most variation in audio. The features parameter will be fixed to **chords**, because it is a stable feature that performs well and roughly the same in many configurations. By default, 12-transpose will be used on the training data only, since it reduces potential bias to certain keys and gives us more training data. We won't merge chords by default during the parsing step. We will still display the results of every classifier (with optimal classifier parameters), since they occasionally produce very different results.

Not all of the experimental results are listed here, because often they do not show anything new. Additional experimental results can be found in Appendix B. Only the most relevant results are shown in this section.

5.1.1.1 *Classifier*

In this experiment we will evaluate which classifier performs the best when using the default parameter configuration. We will find the best settings for every every classifier and return the accuracy that we have obtained with that classifier. These optimal classifier settings are for the corner set with 12-transpose, using the chords feature without the chord merging during the parsing step. It could be possible that these settings are suboptimal for a different configuration of parameters.

These are the classifiers together with their settings that we can tweak:

- KNN
  - k: The larger the value of k, the more stable the classifier becomes and the less overfitting takes place. This means that it won't be influenced by outliers or noise quickly.

  The best value of k turned out to be 1. Increasing the value of k only lowered the accuracy of the classifier.
- NB
  - Has no settings to tweak.
- Decision Tree
  - Maximum depth of the tree.
  - Minimum number of samples required to split an internal node.
  - Minimum number of samples required for a leaf node.

  A deeper tree and a low minimum of samples required for both type of nodes results in more overfitting, because more rules are created this way.

  Having no maximum depth turned out to be the best, having the required samples for an internal and leaf node as low as possible (2 and 1 respectively) also turned out to be the best. This means a rather complex and overfitted tree is built.
- SVM
  - Kernel: This can be Gaussian, Linear, Polynomial or Sigmoid.
  - C: The cost parameter puts a certain penalty on points that are too close to or on the wrong side of the boundary.
  - γ: Kernel specific parameter.

    The Gaussian kernel turned out to produce the best results. We first evaluated the individual effect of increasing C and γ. We used values for C between 1 and 10000, where 1000 gave the best results. For γ we tested values between 0.001 and 25, where 20 gave the best results. When combining C and gamma, simply using the best individual values did not give the best results; we had to find a good balance between them to obtain good accuracies. Eventually the best results were obtained with C = 10 and gamma = 20.
- Random Forest
  - Maximum number of classifiers.

– Parameters of the decision tree.

We used the same settings that we used for the individual decision tree. For the number of classifiers we tested values between 1 and 200. The difference between 50 and 200 classifiers was very small, thus we chose for the simpler model that only has 50 classifiers.

- AdaBoost

  – Maximum number of classifiers.

  For AdaBoost we also tried between 1 and 200 classifiers. Similarly to the random forest, 50 classifiers gave the best results.

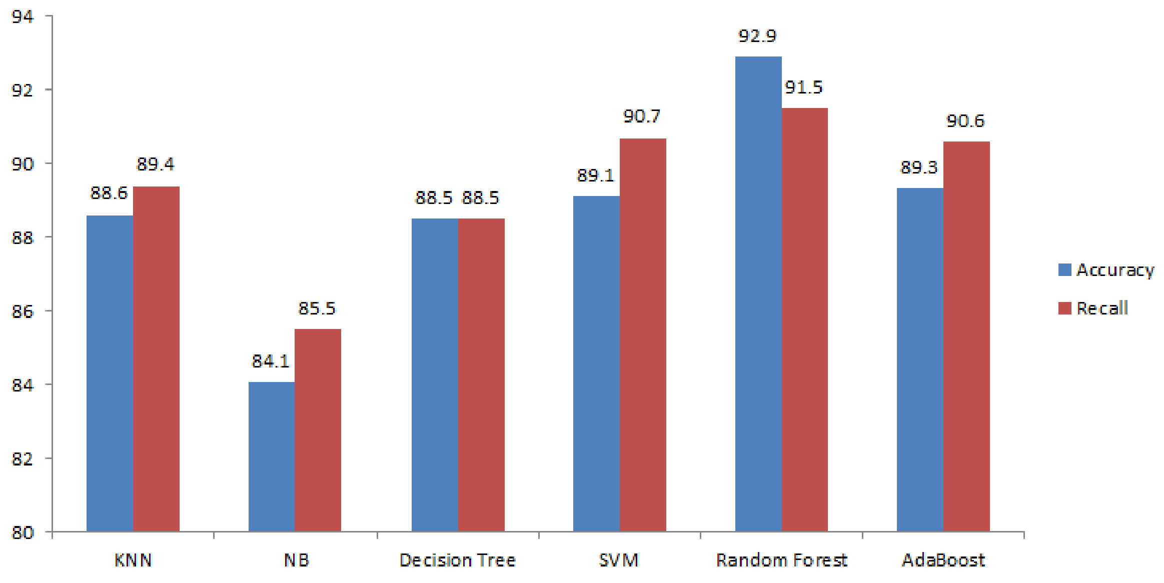


Figure 5.1: Accuracy and recall classification results for the corner data set, where each classifier is optimized. Only the chords feature is used.

Figure 5.1 shows that the random forest performs the best, being the only classifier above 90% accuracy. In other configurations, which can be seen in Tables 5.1 and 5.3, the random forest also performs the best. Similarly, the recall of the random forest is also the highest, but the AdaBoost and SVM classifiers are not far off.

For the remaining experiments we will not include the decision tree and NB classifier, because the NB classifier performs a lot worse than other classifiers and the decision tree is just a simpler (and worse performing) variant of the random forest.

- **Best parameter setting:** Random forest.

5.1.1.2 *Data Set*

In this experiment we will test how a data set influences the classification results. We will briefly reintroduce the five data sets:

- **chordino:** 280 audio files (140 pure h-music, 140 pure non-h-music) from which the Chordino algorithm has extracted the chords.

- **ht:** 280 audio files (140 pure h-music, 140 pure non-h-music) from which the HarmTrace algorithm has extracted the chords.
- **corner:** 380 audio files (190 pure h-music, 140 pure noise, 50 atonal music) from which the HarmTrace algorithm has extracted the chords.
- **ht-small:** Subset of **ht** with 100 audio files (50 pure h-music, 50 pure non-h-music).
- **corner-small:** Subset of **corner** with 100 audio files (50 pure h-music, the same as in **ht-small**, 50 atonal music pieces, the same as in **corner**)

We think that **corner** will obtain a lower accuracy than **ht** because it contains atonal music, which is more likely to be incorrectly classified as h-music than random noise. Because these two data sets aren't of the same size the experiment could be slightly flawed. **ht-small** and **corner-small** are therefore used to further investigate how difficult it is to classify atonal music. The results of the experiments are show in Table 5.1.

	chordino	**ht**	**corner**	**ht-small**	**corner-small**
KNN	96.5	86.1	**88.6**	79.9	**83.2**
SVM	98.2	**89.9**	89.1	**88.6**	83.6
Random Forest	96.8	**93.3**	92.9	**95.0**	92.7
AdaBoost	96.4	**90.2**	89.3	**91.6**	86.1

Table 5.1: Classification accuracies for different data sets and classifiers. In bold is the data set that obtains the highest accuracy (where **ht** and **corner** are compared with each other).

The **chordino** data set obtains the highest classification results. Even though both Chordino and HarmTrace are chord extraction algorithms, it is interesting that the chords produced by Chordino perform considerably better with classification. Two reasons in which Chordino differs from HarmTrace are 1) there are no chord repetitions (adjacent chords that are similar are merged by default) and 2) the non-h-music files of Chordino contain many N-chords, because Chordino does not classify these non-harmonic signals as musical chords.

The normal HarmTrace set and the HarmTrace set with corner cases perform almost evenly. Even though the set with corner cases might contain more difficult cases, it also contains more training data. For a better comparison between corner cases and normal noise, we use the two smaller data sets, **ht-small** and **corner-small**, which have the same amount of data. These results hint that the corner cases are indeed harder to classify.

5.1.1.3 *Chord Merge*

As seen in the previous experiment, the Chordino data set obtains a rather high accuracy. The merged chords might be the cause for this high accuracy. In this experiment we will merge the chords of the **corner** data set and compare it with the non-merged set. These results are shown in Figure 5.2.

The results for chord merge vary per classifier. KNN and SVM obtain a noticeable higher accuracy when using the merged chords. AdaBoost receives a small boost from the merged chords. The performance of the random forest does not improve when merged chords are introduced. The reason that the random forest performs worse may be because of the fact that the classifier creates several complex and overfitted trees. When we merge chords, we throw away some information, which the overfitted trees can then no longer use.

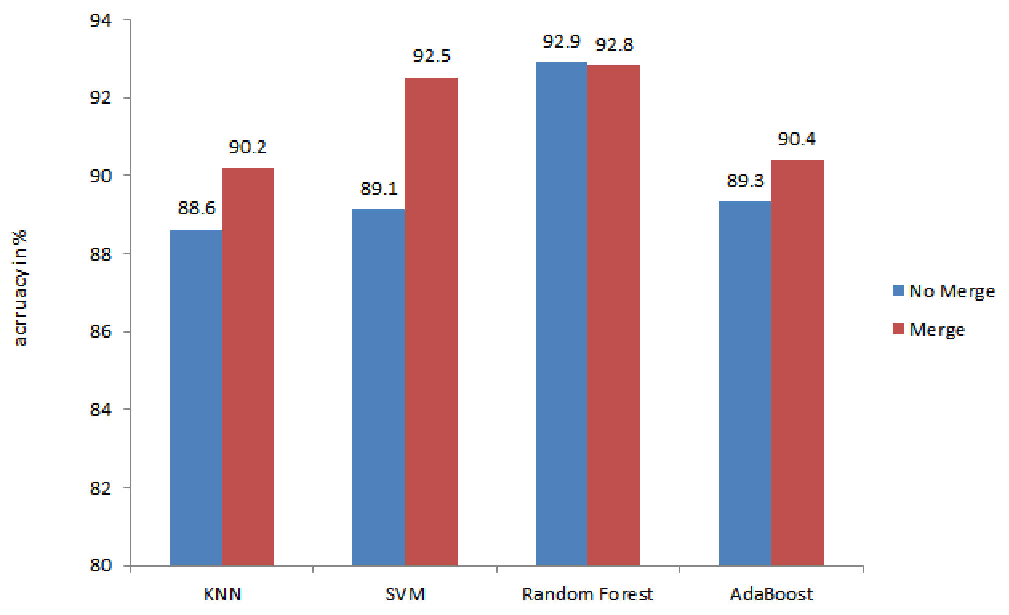


Figure 5.2: Classification accuracies for the corner data set where we examine the effect of merging of adjacent similar chords. Only the chords feature is used.

Even with merged chords, our classifiers still perform considerably worse on the **corner** data set than on the **chordino** data set. We hypothesize that merging chords isn't the only thing that sets CHORDINO apart, but also its assignment of N-chords to non-harmonic audio. This hypothesis is confirmed when we look at Table 5.2. We've done a run on the **chordino** set and have used a chord histogram in which we have omitted the N-chord dimension and have compared this to a run with a full chord histogram. Without the N-chord, the classification accuracy is much lower.

	Full histogram	**Histogram without N-chord**
KNN	96.5	91.6
SVM	98.2	94.5
Random Forest	96.8	96.0
AdaBoost	96.4	80.5

Table 5.2: Classification accuracies for the **chordino** data set, using a chord histogram with and without the N-chord.

For the remaining experiments we will use the merge setting on KNN, SVM and AdaBoost. For the random forest we will not merge the chords.

- **Best parameter setting:**
  - **Random Forest:** No merge.
  - **KNN, SVM and AdaBoost:** Merge.

5.1.1.4 *Features*

In this experiment, of which the results are shown in Table 5.3, we will examine how well every set of features performs. We will briefly repeat the features here:

- **chords:** Uses a normalized chord histogram, storing the relative occurrence of every chord.

- **chords + beat:** Uses the chord histogram and a beat histogram. The beat histogram stores how many times the same chord repeats itself *n* times.
- **chords + duration-sd:** Uses the chord histogram and stores the standard deviation of the duration of all the chords combined.
- **chords + beat + duration-sd:** Uses all three features.

Beat and duration-sd features are only shown in combination with the chord feature, because on their own they perform far below 80% accuracy.

	chords	chords + beat	chords + duration-sd	chords + beat + duration-sd
KNN	**90.2**	88.6	**90.2**	89.7
SVM	**92.6**	90.1	92.3	90.7
Random Forest	92.9	93.4	93.4	**94.2**
AdaBoost	90.5	**91.0**	90.0	90.7

Table 5.3: Classification accuracies for different features and classifiers. For every classifier, in bold is the feature set that obtains the highest accuracy. KNN, SVM and AdaBoost use chord merge, random forest does not use chord merge.

We can see that the random forest works better with every feature we add. The most likely explanation for this is that the random forest can turn into a rather complex classifier that makes use of all these different features. The best accuracy is obtained by the random forest classifier using the chords + beat + duration-sd feature. For the other classifiers, these extra features do not improve the accuracy (by much).

- **Best parameter setting:**
  - **KNN and SVM:** chords
  - **Random forest:** chords + beat + duration-sd
  - **AdaBoost:** chords + beat

5.1.1.5 *Transposition*

In this experiment the effect of transposition is evaluated. The results are shown in Table 5.4. The options regarding transposition are:

- **no transpose:** Do not transpose chord sequences
- **12-train:** Only transpose the training data to all 12 pitches, giving us 12 times as many data points for training
- **12-full:** Transpose both the training and test data, using the voting mechanism

In all cases, using a 12-transpose variant is better than not using one. For the random forest and AdaBoost the effect is less strong than for KNN and SVM. The difference between **12-train** and **12-full** is very small. **12-full** seems to be slightly better, but we cannot draw any conclusions from these numbers. Since both 12-transpose methods perform similarly and 12-train is the simpler we variant, we choose this transposition as our default.

- **Best parameter setting:** 12-train.

	no transpose	12-train	12-full
KNN	85.3	**90.2**	90.1
SVM	85.6	**92.6**	**92.6**
Random Forest	91.2	92.9	**93.2**
AdaBoost	88.0	90.5	**90.9**

Table 5.4: Classification accuracies for different types of transposition and classifiers. For every classifier, in bold is the type of transposition that obtains the highest accuracy.

5.1.1.6 *Best Configurations*

For all previous experiments, we have only reported the accuracy measure, but as stated earlier in Section 3.1.1, the recall measure on h-music is also interesting for us. Fortunately, the recall scores are very similar to the accuracy scores. The classifiers that obtained a high accuracy also obtain the highest recall. The best obtained results are shown in Figure 5.3, in which both the accuracy and the recall can be seen.

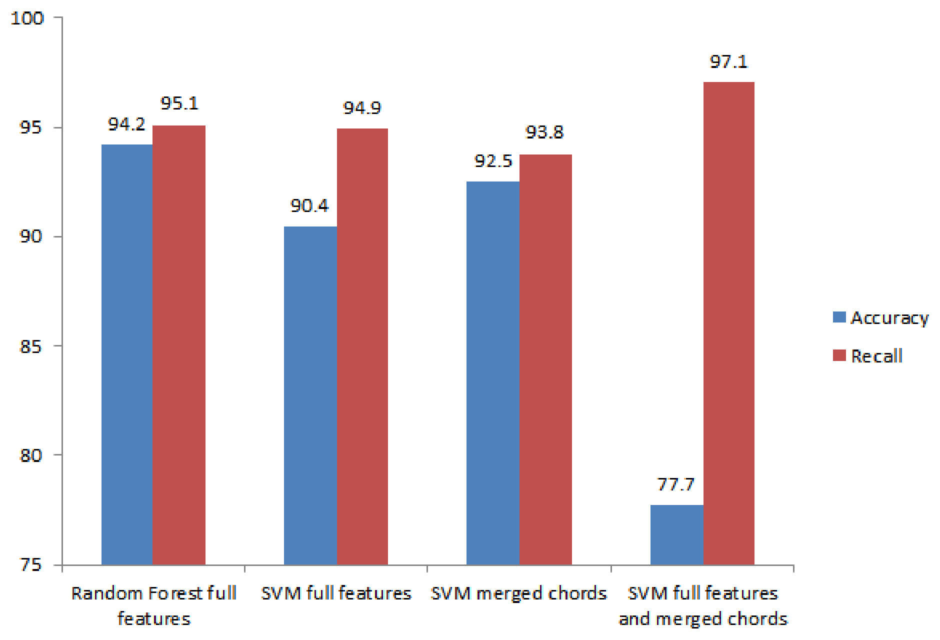


Figure 5.3: Classification results of the best configurations taking into account both the accuracy and recall.

The parameter configuration for the best overall classifier that also obtains the highest accuracy is:

- **Classifier:** Random forest.
- **Merging:** No merge.
- **Features:** chords + beat + duration-sd.
- **Transposition:** 12-train.

SVM takes the second place with multiple configurations. When using full features (chords + beat + duration-sd) and merging chords during the parsing step, a very high recall can be obtained with the SVM classifier. Unfortunately, the accuracy then suffers greatly. Thus using this configuration is generally not recommended, unless recall is of utmost importance.

5.1.2 *Language Model Experiments*

Similar to the feature classification experiments, we have several parameters, of which we change one at a time to evaluate the effects of that parameter. These parameters are: smoothing, data set, chord merging, transposition and n-gram order. For every experiment all order of LMs are shown and the default settings are to use the **corner** data set without merged chords and to use 12-transpose on only the training data. We also use additive smoothing by default, since it is the most simple and easy to understand smoothing method.

5.1.2.1 *Smoothing*

In this experiment we evaluate the effect of smoothing. Four different kinds of smoothed LMs and one unsmoothed LM are tested:

- **No smoothing**
- **Additive smoothing**
- **Good-Turing smoothing**
- **Constant discounting**
- **Witten-Bell smoothing**

These LMs are built from the corner data set with 12-transpose applied on the training set. The results for unmerged chords can be seen in Table 5.5, the results for merged chords are in Figure 5.4. For additive smoothing, we use $\delta = 1$, which means that we add one count to every n-gram.

n	None	Additive	Good-Turing	Constant Discounting	Witten-Bell
2	**91.5**	91.4	91.4	91.3	91.3
3	89.0	**90.3**	90.1	90.0	90.1
4	70.1	**93.7**	93.3	93.2	93.5
5	75.4	**93.5**	92.9	93.2	**93.5**
6	48.6	94.2	93.7	94.4	**94.8**
7	48.3	**94.6**	94.2	94.4	**94.6**
8	51.3	94.6	94.5	94.7	**94.9**

Table 5.5: Classification accuracies for different smoothing methods on different order unmerged LMs. In bold is the smoothing method obtaining the highest accuracy per n-gram order.

For the unmerged chords, the higher the order of LM, the more noticeable the effect of smoothing is. 2- and 3-gram unsmoothed LMs still perform as well as the smoothed LMs, but beyond that, smoothed LMs clearly outperform unsmoothed ones. From this we can conclude that using any form of smoothing is at least as good as using none, thus it is recommended to use some form of smoothing.

There seems to be no strong differences between the smoothing methods themselves. As we are using the LMs for classification, it does not matter if the probabilities of n-grams aren't entirely

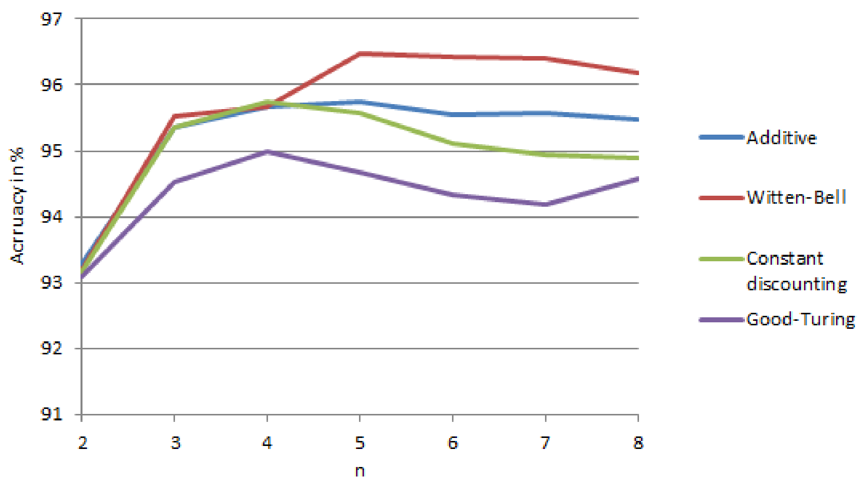


Figure 5.4: Classification accuracies for different order merged LMs using different types of smoothing.

correct. As long as the h-music LM gives higher probabilities to h-music chord sequences than the non-h-music LM does (and vice versa for non-h-music sentences), then the specific probability does not matter. Since additive smoothing appears to be working well on average and is easy to understand, we will continue with this smoothing method for unmerged chords.

When using the chord merge setting (Figure 5.4) beyond $n = 5$, the accuracy of Witten-Bell smoothing is about 1% higher than that of additive smoothing and a lot higher than the other smoothing methods. Thus for the chord merge setting we will make use of Witten-Bell smoothing.

- **Best parameter setting:**
  - **With chord merge:** Witten-Bell smoothing.
  - **Without chord merge:** Additive smoothing.

5.1.2.2 *Chord Merge*

In this experiment we will merge adjacent similar chords of the **corner** data set and compare it with the unmerged set. These results are shown in Figure 5.5.

For every order, the setting with merged chords obtains a considerably higher accuracy. Another advantage of merging chords is that chord sequences are becoming more compact, this means that the LM takes up less space (See Table 5.6), and the querying also takes up less time.

For the remaining experiments we will merge chords by default and use Witten-Bell smoothing. Occasionally, we will also do the experiment without the chord merge setting, in which case additive smoothing is used.

- **Best parameter setting:** Merge.

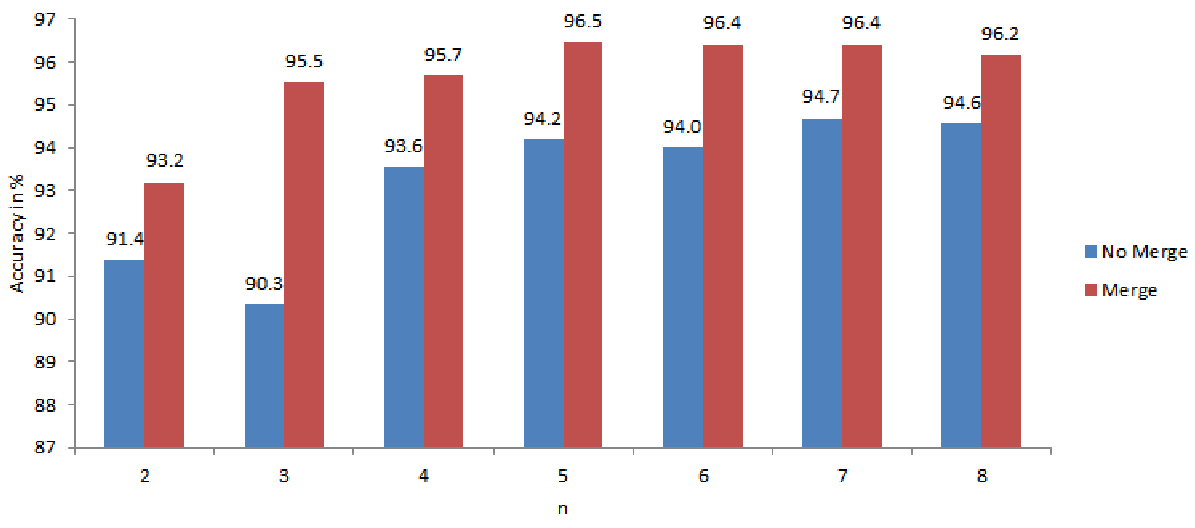


Figure 5.5: Classification accuracies for the corner data set where we examine the effect of merging of adjacent similar chords. The merged-chord configuration uses Witten-Bell smoothing and the non-merged configuration uses additive smoothing.

5.1.2.3 *Data Set*

In this experiment we apply classification on each of the five data sets. The results for merged chords can be seen in Figure 5.6. The results for unmerged chords can be seen in Table B.3 in Appendix B.

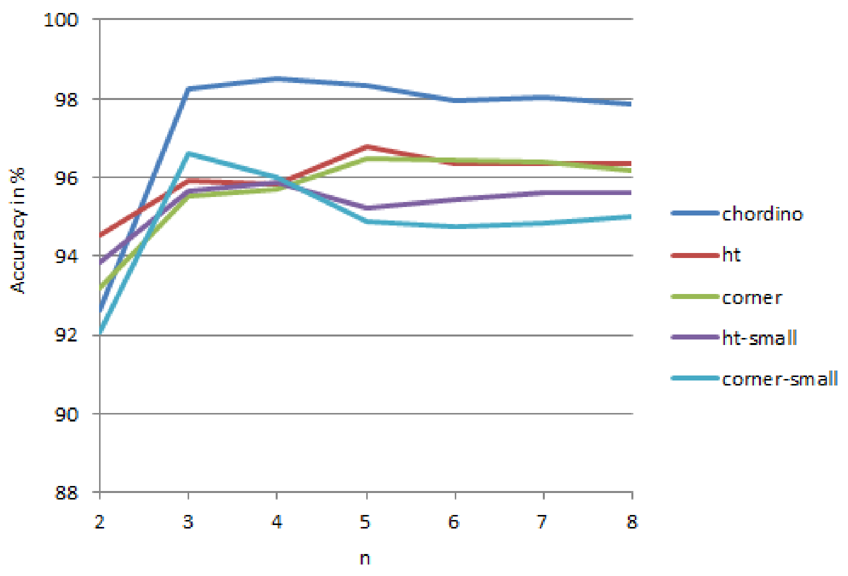


Figure 5.6: Classification accuracies for different order LMs using different data sets with merged chords.

As with the feature classification results, the **chordino** data set is the easiest to classify; every LM classifier scores the highest on this data set. For the HARMTRACE extracted chords, there seem to be no strong difference between the **corner** and **ht** data set. The 50 additional atonal pieces in the corner set do not result in a lower accuracy.

When we look at distinguishing noise from h-music (**ht-small**) compared to distinguishing atonal music against h-music (**corner-small**), there seems to be no strong difference either. The only trend we can see, is that the smaller data sets result in a lower accuracy, which is logical because there is less training data. In Paragraph B.1 of Appendix B we have looked into music styles a bit further with a different experiment that makes use of more than two LMs. Since no increase in accuracy is obtained in these experiments, they are not shown in this section.

5.1.2.4 *Transposition*

We will now look at the effects of transposition. The results from these experiments can be seen in Figure 5.7. In all the order LMs, apart from order 2, any form of transposition increases the accuracy by around 1%. Thus, using some form of 12-transpose is recommended.

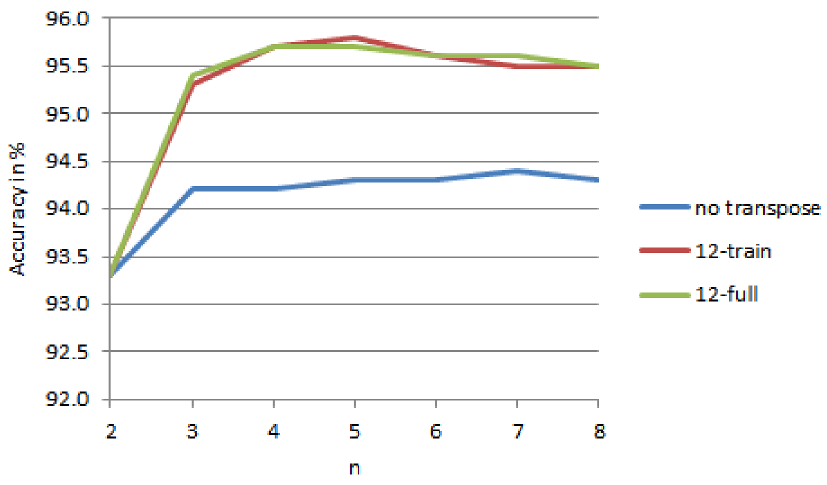


Figure 5.7: Classification accuracies for different order LMs using different kinds of transposition on the merged **corner** data set.

The only downside of using 12-transpose is that we create data that does not really exist. Many songs for example are more likely to appear in the key of C than in the key of C#, although we are assuming that the two are both as likely to happen.

There seems to be no clear difference between applying 12-transpose on only the training set (12-train) or on both the training and testing set with a voting mechanism (12-full). Because of the voting mechanism the accuracy is increased in some cases, but is also decreased in others. Thus, as was the case for the feature classifiers, we prefer to use the simplest form of 12-transpose, which is 12-train.

In Table B.5 of Appendix B the effect of transposition on unmerged smoothed and unsmoothed LMs can be seen. These results reinforce the conclusion that 12-transpose has a positive effect on the accuracy.

- **Best parameter setting:** 12-train.

5.1.2.5 *N-gram Order*

Finally, the effect of the order of the LM is investigated. We test n-gram orders between 2 and 8. A larger value of n means that we look at longer chord sequences, making the model more complex. In Table 5.6 is displayed how much n-grams of each order appear in our noise LM that has an order of 8. Every time we increase the order, the number of n-grams in the LM increases as well, meaning that longer combination of chords are also found plenty in our data. This means that more information is used in the higher order n-grams, since an n-gram model contains all n-grams up to n. Although, the number of possible n-grams increases by a factor 25 (which is the size of the chord vocabulary minus start- and begin of sentence tokens), the number of n-grams in our data only increases by a factor of roughly 2 in the unmerged-chord LMs of orders 4 and above. For the merged-chord LMs, increasing the order past 5 does not provide us with many new n-grams.

n	Unmerged chords	Merged chords
1	27	27
2	675	650
3	8,799	12,324
4	30,183	45,960
5	69,447	17,460
6	186,363	2,232
7	299,247	372
8	440,391	96

Table 5.6: Amount of n-grams that the noise LM with an order of 8 for merged and unmerged chords contains in the **corner** data set. An LM of order n contains the n-grams of order 1 through n.

In Table 5.5 it can be seen that higher order models only work when smoothing is applied. Smoothed LMs obtain an increased accuracy when we increase the order, whereas unsmoothed LMs perform much worse when we increase the order. In Figure 5.5 we can see that both the merged-chord and unmerged-chord LM accuracies are increased when we increase the order of the model. For merged-chord LMs an order of 3 already gives very good results, and an order of 5 is optimal for predictions. For the unmerged-chord setting this logically is more, as the many chord repetitions do not provide much extra information (i.e. Am-Am-C-C-G-G does not provide much more information than Am-C-G).

In language processing, using an order above four often gives poor results, as there isn't enough data to support the model. Since our vocabulary only contains 27 tokens, there are not that many possible combinations of n-grams (compared to a natural language), which means that many of the possible combinations of chords appear in our data. For that reason, our higher order n-gram models are also performing well.

- **Best parameter setting:** 5-gram model.

5.1.2.6 *Factored Language Models*

We will now experiment with FLMs, by splitting up the chords in several factors. The results can be seen in Table 5.7. The n stands for the number of chords used, not the number of factors. For example, when we use both the label and the type as a factor, an n-gram of length two will have four factors. When the duration factor is also used, $n = 2$ results in six factors.

	Factors		
n	**Label**	**Label + type**	**Label + type + duration**
2	91.7	93.0	93.2
3	93.0	84.5	93.5
4	93.4	84.9	92.3
5	94.0	84.0	91.1
6	90.4	83.1	91.3
7	90.7	83.3	timeout
8	91.0	83.5	timeout

Table 5.7: Classification accuracies for FLMs that use different factors and a different number of chords. Chords of the **corner** data set are merged and Witten-Bell smoothing is applied.

First, the most simple configuration is used: We only use the root note of the chord as a factor (e.g. C or G), omitting the chord type. These results are not that high compared to normal LMs, mainly because we are using less information with only one factor.

The next step is adding the chord type as a factor (resulting in, for example, C minor or G major). During the classification phase, when a queried n-gram is not found in the training set (resulting in a backoff to a lower order n-gram), we first remove the type factor of the furthest away chord and then the label of the furthest away chord. We keep backing off to smaller n-grams until there are counts in the training data for the n-gram. For $n = 2$ the results have become better with the additional factor, but for all the other order FLMs the accuracy has dropped by almost 10% with the additional type factor.

The third factor that we will add is the duration of a chord. This is the duration in seconds, rounded down. Unfortunately, this extra factor still does not allow the FLM to become nearly as good as normal LMs. A further consideration is the running time of the FLM models, which becomes exponentially larger as we increase the model order and number of factors. The 7- and 8-gram model with three factors did still not give us any results after running for three full months, so we have canceled these runs (hence the timeout in Table 5.7).

In Table B.7 of Appendix B some other FLM experiments are displayed with unmerged chords. These produce slightly higher accuracies, but still not better than our normal merged LMs. We can safely say that in our situation FLMs do not provide any benefits compared to normal LMs.

5.1.2.7 *Best Configurations*

We will now display the best results of the LM experiments, based on both the accuracy and recall. We only show one configuration, since this configuration obtains both the best accuracy (96.5%) and recall (97.5%):

- **Smoothing:** Witten-Bell smoothing.
- **Merging:** Merge.
- **n-gram order:** 5-gram model.
- **Transposition:** 12-train.

5.2 DISCUSSION

In this chapter we have seen how two different types of classifiers, the feature classifier and language model classifier, perform at the classification task. We will now briefly discuss some important results.

The first thing to note is that the chord extraction algorithm plays an important role. When we applied classification on the **chordino** data set, we obtained much higher accuracies compared to classification on HarmTrace data sets. The fact that the adjacent similar chords are merged together by Chordino's algorithm plays but a small role. More important is the fact that non-harmonic audio get N-chord labels, which is something the HarmTrace algorithm does not do.

In every situation, regardless of the type of classifier or other settings, 12-transpose is a technique that works well. It managed to greatly improve the classification results in all cases. There is no strong difference between using 12-transpose on only the training set or using it on both the training and test set with a voting mechanism. The question remains whether 12-transpose stays beneficial when we have more data. Perhaps with more data, we will notice that most songs are in a similar key, resulting in similar chords being used. 12-transpose then throws this information away.

For the general classifiers, the random forest performs best in almost every situation. When classifying the **corner** data set, which is a broad representation of audio files, making use of the full set of features (chords + beat + duration-sd) an accuracy of 94.2% is obtained and a recall on h-music of 95.1% is obtained. The simpler SVM classifier is about 2% worse and came on the second place. Perhaps in situations where memory or speed play a crucial role, it might be more advantageous to use the SVM classifier instead.

The best LM classifier performs around 2% better than the random forest, with an accuracy of 96.5% and a recall of 97.5%. This LM classifier is created by using two LMs with an order of 5 on which Witten-Bell smoothing has been applied and of which the chords have been merged during the parsing step. Both in terms of accuracy and recall, this classifier performs the best. On the other hand, the random forest classifier has many implementations and libraries available, whereas LM implementations are not widely available or require a bothersome installation process. Random forests also perform slightly better in terms of speed, although this difference is only noticeable when processing a large amount of data.

6

SEGMENTATION OF H-MUSIC AND NON-H-MUSIC

In this chapter we discuss the experiments that are done for the segmentation problem. The goal is to segment chord sequences into parts of h-music and parts of non-h-music. In Section 3.2 we proposed two segmentation algorithms with several parameters, that we will now test. After showcasing the experiments, we will discuss the results that we have obtained.

6.1 EXPERIMENTS AND RESULTS

Since our segmentation algorithm makes use of an h-music-classifier, we have first done some extra classifier experiments. This is shortly followed by the experiments on the two segmentation algorithms, in which we investigate the effect of several parameters.

6.1.1 *Variable Length Classification Experiments*

Even though we have found the optimal classifier in the previous chapter, this classifier may work less well on chord sequences of smaller length, containing both music and speech. These experiments serve to find the best classifier that works on shorter chord sequences. We have done these tests for both a feature and LM classifier.

6.1.1.1 *Variable Length Feature Classification*

In this experiment we will observe what effect the length of a chord sequence has on the classification results. We train the classifier with full length chord sequences, whereas the chord sequences in the test set have a reduced length. Instead of querying a full chord sequence to the trained classifier, we pick a random starting position in the chord sequence and then only use n seconds of the chord sequence for classification.

Similar to the chord merging experiments of Section 5.1.1.3, here merging chords is beneficial as well. The results for different classifiers using merged chords can be seen in Figure 6.1. The results for unmerged chords can be found in Tables B.8 and B.9 of Appendix B. In all cases, 12-train has been used, since it has always proved beneficial.

For all cases, increasing the length of a chord sequence also increases the accuracy. KNN already reaches an optimal accuracy at the one minute mark, whereas the other classifiers still keep

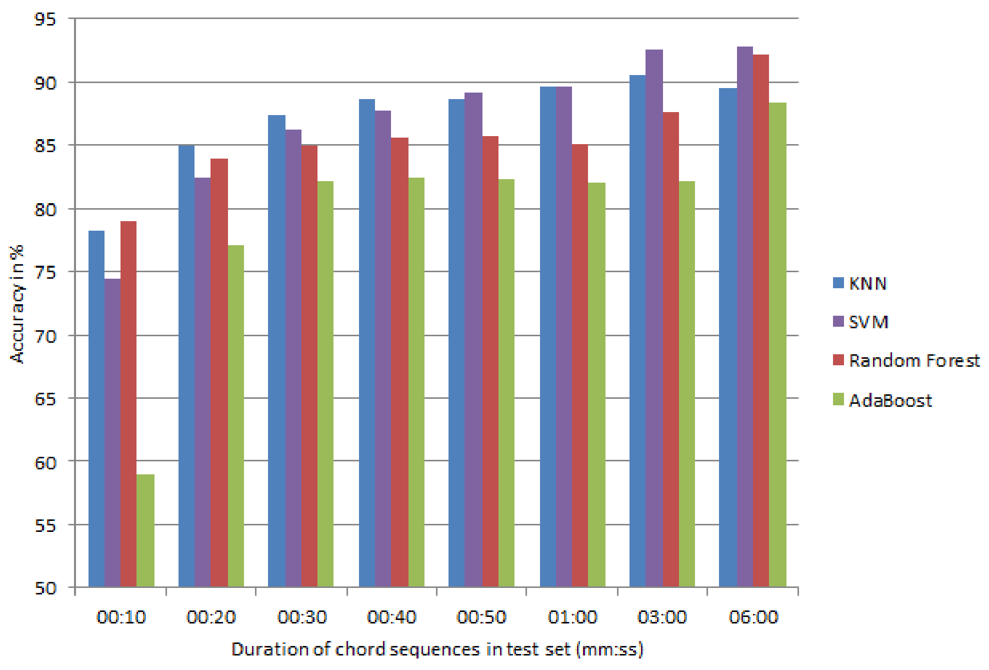


Figure 6.1: Obtained accuracies for different chord sequence lengths with using the **chords** feature and merging adjacent similar chords.

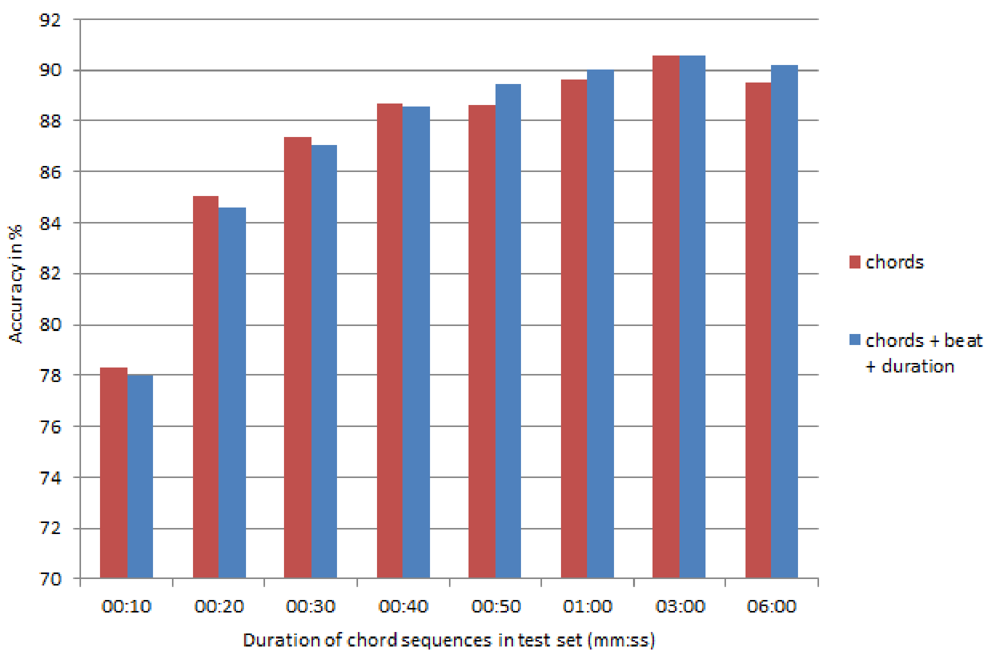


Figure 6.2: Classification results for using two different feature sets for variable length merged chord sequences.

improving beyond one minute. For the durations that are below one minute, KNN outperforms the other classifiers. Beyond one minute, the SVM and random forest classifiers perform better. In segmentation tasks, we generally look at segments that are less than a minute, thus the KNN classifier is the best candidate for segmentation purposes.

In Figure 6.2 we evaluate which feature performs the best for the KNN classifier. In these results we can see that with lower durations, the **chords** feature performs the best. In Tables B.10 and B.11 of Appendix B the effect for the features on the other classifiers can be seen.

The best configuration to use on small sequences is:

- **Classifier:** KNN.
- **Merging:** Merge.
- **Features:** chords.
- **Transposition:** 12-train.

6.1.1.2 *Variable Length LM Classification*

This experiment is done in the same manner as the feature classification experiment: Train LMs on full length chord sequences and classify a test set of chord sequences that contain only n seconds, starting at a random position. The parameters that we have tested in the first experiment are: 1) chord-merging, 2) smoothing and 3) the order of the LM. The results displayed here, which can be seen in Figure 6.3, use LMs with an order of 4. We refer the reader to Appendix B for the results of all the different parameter combinations.

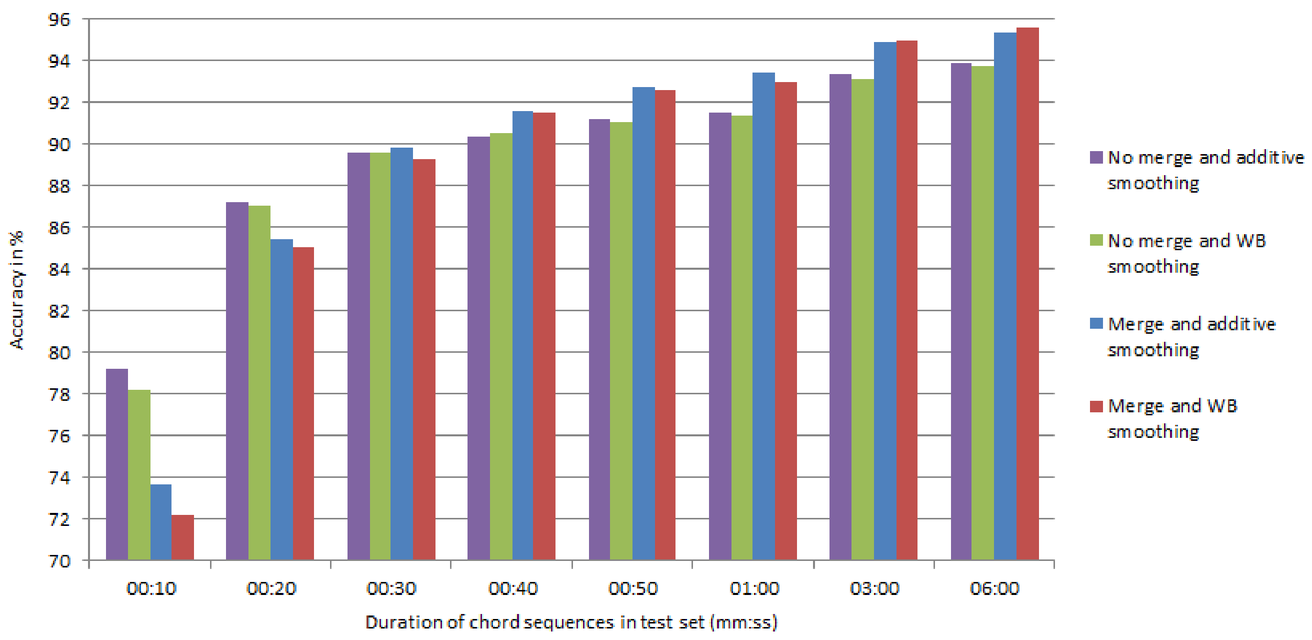


Figure 6.3: Obtained accuracies on chord sequences of variable duration with different LM classifiers.

Similar to the general classifiers, the longer the chord sequence becomes the better the classification performance. When we compare additive smoothing to Witten-Bell smoothing, additive smoothing is slightly better in almost all cases. Lastly, when we compare merging chords to not merging chords, there is an interesting trend. Below a duration of 30 seconds, not merging gives better results. On the other hand, when the duration goes beyond 30 seconds, merging chords becomes beneficial. The reason for that is when merging very short chord sequences, only a handful of chords remain. Classifying on only a few chords logically makes the classification process more difficult.

In Figures 6.4 and 6.5 we compare the different order LMs with one another for unmerged and merged chords respectively. Only additive smoothing is used in these comparisons. For durations between 10 and 30 seconds we use unmerged LMs, since they work better for these short durations. For durations between 30 and 60 seconds we use the merged chords. We do not look at longer durations, since in our research our window size for segments is generally not longer than one minute.

For the unmerged chords 2- and 4-gram models appear to work best. For the merged chords 3-, 4- and 5-gram models work best on average. For simplicity's sake we will say that in general a 4-gram model works best.

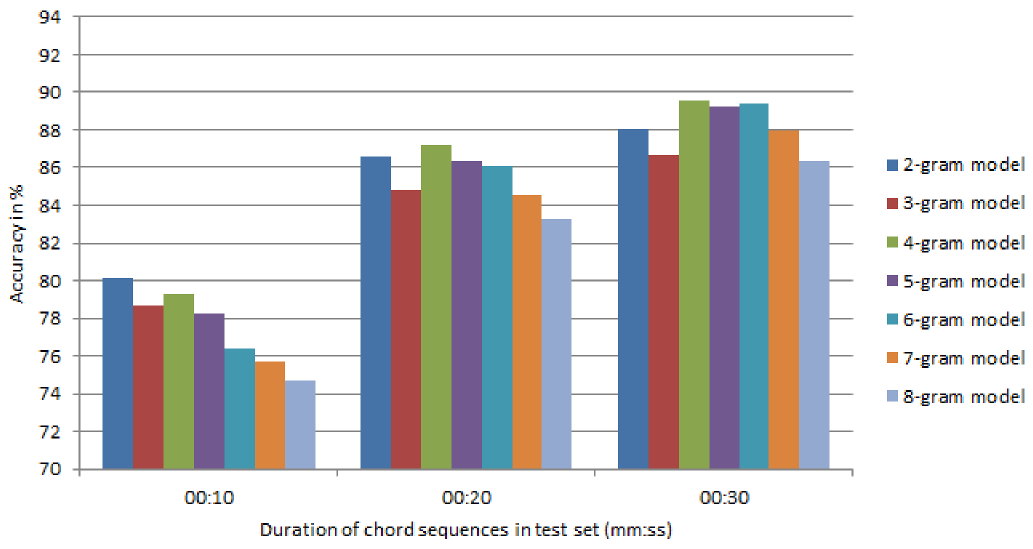


Figure 6.4: Obtained accuracies on chord sequences of variable duration for an LM classifier with unmerged chords.

The best configuration to use on small sequences is:

- **Smoothing:** Additive smoothing.
- **Merging:** No merge below 30 seconds. Merge above 30 seconds.
- **n-gram order:** 4-gram model.
- **Transposition:** 12-train.

6.1.2 *Segmentation Experiments*

Now that we know which classifiers work best on shorter chord sequences, we can continue with the segmentation experiments. Initially, we will try both the LM and KNN classifier. Because we require the classifier to return probabilities now, instead of class labels, it is not useful to use only one neighbor for KNN (as we then only get probabilities of 0 and 1). Class probabilities for KNN are computed by calculating the fraction of neighbors that are of the h-music class. We have chosen

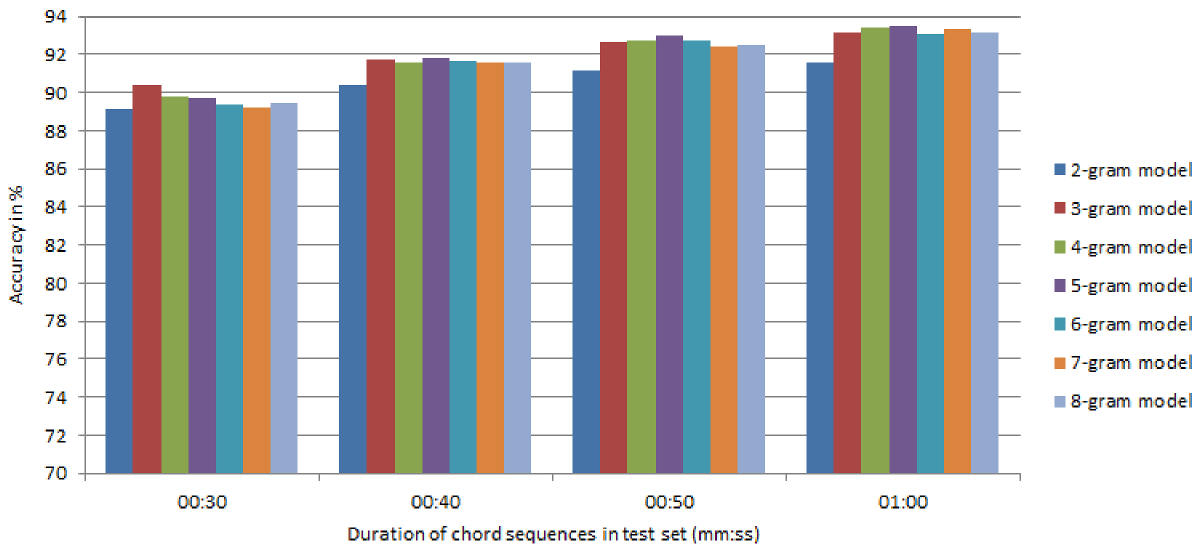


Figure 6.5: Obtained accuracies on chord sequences of variable duration for an LM classifier with merged chords.

$k = 7$, because 1) a higher k means that we obtain more precise probabilities and 2) increasing k beyond 7 gave us considerably worse results.

In each experiment we have applied 4-fold cross validation 25 times. This results in 100 training and test set pairs. In each of these 100 iterations we have have used simulated annealing and have tried to obtain the best segmentation F1-score on the training set, and have then used those parameters to segment the test set. Each of these 100 iterations has resulted in a parameter configuration paired by three scores (F1, border precision, and border recall) that were obtained on the test set. For every parameter we report the range in which the parameter was found (in those 100 iterations). We also report the mode of that parameter together with how many times it occurred (as a percentage), as we can then easily observe whether the parameter varied a lot or was mainly the same value. We also report the mean of each score over those 100 iterations. In Section 6.1.2.4 we give an overview of the obtained results and compare those results with one another.

6.1.2.1 *Computing a Random Baseline*

Before we commence the segmentation experiments, we would first like to know how difficult this problem is (with our specific data set). For that reason, we will calculate a random baseline first. We look at our data set and estimate what the F1, border precision, and border recall scores would be if we would randomly place borders.

Our data set consists of 12 hours of audio, which equals 43200 seconds. The number of borders of all files combined is 177. We will denote the number of predictions that we make as n. The border precision can be calculated as the probability that all n predictions fall 30 seconds within a ground truth border. The probability that one prediction falls within a border is $177 \cdot 30/43200 = 0.123$. Thus the probability that n predictions fall 30 seconds within a ground truth border is 0.123^n. For 177 predictions the border precision would be equal to 0.123^{177}, which is extremely small.

The border recall can be computed as the probability that all ground truth borders fall within 30 seconds of the predicted borders. The probability that only one ground truth border falls 30 seconds within a predicted border is $n \cdot 30/43200$. The probability that all ground truth borders fall within 30 seconds of a prediction is then $(n \cdot 30/43200)^{177}$. Theoretically, with 1440 predictions the border recall would be equal to 1. With 177 predictions, the border recall is equal to 7.26285e-162, which is once again very small.

As the F1 score is not based on border predictions, but based on whether a time frame corresponds to h-music or non-h-music, we can't use the precision and recall scores that we just computed. But since the F1 score is basically computed for a binary classification problem (per time frame), on average, the F1 score would be equal to 50%.

Based on these numbers, we can safely say that border precisions and recalls that are around 50% are already an enormous improvement over random guessing. Therefore, we also think that using a border threshold of 30 seconds is acceptable, as the problem is quite difficult.

6.1.2.2 *Separate Segment Algorithm (SSA)*

We will now investigate the separate segment algorithm (SSA) that was introduced in Section 3.2.1.1. Since the only parameter that this algorithm has is the window size, it does not make sense to use simulated annealing to explore to solution space. For this experiment we will instead iteratively check every window size between 4 and 100 seconds. The remaining steps of the pipeline remain unchanged; when we have found the optimal window size for the training set, we will segment the test set with this window size and obtain a performance score from that.

Results:

- **Window size:** 18-35 sec., *mode*: 32 sec. (23%)

- **F1:** 87.2% (89.4% on the training set)
- **Border precision:** 58.8%
- **Border recall:** 82.7%

This experiment has also been done with an LM classifier (4-gram model with 12-transpose and merged chords).

- **Window size:** 34-57 sec., *mode*: 46 sec. (52%)

- **F1:** 82.7% (84.2% on the training set)
- **Border precision:** 61.7%
- **Border recall:** 70.2%

6.1.2.3 *Sliding Window Algorithm (SWA)*

We will now perform several experiments that test the performance of the sliding window algorithm (SWA). As previously mentioned in the introduction of the algorithm, the SWA has three important procedures: 1) The core algorithm (core SWA) that produces a probability vector, 2) the addition of a bias value for h-music to all probabilities (bias SWA), and 3) smoothing out the smaller segments that are most likely miss-classifications (smoothing SWA). In these experiments we will evaluate the effect of these procedures.

Experiment 1: Core SWA

In the first experiment we will evaluate the core algorithm. Only the step size and window size are used by setting the parameters as follows:

- Step size: Varying between 1 and 10 (interval of 1).
- Window size: Varying between 10 and 100 seconds (interval of 2 seconds).
- Music bias: Fixed to 0.
- Segment size: Fixed to 10 seconds.
- Segment purity: Fixed to 0.

With this parameter configuration the step size and window size, that are part of the core algorithm, can be varied. The music bias is fixed to 0, thus always disabled. The segment size and segment purity are fixed to 10 and 0 respectively. As a result, when at least 0 of the 10 class values in the segment after a border have the same class as the border element (which is always true), the border is valid and no segments are smoothed away.

Results:

- **Step size:** 1-10, *mode*: 2 (16%)
- **Window size:** 16-40 sec., *mode*: 26 sec. (48%)

- **F1:** 89.8% (89.9% on the training set)
- **Border precision:** 23.5%
- **Border recall:** 96.8%

The core SWA seems to outperform the SSA. Only the border precision has dropped, which indicates that many more borders are predicted, of which a large portion are no real borders.

For the LM classifier we have fixed the step size to 4. The reason for this is that the step size did vary greatly in the KNN experiment, thus this parameter does not really influence the algorithm. If we fix the step size, then we only have to vary the window size. This can be done iteratively once again, instead of using simulated annealing, which greatly decreases the running time of the algorithm (else this experiment would take a couple of weeks). The results for the LM classifier are as follows:

- **Step size:** Fixed to 4.
- **Window size:** 6-25 sec., *mode*: 10 sec. (42%)

- **F1:** 87.5% (89.3% on the training set)
- **Border precision:** 62.1%
- **Border recall:** 82.5%

Because the LM classifier perform worse than the KNN classifier and is 5 to 10 times as slow, we will perform the remaining experiments with only the KNN classifier.

Experiment 2: Bias SWA

In the second experiment we will include the music bias to the parameters that we allow to vary. Additionally, we will fix the step size to 4, as it did not seem to have a big influence on the segmentation results. This results in the following parameter configuration:

- Step size: Fixed to 4.
- Window size: Varying between 10 and 100 seconds (interval of 2 seconds).
- Music bias: Varying between 0 and 0.5 (interval of 0.05).
- Segment size: Fixed to 10 seconds.
- Segment purity: Fixed to 0.

Results:

- **Window size:** 38-64 sec., *mode*: 54 sec. (49%)
- **Music bias:** 0.35 (100%)

- **F1:** 93.6% (93.7% on the training set)
- **Border precision:** 32.0%
- **Border recall:** 96.0%

We can see that the addition of the music bias greatly improves the results compared to the core SWA that uses KNN. Only the the border recall suffers slightly and has decreased by 0.8%.

Experiment 3: Smoothing SWA

For this experiment we will apply the segment smoothing step instead of the music bias procedure.

- Step size: Fixed to 4.
- Window size: Varying between 10 and 100 seconds (interval of 2 seconds).
- Music bias: Fixed to 0.
- Segment size: Varying between 10 and 100 seconds (interval of 2 seconds).

- Segment purity: Varying between 0.6 and 1.0 (interval of 0.05).

Results:

- **Window size:** 10-26 sec., *mode*: 10 sec. (53%)
- **Segment size:** 10-98 sec., *mode*: 14 sec. (6%)
- **Segment purity:** 0.6-0.95, *mode*: 0.65 (24%)

- **F1:** 92.2% (93.6% on the training set)
- **Border precision:** 82.4%
- **Border recall:** 72.0%

The F1-score is slightly lower on the test set compared to the bias SWA. Additionally, the border precision has increased considerably, whereas the border recall is slightly lower. This occurs because we remove a lot of small segments (including their borders). Therefore, there are considerably fewer (false) border predictions, which increases the precision. But because there are fewer predicted borders, some real borders are also not found anymore, which decreases the recall.

Experiment 4: Full SWA

In the fourth experiment we will use all three procedures (core, bias and smoothing) of the algorithm and thus use all parameters, except for the step size because of its minimal effect. This results in the following parameter configuration:

- Step size: Fixed to 4.
- Window size: Varying between 10 and 100 seconds (interval of 2 seconds).
- Music bias: Varying between 0 and 0.5 (interval of 0.05).
- Segment size: Varying between 10 and 100 seconds (interval of 2 seconds).
- Segment purity: Varying between 0.6 and 1.0 (interval of 0.05).

Results:

- **Window size:** 10-28 sec., *mode*: 14 sec. (21%)
- **Music bias:** 0.1-0.35, *mode*: 0.25 (36%)
- **Segment size:** 26-70 sec., *mode*: 50 sec. (9%)
- **Segment purity:** 0.6-0.95, *mode*: 0.8 (31%)

- **F1:** 95.1% (96.4% on the training set)
- **Border precision:** 87.7%

- **Border recall:** 72.8%

Compared to only using the bias or segment smoothing, the combination of these two procedures greatly increases the F1-score and the border precision. The border recall is slightly higher than the smoothing SWA, but considerably lower than the bias SWA.

Initially, we allow a prediction of a border to be 30 seconds off to still consider it a correct prediction. Below, we can see how the border precision and recall are affected if we change this value to 10 seconds.

- **Border precision, length 10:** 74.4%
- **Border recall, length 10:** 71.5%

The border recall does not change much compared to the 30 seconds variant, which means that most of the correctly predicted borders are only 10 seconds off from the ground truth. The border precision, however, suffers more, which means that a considerable portion of the predicted borders are between 10 and 30 seconds away from ground truth borders.

Experiment 5: Chordino

We do one final experiment in which we use CHORDINO extracted chords instead of HARMTRACE chords (once again using the 30 second border threshold). In the classification experiment of Section 5.1.1.2 we could see that the **chordino** data set was much easier to classify. We test whether segmentation also works better on CHORDINO extracted chords.

Results:

- **Window size:** 18-44 sec., *mode*: 2 sec.6 (24%)
- **Music bias:** 0.0-0.15, *mode*: 0.0 (71%)
- **Segment size:** 24-98 sec., *mode*: 60 sec. (8%)
- **Segment purity:** 0.65-0.95, *mode*: 0.75 (20%)

- **F1:** 94.5% (94.7% on the training set)
- **Border precision:** 84.7%
- **Border recall:** 80.2%

The F1 score and border precision are slightly lower compared to using HARMTRACE chords. We would like to investigate why CHORDINO performed very well on the classification problem, but not on the segmentation problem. We have tried increasing the probability of N-chords (boostn $>$ 0.01), but this didn't give us far different results. When we take a closer look at the ratio of N-chords in both the **chordino** and the **mixed-chordino** set, they appear to be exactly even (0.063), therefore the N-chords can't be the cause.

When we inspect the files of the **chordino** set more closely, there is a large difference in file size (and thus the number of chords). Many non-h-music files are really small and sometimes only contain one long N-chord, whereas the h-music files and a minority of non-h-music files contain many (short) chords. What's more, the non-h-music files that are speech samples contain the most number of chords (in relation to the length of the file). It is therefore very likely that speech and music share similar (Chordino extracted) chord patterns or at least a greater variation in chords. As the chord structure of speech looks similar to music, Chordino has more difficulty distinguishing music from speech. Our segmentation data set mainly consists of music and speech, hence the reason Chordino doesn't perform as well as it did for the classification problem.

This theory is strengthened by the fact that the best value for the music bias parameter often was 0.0 (71% of the time). Since speech looks similar to music, many speech (non-h-music) segments are classified as music, and therefore, the music bias is no longer required.

6.1.2.4 *Significance Tests*

We will now test whether there is a significant difference between the results of the segmentation algorithms and its variants. We will test whether there is any significant difference between the algorithms of experiment 0 (the separate segment algorithm) through 4 (the full sliding window algorithm with all its procedures). An overview of the obtained scores can be seen in Table 6.1 .

	F1	**Border precision**	**Border recall**
Experiment 0: SSA	87.2	61.7	70.2
Experiment 1: Core SWA	89.8	23.5	96.8
Experiment 2: Bias SWA	93.6	32.0	96.0
Experiment 3: Smoothing SWA	92.2	82.4	72.0
Experiment 4: Full SWA	95.1	87.7	72.8

Table 6.1: Overview of the obtained results of the segmentation experiments on the **mixed** data set. The KNN classifier is used for the segmentation algorithm.

We have 17 data points in the **mixed** data set, and have done 25 iterations of cross validation. This gives us $17 \cdot 25 = 425$ different values. We can either take the mean of the 25 iterations for each of the 17 data points, or keep all the values as separate points. As there sometimes is a difference of more than 5% between values of the same song, we decide to keep each point separate.

We have first performed a Friedman ANOVA test to determine whether there are any significant differences between the results of the three algorithms. The regular ANOVA test cannot be used, as it assumes that our data is normally distributed, which is something we cannot assume. We will follow this up with a post-hoc Tukey HSD test, to determine which algorithms differ significant from each other.

There are significant differences between the algorithms, $\chi^2(4, 2120) = 564$, $p < 0.00001$. With the Tukey HSD test results we can observe a significant difference between all algorithms, except for Bias SWA and Full SWA, and Bias SWA and Smoothing SWA, using $\alpha = 0.01$ (see Table 6.2).

6.2 DISCUSSION

When classifying chord sequences of much shorter length, we have seen that for the feature classifiers, KNN with merged chords and the chords feature performs the best in all cases. For the LM classifiers, an order of 4 with additive smoothing seems to work best, and merging only is beneficial when chord sequences are longer than 30 seconds. When we apply classification on chords of a

Algorithm 1	**Algorithm 2**	**Difference in means**	**p-value**	**Signifcant?**
SSA	Core SWA	-0.0208	< 0.001	Yes
SSA	Bias SWA	-0.0548	< 0.001	Yes
SSA	Smoothing SWA	-0.0438	< 0.001	Yes
SSA	Full SWA	-0.0651	< 0.001	Yes
Core SWA	Bias SWA	-0.0340	< 0.001	Yes
Core SWA	Smoothing SWA	0.0230	< 0.001	Yes
Core SWA	Full SWA	0.0443	< 0.001	Yes
Bias SWA	Smoothing SWA	-0.0110	0.0490	No
Bias SWA	Full SWA	0.0103	0.0755	No
Smoothing SWA	Full SWA	-0.0213	< 0.001	Yes

Table 6.2: Results for the Tukey HSD test using $\alpha = 0.01$.

mixed audio signal that contains both h- and non-h-music, the KNN classifier is the most suitable, both in terms of F1 score and processing speed.

The segmentation problem (for this specific data set) cannot be solved with random heuristics, as we obtain a random baseline with a border precision and recall close to 0. We have obtained baseline scores with the separate segment algorithm, which obtains an F1-score of 87.2% (on the test set). Relatively many false borders are predicted (border precision of 41.2%), but most of the ground truth borders are also found (border recall of 82.7%).

The sliding window algorithm improves on these results in many ways. When using only the core algorithm (core SWA), an F1-score of 89.8% is obtained. This is a statistically significant improvement over the separate segment algorithm. Unfortunately, many false borders are predicted as the border precision is only 23.5%. On the other hand, the border recall is 96.8%. We conclude that the core algorithm predicts too many borders.

Adding either the music bias procedure (Bias SWA) or the smoothing procedure (smoothing SWA), significantly improves the F1 score once again to 93.6% and 92.2% respectively. With the smoothing procedure, there is a large increase in the border precision (from 32.0% to 82.4%), which indicates that many falsely predicted borders are indeed smoothed out.

When combining the music bias and segment smoothing procedure (full SWA), an even better F1-score is obtained: 95.1%. A border precision of 87.7% and a border recall of 72.8% is obtained. The F1 score of the full SWA is significantly better than all algorithms, except for bias SWA.

From these tests we can conclude that the core SWA already performs significantly better than the separate segment algorithm (in terms of F1 score), and that each procedure we add has a positive effect. In many cases, every extra procedure increases the F1 score significantly.

When we compare the training and test set F1 scores with each other, in most cases the training score is higher than the test score. However, these differences in score are generally small and at most 2%. Thus in our eyes, overfitting does not strongly influence the results.

7 ASSESSING THE QUALITY OF CHORDS

We will now discuss how we have tackled the problem of assigning quality scores to chords. The goal of assessing the quality of chords is solved by first gathering user quality ratings. With these ratings we train two different algorithms, a regression method and a sliding window approach, and then try to predict the user ratings ourselves.

Unfortunately, the results of these experiments have been inconclusive so far. We will now provide a small case study, in which we examine the ratings a bit more and report the results from our initial experiments. In the discussion we will try to explain these poor results.

7.1 CASE STUDY ON QUALITY ASSESSMENT

We will first take a closer look at our data. In Section 3.3.1 we stated that we wanted at least 50 songs, and at least 20 ratings per song. We do have 2539 songs, but do not have 20 ratings per song. In most cases we only have 1 or 2 ratings, and in a few rare cases we have between 3 and 6 ratings for a song. A person could give a bad rating to good quality chords and vice versa (either with ill intend, or because the user rates how much he likes the song instead of the chords). When those chords only have that specific ″miss-rating″, the average rating becomes totally useless. Whereas, when there are more than 20 ratings for a song, the probability that all these ratings are ″miss-rated″ is low, leaving us with a more reliable average rating.

To investigate the reliability of these ratings, we have taken a look at some *default* chords. These are chords of songs that have been edited by hand; therefore we know that these chords are of high quality. We have obtained ratings for 17 of these default chords and taken the mean of all these ratings. One would expect these ratings to be high, as we know the quality of these chords is high. This is indeed the case, as the average rating for the 17 default chords is 4.69, which partially rejects the hypothesis that the ratings are unreliable.

7.1.1 *Experiments*

In the quality assessment pipeline of Section 3.3.2 we proposed two methods of predicting ratings for chord sequences: Regression analysis and a sliding window approach. For both methods we report the mean squared error and the R^2 score. Apart from our two proposed models, we also try some other heuristic methods for predicting quality ratings. The results can be found in Table 7.1.

	MSE	R^2
Least squares regression	2.70	-0.01
KNN regression (k = 15)	2.78	-0.04
Sliding window algorithm	2.83	-0.05
Always predict with the average rating of the training set	2.69	0.0
Always predict a rating of 3	2.81	-0.05
Train and test with all data (least squares)	2.69	0.0

Table 7.1: Mean squared error and $R2$ score results for different models that try to predict the rating for a chord sequence.

The first thing to notice from these results is that every model obtains an R^2 of zero or lower, which indicates that the models are of poor quality, as using the average rating as a prediction works as well as using the model. Even when the model is trained and tested with all data, it is still unable to do proper predictions.

7.2 DISCUSSION

We will now discuss the poor results that we have obtained and try to explain them. The first reason for these poor results, could be the lack of data. As we have previously mentioned, for many songs we only have one or two ratings, which makes these ratings unreliable. We also cannot guarantee that Chordify users are rating the quality of the chords and not whether they like or dislike the song. Even though 17 *default* songs obtained high ratings, which is something one would expect of high quality chords, we do not know for sure whether other chords have been also been rated reliably.

The second reason for poor quality predictions, could be related to the features and algorithms that we use for this problem. We do not exactly know why users rate a chord sequence as high or low quality. A low rating might be the result of some local error in the chords that occurs only for a few seconds. In the chord histogram that we use for the regression model, this local error will never be found, since we take the histogram of the entire song. The sliding window algorithm might be able to detect this local error, and return a few low probabilities. But as we take the mean of these probabilities to compute a score, these local errors also fade out. This could potentially be solved by being more strict towards low probabilities, although it might be difficult to tweak this properly.

Another problem with the sliding window approach, is that the algorithm is trained on HarmTrace extracted chord sequences. This means that the sequences that are trained with could also be of low quality. The classifier that is used for the segmentation algorithm will still associate these low quality chord sequences with h-music, and will give these low quality chord sequences a high probability of being h-music.

8 CONCLUSIONS AND FUTURE WORK

We will first summarize the conclusions from our experiments in Section 8.1 and discuss what we have learned about our three problems, which were as follows:

1. **Classification**: Given a chord sequence, can we determine whether the chord sequence corresponds to h-music or non-h-music?
2. **Segmentation**: Can we make a segmentation of the chord sequence such that we obtain h-music and non-h-music segments?
3. **Quality Assessment**: When we deal with a chord sequence that corresponds to h-music, can we give a score to this sequence that reflects to which degree a Chordify user accepts the chord sequence?

As there was not enough time to investigate all problems thoroughly, some future work remains. This future work is discussed in Section 8.2.

8.1 CONCLUSIONS

We have constructed a classifier which is able to classify chord sequences as h-music from non-h-music. In our framework we have tested several parameters which strongly influence the classification results. We have created a model for several feature classifiers (K-Nearest Neighbors, Support Vector Machines, Random Forests and AdaBoost) which uses a chord histogram. This chord histogram stores the relative appearances of each chord in the chord sequence. By using additional information about chord repetitions and durations we manage to obtain an accuracy of 94.2% and a recall on h-music of 95.1%.

We have also tested language models (LMs), which create a probability distribution over chord sequences. An LM for both h- and non-h-music is created, which both return a probability for a queried chord sequence. The queried chord sequence obtains the class of the LM that returns the highest probability. With this LM classifier we have obtained an accuracy of 96.5% and a recall of 97.5%.

When we use Chordino, instead of HarmTrace, to extract chords, accuracy and recall scores of around 98% are obtained. The main reason for that is that Chordino assign N-chords to non-harmonic audio, whereas HarmTrace only assigns N-chords to silence. These N-chords make it easier to distinguish h- from non-h-music.

These classifiers have been used by a segmentation algorithm, such that we can segment chord sequences into segments of h- and non-h-music. We have constructed a sliding window segmentation algorithm that obtains an F1-score of 95.1%. To obtain this score, we had to apply a small counter bias for music, as classifying a mixed signal (of both music and noise) automatically gave a bias to non-h-music. We have also applied a heuristic that smooths out small segments that are only a few chords long, as these small segments are most likely miss-classifications.

We have also tried to predict the quality of a chord sequence according to which degree the user accepts the chord sequence. We have first gathered 5-star user ratings for songs (and thus chord sequences) with our rating system. We have then tried to create a regression model from these chord-rating pairs, which gave us rather poor results; always predicting the average rating (three stars) gave the same results as using the model's predictions. Using classifier probabilities as a quality prediction gave poor results as well. The main causes for these poor results were a lack of reliable data and the fact that our methods work on global features, whereas errors in chord sequences are mostly local errors.

Our framework can be used by Chordify in several ways. Our classifier can be used to determine whether chords of an uploaded audio file should be stored or not. Only when the classifier predicts that the chords correspond to h-music will the chords get stored. As the recall of the classifier is 97.5%, in only 2.5% of the cases the chords will not get stored, even though they should be because they correspond to h-music. As a consequence, the chords must be calculated every time someone uploads this specific audio file.

The segmentation algorithm could potentially be used to only display chords in songs when there is actually h-music playing. Because the impact of false negatives (not showing chords, even though there is music) is much higher, the performance might first need to improve further to F1 scores close to 100% before Chordify would want to use the segmentation algorithm. Although the parameters of the algorithm could be easily tweaked, such that the recall becomes close to 100%, meaning that the number of false negatives is almost non-existent. Our method for assessing the quality of a chord sequence still needs some work as well, before Chordify can use it. This is further discussed in the next section.

8.2 FUTURE WORK

We will now discuss some additional research that could be done to improve our understanding of our research problems and to potentially obtain better results. In Section B.1, we did some small experiment in which we used three LMs instead of only two, where the third LM consisted of atonal music. There was no strong evidence that using three LMs gave us better results, but it might be worth to investigate this more thoroughly. When a large data set is available, of a least thousand entries, for different genres of music and perhaps different kinds of noise, it might prove beneficial to create an LM for every kind of genre music (or noise) and then use these LMs for classification. This improved LM classification could also be applied to other related research areas such as music genre classification.

The second area, in which many improvements can still be made, is that of quality assessment. First, more user ratings (for the same songs) must be obtained, before new experiments can be done. The methods to predict scores may also need to be modified, such that they can detect local errors more easily. The sliding window algorithm is the most suitable for this, as the window works on local parts of the chord sequence.

Koops [41] has already done some research on improving the quality of chord sequences by means of data fusion; users suggest improvements to chord sequences (edits), which are then fused together to obtain a better chord sequence. If these edits are investigated, we might be able to employ machine learning techniques to learn what good edits are, and then also learn what good and bad chord sequences are. With this information it might be easier to assess the quality of a chord sequence and to automatically improve it. Information about good and bad quality chord sequences could be applied to other chord extraction algorithms apart from HarmTrace

Something else that is missing in this research is a comparison with audio based techniques. We have obtained certain results with our algorithms that work on chord sequences, but we do not know how well they perform compared to the more common algorithms that work with the whole spectrum of audio features. It would be interesting to see whether our algorithms are able to keep up with audio based techniques.

Appendices

A IMPLEMENTATIONS

This section describes how the different pipelines are implemented. The entire framework is built with Python and is tested in version 2.6.6 and 2.7.6. We also make use of NumPy 1.8.2, SciPy 0.14.0 and the enum library found here: `https://pypi.python.org/pypi/enum34#downloads`.

A.1 FEATURE CLASSIFICATION

All classification methods come from the scikit-learn[1] module that has been developed by Pedregosa et al. [42]. This module contains a large collection of state-of-the-art machine learning algorithms. We are using version 0.16.1 of this module. These are the links to the documentation of the specific classifiers:

KNN:

`http://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-classification`.

Decision trees:

`http://scikit-learn.org/stable/modules/tree.html#classification`.

NB:

`http://scikit-learn.org/stable/modules/naive_bayes.html#bernoulli-naive-bayes`.

SVM (SVC):

`http://scikit-learn.org/stable/modules/svm.html#classification`

Random Forest:

`http://scikit-learn.org/stable/modules/ensemble.html#random-forests`.

AdaBoost:

`http://scikit-learn.org/stable/modules/ensemble.html#adaboost`.

[1] `http://scikit-learn.org/`

A.2 LANGUAGE MODEL CLASSIFICATION

For the estimation of an LM out of a training set, the SRILM toolkit[2] is used. The functionality of this toolkit is explained in the paper by Stolcke [43]. A C++ binary (`ngram-count`) is called that has as parameters a text file with chords (our training set text file), several options for smoothing and the name/location of the output file. The generated output file is of the ARPA format, which contains probabilities for every n-gram found in the data.

With the `ngram` binary we query a chord sequence to the LM. We have applied several smoothing methods, these are the flags used for each smoothing method:

- **No smoothing:** `-cdiscount 0 -gt3min 1 -gt4min 1 -gt5min 1 -gt6min 1 -gt7min 1 -gt8min 1`
- **Witten-Bell smoothing:** `-ndiscount`
- **Additive smoothing:** `-addsmooth 1`
- **Constant discounting:** `-cdiscount 0.75`
- **Good Turing smoothing:** No extra flags, since this is the default smoothing used.

FLMs are handled by the SRILM toolkit as well. The binary `fngram-count` is called to create an FLM. It uses a specific text file that lists the backoff path of the different factors. The FLMs are outputted to their respective ARPA file. The binary `fngram` then queries the FLMs with a test set and obtains probabilities for chord sequences in this test set. A thorough tutorial for the FLM component is contained in [44].

A.3 REGRESSION METHODS

The regression methods also come from scikit-learn.

Least squares:

`http://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares`

KNN regression:

`http://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbors-regression`

[2] `http://www-speech.sri.com/projects/srilm/`

B

EXTRA RESULTS

This section shows additional results from the experiments. In Table B.1 and B.2 we have listed all the configurations that we have tested with feature and LM classifiers (including the ones outside of the appendix).

Classifier	**Data Set**	**Chord Merge**	**Features**	**Transposition**	**Number of configurations**	**Found in**
KNN, NB, Decision Tree, SVM, Random Forest, AdaBoost	Fixed to corner	Fixed to No Merge	Fixed to chords	Fixed to 12-train	6	Fig. 5.1
KNN, SVM, Random Forest, AdaBoost	*chordino, ht, corner, ht-small, and corner-small*	Fixed to No Merge	Fixed to chords	Fixed to 12-train	20	Table 5.1
KNN, SVM, Random Forest, AdaBoost	Fixed to chordino	Fixed to No Merge	*chords and chords without N-chord*	Fixed to 12-train	20	Table 5.2
KNN, SVM, Random Forest, AdaBoost	Fixed to corner	*Merge and No Merge*	Fixed to chords	Fixed to 12-train	8	Fig. 5.2
KNN, SVM, Random Forest, AdaBoost	Fixed to corner	Fixed to Merge and No Merge	*chords, chords + beat, chords + duration-sd, and chords + beat + duration-sd*	Fixed to 12-train	20	Table 5.3
KNN, SVM, Random Forest, AdaBoost	Fixed to corner	Fixed to Merge and No Merge	Fixed to chords, chords + beat, and chords + beat + duration-sd	*no transpose, 12-train, and 12-full*	12	Table 5.4
KNN, SVM, Random Forest, AdaBoost	Fixed to corner	*Merge and No Merge*	*chords, and chords + beat + duration-sd*	Fixed to 12-train	16	Fig. 6.1 and 6.2, Table B.8, B.9, B.10 and B.11

Table B.1: List of all configurations that were tested for the feature classification experiments. In italic is the parameter of which every value has been tested. If two parameters (columns) are italic in the same row, then all possible combinations of the two parameters have been tested.

Smoothing	**Data Set**	**Chord Merge**	**Transpostion**	**n-gram order**	**Number of configurations**	**Found in**
None, Additive, Good-Turing, Constant Discounting, Witten-Bell	Fixed to corner	*Merge and No Merge*	Fixed to 12-train	*2 through 8*	70	Table 5.5, Fig. 5.4
Fixed to Additive and Witten-Bell	Fixed to corner	*Merge and No Merge*	Fixed to 12-train	*2 through 8*	14	Fig. 5.5
Fixed to Additive and Witten-Bell	*chordino, ht, corner, ht-small, corner-small*	*Merge and No Merge*	Fixed to 12-train	*2 through 8*	70	Fig. 5.6
Fixed to Witten-Bell for Merge. *None and Additive for No Merge*	Fixed to corner	*Merge and No Merge*	*no transpose, 12-train, and 12-full*	*2 through 8*	63	Fig. 5.7, Table B.5
FLM fixed to Witten-Bell	Fixed to corner	*Merge and No Merge*	Fixed to 12-train	*2 through 8 and 1 through 3 factors*	42	Table 5.7 and B.7
Additive and Witten-Bell	Fixed to corner	*Merge and No Merge*	Fixed to 12-train	*2 through 8*	28	Fig. 6.3, 6.4 and 6.5, Table B.12, B.13, B.14 and B.15

Table B.2: List of all configurations that were tested for the LM classification experiments. In italic is the parameter of which every value has been tested. If two parameters (columns) are italic in the same row, then all possible combinations of the two parameters have been tested.

B.1 MULTIPLE LMS

To continue with the question whether it is more difficult to combine two types of sound, we have done another experiment where we use three different LMs: one for h-music, one for noise, and one for atonal music. If either the atonal music or noise LM returns the highest probability in classification, we will classify the chord sequence as non-h-music. If the h-music LM returns the highest probability, we will classify the sequence as h-music. The results from this experiment are shown in Figure B.1.

The differences between using three LMs as opposed to two LMs is very small. For this specific configuration (Witten-Bell smoothing with merged chords), using three LMs negatively impacts the accuracy in most cases.

In Figure B.2, the same experiment is done while using additive smoothing on merged chords. Those results hint that there could be a potential increase in accuracy. In Table B.3 the results of

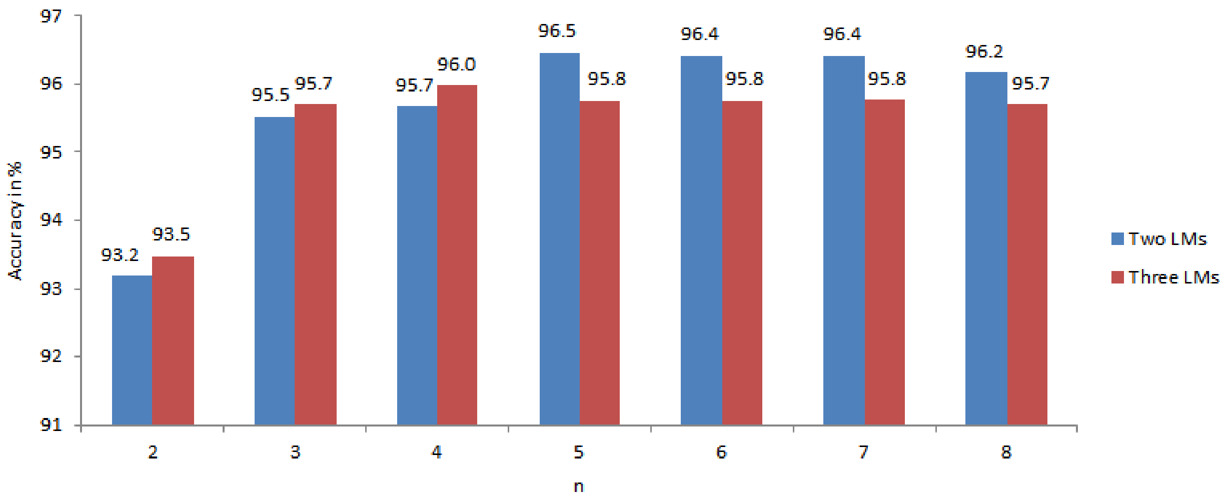


Figure B.1: Classification accuracies for using two LMs versus using three LMs for different order merged LMs of the corner data set using Witten-Bell smoothing.

	No smoothing		Smoothing	
n	**Two LMs**	**Three LMs**	**Two LMs**	**Three LMs**
2	91.5	**92.6**	91.4	**92.6**
3	89.0	**90.1**	90.3	**91.5**
4	70.1	**89.5**	93.6	**93.8**
5	75.4	**88.2**	**94.2**	93.9
6	48.6	**64.6**	94.0	**94.6**
7	48.3	**64.9**	**94.7**	93.1
8	51.3	**58.8**	**94.6**	93.0

Table B.3: Classification accuracies for using two LMs versus using three LMs, for both smoothed and unsmoothed LMs. In bold is the LM count that obtains the highest accuracy, per smoothing category.

using three LMs on unmerged smoothed and unsmoothed LMs can be seen. Unsmoothed models always benefit from three LMs, whereas for smoothed ones the results vary a lot.

Larger differences in accuracy could possibly be achieved if there was more atonal data or when there were more subclasses (for which in turn we would need even more data). Unfortunately with a total data set of 50 atonal pieces, only 5-10 atonal pieces out of the 50 total pieces will be in the test set. Thus only with more data the effect can be properly investigated.

B.2 ADDITIONAL RESULTS

	no transpose	**12-train**	**12-full**
KNN	82.1	**88.6**	**88.6**
SVM	82.9	89.1	**89.3**
Random Forest	91.0	92.9	**93.1**
AdaBoost	87.2	89.3	**89.8**

Table B.4: Classification accuracies for different types of transposition and classifiers. For every classifier, in bold is the type of transposition that obtains the highest accuracy. KNN, SVM and AdaBoost do not use chord merge, random forest does use chord merge.

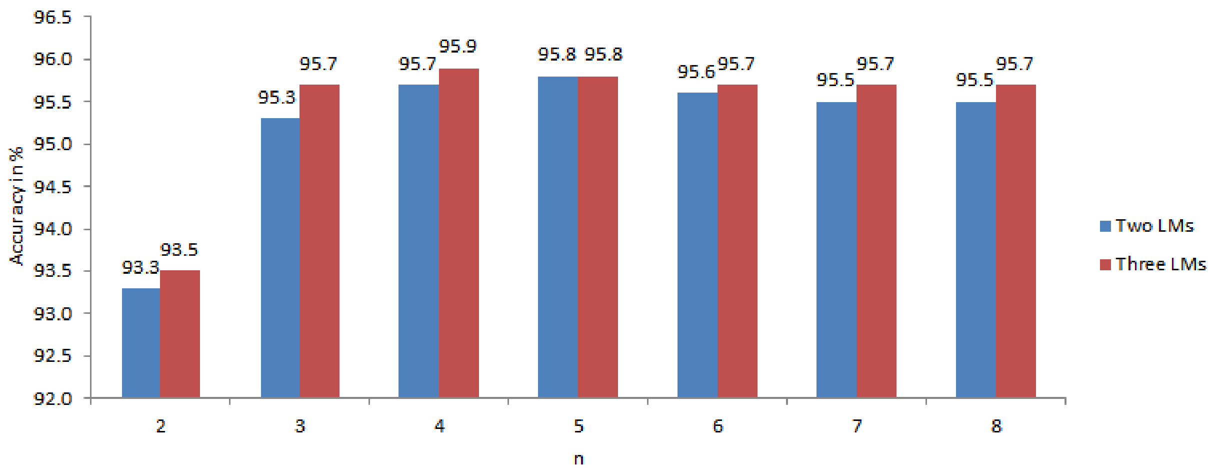


Figure B.2: Classification accuracies for using two LMs versus using three LMs for different order merged LMs of the corner data set using additive smoothing.

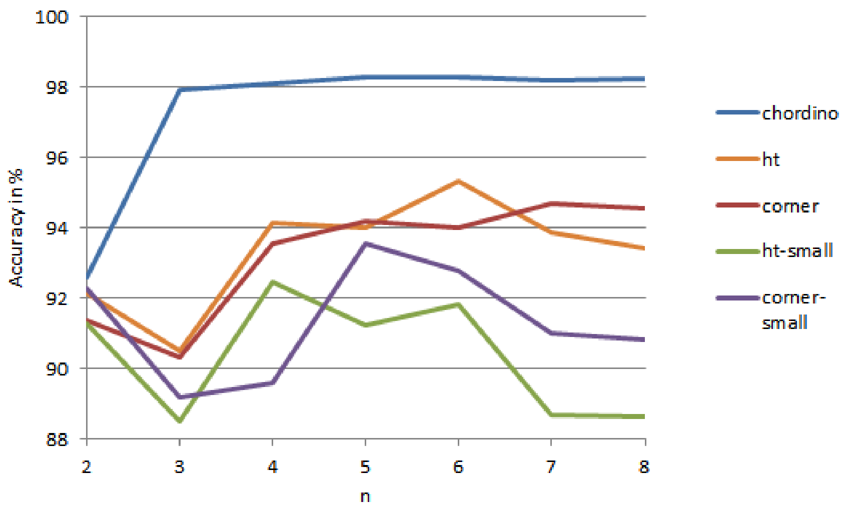


Figure B.3: Classification accuracies for different order LMs using different data sets with unmerged chords.

	No smoothing			Smoothing		
n	no transpose	12-train	12-full	no transpose	12-train	12-full
2	89.0	**91.5**	91.4	91.1	**91.4**	**91.4**
3	87.1	**89.0**	**89.0**	90.0	**90.3**	**90.3**
4	48.7	70.1	**70.7**	91.8	93.6	**93.7**
5	49.3	**75.4**	**75.4**	90.2	**94.2**	93.5
6	40.6	**48.6**	48.4	91.1	94.0	**94.1**
7	41.1	48.3	**48.4**	88.5	**94.7**	**94.7**
8	**52.2**	51.3	**51.6**	87.0	94.6	**94.7**

Table B.5: Classification accuracies for different types of transposition and different smoothing for every n-gram order using the unmerged corner data set. In bold is the transposition method that obtains the highest accuracy (separately measured for smoothed and unsmoothed LMs.

	No smoothing		Smoothing	
n	Two LMs	Three LMs	Two LMs	Three LMs
2	**93.6**	90.3	**93.5**	90.1
3	**93.6**	85.5	**92.2**	91.1
4	**97.9**	87.7	**94.3**	93.6
5	**96.2**	84.1	93.1	**95.2**
6	**94.9**	49.0	93.2	**95.8**
7	**93.7**	47.6	96.6	**98.4**
8	**76.2**	27.3	96.9	**98.4**

Table B.6: Classification accuracies for using two LMs versus using three LMs, for both smoothed and unsmoothed LMs. In bold is the LM count that obtains the highest recall, per smoothing category.

	Factors		
n	Label	Label + type	Label + type + duration
2	89.3	91.4	91.5
3	86.9	90.0	90.0
4	91.8	93.2	93.4
5	91.7	93.3	93.4
6	93.2	94.4	94.5
7	93.0	**94.7**	timeout
8	93.5	**94.3**	timeout

Table B.7: Classification accuracies for unmerged FLMs that use different factors and a different number of chords. In bold is the configuration that achieves the highest accuracy for every n-gram order.

Duration	KNN	Random Forest	AdaBoost	SVM
00:10	**72.6**	61.5	69.6	70.6
00:20	**80.6**	68.1	76.9	80.2
00:30	83.8	71.4	75.4	**84.6**
00:40	85.6	72.8	73.1	**86.0**
00:50	86.1	74.0	72.9	**87.2**
01:00	87.1	74.1	72.2	**88.1**
01:30	88.2	74.1	69.6	**88.9**
02:00	87.9	74.3	68.8	**88.8**
03:00	88.1	81.8	74.9	**89.5**
04:00	88.9	89.5	84.5	**89.6**
06:00	88.8	92.2	88.8	**89.4**

Table B.8: Classification accuracies for chord sequences with varying length. The settings are unmerged chords and the **chords** feature.

Duration	KNN	Random Forest	AdaBoost	SVM
00:10	**73.8**	66.1	62.3	59.1
00:20	**81.5**	71.3	68.9	73.6
00:30	**84.2**	73.6	72.2	80.8
00:40	**86.1**	74.6	73.1	84.2
00:50	**87.7**	75.9	73.3	86.1
01:00	**88.3**	76.8	74.4	87.2
01:30	89.3	76.8	75.8	**89.4**
02:00	89.1	77.2	76.3	**90.1**
03:00	88.5	83.5	81.4	**90.3**
04:00	87.9	90.2	86.4	**90.6**
06:00	87.7	93.1	88.6	**90.6**

Table B.9: Classification accuracies for chord sequences with varying length. The settings are unmerged chords and the **chords + beat + duration-sd** feature.

Duration	KNN	Random Forest	AdaBoost	SVM
00:10	78.3	**78.9**	58.9	74.5
00:20	**85.0**	84.0	77.1	82.5
00:30	**87.4**	85.0	82.2	86.2
00:40	**88.7**	85.6	82.5	87.8
00:50	88.6	85.8	82.3	**89.1**
01:00	**89.7**	85.1	82.1	89.6
01:30	89.9	85.1	80.5	**91.5**
02:00	90.5	84.8	79.1	**91.8**
03:00	90.6	87.6	82.2	**92.5**
04:00	89.7	90.5	85.8	**92.8**
06:00	89.5	92.1	88.4	**92.8**

Table B.10: Classification accuracies for chord sequences with varying length. The settings are merged chords and the **chords** feature.

Duration	KNN	Random Forest	AdaBoost	SVM
00:10	**78.0**	70.7	66.5	58.8
00:20	**84.6**	77.1	75.1	74.2
00:30	**87.1**	78.7	78.7	81.2
00:40	**88.6**	79.9	79.2	84.7
00:50	**89.4**	81.2	79.9	86.9
01:00	**90.0**	81.7	80.5	88.5
01:30	**90.9**	82.1	81.0	89.8
02:00	**90.6**	82.4	80.9	90.5
03:00	90.5	86.1	83.6	**90.7**
04:00	90.2	90.5	86.8	**90.6**
06:00	90.2	92.5	88.6	**90.7**

Table B.11: Classification accuracies for chord sequences with varying length. The settings are merged chords and the **chords + beat + duration-sd** feature.

Duration	2	3	4	5	6	7	8
00:10	**77.2**	74.8	72.2	72.5	72.4	72.5	72.5
00:20	**86.5**	86.3	85.1	83.9	83.7	83.7	83.5
00:30	89.0	**89.9**	89.3	88.8	88.3	88.3	88.2
00:40	90.1	**91.9**	91.5	91.3	90.9	91.1	90.9
00:50	91.1	**92.7**	92.6	92.6	92.5	92.3	92.2
01:00	91.8	93.0	93.0	93.0	93.1	**93.2**	93.1
01:30	92.0	94.0	93.4	93.8	93.9	**94.1**	94.0
02:00	92.6	94.0	94.0	**94.5**	94.6	94.7	94.4
03:00	93.2	94.4	95.0	**95.5**	95.4	95.5	95.4
04:00	93.4	94.9	95.5	**96.3**	96.2	96.2	96.1
06:00	93.4	94.9	95.6	**96.4**	96.4	96.2	96.1

Table B.12: Classification accuracies for chord sequences with varying length. The settings are merged chords and Witten-Bell smoothing.

Duration	**2**	**3**	**4**	**5**	**6**	**7**	**8**
00:10	**78.2**	74.9	73.7	74.7	74.4	74.6	74.6
00:20	**86.5**	86.4	85.4	85.5	84.9	85.0	85.0
00:30	89.1	90.4	89.8	89.7	89.4	89.2	89.5
00:40	90.4	91.7	91.6	91.8	91.7	91.6	91.6
00:50	91.1	92.7	92.8	93.0	92.7	92.4	92.5
01:00	91.6	93.2	93.4	93.5	93.1	93.4	93.2
01:30	92.1	**94.2**	93.8	94.1	94.0	93.9	93.9
02:00	92.6	94.1	**94.3**	**94.3**	94.1	94.0	93.7
03:00	93.3	94.6	94.9	**95.0**	94.9	94.8	94.6
04:00	93.5	94.9	**95.2**	**95.2**	**95.2**	95.0	95.1
06:00	93.6	94.9	**95.4**	95.3	95.2	95.2	95.2

Table B.13: Classification accuracies for chord sequences with varying length. The settings are merged chords and additive smoothing.

Duration	**2**	**3**	**4**	**5**	**6**	**7**	**8**
00:10	**79.7**	78.1	78.2	76.3	76.0	76.1	75.9
00:20	86.2	84.8	**87.0**	86.1	86.7	86.5	85.7
00:30	88.2	86.6	**89.6**	89.2	89.8	89.8	89.4
00:40	88.5	87.6	90.5	90.1	**91.1**	90.7	90.9
00:50	89.2	87.8	91.0	90.8	**91.9**	91.7	91.6
01:00	89.4	87.9	91.4	91.1	**92.3**	**92.3**	92.1
01:30	90.0	88.6	92.1	91.7	**93.0**	**93.0**	**93.0**
02:00	90.3	88.9	92.3	92.2	93.0	93.1	**93.4**
03:00	91.2	89.7	93.1	93.0	93.9	93.8	**94.2**
04:00	91.6	90.3	93.6	93.4	**94.4**	**94.4**	**94.4**
06:00	91.7	90.3	93.7	93.5	**94.6**	94.4	**94.6**

Table B.14: Classification accuracies for chord sequences with varying length. The settings are unmerged chords and Witten-Bell smoothing.

Duration	**2**	**3**	**4**	**5**	**6**	**7**	**8**
00:10	80.09	78.72	79.25	78.24	76.36	75.69	74.73
00:20	86.6	84.8	87.2	86.4	86.1	84.5	83.3
00:30	88.0	86.7	89.6	89.3	89.4	88.0	86.3
00:40	88.7	87.7	90.4	90.5	**90.9**	89.1	87.7
00:50	89.3	87.7	91.2	90.9	**91.5**	90.4	89.0
01:00	89.7	88.2	91.5	91.3	**92.0**	90.7	89.4
01:30	90.0	88.7	**92.6**	92.3	**92.6**	92.1	91.4
02:00	90.3	89.3	93.0	92.9	**93.1**	93.0	92.2
03:00	91.2	90.0	93.4	93.4	93.2	**94.0**	93.7
04:00	91.7	90.5	93.7	93.5	93.8	**94.2**	94.0
06:00	91.8	90.5	93.9	93.5	94.0	94.5	**94.7**

Table B.15: Classification accuracies for chord sequences with varying length. The settings are unmerged chords and additive smoothing.

BIBLIOGRAPHY

[1] C. Harte, M. B. Sandler, A. A. Abdallah, and E. Gómez. Symbolic representation of musical chords: A proposed syntax for text annotations. In *ISMIR*, volume 5, pages 66–71, 2005.

[2] T. Fujishima. Realtime chord recognition of musical sound: A system using common lisp music. *Proc. of ICMC*, pages 464–467, 1999.

[3] G. H. Wakefield. Mathematical representation of joint time-chroma distributions. In *SPIE's International Symposium on Optical Science, Engineering, and Instrumentation*, pages 637–645. International Society for Optics and Photonics, 1999.

[4] L. Oudre, Y. Grenier, and C. Févotte. Chord recognition by fitting rescaled chroma vectors to chord templates. *IEEE Transactions on Audio, Speech, and Language Processing 19.7*, pages 2222–2233, 2011.

[5] A. Sheh and D. P. W. Ellis. Chord segmentation and recognition using em-trained hidden markov models. *Proc. of 4th International Conference on Music Information Retrieval*, pages 185–191, 2003.

[6] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[7] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory 13.2*, pages 260–269, 1967.

[8] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77.2*, pages 257–286, 1989.

[9] W. B. de Haas, J. P. Magalães, and F. Wiering. Improving audio chord transcription by exploiting harmonic and metric knowledge. *Proc. of 13th International Conference on Music Information Retrieval*, pages 295–300, 2012.

[10] W.B. De Haas et al. Music information retrieval based on tonal harmony. 2012.

[11] K. Lee. Automatic chord recognition from audio using enhanced pitch class profile. In *Proc. of the International Computer Music Conference*, 2006.

[12] E. Gómez. Tonal description of polyphonic audio for music content processing. *INFORMS Journal on Computing*, 18(3):294–304, 2006.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[14] T. G. Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. 2000.

[15] L. Breiman. Bagging predictors. *Machine learning 24.2*, pages 123–140, 1996.

[16] R. E. Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. 2003.

[17] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning: data mining, inference and prediction, second edition. 2009.

[18] G. Lu and T. Hankinson. A technique towards automatic audio classification and retrieval. pages 1142–1145, 1998.

[19] P. Dhanalakshmi, S. Palanivel, and V. Ramalingam. Classification of audio signals using svm and rbfnn. *Expert Systems with Applications 36.3*, pages 6069–6075, 2009.

[20] A. Pikrakis, T. Giannakopoulos, and S. Theodoridis. A speech/music discriminator of radio recordings based on dynamic programming and bayesian networks. *IEEE Transactions on Multimedia 10.5*, pages 847–857, 2008.

[21] C. Pérez-Sancho, D. Rizo, and J. M. Iñesta. Genre classification using chords and stochastic language models. *Connection science 21(2-3)*, pages 145–159, 2009.

[22] T. Hedges, R. Hedges, R. Pierre, and F. Pachet. Predicting the composer and style of jazz chord progressions. *Journal of New Music Research*, 43(3):276–290, 2014.

[23] D. Jurafsky and J. Martin. Speech and language processing: An introduction to natural language processing. *Computational Linguistics and Speech Recognition. Prentice Hall, NJ, USA*, 2000.

[24] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language 13.4*, pages 359–393, 1999.

[25] J. A. Bilmes and K. Kirchhoff. Factored language models and generalized parallel backoff. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: companion volume of the Proceedings of HLT-NAACL 2003–short papers-Volume 2*, pages 4–6, 2003.

[26] M. Khadkevich and M. Omologo. Use of hidden markov models and factored language models for automatic chord recognition. *Proc. of 10th International Conference on Music Information Retrieval*, pages 561–566, 2009.

[27] M. R. López and A. Volk. Automatic segmentation of symbolic music encodings: A survey. 2012.

[28] J. Saunders. Real-time discrimination of broadcast speech/music. In *icassp*, pages 993–996. IEEE, 1996.

[29] E. Scheirer and M. Slaney. Construction and evaluation of a robust multifeature speech/music discriminator. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 2, pages 1331–1334. IEEE, 1997.

[30] L. Lu, H. J. Zhang, and S. Z. Li. Content-based audio classification and segmentation by using support vector machines. *Multimedia systems*, 8(6):482–492, 2003.

[31] M. Ferrand, P. Nelson, and G. Wiggins. Unsupervised learning of melodic segmentation: A memory-based approach. In *Proceedings of the 5th Triennial ESCOM Conference*, pages 141–144, 2003.

[32] J. J. Faraway. Practical regression and anova using r., 2002.

[33] R. Lowry. Concepts and applications of inferential statistics. 2014.

[34] R. Macrae and S. Dixon. Guitar tab mining, analysis and ranking. In *ISMIR*, pages 453–458, 2011.

[35] D. Cosley, S. K. Lam, I. Albert, J. A. Konstan, and J. Riedl. Is seeing believing? how recommender system interfaces affect users' opinions. pages 585–592, 2003.

[36] C. C. Preston and A. M. Colman. Optimal number of response categories in rating scales: reliability, validity, discriminating power, and respondent preferences. *Acta psychologica*, 104(1):1–15, 2000.

[37] E. I. Sparling and S. Sen. Rating: how difficult is it? pages 149–156, 2011.

[38] J.A. Burgoyne, J. Wild, and I. Fujinaga. An expert ground truth set for audio chord recognition and music analysis. *Proceedings of the 12th International Society for Music Information Retrieval Conference*, pages 633–638, 2011.

[39] F. Fond, G. Roma, and X. Roma. Freesound technical demo. *Proceedings of the 21st ACM international conference on Multimedia*, pages 411–412, 2013.

[40] M. Mauch and S. Dixon. Approximate note transcription for the improved identification of difficult chords. *Proc. of 11th International Conference on Music Information Retrieval*, pages 135–140, 2010.

[41] H. V. Koops, W. B. de Haas, and A. Volk. Integration of crowd-sources chord sequences using data fusion.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, pages 2825–2830, 2011.

[43] A. Stolcke. Srilm-an extensible language modeling toolkit. *Proc. Intl. Conf. on Spoken Language Processing 2*, pages 901–904, 2002.

[44] K. Kirchhoff, J. Bilmes, and K. Duh. Factored language models tutorial. 2007.