Ruprecht-Karls-Universität Heidelberg
Institut für Informatik
Lehrstuhl für Parallele und Verteilte Systeme

Bachelorarbeit

Maschinelles Lernen von komplexen Inhalten am Beispiel der Erfolgsvorhersage von Musikstücken

Name:	Sebastian Butterweck
Matrikelnummer:	2889264
Betreuer:	Artur Andrzejak
Datum der Abgabe:	17.02.2014

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, 17.02.2014

Zusammenfassung

Diese Arbeit untersucht die Möglichkeit, komplexe Sachverhalte algorithmisch zu erfassen und zu verarbeiten. Dies geschieht am Beispiel von Kompositionen der populären Musik der 1950er bis 1990er Jahre. Ziel ist die Vermessung von musikalischen Eigenschaften und das Erstellen eines Modells, auf dessen Basis Erfolgsvorhersagen für neue, unbekannte Musikstücke getroffen werden können. Die Algorithmen zur Extrahierung der Eigenschaften verarbeiten Musikstücke auf Basis von MIDI-Daten und versuchen, trotz des unerforschten Zusammenhangs zwischen musikalischen Charakteristika und Popularität oder kommerziellem Erfolg, die richtigen Attribute für ein robustes Vorhersagemodell auf Basis von Kompositionsregeln zu finden. Das Modell wird mit Methoden des Data Mining und des maschinellen Lernens erstellt und anschließend auf einem Datensatz von 209 Musikstücken des Genres Rock/Pop validiert.

Abstract

This thesis examines the possibilities of covering and processing complex entities algorithmically using the example of popular music from 1950s to 1990s. The goal ist to measure musical characteristics and to create a model for predicting the success of new and unseen songs. In order to do so, the algorithms extract musical features from MIDI-data on the basis of composition rules and try to find the right attributes to build a robust prediction model, regardless of the deficiently understood relation between musical characteristics and popularity. The model is built on methods of data mining and machine learning and validated with a 209 song dataset containing rock/pop music.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quell- und Pseudocodeverzeichnis

1. Einleitung

1.1. Motivation

1624 veröffentlichte der Literat Martin Opitz sein Werk *Buch von der deutschen Poeterey*, in dem er ein Regelwerk für die deutschsprachige Dichtung aufzustellen versuchte. Sein Ansatz war es, die bis dahin vorrangig lateinische Dichtung zu analysieren, Regeln zu extrahieren und diese auf die deutsche Sprache anzuwenden. Die Regeln sollten verbindlich für gute deutsche Dichtung gelten [5]. Opitz' Werk erfuhr nach eineinhalb Jahrhunderten der uneingeschränkten Zustimmung ab Beginn des 19. Jahrhunderts massive Kritik.

> „Einerseits sei Opitz kein 'großer Künstler', kein wahrhaft schöpferischer Dichter, nur Nachahmer, in dessen Arbeit die »rationalen Elemente vorherrschen, dichterische Phantasie und Empfindung zurücktreten [...]«. "
>
> ([5], 10)

Die Debatte über die algorithmische Analyse und Nachahmung von Kunst ist also kein Phänomen des Computerzeitalters, sondern schon viel älter. Das Beispiel Martin Opitz zeigt, dass es schon früh einen scheinbaren Gegensatz zwischen kreativ schaffenden Künstlern und systematisch analysierenden *Nachahmern* gab. Beide müssen aber nicht zwangsläufig Gegensätze darstellen. Kürzlich berichtete die Süddeutsche Zeitung über Kenichi Yoneda, einen japanischen Künstler, der den Versuch unternommen hat, Malerei algorithmisch zu simulieren [6]. Zu lesen war:

> „»Es geht mir nicht darum, ein schönes Kunstwerk zu erschaffen«, sagt er. Er wolle herausfinden, wie nahe ein Computer kommen kann, einen Menschen nachzuahmen."
>
> [6]

Dies ist der Ansatz, der auch dieser Arbeit zu Grunde liegt. Es geht nicht darum, den Menschen bei der Schaffung von Kunst entbehrlich zu machen. Es geht darum, herauszufinden, welche Gesetzmäßigkeiten bei der Produktion und der Rezeption von Kunst gelten. Jede Art von Kunstwerken ist so komplex, dass Algorithmen sie nur sehr schwer analysieren können. Im Folgenden soll ein Weg aufgezeigt werden, eine so komplexe Struktur wie eine Komposition auf algorithmisch messbare Eigenschaften abzubilden, um sie im Weiteren mit den Methoden des maschinellen Lernens analysieren zu können.

1.2. Ziele

Das Ziele dieser Arbeit ist der Entwurf von Algorithmen, die musikalisch sinnvolle Eigenschaften einer Komposition erfassen und auf Wertebereiche abbilden. Die Eigenschaften werden dabei immer mit dem Hintergrund einer Erfolgsvorhersage ausgewählt. Ziel ist es also, diejenigen Eigenschaften zu finden, die möglicherweise mit der Popularität oder dem kommerziellen Erfolg der Komposition korreliert sind. Im Weiteren soll für diese Eigenschaften ein Modell gefunden werden, das es erlaubt, auf Basis eines ausreichend großen Datensatzes eine Erfolgsvorhersage für Musikstücke zu treffen. Dies dient der Validierung der vermessenen Eigenschaften und soll zeigen, dass es möglich ist, komplexe Inhalte algorithmisch zu erfassen.

1.3. Grenzen

Die Grenzen und Schwierigkeiten dieser Arbeit sind sehr vielfältig. Der Zusammenhang zwischen den Chrakteristika kultureller Güter und deren Popularität ist weitgehend unerforscht [1]. Dies erschwert die Suche nach einem Ansatzpunkt für die Vermessung von Kompositionen. Es ist unklar, ob überhaupt musikalische Kriterien eine Rolle spielen oder ob die bestimmenden Faktoren für die Beliebtheit eines Musikstückes im Zeichen von sozialem Druck und der Vermarktungsstrategie der Musikindustrie stehen [7]. Zudem sind in der Musikgeschichte oft gerade diejenigen Stücke beliebt gewesen, die eine radikale Neuerung mit sich brachten. Ein Modell zur Erfolgsvorhersage kann aber Entscheidungen lediglich auf Basis eines bekannten Datensatzes treffen. Für die Bewertung eines stilistischen Umbruches fehlen schlicht die Referenzstücke. Darüberhinaus beinhalten jegliche musiktheoretischen Regelwerke, die Kompositionen betreffen und Grundlage einer algorithmischen Analyse sein können, oft nur Hinweise darauf, was zur Erschaffung eines erfolgreichen Werkes zu unterlassen ist. Verstößt ein Thema (eine Melodie) gegen

die Regeln, klingt es nicht schön. Ein Einhalten aller Regeln garantiert aber noch kein schönes Thema. Machmal entsteht ein solches erst durch die bewusste Verletzung einer Regel [2]. Dazu kommt das Problem des Rauschens in den Grunddaten, das im Rahmen eines Data Mining Prozesses immer besteht. Die im Datensatz enthaltenen Informationen über die Kompositionen können fehlerhaft oder unvollständig sein, da diese Daten in der Regel nicht von den Komponisten selbst, sondern von Dritten erstellt werden.

1.4. Aufbau

Kapitel 2 vermittelt die nötigen Grundlagen für das Verständnis dieser Arbeit. Dabei werden musikalische Grundbegriffe geklärt, die Funktionsweise des MIDI-Protokolls zur digitalen Erfassung und Speicherung von Kompositionen beschrieben, ein Überblick über die Methoden des Data Mining und des maschinellen Lernens gegeben sowie verwandte Arbeiten aus der Disziplin der Hit Song Science vorgestellt.

In Kapitel 3 werden verschiedene Eigenschaften einer Komposition vorgestellt, mathematisch modelliert und anschließend in algorithmische Pseudocodes übersetzt, die der Abbildung der Eigenschaften auf einen Wertebereich dienen.

Kapitel 4 beschäftigt sich mit der Implementierung der Algorithmen aus dem vorangehenden Kapitel und dem Entwurf eines maschinellen Lern- und Vorhersageprozesses. In diesem Zusammenhang werden der Datensatz für die Analyse ausgewählt, die notwenidgen Daten berechnet und ein Modell für die Vorhersage aufgestellt.

In Kapitel 5 findet schlussendlich die Auswertung der Ergebnisse des Vorhersagealgorithmus' sowie die Analyse hinsichtlich der Zielsetzung statt.

2. Grundlagen

Kapitel 2 dient der Vermittlung der Grundlagen, die für das Verständnis dieser Arbeit notwendig sind. Im Folgenden wird auf musikalische Grundbegriffe eingegangen, die die Grundlage für die Attributgenerierung bilden, und auf die digitale Musikschnittstelle MIDI (Musical Instrument Digital Interface), die als Repräsentation von Musikstücken das Datenformat für den Datensatz darstellt. Es werden Methoden und Algorithmen des Data Mining und des maschinellen Lernens erklärt sowie ein Überblick über die *Hit Song Science*, die Disziplin der Vorhersage des kommerziellen Erfolgs von Musikstücken, gegeben. Dazu werden bestehende Ansätze und Arbeiten aus dem Bereich der Hit Song Science vorgestellt.

2.1. Musik

Zum besseren Verständnis der Arbeit sind einige Grundlagenkenntnisse aus dem musikalischen Bereich nötig. Viele der Algorithmen in Kapitel 3 greifen auf musikalische Phänomene zurück, die hier kurz erörtert werden.

Eine einfache, aber grundlegende und für die vorliegende Arbeit wichtige Unterscheidung ist die zwischen Thema (Melodie), harmonischer Struktur und Rhythmus. Während das Thema die horizontale Komponente eines Stückes darstellt, also die Abfolge von Tönen, ist die Harmonik die vertikale Komponente, also der gleichzeitige Klang von Tönen. Die Rhythmik als dritte und ebenfalls horizontale Komponente beschreibt im Unterschied zum Thema keine Tonhöhen, sondern vielmehr Tonlängen. Da jeder dieser Komponenten eigene Mechanismen und Regeln zu Grunde liegen (vgl. [2], 120), lohnt sich eine getrennte Betrachtung, um die Korrelationen mit dem Erfolg eines Stückes zu finden.

Dynamik

Mit Dynamik bezeichnet man den Verlauf der Tonstärke, also die Differenzierung der Lautstärke, innerhalb eines Stückes. In der musikalischen Notationen werden Dynami-

kanweisungen in den acht Stufen *ppp*, *pp*, *p*, *mp*, *mf*, *f*, *ff*, *fff* (*piano pianissimo*, *pianissimo*, *piano*, *mezzo piano*, analog für *forte*) angegeben. Das MIDI-Datenformat, in dem der Datensatz dieser Arbeit vorliegt und das in Kapitel 2.2 näher beschrieben ist, bietet eine etwas feinere Unterteilung in 127 Stufen an.

Rhythmik & Takt

Ein Musikstück ist immer eine zeitlich koordinierte Abfolge von Tönen. Die Koordination betrifft dabei nicht nur eine einzelne Stimme, sondern das Zusammenspiel aller Stimmen. Aus diesem Grunde ist die Unterteilung eines Stückes in für Musiker intuitiv klare Einheiten nötig. Diese Einheit ist der *Grundschlag*. Mehrere Grundschläge werden zum nächstgrößeren Maß, dem *Takt*, zusammengefasst. Die *Taktart* gibt hierbei Aufschluss über die Beziehung zwischen Grundschlag und Takt. Sie ist immer als Bruch dargestellt und gibt im Zähler die Zahl der Grundschläge pro Takt sowie im Nenner den Notenwert eines Grundschlages an.

$$\frac{4}{4} \tag{2.1}$$

definiert einen Takt als zeitlichen Abschnitt mit der Länge von vier aufeinanderfolgenden Viertelnoten. Der numerische Wert der Taktart gibt also immer die Länge eines Taktes an. Deutlich wird das an der etwas selteneren Taktart

$$\frac{6}{8}, \tag{2.2}$$

die einem Takt die Länge von 6 Achtelnoten, also $0,75$ zuordnet.

Auf der Basis dieses Systems entstehen Rhythmusstimmen, die dem jeweiligen Stück ein *Metrum* geben, indem sie mehr oder weniger regelmäßig bestimmte Grundschläge betonen. Im einfachsten und in der populären Musik sehr verbreiteten Fall werden die Grundschläge 1 und 3 bzw. 2 und 4 betont. Dies setzt natürlich einen $\frac{4}{4}$-Takt voraus, dem ebenfalls häufigsten in der populären Musik. Erst auf Basis dieses Metrums, das dem Hörer Orientierung gibt, kann mit Methoden wie z.B. der *Synkope* ein Bruch der Regelmäßigkeit erfolgen, der ein beliebtes Kompositionsmittel ist, ein Stück für den Hörer interessant zu gestalten.

Die Synkope stellt die Betonung eines Schlages dar, der im Metrum eines Stückes eigentlich unbetont geblieben wäre. Eine Möglichkeiten, die Synkope zu realisieren, ist es, unbetonte Grundschläge zu betonen. Angenommen, das Metrum eines Stückes betont die Schläge 1 und 3. Dann beinhaltet der in Abb. 2.1 dargestellte Takt eine Synkope, da durch die (teilweise übergebundenen) halben Noten nur noch auf den Schlägen 2 und 4

Noten einsetzen. Die Schläge 1 und 3 werden dadurch nicht mehr betont. Eine alternative

Abbildung 2.1.: Notenbeispiel: Synkope auf Grundschlägen

Art der Synkope stellt eine Betonung von Schlägen zwischen den Grundschlägen dar. In Abb. 2.2 ist ein Notenbeispiel zu sehen, in dem Schlag 2 nicht betont wird. Zwischen Schlag 2 und 3 (der Musiker sagt: *auf 2 und*) beginnt dann eine halbe Note. Diese Note stellt die Synkope dar, da sie zwischen die Schläge fällt.

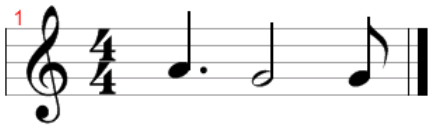

Abbildung 2.2.: Notenbeispiel: Synkope zwischen Grundschlägen

Aufbau eines Themas

Das Thema eines Stückes ist immer untergliedert in Teilabschnitte. Diese Untergliederung ist dabei kein künstliches Konstrukt, sondern entsteht aus dem Umstand, dass das menschliche Gehirn bei der Rezeption von Musik automatisch nach kleineren Sinnabschnitten sucht, die es - abgetrennt - leichter verarbeiten kann (vgl. [2], 163-164). Die kleinste thematische Einheit ist das Motiv. Die Länge eines solchen Motivs kann nicht eindeutig definiert werden. Ein Motiv ist immer das, was intuitiv als zusammengehörig empfunden wird. Das können auch schon zwei Töne sein. Ein guter Indikator für das Ende eines Motivs ist immer eine Pause. Ein Motiv zeichnet sich dadurch aus, dass es im Verlauf des Stückes immer wieder auftaucht und variiert wird. Es beinhaltet also eine Idee, die den musikalischen Charakter eines Stückes über dessen gesamten Verlauf profiliert.

Die nächstgrößere Einheit ist die Phrase. Sie fasst mehrere Motive zu einer Einheit zusammen.

Harmonik

Die Harmonik wurde in diesem Kapitel als die vertikale Komponente eines Stückes bezeichnet. Dies ist insoweit richtig, als sie das gleichzeitige Zusammenspiel von mehreren Tönen, den Akkord, betrifft. Auch die Harmonik hat allerdings eine horizontale Komponente, wenn es um die Abfolge von Akkorden geht.

Innerhalb eines Akkordes spielt das Tongeschlecht eine wesentliche Rolle. Unterschieden wird dort zwischen Dur- und Mollakkorden. Mit Durakkorden (von lat. *durus* „hart“) wird ein heller, klarer Höreindruck verbunden, mit Mollakkorden (von lat. *mollis* „weich“) ein eher dunkler und sanfter. Ein Durakkord ist aufgebaut aus einer großen Terz (= 4 Halbtonschritte) und einer kleinen Terz (= 3 Halbtonschritte), ausgehend vom Grundton. Die Reihenfolge ist beim Moll-Akkord genau umgekehrt. Abb. 2.3 zeigt einen C-Dur und einen c-Moll Akkord. Sie unterscheiden sich im mittleren Ton, der bei Moll einen Halbton tiefer liegt.

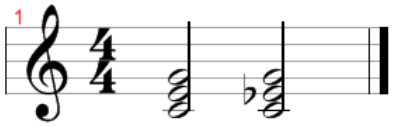

Abbildung 2.3.: Notenbeispiel: C-Dur und c-Moll Akkord

Wie bereits erwähnt, hat auch die Harmonik eine horizontale Komponente. So kann ein Tongeschlecht nicht nur für einen Akkord, sondern auch für eine Abfolge von Akkorden oder Tönen angegeben werden. Man unterscheidet bei Tonleitern ebenfalls zwischen Dur- und Molltonleitern. In einer Durtonleiter sind die Beziehungen zwischen zwei benachbarten Tönen genau festgelegt. Folgende Abfolge von Halbtonschritten findet sich in einer Durtonleiter: 2-2-1-2-2-2-1. Abb. 2.4 zeigt eine C-Dur Tonleiter.

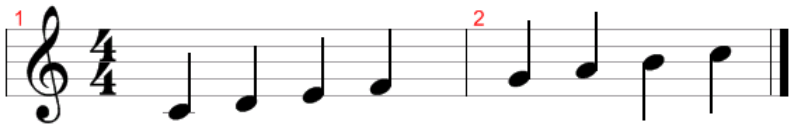

Abbildung 2.4.: Notenbeispiel: C-Dur Tonleiter

Von der Molltonleiter gibt es verschiedene Versionen. Natürliches Moll ist aus folgenden Halbtonschritten aufgebaut: 2-1-2-2-1-2-2. Um aber den sogenannten Leittoneffekt auf der 7. Stufe zu erzeugen, verwendet man harmonisches Moll. Ein Leitton ist ein

Ton, der beim Erklingen Spannung und Erwartung einer Auflösung im Folgeton erzeugt, der einen Halbton höher oder tiefer liegt. Dieses Phänomen tritt auf der 7. Stufe von Durtonleitern, nicht aber in natürlichem Moll auf. Harmonisches Moll mit dem Halbtonschrittschema 2-1-2-2-1-3-1 erzeugt den Leittoneffekt ebenfalls auf der 7. Stufe. Da der Leitton ein Mittel der Harmonieführung darstellt, lohnt bei entsprechenden Analysen eines Stückes die Betrachtung der harmonischen Molltonleiter besonders. Abb. 2.5 zeigt eine harmonische c-Moll Tonleiter.

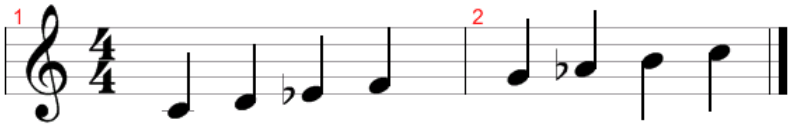

Abbildung 2.5.: Notenbeispiel: Harmonische c-Moll Tonleiter

2.2. MIDI

Das Musical Instrument Digital Interface, kurz MIDI, ist ein Protokoll, das ein Zusammenschluss aus Industrieunternehmen, die MIDI Manufacturers Association, entwickelt hat. Teil dieses Protokolls ist eine Schnittstellenbeschreibung für die Kommunikation zwischen Hard- und Software, die im musikalischen Bereich Verwendung findet. So steuern beispielsweise sogenannte Sequenzerprogramme digitale Instrumente wie Synthesizer oder Keyboards. Das Interface dient in erster Linie dazu, Geräte, die den MIDI-Standard unterstützen, zu synchronisieren und zu steuern. Die Steuerung geschieht durch ON- und OFF-Events, die jeweils Beginn und Ende einer Note darstellen. Gleichzeitig mitübertragen wird dabei die *velocity*, die die Lautstärke der Note angibt. Die Bezeichnung leitet sich vom Drücken der Tasten eines Klaviers ab. Je schneller eine Taste heruntergedrückt wird, je schneller also der Hammerkopf die Saite trifft, desto lauter erklingt die Note. Dieses simple Protokoll erlaubt eine Echtzeitsteuerung von Musikgeräten, beispielsweise durch eine Musiksoftware am Computer [8].

Das MIDI-Protokoll unterstützt neben den einfachen ON- und OFF-Events eine Reihe weiterer Funktionen, wie z.B. Zeitsynchronisation oder Steuerung von Effektgeräten und Pedalen. Der wesentliche Teil für diese Arbeit sind allerdings die ON- und OFF-Events. Neben der Echtzeitsteuerung hat sich MIDI auch zu einem Dateiformat entwickelt. Dies wird möglich, wenn man den Eventstrom nicht an ein Gerät sendet, sondern in einer Textdatei protokolliert. Speichert man die Events mit zusätzlichen Timestamps versehen

ab, kann man sie später wieder zur Kontrolle eines Abspielgerätes verwenden. Die Timestamps werden dabei immer relativ zu ihrem Vorgänger berechnet. Tabelle 2.1 zeigt dies beispielhaft. Die Spalte *velocity* gibt die bereits beschriebene Lautstärke der Note an, die zwischen 0 und 127 liegen kann. Ein ON-Event mit $velocity = 0$ ist dabei äquivalent zu einem OFF-Event. Die Spalte *pitch* gibt die Tonhöhe an, die ebenfalls im Intervall [0,127] liegt. Jeder Wert repräsentiert dabei eine Halbtonstufe. Der theoretisch mögliche Tonumfang beträgt somit fast 11 Oktaven. Alle Werte, die *modulo* 12 Null ergeben, werden der Note C zugeordnet, alle, die *modulo* 12 Eins ergeben, dem C#, usw. Das Beispiel würde einen gebrochenen C-Dur Dreiklang erklingen lassen, bei dem jede Note fünf Zeiteinheiten lang gehalten wird.

type	timestamp	velocity	pitch
ON	0	100	72
OFF	5	0	72
ON	0	100	76
ON	5	0	76
ON	0	100	79
OFF	5	0	79

Tabelle 2.1.: ON- und OFF-Events

MIDI Toolbox

Diese Speicherform ist so für die Berechnungen der Attribute allerdings nicht zu gebrauchen, da sie zu ineffizientem Suchen nach Noten zu einem gegebenen Zeitpunkt innerhalb eines Stücks führt. Sei on_i der Zeitpunkt des ON-Events der Note i und off_i der des korrespondierenden OFF-Events. Besteht ein Track nun aus n Noten, also $2n$ Events, müssten für die Suche nach allen zu einem Zeitpunkt t aktiven Noten alle $2n$ Events betrachtet werden. *Aktiv* meint in diesem Zusammenhang, dass $on_i < t < off_i$ gilt. Speichert man hingegen die Noten als Tupel $(on_i, length_i)$, wobei $length_i$ der Dauer der Note i entspricht, muss man, um die für den Zeitpunkt t aktiven Noten zu bestimmen nur alle $(on_i, length_i)$ mit $on_i < t$ betrachten, im Durchschnitt also bloß $n/2$.

Das MIDI-Framework von Eerola *et al.* [9] leistet dieses Parsing bereits und konvertiert die Midi-Events in eine *notematrix*, deren Aufbau in Abb. 2.6 zu sehen ist. Sie enthält für jede Note eine Struktur, die Notenhöhe (*pitch*), Intensität (*velocity*), Startzeitpunkt (*on*) und Länge (*dur*) in Sekunden und in Schlägen $(onBeat, durBeat)$ sowie den Kanal (*channel*) speichert. Letzterer dient zur Unterscheidung der verschiedenen Stimmen eines Stückes.

$$\begin{pmatrix} onBeat & durBeat & channel & pitch & velocity & on & dur \\ 1 & 0.5 & 8 & 60 & 110 & 0 & 0.25 \\ 2 & 0.5 & 4 & 64 & 90 & 0.5 & 0.25 \\ 2.5 & ... & & & & & \\ ... & & & & & & \end{pmatrix}$$

Abbildung 2.6.: Matrixstruktur der MIDI-Daten

Algorithmus von Krumhansl-Kessler

Ein weiterer wichtiger Bestandteil des MIDI-Frameworks [9] ist die Funktion *kkkey* zur Berechnung der Tonart eines Stückes. Die Funktion implementiert den Algorithmus von Krumhansl-Kessler [10], der auf Forschungsergebnissen [11] derselben beruht.

Der Algorithmus stellt zunächst eine Tonhierarchie eines Stückes auf. In dieser Tonhierarchie erhält ein Ton einen höheren Wert, je länger er innerhalb des Stückes gespielt wird. Diese Hierarchie wird als 12-dimensionaler Vektor repräsentiert, in dem entsprechend der 12 Töne einer Oktave jeder Ton einen Wert erhält, der angibt, wie stark er im Stück vorhanden ist (vgl. [10], 78). Des Weiteren existieren als Ergebnis der Untersuchungen [11] Referenzvektoren für alle 24 Tonarten. Diese Vektoren wurden auf Stücken der westeuropäischen Musik trainiert, für die die Tonart bekannt ist. Ein Beispiel geben Krumhansl *et al.* für die Tonart C-Dur.

> „The vector for C major is (6.35, 2.23, 3.48, 2.33, 4.38, 4.09, 2.52, 5.19, 2.39, 3.66, 2.29, 2.88). By convention, the first number corresponds to the tone C, the second to C# (Db), and so on."
>
> ([10], 80)

Gegeben die Tonprofile für alle 24 Tonarten K_i und das Tonprofil des zu bestimmenden Stückes I, berechnet man die Korrelation zwischen I und K_i für $i = 1...24$. Es resultieren 24 Korrelationswerte r_i. Die Tonart des zu bestimmenden Stückes ist nun genau dasjenige i, für das r_i maximal, K_i also am stärksten mit I korreliert ist (vgl. [10], 80).

2.3. Hit Song Science

Die Erfolgsvorhersage von Songs, die *Hit Song Science* hat sich in der Wissenschaft als Spezialdisziplin des Music Information Retrieval (MRI), das wiederum eine Form des Data Minings ist, etabliert. Li *et al.* fragen:

> „Can someone predict whether your recently produced song will become a hit?“
>
> ([1], 306)

Und sie geben die Antwort gleich selbst.

> „Any pop song composer would probably laugh at this question and respond: How could someone predict the success of what took so much craft, pain, and immeasurable creativity to produce?“
>
> ([1], 306)

Der Musikgeschmack jedes Menschen ergibt sich aus seiner persönlichen Vorgeschichte und seinen Empfindungen (vgl. [1], 306). Die Gesetzmäßigkeiten, die dort gelten, sind sehr komplex und unerforscht. Letztlich lässt sich das Vorhaben der Erfolgsvorhersage auf die Frage zuspitzen: Kann man den Geschmack des Publikums, so individuell er für jeden einzelnen ist, auf ein allgemeines Maß bringen? Li *et. al* sehen die Hauptaufgabe im Finden der richtigen Attribute:

> „[...] the goal ist to understand better the relation between intrinsic characteristics of songs [...] and their popularity, regardless of the complex and poorly understood mechanisms of human appreciation and social pressure at work.“
>
> ([1], 311)

Darüberhinaus bringt schon die Definition des Erfolgs Schwierigkeiten mit sich. Die Qualität oder das *Ohrwurmpotential* von Musik sind subjektive Eindrücke. Ein Stück, das den Einen tief berührt, lässt einen Anderen völlig kalt. Hinzu kommen soziale Effekte, die bei der Rezeption von Musik einen nicht zu vernachlässigenden Einfluss haben. Dies untersuchten Sagalnik *et al.* mit einem Experiment, in dem sie einen künstlichen Kulturmarkt von 14.341 Personen schufen, die kein Vorwissen über die auf dem Markt zur Verfügung stehenden Musikstücke besaßen [7]. Durch die gezielte Bereitstellung oder Verweigerung von Informationen über die Bewertungen von anderen Marktteilnehmern untersuchten sie den Zusammenhang zwischen Qualität und Popularität der Songs und kamen zu dem Schluss:

> „Increasing the strength of social influence increased both inequality and unpredictability of success.“

([7], 854)

Es stellt sich also neben der Frage der richtigen Attribute ebenso die Frage nach dem richtigen Label.

Im Folgenden sollen zwei Ansätze vorgestellt werden, durch die musikalische Attribute auf gänzlich unterschiedliche Art und Weise gefunden werden. Im Unterschied zur vorliegenden Arbeit, in der die Attribute aus MIDI-Daten extrahiert werden, machen diese Ansätze Audiodaten und soziale Metadaten zur Grundlage ihrer Vorhersage.

2.3.1. Audiodatenbasierte Ansätze

Li *et al.* beschreiben an einem weniger komplexen Sachverhalt die Möglichkeiten der Generierung musikalischer Audiofeatures (vgl. [1], 312 f.). Das Untersuchungsgebiet stellt dabei nicht die menschliche Musik, sondern Vogelgesang dar. Dies bietet zum einen den Vorteil einer weniger komplexen Musik, die somit einfacher vermessbar wird. Zum Anderen wird angenommen, dass der Einfluss sozialen Drucks, der im menschlichen Bereich große Auswirkung auf Erfolg und Popularität hat, geringer ist. Die Popularität von Vogelgesang wird in der Attraktivität gegenüber Weibchen gemessen. Viele Männchen produzieren den Gesang gerade zum Zweck der Partnersuche. Ein *Hit* ist in diesem Zusammenhang also ein Gesang, der ein Weibchen anzieht. Die Attribute des Vogelgesangs müssen ganz zwangsläufig aus Audiodaten generiert werden, denn von Vogelsongs stehen weder MIDI-Dateien, noch soziale Metadaten zur Verfügung. In mehreren Arbeiten zu diesem Thema haben sich zwei Attribute als besonders aussagekräftig herausgestellt: Die Trillerrate, also die Zahl der Silben pro Sekunde und die Frequenz des Gesangs in Hz, also die Tonhöhe. Je höher beide Attribute, desto attraktiver wirken die Gesänge. Jedoch stehen sie im Gegensatz zueinander: Es ist für den Vogel also schwer, beide zu maximieren. Deshalb stellt natürlicher Vogelgesang immer einen Kompromiss dar. Künstlicher Gesang hingegen kann beide Attribute maximieren. Abb. 2.7 zeigt die natürlichen Gesänge in schwarz, die künstlichen in grau. Tatasächlich hat sich herausgestellt, dass die optimierten Gesänge am attraktivsten auf Weibchen wirken.

Dieser Ansatz zeigt, wie prinzipiell aus Audiodaten Attribute gewonnen werden. Li *et al.* beschreiben, dass im Bereich der menschlichen Musik auf Basis audiobasierter Attribute Modelle generiert werden können, die signifikant besser vorhersagen als zufällige Modelle. Die Zuhilfenahme von Songtexten hat dabei keinen positiven Effekt auf die Leistung des Modells (vgl. [1], 314).

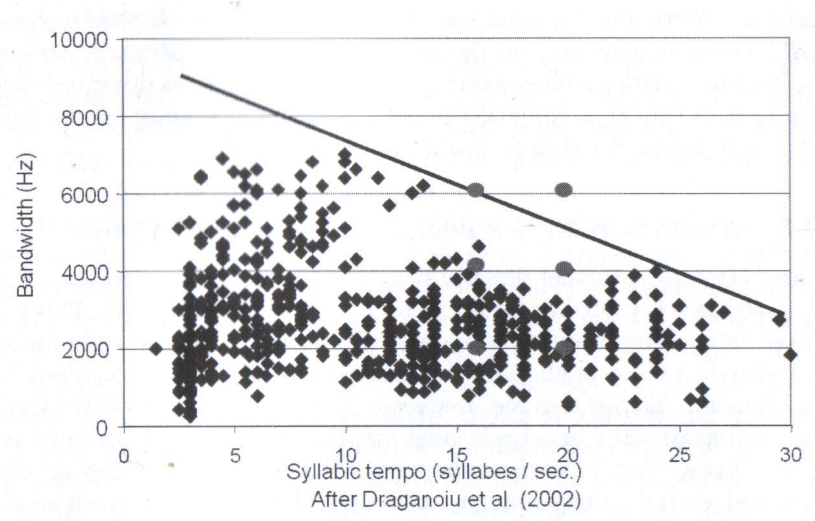

Abbildung 2.7.: Natürliche und künstliche Vogelgesänge nach Trillerrate und Frequenz ([1], 313)

2.3.2. Metadatenbasierte Ansätze

Ein interessanter Ansatz, um im musikalischen, aber auch in anderen Bereichen Vorhersagen zu treffen, ist das Projekt PredictWise von Microsoft Research Ökonom David Rothschild. Barbara Steiger schildert in einem Artikel [3], wie Rothschild Großbefragungen unter Xbox- oder Glücksspiel-Website-Nutzern durchführt, um in sehr hohem Maße Daten zu generieren. PredictWise hat unter anderem Vorhersageerfolge bei der Präsidentschaftswahl der USA 2012 und bei der Verleihung der Academy Awards 2013 erzielt. Im musikalischen Zusammenhang erweist sich das Projekt als interessant, da es erfolgreich den Gewinner des Eurovision Song Contests 2013, Dänemark, vorhergesagt hat. Dazu wurden nicht nur Befragungen durchgeführt, sondern auch soziale Kanäle ausgewertet. Beispielweise hatte die Zahl der Youtube-Views starken Einfluss auf die Reihenfolge der 10 wahrscheinlichsten Gewinner. Ebenfalls ausgewertet wurden Twitter-Daten. Tab. 2.2 zeigt die Vorhersagen von PredictWise, Tab. 2.3 die tatsächlichen Platzierungen.

2.4. Data Mining

Mit den Methoden des Data Mining ist es möglich, aus einem großen Datensatz Zusammenhänge zu extrahieren, die eine Aussage über zukünftige Daten zulassen. Nisbet *et al.* definieren Data Mining wie folgt.

Land	Siegwahrscheinlichkeit	Land	Siegwahrscheinlichkeit
Denmark	41%	Montenegro	0.3%
Ukraine	11%	Armenia	0.2%
Norway	11%	Austria	0.2%
Russia	6%	Switzerland	0.2%
Azerbaijan	5%	Spain	0.2%
Italy	4%	Slovenia	0.2%
Georgia	3%	Romania	0.2%
Germany	3%	Belgium	0.2%
Netherlands	2%	Bulgaria	0.2%
Sweden	2%	Hungary	0.2%
San Marino	2%	Cyprus	0.2%
Finland	1%	Albania	0.1%
Great Britain	1%	Lithuania	0.1%
Rep. Of Ireland	1%	Estonia	0.1%
Greece	1%	Croatia	0.1%
France	0.6%	Israel	0.1%
Belarus	0.5%	FYR Macedonia	0.1%
Moldova	0.5%	Iceland	0.1%
Malta	0.5%	Latvia	0.1%
Serbia	0.4%		

Tabelle 2.2.: Vorhersage von Microsofts PredictWise für den Eurovision Song Contest 2013 [3]

> „Statistical analysis and data mining are two methods for simulating the unconscious operations that occur in the human brain to provide a rationale for decision making and actions.“
>
> ([12], xxiii)

Grundlage im Data Mining Prozess ist der *Datensatz*, also die Gesamtheit der zur Verfügung stehenden Daten. Ziel ist es, ein *Modell*, eine Datenrepräsentation zu finden, auf deren Grundlage ein Algorithmus die Aufgabe der Simulation und der Entscheidungsfindung übernehmen kann. Die Struktur des Datensatzes wird bestimmt durch die *Attribute*. Ein Attribut ist eine messbare Eigenschaft des realen Objektes, das der Datensatz repräsentiert. Der Datensatz besteht aus den Attributwerten seiner einzelnen Instanzen, *Samples* genannt. Eines der Attribute nimmt eine besondere Funktion ein. Das *Label*, die Zielvariable ist dasjenige Attribut, das auf Basis der übrigen vorhergesagt werden soll. Entsprechend steht es für die zu vorhersagenden Samples nicht zur Verfügung. Der Zusammenhang zwischen den Attributen und dem Label muss auf Basis

Platzierung	Land	Platzierung	Land
1.	Dänemark	14.	Schweden
2.	Aserbaidschan	15.	Georgien
3.	Ukraine	16.	Weißrussland
4.	Norwegen	17.	Island
5.	Russland	18.	Armenien
6.	Griechenland	19.	Großbritannien
7.	Italien	20.	Estland
8.	Malta	21.	Deutschland
9.	Niederlande	22.	Litauen
10.	Ungarn	23.	Frankreich
11.	Moldau	24.	Finnland
12.	Belgien	25.	Spanien
13.	Rumänien	26.	Irland

Tabelle 2.3.: Ergebnisse des Eurovision Song Contest 2013 [4]

des Datensatzes erkannt werden, so dass er für neue, unbekannte Daten vorhergesagt werden kann. Der Vorgang der Vorhersage nennt sich *Klassifikation*, sofern die Zielvariable diskret ist. Im Zuge des Lernprozesses muss darauf geachtet werden, *Overfitting* zu vermeiden. Dabei handelt es sich um eine Überanpassung des Modells. Die Vorhersage gelingt dann auf den Testdaten, auf neuen, unbekannten Daten hingegen weniger. Das Modell generalisiert also schlecht (vgl. [12], 279).

Typischerweise gliedert sich das Vorgehen beim Data Mining in drei Schritte.

Vorverarbeitung

Zunächst muss ein Datensatz durch die Sammlung von Samples und Messung von Attributwerten aufgestellt werden. Danach folgt die Verarbeitung des Datensatzes. Dabei werden die Attribute transformiert und selektiert, um die ideale Attributmenge für die weiteren Schritte zu finden. Dieser erste Schritt ist sehr wichtig für die Genauigkeit der Vorhersage.

Modellierung

Anhand der Attributstruktur und des vorliegenden Datensatzes muss nun eine sinnvolle Datenrepräsentation und ein Algorithmus gefunden werden, der das Label vorhersagt. Das Modell wird aus dem Datensatz generiert; dieses Vorgehen nennt sich Training.

Evaluation

Letztlich muss die Genauigkeit und Korrektheit des vorliegenden Modells bewertet werden. Das übliche Vorgehen ist, einen Teil des Datensatzes nicht ins Training einfliessen zu lassen und ihn stattdessen zur Evaluation zu verwenden. Dies hat den Vorteil, dass man Evaluation auf Daten betreibt, für die die Zielvariable schon bekannt ist, die aber nicht in die Erstellung des Modells eingegangen sind. Vergleicht man nun die reale und die vorhergesagte Zielvariable, so lässt sich festellen, ob der Algorithmus zuverlässig arbeitet und welche Fehlerquote er hat.

2.4.1. Maschinelles Lernen

Maschinelles Lernen ist als Teilmenge des Data Mining Prozesses zu verstehen. Die Algorithmen dieser Kategorie kommen in den Schritten der Vorverarbeitung und Modellierung zur Anwendung. Sie beschäftigen sich mit der Selektion und Tranformation von Attributmengen sowie mit dem Lernen eines Datenmodells. Im Folgenden werden die Methoden des Maschinellen Lernens vorgestellt, die im Rahmen dieser Arbeit Anwendung finden.

2.4.1.1. Korrelationsbasierte Attributselektion

Attributselektion ist ein wichtiger Schritt vor Erstellung eines Modells, da dieses im Allgemeinen eine bessere Grundlage für genaues Vorhersagen bietet, wenn der Datensatz aus wenigen Attributen aufgebaut ist. Da in jedem Datensatz die Gefahr besteht, dass Attribute redundante Informationen enthalten - beispielsweise, weil sie durch Parametrisierung derselben Messmethode entstanden - sind Methoden notwendig, um die für die Vorhersage nützlichsten Attribute aus der Menge aller auszuwählen.

Grundsätzlich bestehen zwei verschiedene Ansätze, Feature Selection zu betreiben.
1. Ansatz: In die Kategorie der *Ranker* fallen Methoden, die generelle statistische Merkmale der Attribute - wie z.B. die Korrelation mit dem Label - messen und sie danach bewerten (vgl. [12], 78). Eine willkürlich festzulegende Anzahl der bestbewerteten Attribute kann anschließend ausgewählt werden. Diese Methode bringt allerdings den Nachteil mit sich, dass sie Attribute isoliert betrachtet und nicht bewertet, ob z.B. redundante Informationen in mehreren Attributen enthalten sind.
2. Ansatz: Um dieses Problem zu lösen, existieren *Subset Evaluator*. Diese werten Teilmengen der Gesamtheit aller Attribute aus. Auf jeder Teilmenge wird testweise ein Vorhersagealgorithmus ausgeführt. Die Teilmenge wird nach der Genauigkeit der Vor-

hersage bewertet und die bestbewertete ist die ideale Attributmenge für die endgültige Vorhersage (vgl. [12], 82). Diese Methode ist gerade mit einer sehr großen Zahl an Attributen extrem rechenaufwändig. Manche Methoden werten daher nicht alle Teilmengen aus, können aber so auch kein optimales Ergebnis garantieren.

Da im Rahmen dieser Arbeit die korrelationsbasierte Selektionsmethode Anwendung findet, soll im folgenden ihre Funktionsweise erläutert werden. Das ausschlaggebende Kriterium für die Bewertung eines Attributes ist hier die Korrelation mit dem Label. Es handelt sich um eine reine Ranker-Methode, so dass keine Korrelation der Attribute untereinander in die Bewertung einfließt. Um die Korrelation zweier Attribute zu bestimmen, benötigt man zunächst die Kovarianzformel. Bezeichne A einen Attributvektor und L den Labelvektor, sowie n deren Dimension. Dann ist die Kovarianz definiert als

$$cov(A, L) = \frac{1}{n-1} \sum_{i=1}^{n} (A_i - avg(A))(L_i - avg(L)). \tag{2.3}$$

Die Korrelation von A mit L ergibt sich somit aus

$$cor(A, L) = \frac{cov(A, L)}{\sqrt{cov(A, A)}\sqrt{cov(L, L)}}. \tag{2.4}$$

Die Korrelation liegt im Intervall [-1,+1]. Eine negative Korrelation gibt dabei ein gegengleiches Verhalten beider Attribute an, eine positive Korrelation hingegen ein ähnliches Verhalten. Sind beide unabhängig voneinander, so ist die Korrelation gleich null (vgl. [13], 92 f.). Ein korrelationsbasierter Feature Selection Algorithmus wertet nun für jedes Attribut die Korrelation mit dem Label aus und sortiert die Attribute nach ihren Korrelationswerten. Eine ideale Teilmenge für die Erstellung des Modells liefert diese Methode auf die Weise noch nicht. Es muss zuätzlich sichergestellt werden, dass die Attribute in der Teilmenge nicht untereinander korreliert sind, also keine redundanten Informationen enthalten.

2.4.1.2. Naive Bayes Klassifikationsalgorithmus

Im Rahmen dieser Arbeit findet der Klassifikationsalgorithmus *Naive Bayes* Anwendung. Naive Bayes ist ein wahrscheinlichkeitsbasierter Algorithmus, der auf dem *Satz von Bayes* aufbaut und zusätzlich die Unabhängigkeit aller Attributvariablen annimmt. Diese naive und im Allgemeinen sicher falsche Annahme, die allerdings die Genauigkeit des Algorithmus' nicht einschränkt, begründet die Herkunft des Namens Naive Bayes

(vgl. [12], 256).

Betrachten werden soll zunächst der einfache Fall einer eindimensionalen Attributmenge. Dieses Attribut sei w. Der Klassenwert der Samples sei c. Gesucht ist also die Wahrscheinlichkeit

$$p(c|w) \tag{2.5}$$

für einen Klassenwert c, gegeben Attributwert w. Berechnet man diese Wahrscheinlichkeit für jeden möglichen Klassenwert, so fällt die Wahl des Klassifikationsalgorithmus auf das Label mit der höchsten Wahrscheinlichkeit. Nach dem Satz von Bayes (vgl. [13], 425) gilt:

$$p(c|w) = \frac{p(w|c) * p(c)}{p(w)} \tag{2.6}$$

$p(w)$ bleibt für alle c konstant, kann also vernachlässigt werden. Folglich muss nur

$$p(c|w) = p(w|c) * p(c) \tag{2.7}$$

für alle c betrachtet werden. Diese Werte lassen sich leicht aus dem Trainingsdatensatz ablesen. $p(c)$ kann durch die relative Häufigkeit des Klassenwertes c im Datensatz ausgedrückt werden. $p(w|c)$ ist die relative Häufigkeit des Attributwertes w innerhalb aller Samples mit Klassenwert c.

Ein Beispiel zeigt Tabelle 2.4. Berechnet werden soll das wahrscheinlichste Label für $w = 4$. Für $c = 1$ gilt

$$p(4|1) * p(1) = \frac{1}{3} * \frac{3}{5} = \frac{4}{15}. \tag{2.8}$$

Für $c = 0$ gilt

$$p(4|0) * p(0) = 1 * \frac{2}{5} = \frac{2}{5}. \tag{2.9}$$

$w = 4$ würde also dem Label 0 zugeordnet.

w	c
5	1
4	1
5	1
4	0
4	0

Tabelle 2.4.: Beispieldatensatz mit eindimensionaler Attributmenge

Nicht wesentlich komplizierter gestaltet sich nun der mehrdimensionale Fall. Jedes Sample setzt sich aus einer Menge von Attributwerten $w = w_1\ w_2\ ...\ w_m$ zusammen. Im

Allgemeinen weiß man nicht, ob diese Variablen abhängig voneinander sind. Da sie alle dazu dienen, dasselbe Label zu beschreiben, ist sogar eher zu vermuten, dass sie abhängig sind. Um die Berechnung des Naive Bayes Algorithmus' aber einfach zu halten, wird die Unabhängigkeitsannahme getroffen (vgl. [14], 185). Damit gilt

$$p(w|c) = p(w_1\ w_2\ ...\ w_m|c) = \prod_{i=1}^{m} p(w_i|c). \tag{2.10}$$

Dies erlaubt nun die Behandlung von mehrdimensionalen Attributmengen mit dem Naive Bayes Algorithmus, indem die Wahrscheinlichkeit für jeden Klassenwert c - gegeben sind die Attributwerte $w = w_1\ w_2\ ...\ w_m$ - mittels

$$p(c|w) = \prod_{i=1}^{m} p(w_i|c) * p(c) \tag{2.11}$$

bestimmt werden kann.

Die Unabhängigkeitsannahme bringt einen großen Vorteil mit sich. Ist die Dimensionalität des Datensatzes sehr hoch, versagen viele Modelle, die - wie z.B. Entscheidungsbäume - die Abhängigkeit der Variablen beachten. Naive Bayes kommt durch die unabhängige Betrachtung jedes Attributs auch mit hochdimensionalen Attributmengen zurecht. Nisbet *et al.* schreiben darüber:

> „In fact, the Naïve Bayesian Classifier technique is particularly suited when the number of variables (the *dimensionality* of the inputs) is high."
>
> ([12], 51)

2.4.1.3. Überwachte Diskretisierung

Das Ziel einer Diskretisierung ist die Transformation eines numerischen Attributs in ein diskretes. Dazu ist sogenanntes *binning* erforderlich, also die Einteilung des Wertespektrums des numerischen Attributs in Abschnitte - *bins* - und die Einordnung aller Werte in diese (vgl. [12], 73).

Beispielhaft sei ein Attribut mit einem Wertspektrum von $[0, 100]$ angenommen. Abb. 2.8 zeigt dieses Attribut und sieben durch Punkte markierte Attributwerte. Die Farbe symbolisiert dabei die Zugehörigkeit der Datenpunkte zu einer von zwei Klassen. Eine Möglichkeit, ein solches Attribut zu diskretisieren, besteht darin, das Spektrum in gleich große Abschnitte aufzuteilen. Die drei bins im Beispiel sind auf diese Weise entstanden

und decken die Intervalle $[0, 33\frac{1}{3}]$, $[33\frac{1}{3}, 66\frac{2}{3}]$ und $[66\frac{2}{3}, 100]$ ab. Die so vorgenommene Diskretisierung nennt sich unüberwacht, da sie eine Einteilung ohne Ansehen der Klassenwerte vornimmt. Im ersten bin wird ersichtlich, wozu das führt. Es befinden sich zwei Datenpunkte jeder Klasse in einem bin, obwohl die Einteilung geschickter hätte gewählt werden können.

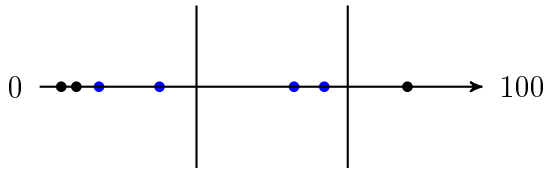

Abbildung 2.8.: Unüberwachte Diskretisierung eines numerischen Attributs

In Abgrenzung zur unüberwachten Diskretisierung gibt es die Möglichkeit, unter Beachtung der Klassenwerte - also überwacht - zu diskretisieren. Abb. 2.9 zeigt, wie das aussehen kann. Die bins sind nicht mehr gleich groß, sondern passen sich den Klassenwerten an. Durch das binning wurde eine optimale Aufteilung der Datenpunkte gefunden, da jeder bin nur noch diejenigen eines Klassenwertes enthält. In diesem Beispiel wäre eine Klassifizierungsregel sehr einfach gefunden, da jeder Datenpunkt im mittleren bin der blauen Klasse angehört, jeder übrige der schwarzen Klasse.

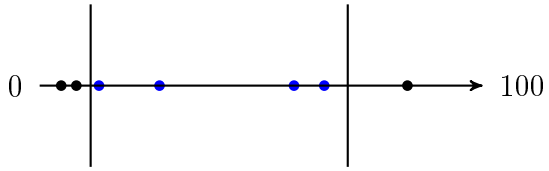

Abbildung 2.9.: Überwachte Diskretisierung eines numerischen Attributs

Algorithmen der überwachten Diskretisierung betrachten und minimieren die Entropie einer Aufteilung. Die Entropie einer Aufteilung gibt dabei die *Reinheit* der bins an. Die Entropie liegt niedrig, wenn viele Klassenwerte der dominanten Klasse im bin enthalten sind. Eine hohe Entropie hingegen liegt vor, wenn die Klassenwerte gleichmäßig verteilt im bin vorliegen. In Abb. 2.9 haben alle bins die niedrigstmögliche Entropie, da sie lediglich Klassenwerte einer Klasse enthalten.

Ein Algorithmus zur überwachten Diskretisierung kann diese Entropie nun heranziehen, um den Punkt zu finden, der ein numerisches Attribut optimal in zwei bins aufteilt.

Dazu wird die Entropie beider resultierender bins für jeden möglichen Spaltungspunkt berechnet und der ideale ausgewählt. Dieses Verfahren kann rekursiv auf beide bins angewandt werden.

Diskretisierung ist ein mächtiges Instrument, gerade um die Genauigkeit eines Naive Bayes Klassifizierers zu steigern. Der Naive Bayes Algorithmus berechnet, wie in Kapitel 2.4.1.2 beschrieben, Wahrscheinlichkeiten von Attributen, gegeben ein Klassenwert ($p(w|c)$). Durch Diskretisierung wird die Vielfalt der Attributwerte reduziert und die Zahl der Instanzen pro Attributwert erhöht. Dadurch werden die berechneten Wahrscheinlichkeiten verlässlicher.

2.4.1.4. Untersuchung von Zeitreihen

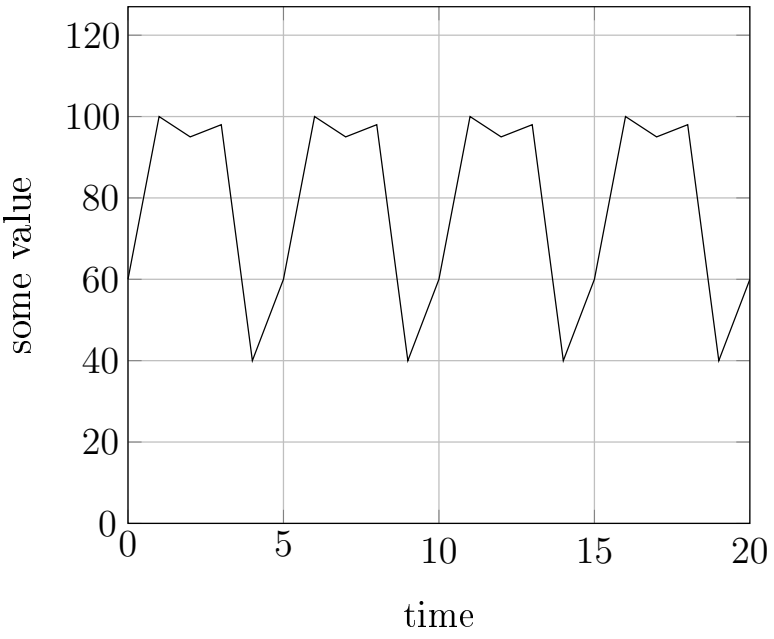

Abbildung 2.10.: Eine Zeitreihe mit saisonalem Anteil

Die mathematische Zeitreihenanalyse stellt einige Mittel bereit, die für die Analyse vom Musikstücken interessant sind. Aus Sicht des Data Mining liegt eine Zeitreihe vor, wenn die Indexvariable eines Datensatzes kontinuierlich steigt. Aus dem ganzen Methodenspektrum, das die Zeitreihenanalyse bietet, ist die Aufstellung eines Korrelogrammes (vgl. [15], 27) für die Musikanalyse besonders interessant. Ein Korrelogramm hilft bei der Identifikation saisonaler Anteile einer Zeitreihe. Grundlage hierfür ist die Definition der Korrelation in Kapitel 2.4.1.1.

Die Korrelation zweier Signale beschreibt, wie ähnlich sich beide verhalten. Eine hohe positive Korrelation gibt dabei ein ähnliches Verhalten an, eine hohe negative Korrelation ein gegengleiches Verhalten. Sind beide Signale unabhängig, liegt die Korrelation bei Null. Die Idee eines Korrelogramms ist es nun, die Korrelation für eine Zeitreihe und diejenige Reihe zu berechnen, die durch Verschiebung der Zeitreihe um ein bestimmtes Intervall entsteht. Dieses Maß bezeichnet man als Autokorrelation.

Enthält die Zeitreihe saisonale Anteile, wiederholt sich ihr Verhalten also in einem periodischen Intervall, dann ist die Autokorrelation in genau diesem Intervall hoch. Misst man die Autokorrelation für verschiedene Intervalle und trägt sie in einem Diagramm auf, nennt sich dies Korrelogramm. Abb. 2.10 zeigt ein Beispiel für eine Zeitreihe, die sich im Intervall 5 periodisch wiederholt. Entsprechend ist im Korrelogramm in Abb. 2.11 zu sehen, dass die Autokorrelationswerte für eine Verschiebung (*lag*) um 5 bzw. 10 Einheiten sehr hoch liegen. Für die Analyse saisonaler Anteile der Reihe ist klar, dass der stärkste saisonale Anteil im Intervall 5 auftritt.

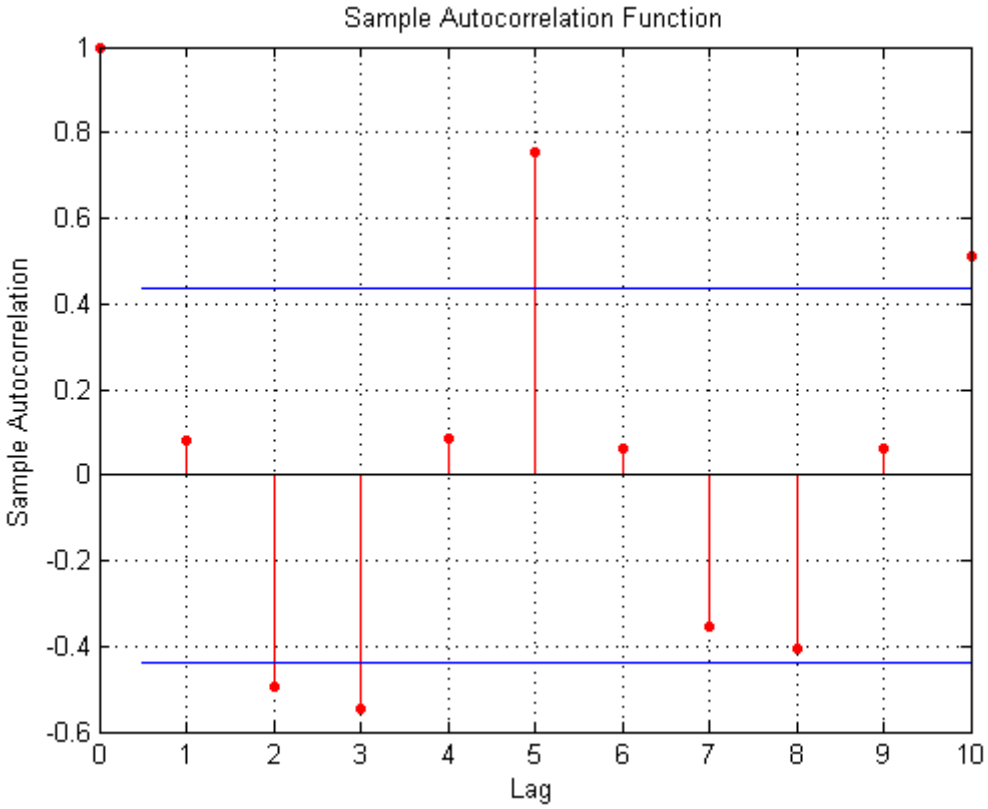

Abbildung 2.11.: Korrelogramm

Diese Untersuchungsmethode lässt sich für die musikalische Analyse verwenden. Ein

Musikstück ist ebenfalls eine Zeitreihe: Die Tonhöhe stellt den Wert der Reihe, das Metrum die Indexvariable dar. Komponisten arbeiten oft mit dem Element der Wiederholung und Variation. So werden Motive im Verlauf des Stückes wiederaufgegriffen und abgewandelt. Dabei spielt der Komponist mit der Erwartung des Hörers. Er weckt sie durch Wiederholung und bricht mit ihr durch plötzliche Variation. Nehmen die Wiederholungen allerdings einen zu großen Anteil ein, wirkt das Stück monoton. Dominieren die Variationen, kann der Hörer die Orientierung verlieren (vgl. [2], 120-123). Dieses Phänomen lässt sich somit sehr gut durch die Autokorrelation ausdrücken, stellt sie doch ein Maß dafür dar, wie stark saisonale Anteile sind. Berechnet man die Autokorrelation für Verschiebungen, die in der Musik gängig sind (2, 4, 8, 16 oder 32 Takte), kann sie ein gutes Indiz für den Grad der Wiederholung in einer Komposition sein.

2.4.2. WEKA Data Mining Software

Im Rahmen dieser Arbeit kommt das WEKA Data Mining Tool zum Einsatz, das an der Universität von Waikato, Neuseeland entwickelt wurde. Hall *et al.* beschreiben den Zweck der Software wie folgt.

> „The Waikato Environment for Knowledge Analysis (WEKA) came about through the perceived need for a unified workbench that would allow researchers easy access to state-of-the-art techniques in machine learning."
>
> ([16], 10)

Das Tool bietet für das gesamte Spektrum des Data Minings - von der Vorverarbeitung, Selektion und Klassifikation bis zur Evaluation - individuell anpassbare Implementierungen der Standardalgorithmen. Für die Algorithmen des maschinellen Lernens, die in Kapitel 2.4.1 beschrieben wurden, sowie die Evaluation in Kapitel 5 kommt im Rahmen dieser Arbeit die Implementierung in WEKA 3.7.10 [16] zum Einsatz.

3. Attribute

Im folgenden Kapitel werden auf Basis von musikalischen Überlegungen Attributideen zur Vermessung von Musikstücken entwickelt, mathematisch modelliert und in die zur Berechnung nötigen Algorithmen übersetzt. Die Daten, auf die in den Pseudocodes zugegriffen werden, beziehen sich auf die in Kapitel 2.2 beschriebenen Grundlagen von MIDI-Dateien sowie auf die Speicherform einer *notematrix* des MIDI-Frameworks [9]. Um Übersichtlichkeit zu gewährleisten, liefert ein direkter Zugriff auf eine Notenvariable (z.B. note[i]) in den Pseudocodes dieses Kapitels immer den *pitch* zurück. Alle anderen Eigenschaften einer Note werden explizit durch den Punktoperator (z.B. note[i].velocity) abgefragt.

3.1. Wahl des Labels

Da das Ziel dieser Arbeit eine Erfolgsvorhersage ist, gilt die erste Überlegung der Frage nach der Messbarkeit von Erfolg. Wie in Kapitel 2.3 beschrieben, ist diese Frage schwer zu beantworten, da messbare Qualität und Erfolg nicht stark korreliert sind, wenn soziale Effekte die Messung beeinflussen. Generell ist es aber sehr schwierig, soziale Effekte auszuschließen. Damit geht immer eine Reduzierung der Stichprobengröße einher, da die Messung unter kontrollierten Bedingungen stattfinden muss. Will man versuchen, einen Durchschnittsgeschmack des Musikhörers - sofern er existiert - zu treffen, muss eine breit erhobenes Erfolgsmaß herangezogen werden. Dadurch lassen sich fast automatisch soziale Effekte nicht ohne Weiteres ausschließen. Die Chartplatzierung, die ein frei zugängliches und in großem Umfang erhobenes Popularitätsmaß darstellt, bietet eine Möglichkeit, dies zu realisieren. Das binäre Label wird auf Basis der Chartpositionierungen der UK Single-Charts der Jahre 1952-2013 [17] erhoben. Ein *Hit* ist dabei ein Song, der mindestens eine Woche lang auf Position 1 der Charts gestanden hat; alle anderen werden der Kategorie *kein Hit* zugeordnet. Eine Liste der Chartpositionen aller Songs des Datensatzes findet sich im Anhang A.1.

3.2. Wahl der Attribute

3.2.1. Distanz und Autokorrelation

Li *et al.* beschreiben die Berechnung einer Ähnlichkeitsmatrix, der eine sehr grobe Repräsentation eines Songs zu Grunde liegt (vgl. [1], 56-58). Aus dem Audiosignal wird eine Kontur extrahiert, beispielweise um die Lautstärke oder Tonhöhe wiederzugeben. Dieses Signal wird dann zu einer Ähnlichkeitsmatrix verarbeitet, indem die Ähnlichkeit jedes Punktes mit jedem anderen in der Konturkurve berechnet wird.

Dieses Vorgehen bleibt, solange ein Audiosignal zu Grunde liegt, sehr oberflächlich. Liegen aber Midi-Daten zu Grunde, kann man dieses Verfahren benutzen, um eine sehr detaillierte Ähnlichkeitsberechnung durchzuführen. Diese greift an jedem Berechnungspunkt auf jede gespielte Note zu und nicht nur auf einen Gesamteindruck des Songs. Zudem kann die Berechnung nach Spuren aufgeschlüsselt erfolgen, beispielweise getrennt für das Thema, den Rhythmus und die harmonische Struktur. So kann eine größere Vielfalt an Attributen generiert werden, die sich im Nachhinein darauf untersuchen lässt, welche Aspekte den größten Einfluss auf das Label haben.

Abb. 3.1 zeigt ein Notenbeispiel: Die ersten 2 Takte aus dem Song *Had To Cry Today* von *Blind Faith*. Dabei handelt es sich um eine typische Ohrwurmmelodie, die sich in besonderem Maße für die folgende Analyse eignet. Im zweiten Takt wird nämlich das Motiv des ersten Taktes - etwas variiert und um ein bestimmtes Intervall nach unten verschoben - wiederholt. Dieses Kompositionsschema findet oft Verwendung, um Ohrwurmmelodien zu erzeugen (vgl. [2], 120-123). Der Hörer findet ein bereits bekanntes Motiv wieder, Erinnerungen werden geweckt, gleichzeitig erzeugt aber die leichte Abwandlung ein Überraschungsmoment, einen Bruch mit der Erwartung des Hörers.

Im Folgenden soll versucht werden, diese Parameter einer Komposition mathematisch zu erfassen. Die Variation eines Themas soll dabei über ihre Standardabweichung berechnet werden. Hinsichtlich der Frage, wie stark Motive eines Themas einander ähneln, kann die Autokorrelation Aufschluss geben.

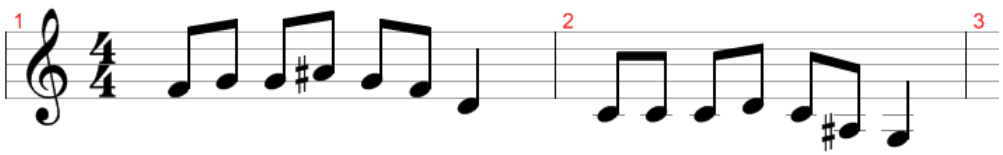

Abbildung 3.1.: Notenbeispiel: *Had To Cry Today* von *Blind Faith*

Distanz

Zunächst wird eine Distanzmatrix DM aufgestellt. Das Thema teilt man dazu in m gleichlange Intervalle auf, im Beispiel 3.1 ist $m = 16$. Diese Unterteilung bietet sich an, da die kürzesten vorkommenden Noten 8tel-Noten sind. Im Anschluss muss die Distanz jedes Intervalls zu jedem anderen mittels einer Distanzfunktion berechnet und an entsprechender Stelle in die Matrix eingetragen werden, so dass für $DM = (dm)_{i,j}$

$$dm_{i,j} = |diff_{ht}(i,j)| \tag{3.1}$$

gilt. Jeder Koeffizient $dm_{i,j}$ stellt dann den betraglichen Abstand in Halbtönen zwischen zwei Noten i und j dar. Eine komplexere Funktion ist für dieses Thema nicht notwendig, da nur einzelne Töne und keine Mengen von Tönen, wie sie beispielsweise in harmonischen Strukturen vorkämen, verglichen werden. Abb. 3.2 zeigt den Vergleich zweier Noten. Mit diesem Vorgehen stellt man, wie in Abb 3.3 gezeigt, eine $m \times m$-Distanzmatrix auf.

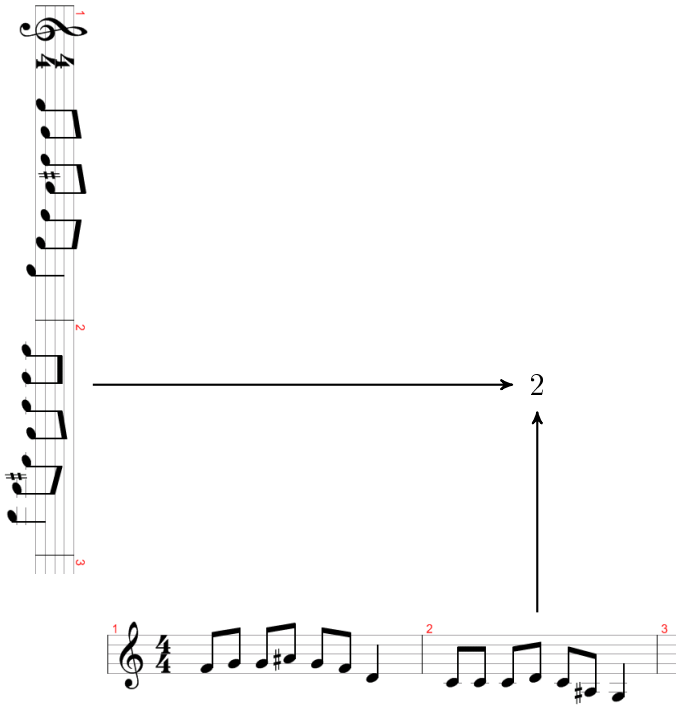

Abbildung 3.2.: Distanzberechnung am Beispiel *Had To Cry Today* von *Blind Faith* mit $dm_{i,j} = |diff_{ht}(i,j)|$

Auf Basis dieser Matrix lässt sich nun ein Distanzmaß D_x für eine Verschiebung um x

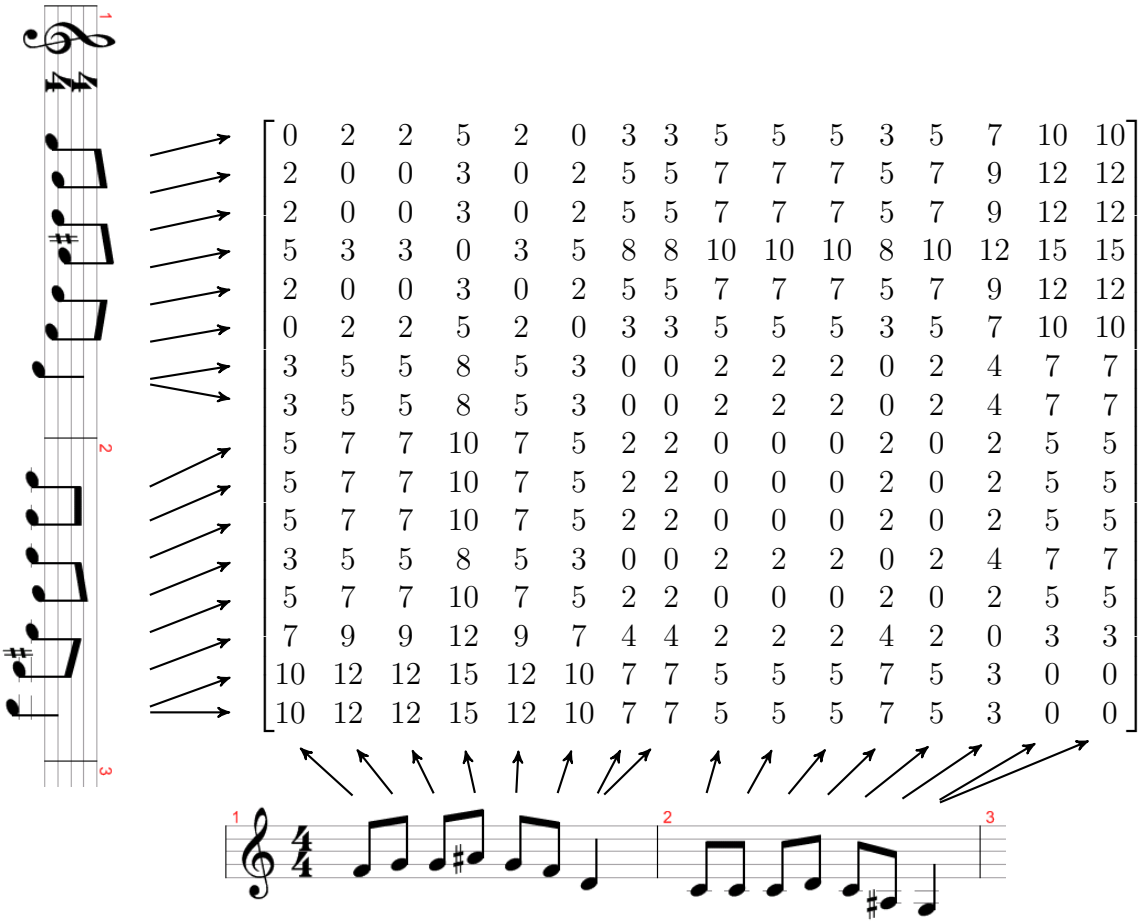

Abbildung 3.3.: Distanzmatrix für *Had To Cry Today* von *Blind Faith* mit $m = 16$ und $dm_{i,j} = |diff_{ht}(i,j)|$

Takte berechnen. Seien

- $t = \#Takte$,
- $b = \frac{m}{t}$ (Intervalle pro Takt) und
- $c_x(j) = (j + bx) \mod m$,

dann sind die für die Berechnung relevanten Koeffizienten von DM alle

$$dm_{c_x(j),j} \tag{3.2}$$

für j = 0 ... (m-1). Aus diesen Koeffizienten

$$\begin{pmatrix} dm_{c_x(0),0} \\ ... \\ dm_{c_x(m-1),m-1} \end{pmatrix} \tag{3.3}$$

wird der Mittelwert

$$avg_x = \frac{1}{m} \sum_{j=0}^{m-1} dm_{c_x(j),j} \tag{3.4}$$

sowie die korrigierte Varianz

$$s_x^2 = \frac{1}{m-1} \sum_{j=0}^{m-1} \left(dm_{c_x(j),j} - avg_x \right)^2 \tag{3.5}$$

berechnet.
Das Distanzmaß ist gerade diese Wertepaar

$$D_x = (avg_x, s_x^2). \tag{3.6}$$

avg_x gibt das Intervall an, um das durchschnittlich verschoben wurde. Dominiert in einem Thema ein dem menschlichen Ohr nicht vertrautes Intervall, so ist dies in der Klassifikation an avg_x abzulesen, ebenso im Falle eines vertrauten Intervalls.

s_x^2 hingegen repräsentiert das Überraschungsmoment des Themas. Ist s_x^2 klein, so werden Motive oft um das gleiche Intervall verschoben. Das Thema wirkt langweilig. Ist s_x^2 etwas größer, enthält das Thema einige motivische Varianten, mithin könnte es interessant wirken. Ist s_x^2 nun sehr groß, haben die einzelnen Motive fast keine Ähnlichkeit mehr miteinander. Den Hörer könnte das verwirren und das Thema so schwerlich zum Ohrwurm werden. Jourdain sagt dazu:

> „Unser Gehirn kann der Musik jedoch keinen Sinn mehr entnehmen, wenn sie zu viele Schwankungen in der Tonhöhe aufweist."
>
> ([2], 92)

Zudem lassen sich diese Attribute durch die Variation von x sehr gut parametrisieren. $x = 1, 2, 4, 8, 16, 32$ sind gute Wahlen für Attribute, da musikalische Motive sich oft in genau diesen Taktabständen wiederholen.

$$\begin{bmatrix}
0 & 2 & 2 & 5 & 2 & 0 & 3 & 3 & \mathbf{5} & 5 & 5 & 3 & 5 & 7 & 10 & 10 \\
2 & 0 & 0 & 3 & 0 & 2 & 5 & 5 & 7 & \mathbf{7} & 7 & 5 & 7 & 9 & 12 & 12 \\
2 & 0 & 0 & 3 & 0 & 2 & 5 & 5 & 7 & 7 & \mathbf{7} & 5 & 7 & 9 & 12 & 12 \\
5 & 3 & 3 & 0 & 3 & 5 & 8 & 8 & 10 & 10 & 10 & \mathbf{8} & 10 & 12 & 15 & 15 \\
2 & 0 & 0 & 3 & 0 & 2 & 5 & 5 & 7 & 7 & 7 & 5 & \mathbf{7} & 9 & 12 & 12 \\
0 & 2 & 2 & 5 & 2 & 0 & 3 & 3 & 5 & 5 & 5 & 3 & 5 & \mathbf{7} & 10 & 10 \\
3 & 5 & 5 & 8 & 5 & 3 & 0 & 0 & 2 & 2 & 2 & 0 & 2 & 4 & \mathbf{7} & 7 \\
3 & 5 & 5 & 8 & 5 & 3 & 0 & 0 & 2 & 2 & 2 & 0 & 2 & 4 & 7 & \mathbf{7} \\
\mathbf{5} & 7 & 7 & 10 & 7 & 5 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 5 & 5 \\
5 & \mathbf{7} & 7 & 10 & 7 & 5 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 5 & 5 \\
5 & 7 & \mathbf{7} & 10 & 7 & 5 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 5 & 5 \\
3 & 5 & 5 & \mathbf{8} & 5 & 3 & 0 & 0 & 2 & 2 & 2 & 0 & 2 & 4 & 7 & 7 \\
5 & 7 & 7 & 10 & \mathbf{7} & 5 & 2 & 2 & 0 & 0 & 0 & 2 & 0 & 2 & 5 & 5 \\
7 & 9 & 9 & 12 & 9 & \mathbf{7} & 4 & 4 & 2 & 2 & 2 & 4 & 2 & 0 & 3 & 3 \\
10 & 12 & 12 & 15 & 12 & 10 & \mathbf{7} & 7 & 5 & 5 & 5 & 7 & 5 & 3 & 0 & 0 \\
10 & 12 & 12 & 15 & 12 & 10 & 7 & \mathbf{7} & 5 & 5 & 5 & 7 & 5 & 3 & 0 & 0
\end{bmatrix}$$

Abbildung 3.4.: Relevante Koeffizienten für D_1

Im Beispiel *Had To Cry Today* mit $x = 1$ nimmt D - berechnet aus der DM in Abb. 3.4 - den Wert $D_1 = (avg_1, s_1^2) = (6.875, 0.65)$ an. Da dieses Thema schon ein großes Ohrwurmpotenzial gezeigt hat, lassen sich die Werte leicht erklären. Die mittlere Verschiebung avg_1 entspricht mit ungefähr 7 Halbtönen der reinen Quinte, einem sehr konsonanten und somit angenehm zu hörenden Intervall. Die Abweichung s_1^2 liegt mit 0.65 recht niedrig und bietet eine gewisse motivische Variante, wenn auch keine große. Der Hörer kommt also in den Genuss der klassischen Ohrwurmkriterien, er entdeckt Bekanntes wieder, während gleichzeitig mit seinen Erwartungen gebrochen wird.

Autokorrelation

Ein weiteres Maß für den Zusammenhang einer Zeitreihe mit sich selbst - um ein bestimmtes Zeitintervall verschoben - ist die Autokorrelation. In Kapitel 2.4.1.4 wurde gezeigt, wie sie benutzt werden kann, um regelmäßig wiederkehrende Komponenten einer Zeitreihe zu identifizieren. Die Autokorrelation

$$ac(Y) = \frac{cov(Y)}{var(Y)} \tag{3.7}$$

mit der Kovarianzformel

$$cov(Y) = \frac{\sum_{i=0}^{m-(bx)-1}(Y_i - avg(Y))(Y_i^x - avg(Y^x))}{m-1} \tag{3.8}$$

wird hier berechnet für ein Signal Y (ein Thema) und dasjenige Signal Y^x, das aus Y durch Verschiebung um x Takte entstanden ist. Die Autokorrelation, durch die Varianz normiert und somit im Intervall $[-1, +1]$, zeigt bei

- positiven Werten ein ähnliches Verhalten,
- Werten um 0 ein unabhängiges Verhalten und
- negativen Werten ein gegengleiches Verhalten

beider Reihen an. Betrachtet wird als Signal in diesem Fall allerdings nicht das Thema selbst, sondern die Intervalle zwischen den Noten. Ein Thema wie beispielsweise in Abb 3.1 würde sonst (für $x = 1$) eine negative Autokorrelation liefern, da der erste Takt (fast) komplett über dem Mittelwert des Themas, der zweite nur darunter liegt. Die Analyse vermutet ein gegengleiches Signal (musikalisch: gespiegelt) und liefert einen negativen Wert. Betrachtet man hingegen bloß die Intervalle als Signal, fallen Unterschiede in der Tonlage weg und die Verhältnisse der Noten zu ihren Nachbarnoten nehmen einzig Einfluss auf den Autokorrelationswert.

Mit einer anderen Ähnlichkeitsfunktion ausgestattet, kann das obige Verfahren auch hier benutzt werden, um eine Autokovarianzmatrix ACM aufzustellen, aus der dann das durch die Verschiebung x parametrisierte Autokorrelationsattribut ac_x generiert wird. Es bietet sich für die Berechnung der Kovarianz an, den Mittelwert der Intervalle

	0.4	−1.6	1.4	1.4	0.4	1.4	−1.6	0.4	−1.6	−1.6	0.4	0.4	0.4	1.4	−1.6
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
−1.6	−0.64	2.56	−2.24	−2.24	−0.64	−2.24	2.56	−0.64	2.56	2.56	−0.64	−0.64	−0.64	−2.24	2.56
1.4	0.56	−2.24	1.96	1.96	0.56	1.96	−2.24	0.56	−2.24	−2.24	0.56	0.56	0.56	1.96	−2.24
1.4	0.56	−2.24	1.96	1.96	0.56	1.96	−2.24	0.56	−2.24	−2.24	0.56	0.56	0.56	1.96	−2.24
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
1.4	0.56	−2.24	1.96	1.96	0.56	1.96	−2.24	0.56	−2.24	−2.24	0.56	0.56	0.56	1.96	−2.24
−1.6	−0.64	2.56	−2.24	−2.24	−0.64	−2.24	2.56	−0.64	2.56	2.56	−0.64	−0.64	−0.64	−2.24	2.56
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
−1.6	−0.64	2.56	−2.24	−2.24	−0.64	−2.24	2.56	−0.64	2.56	2.56	−0.64	−0.64	−0.64	−2.24	2.56
−1.6	−0.64	2.56	−2.24	−2.24	−0.64	−2.24	2.56	−0.64	2.56	2.56	−0.64	−0.64	−0.64	−2.24	2.56
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
0.4	0.16	−0.64	0.56	0.56	0.16	0.56	−0.64	0.16	−0.64	−0.64	0.16	0.16	0.16	0.56	−0.64
1.4	0.56	−2.24	1.96	1.96	0.56	1.96	−2.24	0.56	−2.24	−2.24	0.56	0.56	0.56	1.96	−2.24
−1.6	−0.64	2.56	−2.24	−2.24	−0.64	−2.24	2.56	−0.64	2.56	2.56	−0.64	−0.64	−0.64	−2.24	2.56

Abbildung 3.5.: Autokovarianzmatrix für *Had To Cry Today* von *Blind Faith*

vor der Aufstellung der Matrix von den einzelnen Noten abzuziehen. Abb 3.5 zeigt die Matrix im bekannten Beispielfall *Had To Cry Today*. Auf den Achsen aufgetragen sind diesmal die numerischen Intervallwerte, abzüglich des Mittelwertes. Der Mittelwert des Intervallvektors

$$Y = \begin{pmatrix} 2 & 0 & 3 & 3 & 2 & 3 & 0 & 2 & 0 & 0 & 2 & 2 & 2 & 3 & 0 \end{pmatrix} \tag{3.9}$$

beträgt

$$avg(Y) = \frac{1}{m} \sum_{i=0}^{m-1} Y_i = 1.6, \tag{3.10}$$

die korrigierte Varianz

$$var(Y) = \frac{1}{m-1} \sum_{i=0}^{m-1} (Y_i - avg(Y))^2 = 1.5429. \tag{3.11}$$

Der um den Mittelwert bereinigte Vektor

$$\bar{Y} = \begin{pmatrix} 0.4 & -1.6 & 1.4 & 1.4 & 0.4 & 1.4 & -1.6 & 0.4 & -1.6 & -1.6 & 0.4 & 0.4 & 0.4 & 1.4 & -1.6 \end{pmatrix} \tag{3.12}$$

dient nun als Grundlage für die Matrix $ACM = (acm)_{i,j}$ mit

$$acm_{i,j} = \bar{Y}_i * \bar{Y}_j = (Y_i - avg(Y))(Y_j - avg(Y)). \tag{3.13}$$

Die Kovarianz für eine Verschiebung um x Takte berechnet sich aus den aus Gleichung 3.2 bekannten Koeffizienten $acm_{c_x(j),j}$ für j = 0 ... (m-(bx)-1).

$$cov_x(Y) = \frac{\sum_{j=0}^{m-(bx)-1} acm_{c_x(j),j}}{m-1} \tag{3.14}$$

So lässt sich insgesamt die Autokorrelation als

$$ac_x(Y) = \frac{cov_x(Y)}{var(Y)} \tag{3.15}$$

beschreiben, was im Beispiel *Had To Cry Today*

$$ac_1(Y) = \frac{cov_1(Y)}{var(Y)} \approx \frac{0.55143}{1.5429} \approx 0.3574 \tag{3.16}$$

liefert. Dieser Wert gibt eine nicht sehr starke Autokorrelation an (möglich ist ein Höchstwert von 1) und berücksichtig damit den zwar ähnlichen Tonverlauf in beiden Takten, aber auch die Unterschiede in den jeweils ersten Hälften der Takte.

Ähnlichkeitsmaß für Notenmengen

Ein Problem, das bei der Berechnung der Distanz- bzw. Autokovarianzmatrix zwangsläufig auftritt, ist der Vergleich zweier Mengen von Noten. Möchte man die Autokorrelation für Harmoniestimmen berechnen, muss die Ähnlichkeitsfunktion nicht nur einzelne Noten, sondern Akkorde oder auch Mengen von Noten ohne harmonischen Zusammenhang als Argumente akzeptieren können. Klassische Verfahren zur Beurteilung der Ähnlichkeit von Mengen reichen im musikalischen Zusammenhang jedoch nicht aus. Der Jaccard-Index, der die Ähnlichkeit zweier Mengen nach der Zahl ihrer gemeinsamen Elemente beurteilt, wird der musikalischen Komplexität genausowenig gerecht, wie die Ermittlung der paarweisen Distanz aller oder einiger relevanter Notenkombinationen mit anschließender Durchschnittsbildung (oder sonstiger Aggregation). Eine Aggregation einzelner

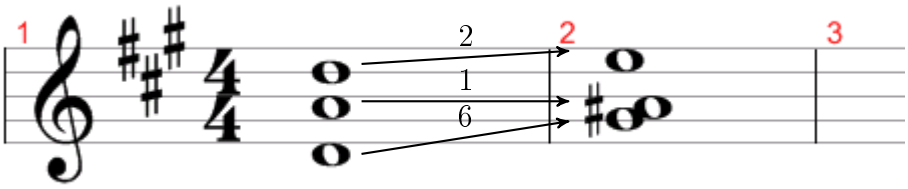

Abbildung 3.6.: Informationsverlust bei Aggregation

Ähnlichkeiten zu bilden, würde bedeuten, musikalischen Informationen zu verlieren. Die Zahl der Elemente, die aggregiert werden, ist zu gering, um einen tatsächlichen Trend erkennen zu können. Beispiel 3.6 zeigt, wie dies zu Informationsverlust bzw. sogar Verzerrung führen kann. Als elementare Ähnlichkeitsfunktion liegt hier wieder der Abstand in Halbtönen zu Grunde. Bildet man den Durchschnitt der drei Intervalle, erhält man $\frac{6+1+2}{3} = 3$ Halbtöne, die *kleine Terz*. Dieses Intervall ist beispielweise Bestandteil jedes Molldreiklangs und nicht ungewohnt zu Hören, ganz im Gegensatz zum Akkordübergang aus Abb. 3.6, der sehr dissonant klingt. Die Durchschnittsbildung hat in diesem Fall die vorliegenden harmonischen Verhältnisse vollkommen falsch wiedergegeben.

Ideal ist also, einen Repräsentanten für eine Notenmenge zu wählen, um diesen stellvertretend zur Ähnlichkeitsberechnung heranzuziehen. Der Grundton eines Akkordes bietet sich dafür an. In der MIDI Toolbox [9] wird ein Algorithmus zur Verfügung gestellt, der dies bereits leistet. Der Algorithmus liefert dabei auch Grundtöne für unvollständige Akkorde, indem er den wahrscheinlichsten Grundton angibt. Abb. 3.7 zeigt beispielhaft, wie zwei Akkorde mit dieser Methode verglichen werden. Als elementare Ähnlichkeitsfunktion dient wieder der Halbtonabstand. Berechnet wird er zwischen den beiden Grundtönen - im Beispiel sind es C und F, die beiden tiefsten Noten.

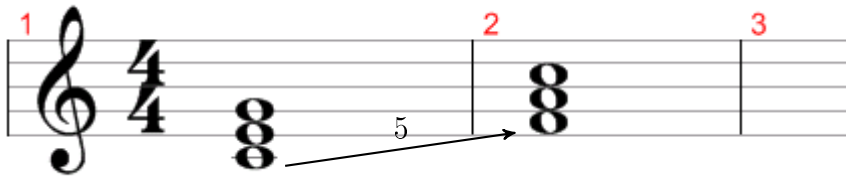

Abbildung 3.7.: Akkordähnlichkeit

Code 3.1 skizziert das beschriebene Verfahren zur Berechnung der Ähnlichkeit zweier Notenmengen und deckt dabei auch die Sonderfälle ab, in denen eine oder beide Mengen einelementig sind. Er ist dann ebenso auf Themen anwendbar. Neben den Notenmengen g und h wird eine Funktion f als Parameter übergeben, die eine elementare, auf Notenpaare anwendbare Ähnlichkeitsfunktion darstellt.

Code 3.1: Pseudocode: Ähnlichkeitsberechnung

```
1  similarity(g, h, f):
2  sim ← 0
3
4  //function kkkey [9] returns key of chord
5  //function map maps return value of kkkey to midi pitch value
6  if (size(g) = 1) do
7          val_g ← g
8  else do
9          val_g ← map(kkey(g))
10 end
11 if (size(h) = 1) do
12         val_h ← h
```

```
13 else do
14         val_h ← map(kkey(h))
15 end
16
17 sim ← f(val_g, val_h)
18
19 return sim
```

Algorithmus

Zunächst bedarf es eines Verfahrens, die *aktiven* Noten zu einem Zeitpunkt t zu ermitteln.

Code 3.2: Pseudocode: Aktive Noten

```
1 notes ← set of notes
2
3 seekActiveNotes(t, notes):
4 activeNotes ← ∅
5 i ← 0
6
7 while (notes[i].on ≤ t) do
8         if (notes[i].on ≤ t < notes[i].off) do
9                 activeNotes ← activeNotes ∪ notes[i]
10        end
11        i ← i + 1
12 end
13
14 return activeNotes
```

Auf Basis der Funktionen *seekActiveNotes* und *similarity* kann abhängig von der elementaren Ähnlichkeitsfunktion f eine der beiden Matrixvarianten berechnet werden. $f(x, y) = |diff_{ht}(x, y)|$ führt zur Ähnlichkeits-, $f(x, y) = (x - avg(X)) * (y - avg(Y))$ zur Autokovarianzmatrix.

Code 3.3: Pseudocode: Matrixberechnung

```
1 notes ← set of notes
2 m ← number of intervals
```

```
3 f ← some basic similarity function
4
5 cache(notes, m):
6 cache ← ∅
7
8 for k ← 0 to m−1, step max(notes.off)/m:
9         cache[k] ← seekActiveNotes(k, notes)
10 end
11
12 return cache
13
14 computeMatrix(cache, m, f):
15 a ← (a)_{m,m}
16
17 for i ← 0 to m−1:
18         for j ← 0 to m−1:
19                 a[i,j] ← similarity(cache[i], cache[j], f)
20         end
21 end
22
23 return a
```

Weiterhin ist zu beachten, dass bei der Aufstellung der Autokovarianzmatrix - wie bereits erwähnt - kein Notenvektor, sondern ein Intervallvektor zu Grunde liegt. Bei der Berechnung der Intervalle wird wieder die Funktion *similarity* herangezogen, die etwaige Akkorde auf ihre Grundtöne herunterbricht und die Intervalle zwischen diesen berechnet.

Letztendlich ist eine weitere Funktion nötig, die aus den Matrizen Attribute generiert.

Code 3.4: Pseudocode: Ähnlichkeits- und Autokorrelationsattribut

```
1 notes ← set of notes
2 m ← number of intervals
3
4 computeD(notes, m):
5
6 t ← number of bars
```

```
7  f ← |diff_ht(x, y)|
8  cache ← cache(notes, m)
9  SM ← computeMatrix(cache, m, f)
10
11 step ← [1/2 1 2 4 8 16 32]
12 coefficients ← ∅
13 attributes ← ∅
14 for x ← 0 to 6:
15         for j ← 0 to m−1:
16                 coefficients ←
17                 coefficients ∪ SM[j + m/t * step[x] mod m, j]
18         end
19         count ← size(coefficients)
20         avg_x ← 1/count sum(coefficients)
21         // subtraktion and exponentiation by element
22         s_x^2 ← 1/(count−1) sum((coefficients − avg_x)^2)
23         coefficients ← ∅
24         attributes ← attributes ∪ [avg_x, s_x^2]
25 end
26
27 return attributes
28
29 computeAC(notes, m):
30
31 t ← number of bars
32
33 cache ← cache(notes, m)
34 // turning vector of notes into vector of intervals
35 g ← |diff_ht(x, y)|
36 intervals ← ∅
37 for c ← 0 to m−1:
38         intervals[c] ← similarity(cache[c], cache[c+1], g)
39 end
40
41 avg ← sum(intervals)/size(intervals)
```

```
42 f ← (x − avg) ∗ (y − avg)
43
44 ACM ← computeMatrix(intervals, m, f)
45
46 step ← [1/2 1 2 4 8 16 32]
47 coefficients ← ∅
48 attributes ← ∅
49 avgInt ← 1/size(intervals) sum(intervals)
50 // subtraktion and exponentiation by element
51 var ← 1/(size(intervals)−1) sum((intervals − avg)^2)
52 for x ← 0 to 6:
53         for j ← 0 to m−(m∗step[x])/t−1:
54                 coefficients ←
55                 coefficients ∪ SM[j + m/t ∗ step[x] mod m, j]
56         end
57         count ← size(coefficients)
58         cov_x ← 1/(count−1) sum(coefficients)
59         ac_x ← cov_x/var
60         coefficients ← ∅
61         attributes ← attributes ∪ ac_x
62 end
63
64 return attributes
```

3.2.2. Rhythmik

Im Gegensatz zu z.B. traditionellen afrikanischen Musikstilen, in denen elaborierte, komplexe Rhythmen eine zentrale musikalische Rolle spielen, übernimmt die Rhythmik in der populären westlichen Musik eine eher unterstützende Funktion (vgl. [2], 159-161). Ein stetiger, gleichmäßiger Grundschlag begleitet ein Stück und betont dabei manche Schläge mehr, manche weniger. Es entsteht ein Metrum, das ein Stück in gleichgroße Zeitabschnitte aufteilt. Die populäre Musik hat dabei keine große Vielfalt an Varianten hervorgebracht. In den meisten Fällen begnügt sie sich mit $\frac{3}{4}$- oder $\frac{4}{4}$-Metren. Komplexere Metren wie $\frac{7}{4}$ oder gar häufige Wechsel zwischen Metren innerhalb eines Stückes sind seltener anzutreffen (vgl. [2], 166-167). Lediglich eine Andeutung komplexerer Rhyth-

mik hielt mit dem Konzept der *Synkope* Einzug in die populäre Musik, bei dem „Schläge außerhalb der normalen Zählzeit mit einem Akzent versehen werden“ ([2], 169).

Da all diese Konzepte in der populären Musik weit verbreitet sind und wenig variiert werden, lohnt es kaum, die Ausgestaltung des Metrums einer genauen Analyse zu unterziehen. Interessant ist hingegen die Frage, inwieweit die Abwesenheit eines solchen Metrums zur Irritation des Hörers führen und somit die Erfolgschance eines Stückes beeinflussen kann. Jourdain sagt dazu:

> „Wenn das Gehirn ein Gefühl für den Schlag entwickelt hat, nimmt es ihn auch dann vorweg, wenn einzelne Schläge unhörbar sind oder auf lang anhaltende Noten fallen. Dennoch muss eine Abfolge von Schlägen immer von neuem aufgefrischt werden, damit ihre Antizipation durch das Gehirn nicht nachläßt. Genauso wie die Harmonik einer andauernden Bestätigung des tonalen Zentrums bedarf, braucht der Rhythmus eine beständige Wiederholung des Grundschlages. Ein Ausfall von nur wenigen Sekunden kann den Hörer ziemlich verwirren.“
>
> ([2], 165)

Ziel eines rhythmischen Attributes muss es also sein, zu messen, wie stark die Grundschläge eines Stückes betont werden. Dazu kann man das Verhältnis der Noten, die auf die Grundschläge eines Stückes fallen, zur Zahl aller Grundschläge berechnen, bzw. anstatt der bloßen Anzahl der Noten ihre Lautstärke - also den Grad ihrer Betonung - heranziehen.

Betonungsverhältnis

Wie aus Kapitel 2.2 bekannt, speichert die Midi-Datenstruktur die *velocity*, also die Intensität jeder Note ab. Dieser diskrete Wert liegt in einem Bereich von $[0, 127]$. Zusätzlich ist für jedes Stück die Taktart $T = \frac{z}{n} \in \{\frac{3}{4}, \frac{4}{4}, \frac{2}{2}, ...\}$ sowie die Länge q einer Viertelnote in Mikrosekunden angegeben. Ein Stück der Länge t (in Mikrosekunden) besitzt also eine maximale Gesamtintensität I_{max} auf den Grundschlägen von

$$I_{max} = \frac{t * n}{q * 4} * 127. \tag{3.17}$$

Die Zahl der Viertelnoten mit $\frac{n}{4}$ zu multiplizieren ist notwendig, da zwei Taktarten, die im mathematischen Sinne gleich sind, andere Betonungsmuster vorgeben können. $\frac{2}{2} = \frac{4}{4}$

gilt zwar, und somit enthalten Takte beider Taktarten Noten gleicher Gesamtlänge, aber im $\frac{2}{2}$-Takt werden nur die Zählzeiten 1 und 3, also zwei halbe Noten betont, im $\frac{4}{4}$-Takt hingegen vier Viertelnoten, die Zählzeiten 1, 2, 3 und 4.
Angenommen, die Funktion $vel(x)$ liefert die *velocity* der auf den Zeitpunkt x fallenden Note, dann bezeichnet

$$I = \sum_{i=1}^{\frac{t*n}{q*4}} vel(i * q * \frac{4}{n}) \tag{3.18}$$

die Intensität auf den Grundschlägen. Um tatsächliche Betonungen von Grundschlägen von den restlichen Noten zu unterscheiden, soll *vel* nur diejenigen betrachten, die genau auf dem Zeitpunkt x einsetzen und diejenigen, die davor beginnen und danach enden, ignorieren.

$$R = \frac{I}{I_{max}} \tag{3.19}$$

gibt dann gerade den Grad der Betonung auf den Grundschlägen an.

Zusätzlich zu diesem allgemeinen rhythmischen Maß bietet es sich an, die Häufigkeit der unbetonten Grundschläge zu untersuchen. Beispielsweise zählt man dazu alle Grundschläge mit einer *velocity* unter einem bestimmten Grenzwert $p \leq 127$ und setzt diese in Relation zur Gesamtzahl der Grundschläge $\frac{t*n}{q*4}$. Damit stellt man sicher, dass unbetonte Stellen im Stück stärker ins Gewicht fallen und verhindert, dass z.B. viele stark betonte Schläge wenige unbetonte ausgleichen. Mit

$$threshold(x) = \begin{cases} 1 & \text{falls } vel(x) < p \\ 0 & \text{sonst} \end{cases} \tag{3.20}$$

erhält man so das Maß

$$R = \sum_{i=1}^{\frac{t*n}{q*4}} threshold(i * q * \frac{4}{n}) * \frac{q*4}{t*n}. \tag{3.21}$$

Ein weiterer Vorteil liegt in der Parametrisierbarkeit dieses Attributs. Durch Wahl verschiedener Grenzwerte p lässt sich eine Vielzahl von Attributen generieren, aus denen bei der Klassifikation das am stärksten mit dem Label korrelierte ausgewählt werden kann.

Algorithmus

Der Algorithmus zur Berechnung des Betonungsverhältnisses R fordert ein Verfahren zur Definition der *velocity* von Notenmengen. Die Definition von R erfolgte zunächst mit einer Funktion $vel(x)$, die voraussetzt, dass zu jedem Zeitpunkt x nur eine Note aktiv ist. Sind gleichzeitig mehrere Noten aktiv, bietet es sich an, den maximalen *velocity*-Wert zurückzuliefern, da solche mit niedrigerem Wert im Zusammenspiel eher untergehen.

Code 3.5: Pseudocode: Betonungsverhältnis

```
1  notes ← set of notes
2  T = z/n ← time signature
3  q ← duration of a quarter note
4  t ← overall duration
5  p ← some threshold
6
7  ratioOfAccentuation(notes, T, q, t, p):
8  velocity ← [0 ... 0]
9  count ← 0
10
11 for i ← 0 to size(notes)-1:
12         if (notes[i].on mod (q*4/n) = 0) do
13                 idx ← notes[i].on / (q*4/n)
14                 if (notes[i].velocity > velocity[idx]) do
15                         velocity[idx] ← notes[i].velocity
16                 end
17                 if (notes[i].velocity < p) do
18                         count ← count + 1
19                 end
20         end
21 end
22
23 I_1 = sum(velocity)
24 I_2 = count
25 return [I_1*q*4/(t*n*127), I_2*q*4/(t*n)]
```

3.2.3. Dynamik

Altenmüller *et al.* untersuchten in [18] mittels eines Versuchs, bei dem körperliche Reaktionen von Musikrezipienten beim Hören gemessen wurden, welche musikalischen Erlebnisse das Nervensystem besonders reizen. Das Experiment ergab, dass - neben einigen vom Rezipienten abhängigen Kriterien - die starke Zunahme der Dynamik - also plötzliches Anschwellen - einen zwar „fast plumpen“, aber dennoch sehr wirksamen „Weg“ ([18], 63) darstellt, um „Gänsehaut“ ([18], 63) beim Hörer zu verursachen. Wenn also ein plötzlicher Anstieg der Lautstärke ein Weg ist, den Hörer eines Stückes emotional zu beeinflussen, muss es das Ziel eines Dynamikattributs sein, eben solche Stellen in einem Stück zu identifizieren und ihre Häufigkeit und Intensität zu messen.

Dynamikgradient

Voraussetzung für die Berechnung des Attributs ist eine Funktion $gradient(x, y)$, die für zwei Zeitpunkte x und y in einem Stück der Länge t (mit $x < y \leq t$) die Lautstärkesteigung von x nach y berechnet. Dazu sei die Funktion $vel(x)$ diejenige, die die maximale *velocity* aller zum Zeitpunkt x *aktiven* Noten zurückgibt.

$$gradient(x, y) = \frac{vel(y) - vel(x)}{y - x} \tag{3.22}$$

Mit Hilfe dieser Funktion und einem festen Abstand $d = y - x$ wird nun das Stück in Intervalle unterteilt, auf denen sich jeweils der Gradient berechnen lässt. Diese Unterteilung lässt gewisse Unebenheiten im Dynamikverlauf verschwinden und sorgt so für ein geglättetes Signal. Man stellt also die endliche Folge der Gradienten

$$\begin{aligned}(f_n) &= gradient(d * n, d * n + d)\ \forall\ n = 0\ ...\ \frac{t-d}{d} \\ &= (gradient(0, d), ..., gradient(t - d, t))\end{aligned} \tag{3.23}$$

für ein Stück der Länge t auf. Mithilfe dieser Folge lassen sich nun Stellen plötzlicher Lautstärkezunahme identifizieren. Dies sind zusammenhängende Teilfolgen von f_n, für die - für ein festes p - gilt:

$$\exists\ 0 \leq N \leq \frac{t-d}{d} : (\forall\ n \leq N : f_n \leq p) \wedge (\forall\ n > N : f_n > p) \tag{3.24}$$

Es reicht in diesem Fall nicht, bloß nach besonders hohen Gradienten zu suchen, um Stellen plötzlicher Lautstärkezunahme zu finden. Damit die Wendung für den Hörer

überraschend kommt, muss er sich an ein dynamisches Niveau gewöhnt haben. Die zusätzliche Untersuchung der dem hohen Gradienten vorausgehenden Folgenglieder und die Zusatzbedingung, dass sie unter einem bestimmten p liegen müssen, stellt dies sicher.

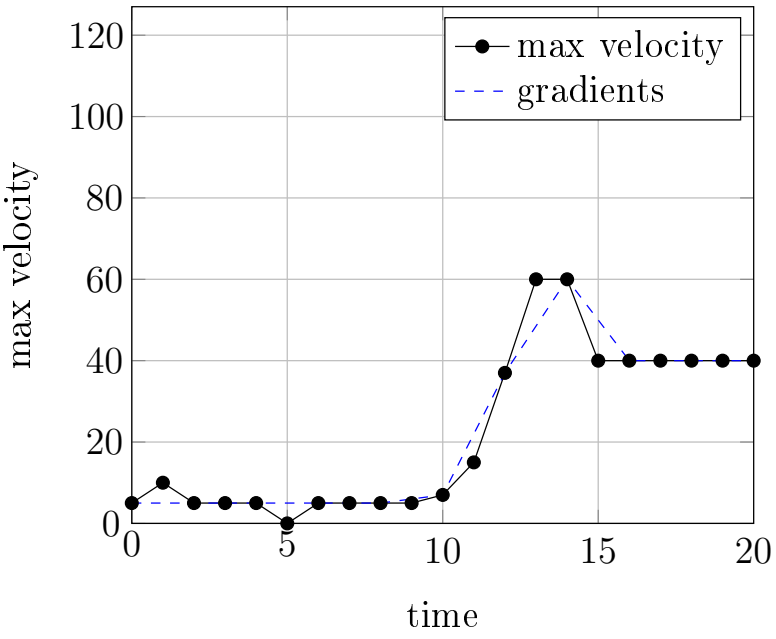

Abbildung 3.8.: Beispiel eines Dynamikverlaufs mit Gradienten für $d = 2$

Abb. 3.8 zeigt die beispielhafte Untersuchung einer zwanzig Zeiteinheiten langen Sequenz. In schwarz ist die *velocity* eingetragen, in blau die Steigungsgeraden zwischen jeweils zwei Punkten mit Abstand $d = 2$. Eine Analyse dieser Werte ergibt die zehn in Abb. 3.9 gezeigten Gradienten. Für $p = 5$, durch die rote Linie gekennzeichnet, liefert das Kriterium 3.24 genau eine Sequenz plötzlicher Lautstärkezunahme, nämlich die ersten sieben Gradienten. Das entspricht dem Zeitabschnitt bis zur 14. Zeiteinheit in der ursprünglichen Sequenz.

Algorithmus

Der folgende Algorithmus wendet das beschrieben Verfahren auf ein Stück an und identifiziert dabei alle Sequenzen großer Lautstärkezunahme. Die Hilfsfunktionen *gradient* und *velocity* entsprechen dabei den oben eingeführten Funktionen zur Berechnung der maximalen *velocity* und des Gradienten. Die Hauptfunktion findet mit ihrer Hilfe alle gesuchten Sequenzen und charakterisiert sie, indem der maximale Gradient sowie die Anzahl der Sequenzen ermittelt werden. Diese Werte dienen als Attribute und können

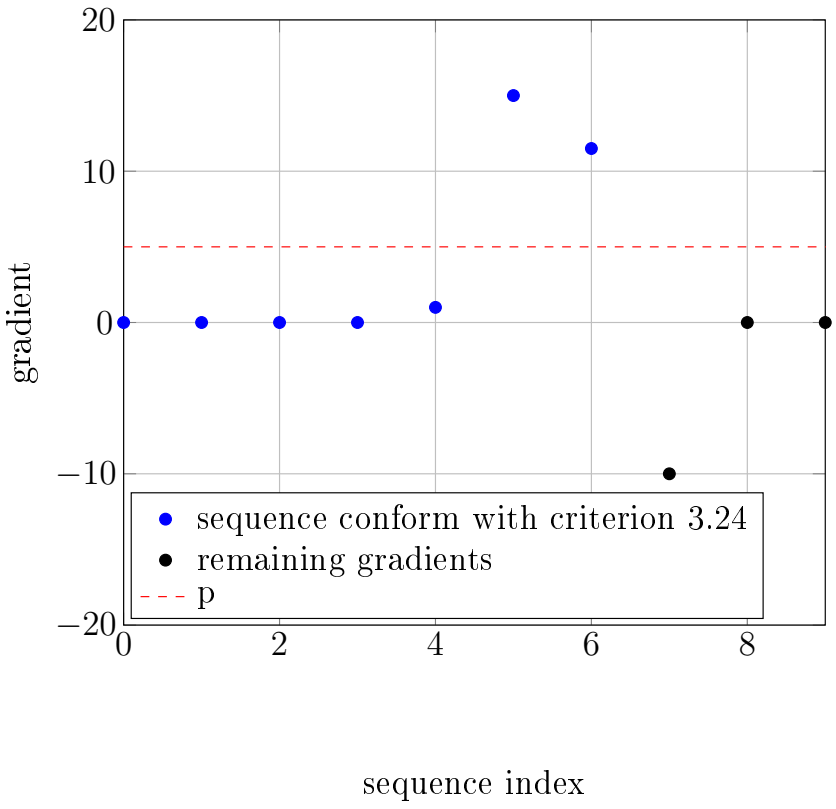

Abbildung 3.9.: Beispiel einer Sequenz plötzlicher Lautstärkezunahme mit $p = 5$

durch verschieden Wahlen von p und d gut parametrisiert werden.

Code 3.6: Pseudocode: Hilfsfunktionen Dynamiksteigung

```
1  notes ← set of notes
2
3  gradient(x, y, notes):
4
5  return (velocity(y,notes)−velocity(x,notes))/(y−x)
6
7  velocity(x, notes):
8  \\function seekActiveNotes see code 3.2
9  activeNotes ← seekActiveNotes(x, notes)
10 velocity ← 0
11
```

```
12 for i ← 0 to size(activeNotes)−1:
13         if (activeNotes[i].velocity > velocity) do
14                 velocity ← activeNotes[i].velocity
15         end
16 end
17
18 return velocity
```

Code 3.7: Pseudocode: Dynamiksteigung

```
1 notes ← set of notes
2 t ← length of the song
3 d ← some fixed distance
4 p ← some fixed critical value
5
6 getSequencesOfIncreasingVelocity(notes, t, d, p):
7 gradients ← ∅
8 tempSequence ← ∅
9 sequences ← ∅
10 count ← 0
11 state ← 0
12
13 for i ← 0 to t−d:
14         gradients[i] ← gradient(i∗d, i∗d+d, notes)
15 end
16
17 for count ← 0 to size(gradients)−1:
18         if (gradients[count] < p AND (state = 0 OR state = 1)) do
19                 tempSequence ← tempSequence ∪ gradients[count]
20                 state ← 1
21         else if (gradients[count] > p AND (state = 1 OR state = 2)) do
22                 tempSequence ← tempSequence ∪ gradients[count]
23                 state ← 2
24         else if (gradients[count] < p AND state = 2) do
25                 sequences ← sequences ∪ tempSequence
26                 tempSequence ← ∅
```

```
27                      state ← 0
28              end
29      end
30
31      maximum ← 0
32      lowGradients ← 0
33
34      for i ← 0 to size(sequences):
35              if (max(sequences[i]) > maximum) do
36                      maximum ← max(sequences[i])
37              end
38      end
39
40      numSeq ← size(sequences)
41
42      return [maximum, numSeq]
```

3.2.4. Kontur und Struktur eines Themas

Das Thema eines Stückes setzt sich im Wesentlichen aus zwei algorithmisch gut messbaren Bestandteilen zusammen: Denen einer Kontur und einer Struktur. Die Kontur ist die Abfolge von Tonhöhen, die Struktur die Abfolge von Tonlängen. Die ausschlaggebende Frage ist: Welche Umstände irritieren einen Rezipienten, sorgen also für ein weniger eingängiges Thema? Jourdain stellt einige Kompositionsregeln auf, über die er sagt:

> „Vergleicht man diese Regeln mit bekannten Melodien, findet man sie fast immer erfüllt."
>
> ([2], 118)

Gewiss garantiert eine Erfüllung aller Vorschriften kein Thema mit Ohrwurmpotential. Aber umgekehrt führt eine zu deutliche Nichteinhaltung wahrscheinlich zu einem wenig reizvollen Thema, woraus sich auf eine Korrelation mit dem Label schließen lässt. Jourdains Regeln betreffen allerdings ausschließlich den Aspekt der Kontur. Die Struktur betreffend gibt es keine so eindeutigen Kriterien, die zeigen, wie ein gutes Thema kreiert wird. Jede erfolgreiche Komposition wartet mit anders strukturierten Themen auf. Jourdain beschreibt das Fehlen derartiger Regeln:

> „Beachten Sie, daß alle erwähnten grundlegenden Regeln den harmonischen Aspekt einer Melodie betreffen. Verändert man innerhalb einer Melodie einen einzigen Ton, kann das zu harmonischen Unstimmigkeiten führen und die Melodie zerstören. Eine Melodie kann aber auch leicht zunichte gemacht werden, wenn man die Länge eines Tones verändert. Merkwürdigerweise schweigen sich die Kompositionslehren über rhythmische Vorschriften bei Melodien aus."
>
> ([2], 120)

Was sich aber messen lässt, ist ein aus der Struktur hervorgehender Höreindruck, ein Auftauchen dominanter Tonlängen. Die *Sony Corporation* verwendet in ihrem musikalischen Klassifikationsalgorithmus *12 Tone Analysis*, auf dessen Basis Hörempfehlungen gegeben werden, ein Attribut, welches die *Energie* eines Stücks misst: „perceived energy" [19]. Entscheidend ist dort die Frage, ob ein Stück eher in die Kategorie „quiet" oder „bright and lively" [19] passt, ob also vorwiegend längere oder kürzere Notenwerte dominieren. Nimmt man an, dass Themen, die in dieser Frage nicht festgelegt sind, einen unübersichtlichen Eindruck auf den Rezipienten machen, so ist hierin nicht nur ein Attribut für die Klassifikation jeweils nach Musikgeschmack, sondern auch für die Erfolgsklassifikation zu finden.

Konturmerkmale

Im Kapitel 3.2.1 war die Schwankung von Tonhöhen bei der Ähnlichkeitsbetrachtung schon einmal ein Kriterium. Berechnet wurde dort die Varianz bei der Verschiebung des Themas um ein bestimmtes Zeitintervall, also die Schwankung zwischen Teilen eines Stückes. Hier soll die Varianz innerhalb eines Themas ein Attribut darstellen. Jourdain rät von zu vielen Schwankungen ab.

> „Das *Gesetz der geschlossenen Gestalt* besagt, daß unser Gehirn vollständige Muster bevorzugt. Sprünge in einer Melodie unterbrechen deren ebenmäßige Kontur, weshalb sie weitgehend vermieden werden."
>
> ([2], 112)

Misst man diese mittels der korrigierten Varianz

$$s^2 = \frac{1}{n-1} \sum_i (note_i - avg)^2 \tag{3.25}$$

für ein Thema mit n Noten, so beschreibt man, wie eingängig dessen Kontur für den Hörer ist. Zu beachten ist in diesem Fall, dass die Berechnung ausschließlich für Themen zu einem sinnvollen Ergebnis führen kann, nicht für Harmoniestimmen. Die dann in die Berechnung einfließenden Akkorde böten ein zu großes Spektrum an Tönen und lieferten so automatisch eine hohe und zudem für verschiedene Harmoniestimmen wahrscheinlich sehr ähnliche Varianz. Zudem sagt Jourdain nichts über einen Zusammenhang zwischen den Schwankungen und dem Erfolgspotential von Harmonien aus.

Eine weitere Regeln betrifft die Hoch- und Tiefpunkte eines Themas.

> „Innerhalb einer Melodie soll der höchste Ton nur einmal auftreten, nach Möglichkeit auch der tiefste Ton."
>
> ([2], 119)

Ziel diese Konturmerkmals muss es also sein, zu messen, wie oft der jeweils höchste bzw. tiefste Ton eines Themas über die gesamt Dauer auftritt - ins Verhältnis gesetzt zur Länge. Liefert die Funktion

$$equals_{max}(x) = \begin{cases} 1 & \text{falls } x = max \\ 0 & \text{sonst} \end{cases} \tag{3.26}$$

für maximale x eine 1 zurück, so lässt sich dieses Merkmal mit der Formel

$$\frac{count(max, note)}{length} = \frac{\sum_i equals_{max}(note_i)}{length} \tag{3.27}$$

beschreiben. Analog kann das Attribut für die Suche nach Minima konstruiert werden.

Tonlängenverhältnis

Für die Berechnung des Tonlängenverhältnisses, das die Anzahl der kürzeren Noten in Relation zu der Gesamtzahl der Noten n setzt, ist ein Grenzwert p nötig, der zwischen langen und kurzen Tönen unterscheidet. Liefert die Funktion $dur(x)$ die Länge einer Note x, so kann mittels

$$short_p(x) = \begin{cases} 1 & \text{falls } dur(x) \leq p \\ 0 & \text{sonst} \end{cases} \tag{3.28}$$

die Zahl der kürzeren Noten bestimmt werden. p kann dabei benutzt werden, um das Attribut zu parametrisieren. Insgesamt ist das Tonlängenverhältnis

$$R = \frac{1}{n} \sum_{i} short_p(i). \tag{3.29}$$

Beispielhaft sollen diese Merkmale einmal für Henry Mancinis *Pink Panther* (Abb. 3.10) bestimmt werden. Die korrigierte Varianz des Themas beträgt $s^2 = 10.7702$ - ein relativ hoher Wert, der den belebten Chrakter des Stückes wiedergibt. Diesen Eindruck unterstützt ebenfalls das Tonlängenverhältnis $R = \frac{29}{32}$ für $p = \frac{1}{4}$. Fast alle Noten sind für diese Wahl von p kurze Noten, das Thema wird nur von einigen Ruhepunkten in den Takten 3, 4 und 7 unterbrochen. Dass Themenkontur- und struktur beide einen ähnlichen Höreindruck vermitteln - etwa nach dem Bild des rasch umhertrippelnden rosaroten Panthers, das viele sofort mit diesem Thema assoziieren -, unterstreicht das Ohrwurmpotential des Themas. Kontur und Struktur sind gut aufeinander abgestimmt und unterstützen sich gegenseitig. Zusätzlich liegt sowohl die Häufigkeit des höchsten, als auch die des tiefsten Tones bei jeweils einem Vorkommen pro acht Takten, wodurch Mancinis Komposition die Gültigkeit von Jourdains Regel belegt.

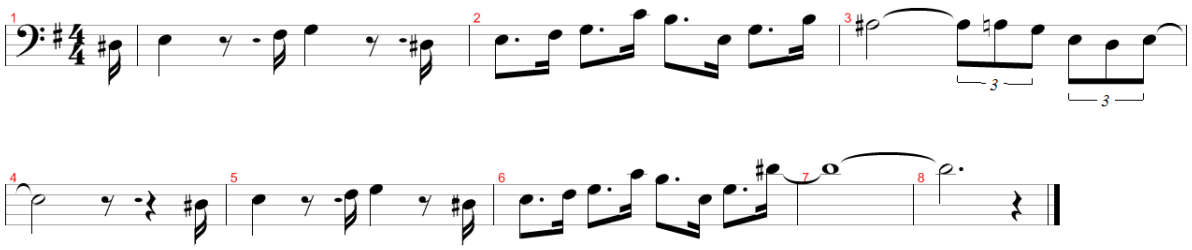

Abbildung 3.10.: Notenbeispiel: *Pink Panther* von *Henry Mancini* (vgl. [2], 121)

Algorithmus

Code 3.8: Pseudocode: Themenmerkmale

```
notes ← set of notes
p ← some fixed critical value, distinguishing short from long notes

equals(x, y):
if (x = y) do
        return 1
```

```
7  end
8  return 0
9
10 short(x, p):
11 if (x ≤ p) do
12         return 1
13 end
14 return 0
15
16 melody(notes, p):
17 max ← max(notes)
18 min ← min(notes)
19 n ← size(notes)
20 length ← overall length of notes
21 avg ← 0
22
23 for i ← 0 to n:
24         avg ← avg + notes[i]
25 end
26 avg ← avg/n
27
28 variance ← 0
29 ratioMax ← 0
30 ratioMin ← 0
31 ratioLength ← 0
32
33 for i ← 0 to n:
34         variance ← variance + (notes[i] − avg)²
35         ratioMax ← ratioMax + equals(max, notes[i])
36         ratioMin ← ratioMin + equals(min, notes[i])
37         ratioLength ← ratioLength + short(notes[i].dur, p)
38 end
39 variance ← variance/(n−1)
40 ratioMax ← ratioMax/length
41 ratioMin ← ratioMin/length
```

```
42 ratioLength ← ratioLength/n

43

44 return [variance ratioMax ratioMin ratioLength]
```

3.2.5. Tonales Zentrum

Auf der harmonischen Ebene eines Stückes wird Spannung vom Komponisten durch das Spiel mit dem tonalen Zentrum erzeugt. Hierbei handelt es sich um den durch die Tonart bestimmten „harmonischen Ausgangszustand“, der für den Hörer die Referenz und den „Bezugspunkt“ darstellt, „auf den alle Töne, Intervalle und Akkorde bezogen und mit dem sie verglichen werden“ ([2], 141). Das Geheimnis einer guten Komposition ist, laut Jourdain, Spannung durch bloße Andeutung und gleichzeitige Entfernung vom Zentrum zu erzeugen und diese zu gegebener Zeit wieder durch eine Rückkehr aufzulösen. Dieser stetige Wechsel zwischen Entspannung und Spannung charakterisiere eine gute Komposition (vgl. [2], 141).

> „Weniger begabte Komponisten kehren zu schnell und zu mechanisch zu ihrem tonalen Zentrum zurück oder entfernen sich so weit von ihm, daß der Hörer kaum erkennt, wenn er dort schließlich wieder angekommen ist. Der Trick besteht darin, die richtige Balance zwischen Bekräftigung des tonalen Zentrums und dessen Verletzung zu finden.“
>
> ([2], 141)

Aus diesen konkreten Kriterien, die Jourdain an die Hand gibt, lässt sich als Attributziel zunächst die Bestimmung des Verhältnisses zwischen tonarteigenen und -fremden Tönen formulieren. Ein zur Seite der tonartfremden Töne verschobenes Verhältnis gibt Anlass zur Vermutung, dass die Harmonie sich zwischenzeitlich zu oft oder zu lange vom Zentrum entfernt und den Hörer seiner tonalen Orientierung beraubt. Ein zur tonarteigenen Seite verschobenes Verhältnis bedeutet hingegen eine zu einfach gestrickte Komposition, die sich nie weit vom Zentrum wegbewegt.

Harmonieverhältnis

Zur Bestimmung des Verhältnisses ist zunächst die Kenntnis der Tonart nötig. Das MIDI-Framework [9] bietet eine Funktion *kkkey*, die, wie in Kapitel 2.2 beschrieben, die Tonart eines Stückes mit dem Algorithmus von Krumhansl-Kessler berechnet. Ausgehend von dieser Tonart kann die Menge der tonarteigenen Noten bestimmt werden.

Diese Menge entspricht den Tönen der Dur- oder Molltonleiter in der jeweiligen Tonart, für die es in der Musik feste Gesetzmäßigkeiten gibt. Ausgehend von einem Grundton GT wird eine Durtonleiter mit den folgenden Halbtonschritten gebildet.

$$Scale_{maj}(GT) = \{GT,\ GT+2,\ GT+4,\ GT+5,\ GT+7,\ GT+9,\ GT+11,\ GT+12\} \tag{3.30}$$

Der letzte Ton ist wieder der Grundton, um eine Oktave erhöht. Im Falle der C-Dur Tonleiter entspricht dies dem Notenbeispiel in Abb. 3.11. Die Molltonleiter kann auf drei

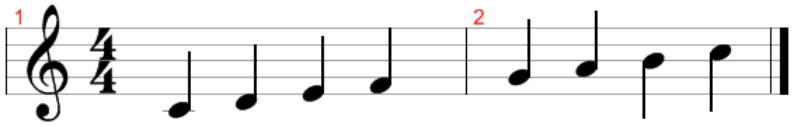

Abbildung 3.11.: Notenbeispiel: C-Dur Tonleiter

unterschiedliche Weisen gebildet werden, die natürliche, harmonische und melodische. Da harmonisches Moll für die Bildung von Harmonien die größte Bedeutung hat (siehe Kapitel 2.1), soll im Falle einer Molltonart diese Variante zur Untersuchung herangezogen werden. Die Bildungsregel lautet in diesem Fall:

$$Scale_{min}(GT) = \{GT,\ GT+2,\ GT+3,\ GT+5,\ GT+7,\ GT+8,\ GT+10,\ GT+12\} \tag{3.31}$$

Für ein Stück mit dem Notenvektor N der Dimension n und der Tonart GT des Tongeschlechts g kann das Harmonieverhältnis also mit der Funktion

$$test(x, S) = \begin{cases} 1 & \text{falls } x \in S \\ 0 & \text{sonst} \end{cases} \tag{3.32}$$

als

$$R = \frac{1}{n} \sum_{i=1}^{n} test(N_i, Scale_g(GT)) \tag{3.33}$$

definiert werden.

Henry Mancinis *Pink Panther* liegt die Tonart G-Dur zu Grunde, die entsprechend des Notenbeispiels im Bassschlüssel in Abb. 3.12 gezeigt wird. In Abb. 3.13 sind die tonartfremden Töne, die nicht in der G-Dur Tonleiter vorhanden sind, rot markiert. Dies ist bei sechs der insgesamt 32 Tönen der Fall, womit das Harmonieverhältnis bei $R = \frac{13}{16}$ liegt.

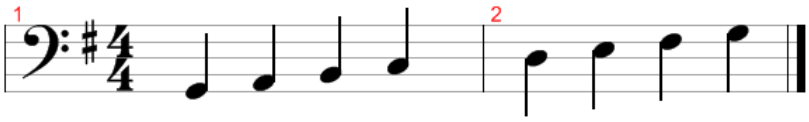

Abbildung 3.12.: Notenbeispiel: G-Dur Tonleiter

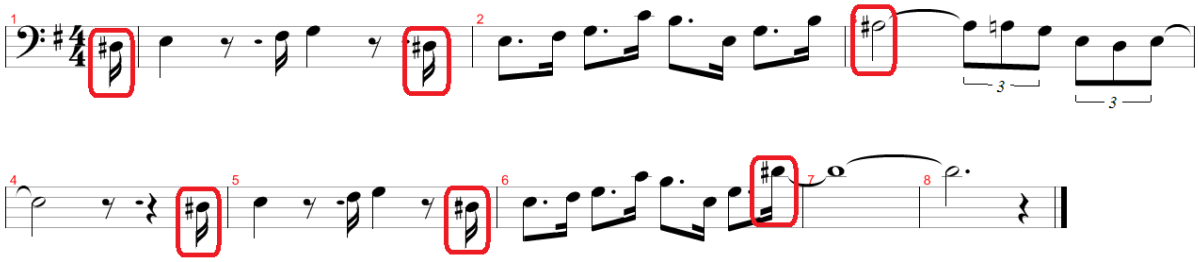

Abbildung 3.13.: Notenbeispiel: Tonartfremde Töne in *Pink Panther* von *Henry Mancini* (vgl. [2], 121)

Algorithmus

Beim Test, ob ein Ton in einer Tonleiter enthalten ist, ist zu beachten, dass Töne im Abstand von 12 Halbtönen, also der Oktave, äquivalent sind. Ein C mit einem *pitch-Wert* von 60 und eines mit einem *pitch-Wert* von 72 müssen in dieser Analyse gleich behandelt werden. Um diesen Umstand algorithmisch umzusetzen, testet die Funktion *test* (Zeile 80 in Code 3.9), ob sich die zu testende Note *note* in der selben Restklasse *modulo 12* wie eine der Noten der Tonleiter *scale* befindet. Zur Berechnung wird nicht die Tonleiter *scale* selbst, sondern ihr Rest *modulo 12* herangezogen, damit etwaige Töne mit $pitch > 12$ eine Oktave nach unten verschoben werden. Beispielweise muss die F-Dur Tonleiter

$$[5\ 7\ 9\ 10\ 12\ 14\ 16\ 17] \tag{3.34}$$

in

$$[5\ 7\ 9\ 10\ 0\ 2\ 4\ 5] \tag{3.35}$$

transponiert werden.

Code 3.9: Pseudocode: Harmonieverhältnis

```
1 notes ← set of notes
2
3 scale(g, gt):
```

```
4  if (g = 'major') do
5          // mod by element
6          return [gt gt+2 gt+4 gt+5 gt+7 gt+9 gt+11 gt+12] mod 12
7  else if (g = 'minor') do
8          // mod by element
9          return [gt gt+2 gt+3 gt+5 gt+7 gt+8 gt+10 gt+12] mod 12
10 end
11
12 test(note, scale):
13 for i ← 0 to 7:
14         if (note mod 12 = scale[i]) do
15                 return TRUE
16         end
17 end
18
19 return FALSE
20
21 ratioOfScalicNotes(notes):
22 n ← size(notes)
23 count ← 0
24
25 //function kkkey [9] returns key of chord
26 //function map maps return value of kkkey
27 //to 'major' or 'minor' and key
28 (g, gt) ← map(kkkey(notes))
29 scale ← scale(g, gt)
30
31 for i ← 0 to n:
32         if (test(notes[i], scale)) do
33                 count ← count + 1
34         end
35 end
36
37 return [1/n count]
```

3.2.6. *k*-Folgen

Während in Kapitel 3.2.5 die Tonalität eines Stückes mit einem eher allgemeinen Maß untersucht wurde, das einen Gesamteindruck des Stückes wiedergibt, soll hier die Suche nach speziellen Mustern im Vordergrund stehen. Es existieren bereits Ansätze, wie der in der Funktion *narmour* [9] implementierte Algorithmus von E. Narmour, die sich die Suche nach solchen Mustern zum Ziel gesetzt haben. Dabei liegt eine Analyse musikalischer Zusammenhänge zu Grunde, die Anhaltspunkte liefert, welche thematischen Strukturen interessant sind, um diese algorithmisch zu identifizieren. Beispielsweise werden Themen auf das Auftreten von *closure*, also musikalischer Auflösung, untersucht.

> „closure occurs when registral direction changes, or when a large interval is followed by a smaller interval“
>
> ([9], narmour.m, Zeile 85 f.)

In Abgrenzung dazu soll die Analyse von k-Folgen generisch sein. Es werden keine Kriterien zur Identifikation in Themen oder Harmonien vorgegeben. Stattdessen extrahiert der Algorithmus die Häufigkeit aller Folgen von k Tönen oder Akkordgrundtönen und bestimmt im Nachgang mittels Feature Selection Algorithmen (Kapitel 4.3), welche davon mit dem Label korreliert sind. Dies öffnet die Analyse für jede mögliche musikalische Struktur, wobei unbedeutende automatisch aussortiert werden. Zur Reduktion der Dimensionalität, die aufgrund der großen Menge verschiedener k-Folgen zu erwarten ist, werden keine Tonlängen, sondern lediglich Tonhöhen betrachtet. Z.B. ist eine Folge von drei halben Noten hier also identisch mit einer Folge von drei Achtelnoten derselben Tonhöhen. Um eine noch weitergehende Reduktion zu erreichen, werden alle Töne in dieselbe Oktave transponiert. Ein C” wird also nicht anders behandelt als ein C’. Darüberhinaus dienen nicht die absoluten Tonhöhen, sondern die dazwischenliegenden Intervalle zur Berechnung der Häufigkeiten. Dies kommt musikalisch einer Transposition in dieselbe Tonart gleich.

Sequentialisierung

Ein Problem, das bei der Identifikation von Tonfolgen auftritt, ist das Vorkommen von sich überschneidenden Noten. Eine Folge kann man nur dann bestimmen, wenn ihre Elemente nacheinander auftreten. In Musikstücken ist aber zu erwarten, dass sich Noten überschneiden, so wie in Abb. 3.14 gezeigt. Auf den Zählzeiten 1, 2 und 3 wird ein C gespielt, auf den Zählzeiten 3 und 4 ein F, auf Zählzeit 3 erklingen also beide. Wegen

des simultanen Auftretens beider Noten kann die Tonfolge an dieser Stelle nicht einfach (C, F) lauten. Ein Weg zur Auflösung der Überschneidung ist der, den Zeitraum, in

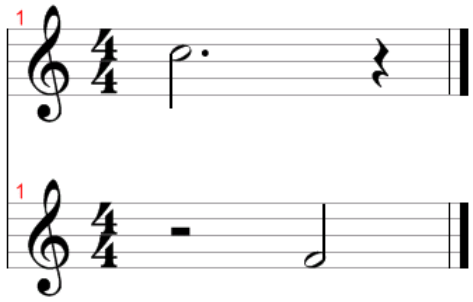

Abbildung 3.14.: Notenbeispiel: Notenüberschneidung

dem sie auftritt, als zusätzliches Element der Tonfolge zu betrachten. Die Folge lautete dann (C, {C,F}, F). Um die Notenmenge in eine elementare Note umzuwandeln, kann die bereits bekannte Funktion *kkkey* [9] dienen. Im vorliegenden Fall liefert sie ein F als Repräsentant der Menge, woraus sich letztendlich die Folge (C, F, F) ergibt.

Die Idee des Algorithmus 3.10 ist es, eine Tonfolge aus den Start- und Endzeiten aller Noten, die aus den Mididaten bekannt sind, zu generieren. Die Menge dieser Zeiten wird sortiert, und alle mehrfach vorkommenden Zeitpunkte werden bis auf einen Vertreter eliminiert. Generiert man nun an jedem vermerkten Zeitpunkt ein Element der Tonfolge aus den jeweils aktiven Noten und behält dabei die Ordnung bei, erhält man gerade die Tonfolge, in dem jeder Überschneidungszeitraum ein eigenes Element darstellt. Wählt man die Funktion *map* in Zeile 18 so, dass sie alle Eingabewerte in dieselbe Oktave abbildet, so erreicht man eine Einschränkung der Notenvielfalt auf ein Spektrum von 12 Tönen. Dadurch wird die Anzahl verschiedener Notenfolgen stark begrenzt, was eine Reduktion der Dimensionalität der Attributmenge bewirkt. In Abb. 3.15 ist verdeutlicht, wie die Start- und Endzeitpunkte der beiden Noten aus Abb. 3.14 zur Folge (C,F,F) führen. Die relevanten Zeitpunkte sind dabei durch die schwarz gestrichelten, senkrechten Linien gekennzeichnet.

Code 3.10: Pseudocode: Sequentialisierung

```
1 notes ← set of notes
2
3 sequentialize(notes):
4 times ← ∅
5 sequence ← ∅
```

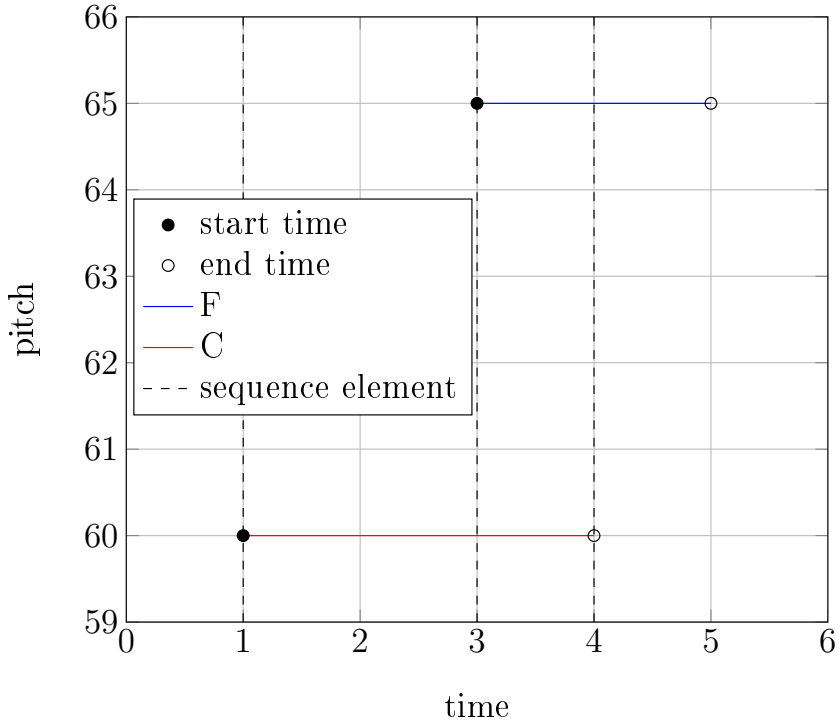

Abbildung 3.15.: Sequentialisierung von Überschneidungen

```
6
7  for i ← 0 to size(notes)−1:
8          times ← times ∪ notes[i].start ∪ notes[i].end
9  end
10
11 times ← unique(times)
12 times ← sort(times)
13
14 for i ← 0 to size(times)−1:
15         // function kkkey [9] returns key of chord
16         // function map maps return value of kkkey to midi pitch
17         // function seekActiveNotes see code 3.2
18         sequence[i] ← map(kkkey(seekActiveNotes(times[i], notes)))
19 end
20
21 return sequence
```

Algorithmus

Mit Hilfe der Funktion zur Sequentialisierung von nicht-monophonen Stimmen erfolgt die Analyse von k-Folgen mit dem Algorithmus 3.11, der eine *map* zurückgibt, die allen vorkommenden k-Folgen ihre Häufigkeit zuordnet. Monophon bedeutet in diesem Zusammenhang, dass zu jedem Zeitpunkt höchstens eine Note aktiv ist.

Code 3.11: Pseudocode: k-Folgen

```
1  notes ← set of notes
2
3  kSequences(notes, k):
4  frequencies ← empty map
5  sequence ← ∅
6  intervals ← ∅
7
8  notes ← sequentialize(notes)
9  //transforming vector of notes into vector of intervals
10 for i ← 0 to size(notes)-1:
11         intervals[i] ← |notes[i]-notes[i+1]|
12 end
13
14 for i ← 0 to size(intervals)-k:
15         sequence ← intervals[i:i+k-1]
16         seqName ← toString(sequence)
17         if (exists(frequencies[seqName])) do
18                 //increase frequency
19                 frequencies[seqName] ← frequencies[seqName] + 1
20         else do
21                 //put new entry to map
22                 frequencies[seqName] ← 1
23         end
24 end
25
26 return map
```

3.2.7. Identifikation von Schlüsselstellen

Alle bisher in den Kapiteln 3.2.1 bis 3.2.6 behandelten Attribute betrachten jeweils einen musikalischen Aspekt eines Songs isoliert. Es ist allerdings für eine umfassende Analyse geboten, ein Musikstück als Ganzes zu betrachten und Aspekte an den Schnittstellen zwischen Harmonie, Thema, Rhythmik und Dynamik zu untersuchen. Gerade wenn sich unterschiedliche Strukturen in einem Stück gegenseitig unterstützen, wenn beispielsweise harmonisch interessante Akkordfolgen auf rhythmisch besonders betonte Passagen fallen, kann erfolgreiche Musik entstehen (vgl. [2], 118 f.). Um solche Schlüsselstellen in einem Stück zu identifizieren, greift man als einfachste Methode auf besonders akzentuierte Stellen, also z.B. lokale Lautstärkemaxima, zu. Dies kann zum einen allgemein für das gesamte Stück Anwendung finden, sollte in jedem Fall aber auch isoliert für die Rhythmusstimmen vorgenommen werden, da diesen die Aufgabe der Betonung interessanter Stellen zukommt. Jourdain stellt einige diese Schlüsselstellen betreffende Regeln auf:

> „Wenn einige der restlichen fünf chromatischen Töne verwendet werden, sollten sie an einer unbetonten Stelle auftauchen, um die zugrundeliegende Harmonie nicht zu zerstören."
>
> ([2], 118)

Hieraus lassen sich die Verhältnisse derjenigen Schwerpunktstellen, die das genannte Kriterium erfüllen, zu allen identifizierten berechnen. Um die Analyse für die Frage offen zu halten, ob bestimmte Intervallsprünge von und zu Schlüsselstellen bevorzugt in erfolgreicher Musik zu finden sind, sollen darüberhinaus die häufigsten Sprünge identifiziert werden.

Zusätzlich können Schwerpunkte auch durch den Intervallverlauf definiert sein. Jourdain schreibt:

> „Das Ohr nimmt Sprünge immer als betont wahr (das Gehirn achtet besonders auf Sprünge, da sie die Grenzen von Melodieabschnitten markieren)."
>
> ([2], 119)

Die Beschaffenheit dieser großen Intervallsprünge hinsichtlich ihrer Ausgangs- und Zieltöne ist also ebenfalls von Bedeutung. Jourdain empfiehlt für erfolgreiche Kompositionen, nur von tonarteigenen zu ebensolchen Tönen zu springen, um der musikalischen

Spannung, die dem Sprung innewohnt, die Entspannung eines harmonischen Übergangs entgegenzusetzen (vgl. [2], 119). Hier zeigt sich, wie thematische und harmonische Struktur gegensätzlich arbeiten können, um interessante Musik zu erzeugen.

Identifikation über Betonung

Um lokale Lautstärkemaxima zu identifizieren, müssen alle Punkte eines Stückes gefunden werden, die in einem Teilintervall fester Länge $2r$ maximale Lautstärke besitzen und dabei nicht am Rand des Teilintervalls liegen. Fasst man also ein Stück als Zeitraum $[0, m]$ auf, versehen mit einer Funktion $vel(x)$, die für $0 \leq x \leq m$ die Lautstärke zum Zeitpunkt x zurückliefert, so ist die Menge der Schlüsselstellen

$$S = \{s \in [0, m] \mid \forall x \in [s - r, s + r] : vel(x) \leq vel(s)\} \tag{3.36}$$

Zur Berechnung der Attributwerte ist weiterhin eine Funktion $pitch(x)$ nötig, die die Tonhöhe im Falle eines einzelnen Tones bzw. den Grundton im Falle eines Akkordes an der Stelle $x \in [0, m]$ angibt. Mit der aus Kapitel 3.2.5 bekannten Funktion $test(x, Scale)$ und der dortigen Definition einer Tonleiter $Scale_g(GT)$ lässt sich das Verhältnis der Schlüsselstellen mit tonarteigenen Tönen zu allen Schlüsselstellen als

$$R_1 = \frac{1}{|S|} |\{s \in S \mid test(pitch(s), Scale_g(GT)) = 1\}| \tag{3.37}$$

definieren. Will man weiterhin ermitteln, welche Intervallsprünge zu und von den $s \in S$ führen, so wird ein Verfahren für die Ermittlung der Nachbarstellen von s benötigt. Da die Unterteilung der MIDI-Daten in Zeitintervalle zu fein ist, reicht es nicht, bloß $s - 1$ und $s + 1$ zu wählen, da diese möglicherweise noch innerhalb der selben Note liegen wie s. Mit einer Funktion $note(x)$, die einem Zeitpunkt $x \in [0, m]$ die Identität seiner Note in den MIDI-Daten zuordnet, ist der Vorgänger von s also

$$v(s) = max(\{x \in [0, s - 1] \mid note(x) \neq note(s)\}) \tag{3.38}$$

sowie analog der Nachfolger

$$n(s) = min(\{x \in [s + 1, m] \mid note(x) \neq note(s)\}). \tag{3.39}$$

Aus den Mengen der Vorgänger- und Nachfolgerintervalle, die sich durch $|pitch(v(x)) - pitch(x)|$ bzw. $|pitch(n(x)) - pitch(x)|$ berechnen lassen, kann man nun die am häufigsten

vorkommenden identifizieren.

Algorithmus

Code 3.12 extrahiert die Schlüsselstellen aus Notenmengen. Dabei wird jeweils für alle Zeitpunkte i mit dem Abstand t getestet, ob sie im Intervall $[i - r, i + r]$ das lokale Maximum sind. Die Funktion soll dabei auf die perkussiven und auf die übrigen Noten getrennt angewandt werden.

Code 3.12: Pseudocode: Schlüsselstellen über Betonung

```
1  notes ← set of notes
2  t ← resolution for segmentation of time
3  r ← interval in which local maximum is to be found
4
5  outOfRange(x, n):
6  if (x < 0) do
7          return 0
8  else if (x > n) do
9          return n
10 else do
11         return x
12 end
13
14 keyPointsFromVelocity(notes, r, t):
15 n ← size(notes)
16 max ← 0
17
18 keyPoints ← ∅
19
20 for i ← 0 to n-1:
21         if (notes[i].off > max) do
22                 max ← notes[i].off
23         end
24 end
25
26 for i ← 0 to max, step t:
```

```
27          isMax ← TRUE
28          for j ← outOfRange(i−r,max) to outOfRange(i+r,max), step t:
29                  \\function velocity returns summed up velocity
30                  if (velocity(j, notes) ≥ velocity(i, notes)) do
31                          isMax ← FALSE
32                  end
33          end
34          if (isMax)
35                  keyPoints ← keyPoints ∪ i
36          end
37  end
38
39  return keyPoints
```

Die Funktion *attributesFromKeyPoints* in Code 3.13 berechnet aus den Schlüsselstellen die beschriebenen Attribute. Dabei wird in Zeile 19 geprüft, ob die Note oder der Akkordgrundton, der sich an der jeweiligen Schlüsselstelle befindet, der Tonart des Stückes entstammt, während ab Zeile 23 die Vorgänger- und Nachfolgerintervalle der Schlüsselstellen mit Hilfe der Funktion *neighbours* in Code 3.14 bestimmt werden. Wichtig ist, dass die Grundtöne von etwaigen Akkorden zurück in ihre Ursprungsoktave gerechnet werden. Nur so lässt sich das tatsächliche Intervall bestimmen. Die Funktion *kkkey* [9] berechnet den Grundton ohne Berücksichtigung dieser Ursprungsoktave, map_2 muss also diese Umrechnung leisten.

Code 3.13: Pseudocode: Attribute aus betonten Schlüsselstellen

```
1  attributesFromKeyPoints(keyPoints, notes):
2  //function kkkey [9] returns key of piece
3  //function map1 maps return value of kkkey
4  //to 'major' or 'minor' and key in first octave
5  (g, gt) ← map1(kkkey(notes))
6  // function scale see code 3.9
7  scale ← scale(g, gt)
8
9  n ← size(keyPoints)
10
11 scalicKeyPoints ← 0
```

```
12 forerunners ← ∅
13 followers ← ∅
14
15 for i ← 0 to n−1:
16         // function seekActiveNotes see code 3.2
17         // function kkkey [9] returns key of chord
18         // function test see code 3.9
19         if (test(kkkey(seekActiveNotes(keyPoints[i],
20             notes)), scale)) do
21                 scalicKeyPoints ← scalicKeyPoints + 1
22         end
23         v ← neighbour(i, notes, 'down')
24         w ← neighbour(i, notes, 'up')
25         //function map2 maps return value of kkkey to midi pitch in
26         //originating octave
27         v ← map2(kkkey(seekActiveNotes(v, notes)))
28         w ← map2(kkkey(seekActiveNotes(w, notes)))
29         i ← map2(kkkey(seekActiveNotes(i, notes)))
30         forerunners ← forerunners ∪ |v−i|
31         followers ← followers ∪ |w−i|
32 end
33
34 frequencies ← [0 ... 11]
35
36 for j ← 0 to 11
37     frequencies(j) ← count(forerunners = j);
38     frequencies(j) ← frequencies + count(followers = j);
39 end
40
41 ratio ← scalicKeyPoints / n
42 interval ← indexOf(max(frequencies), frequencies)
43
44 return [ratio interval]
```

Der Algorithmus 3.14 berechnet Nachbarnoten der Note, die zum Zeitpunkt x aktiv ist - abhängig vom Parameter *mode* Vorgänger oder Nachfolger. Da der Schlüsselstellenbe-

rechnung aus Betonungen keine Informationen über die Tonfolge zu Grunde liegen - die Schlüsselstellen also nur als Zeitpunkte im Stück identifiziert werden, nicht als Indizes in einem Notenvektor - ist diese Funktion nötig, will man etwas über die hin- und wegführenden Intervalle erfahren. Um diese Berechnung durchzuführen, prüft der Algorithmus zwei Kriterien. Im einfachen Fall liegt zum Zeitpunkt x ein anderer Ton oder Akkord vor als bei seinem potentiellen Nachbarn. Dann ist der Nachbar identifiziert. Ist die Tonhöhe allerdings beim Nachbarn die gleiche wie zum Zeitpunkt x, dann wird die Identität beider Noten über die Position in der *notematrix* verglichen. Ist sie unterschiedlich, so ist der Nachbar gefunden.

Code 3.14: Pseudocode: Nachbarnoten identifizieren

```
1  neighbour(x, notes, mode):
2  // function seekActiveNotes see code 3.2
3
4  max ← 0
5  for i ← 0 to n−1:
6          if (notes[i].off > max) do
7                  max ← notes[i].off
8          end
9  end
10
11 if (mode = 'up') do
12         i ← x+1
13 else if (mode = 'down')
14         i ← x−1
15 end
16
17 while (x ≠ 0 AND x ≠ max):
18         // chord or note has changed, neighbour found
19         if (seekActiveNotes(i, notes) ≠
20             seekActiveNotes(x, notes)) do
21                 return i
22         // no change, but note x is in different row
23         // of matrix 'notes' than note i is
24         else if (notes.indexOf(seekActiveNotes(i ,notes)) ≠
25             notes.indexOf(seekActiveNotes(x ,notes))) do
```

```
26                  return i
27          else do
28                  if (mode = 'up') do
29                          i ← x+1
30                  else if (mode = 'down')
31                          i ← x−1
32                  end
33          end
34  end
35
36  return NULL
```

Identifikation über Intervalle

Für die Identifikation von Intervallsprüngen lässt sich der Algorithmus 3.10 zur Sequentialisierung aus Kapitel 3.2.6 verwenden. Im Unterschied zur dortigen Verwendung darf allerdings die Funktion *map* (Zeile 18) den Notenwert ihrer Eingabe nicht in eine bestimmte Oktave versetzen, sondern muss den Grundton eines etwaigen Akkordes, den *kkkey* [9] (Zeile 18) zurückliefert, in seine ursprüngliche Oktave zurückrechnen. Das führt zu einem Erhalt der tatsächlichen Intervallsprünge. Aus dem resultierenden Notenvektor können anschließend alle Notensprünge, die größer als ein bestimmter Grenzwert p sind, wie folgt bestimmt werden.

$$isLarge(note_i, note_j) = \begin{cases} 1 & \text{falls } |note_i - note_j| \geq p \\ 0 & \text{sonst} \end{cases} \tag{3.40}$$

Um nun zu testen, ob Ausgangs- und Zielton des großen Sprunges tonarteigen sind, kann erneut die Funktion $test(x, Scale)$ aus Kapitel 3.2.5 dienen.

$$isScalic(note_i, note_j) = \begin{cases} 1 & \text{falls } isLarge(note_i, note_j) = 1 \\ & \wedge\, test(note_i, Scale_g(GT)) = 1 \\ & \wedge\, test(note_j, Scale_g(GT)) = 1 \\ 0 & \text{sonst} \end{cases} \tag{3.41}$$

Das Verhältnis der großen Intervallsprünge innerhalb der Tonart zu allen großen Intervallsprüngen für einen Notenvektor der Dimension n ist dann

$$R_2 = \frac{\sum_{i=1}^{n-1} isScalic(note_i, note_{i+1})}{\sum_{i=1}^{n-1} isLarge(note_i, note_{i+1})}. \tag{3.42}$$

Algorithmus

Code 3.15 berechnet mit Hilfe der Funktion *sequentialize* aus Code 3.10 einen monophonen Notenvektor, aus dem bestimmt wird, ob die großen Intervallsprünge weg von und hin zu tonarteigenen Noten erfolgen.

Code 3.15: Pseudocode: Schlüsselstellen über Intervalle

```
1  notes ← set of notes
2  p ← some fixed critical value
3
4  keyPointsFromIntervals(notes, p):
5  // function sequentialize see code 3.10
6  notes ← sequentialize(notes)
7  n ← size(notes)
8  largeIntervals ← 0
9  scalicIntervals ← 0
10
11 //function kkkey [9] returns key of piece
12 //function map maps return value of kkkey
13 //to 'major' or 'minor' and key in lowest octave
14 (g, gt) ← map(kkkey(notes))
15 // function scale see code 3.9
16 scale ← scale(g, gt)
17
18 for i ← 0 to n-2:
19         if (|notes[i] - notes[i+1]| ≥ p) do
20                 largeIntervals ← largeIntervals + 1
21                 // function test see code 3.9
22                 if (test(notes[i], scale)
23                     AND test(notes[i+1], scale)) do
24                         scalicIntervals ← scalicIntervals + 1
```

```
25                          end
26                  end
27 end
28
29 return  scalicIntervals / largeIntervals
```

4. Lernprozess & Vorhersage

In Kapitel 4 wird der Prozess des maschinellen Lernens dokumentiert. Kapitel 4.1 beschreibt die Sammlung der Daten, Kapitel 4.2 die Implementierung der Algorithmen aus Kapitel 3 sowie die Berechnung der Featurevektoren. Kapitel 4.3 veranschaulicht die Auswahl der Attribute für die Klassifikation, die in Kapitel 4.4 beschrieben wird.

4.1. Wahl des Datensatzes

Die Auswahl des Datensatzes stellt im vorliegenden Fall ein gewisses Dilemma dar. Einerseits kann der Datensatz für Data Mining Analysen nicht groß genug sein. Je mehr Samples zur Verfügung stehen, desto genauer die Vorhersage. Andererseits sollten alle Musikstücke aus dem gleichen oder zumindest einem vergleichbaren Genre stammen. Beethovens *5. Sinfonie*, Miles Davis' *So What* und *How much is the fish* von Scooter zusammen zu analysieren, dürfte wenig erfolgreich sein. Die musikalischen Konventionen verschiedener Zeiten und verschiedener Stile sind zu unterschiedlich, um diese Stücke mit denselben Attributen zu vermessen. Aus diesem Grunde muss das Sammeln der Daten auf einen zeitlich und stilistisch eingeschränkten Bereich begrenzt bleiben. Hinzu kommen praktische Probleme, die das Sammeln erschweren. Beispielsweise existieren von nur wenigen Stücken gut arrangierte MIDI-Dateien. Außerdem wurden nicht alle verfügbaren Stücke als Singles veröffentlicht, wodurch keine Informationen über die Chartposition zur Verfügung stehen. Um einen guten Kompromiss zwischen diesen Einschränkungen zu finden, ist der Datensatz aus Rock/Pop-Songs der 50er bis 90er Jahre aufgebaut. Diese Kategorie bietet vergleichsweise noch viele verwendbare Daten. Eine genaue Auflistung der 209 verwendeten Stücke findet sich im Anhang A.1. Unter ihnen sind 26 Nr.-1-Hits, also Hits im Sinne der Definition des Labels in Kapitel 3.1. Die MIDI-Dateien aller Stücke im Datensatz wurden aus der MIDI-Datenbank mididb.com [20] bezogen.

4.2. Berechnung der Featurevektoren

Der erste Schritt zur Berechnung der Featurevektoren ist die Umwandlung der MIDI-Dateien in eine *Notenmatrix* (siehe Kapitel 2.2). Eine Inkompatibilität des MIDI- Frameworks [9] mit 64-bit-Betriebssystemen erfordert hierbei einen Zwischenschritt. Zunächst müssen die MIDI-Dateien mit der *mf2t.exe*, die die Autoren des MIDI-Frameworks [9] veröffentlicht haben, in Textdateien konvertiert werden. Die ausführbare Datei ist ebenfalls unter der im Literaturverzeichnis angegebenen URL des Frameworks abrufbar. Mit der Funktion *mftxt2nmat()* können Notenmatrizen aus den Textdateien generiert werden.

Die Implementierung der Pseudocodes aus Kapitel 3 erfolgt in Matlab. Der Quellcode ist in Anhang A.2 zu finden. Die sequentielle Berechnung aller Attribute dauerte auf dem Testsystem für 209 Songs ca. 8 Stunden. Insgesamt generiert die Funktion *getFeatureVectors* (siehe Code A.1) 7332 Attribute. Die entsprechende 209x7334 MATLAB Matrix ist aus Platzgründen nicht in dieser Arbeit abgedruckt, sondern steht unter https://www.dropbox.com/s/2t0xbhcg81r0vp7/matrix.mat zum Download zur Verfügung. Die zwei zusätzlichen Spalten der Matrix sind die fortlaufende Indexvariable und das Label. Neben der Attributmatrix M selbst steht unter dem Downloadlink ein Cellarray m zur Verfügung, dass den Spaltenindex von M als Schlüssel und die Attributbezeichnung als Wert enthält.

Im Folgenden ist eine Dokumentation aller zur Attributberechnung nötigen Funktionen aufgeführt. Die Dokumentation enthält einen Punkt *Benennungsschema*. Dieser gibt Aufschluss darüber, wie die Attribute in dem Cellarray, dass die Funktion *getFeatureVectors* zurückgibt, benannt sind. Der Parameter *Stimmgruppe* enthält dabei die Information, ob das Thema (m), die harmonische Struktur (h) oder der Rhythmus (r) analysiert wurde. Entsprechend kann er den Wert m, h oder r annehmen.

getFeatureVectors(file)	Diese Funktion berechnet die Attributwerte für alle Songs in der durch *file* angegebenen Indexdatei.
Input	
file	Indexdatei mit folgendem Aufbau: path; entrance chart; peak chart; meldody channels; harmony channels; rhythm channels path: Pfad zur Textdatei des Songs entrance chart: Einstiegschartposition peak chart: Beste Chartposition melody channles: MIDI Kanäle, die Themen enthalten (jeweils kommasepariert) harmony channles: MIDI Kanäle, die harmonische Strukturen enthalten (jeweils kommasepariert) rhythm channles: MIDI Kanäle, die Rhythmen enthalten (jeweils kommasepariert)
Output	
M	Matrix mit Samples als Zeilen und Attributwerten als Spalten
m	Cellarray mit Spalten von M als Schlüssel und Attributbezeichnungen als Wert
Quellcode im Anhang	A.1

computeD(notes, step, timeSigNum)	Diese Funktion berechnet die Autodistanz des Stückes *notes*, bei einer Verschiebung um *step* Takte.
Input	
notes	Eine Notenmatrix
step	Verschiebung in Takten
timeSigNum	Zähler der Taktart (=Schläge pro Takt)
Output	
avg	Durchschnittliche Distanz zwischen dem Stück und dem um *step* Takte verschobenen Stück
s2	Varianz
Benennungsschema	D_avg_Stimmgruppe_step D_s2_Stimmgruppe_step
Quellcode im Anhang	A.2

computeAC(notes, step, timeSig)	Diese Funktion berechnet die Autokorrelation des Stückes *notes*, bei einer Verschiebung um *step* Takte.
Input	
notes	Eine Notenmatrix
step	Verschiebung in Takten
timeSig	Zähler der Taktart (=Schläge pro Takt)
Output	
ac	Autokorrelation zwischen dem Stück und dem um *step* Takte verschobenen Stück
Benennungsschema	AC_Stimmgruppe_step
Quellcode im Anhang	A.3

ratioOfAccentuation(notes, p)	Diese Funktion berechnet zwei Varianten des Betonungsverhältnisses.
Input	
notes	Eine Notenmatrix
p	Grenzwert für unbetonte Schläge (0-127)
Output	
r1	Verhältnis der Lautstärke auf den Grundschlägen zur maximalen Lautstärke auf den Grundschlägen (=127*#Grundschläge)
r2	Verhältnis der Anzahl der unbetonten Grundschläge (< p) zur Anzahl aller Grundschläge
Benennungsschema	rOA_1_Stimmgruppe_p rOA_2_Stimmgruppe_p
Quellcode im Anhang	A.4

sequencesOfIncreasingVelocity(notes, d, p)	Diese Funktion berechnet die Anzahl der Sequenzen plötzlicher Lautstärkezunahme und den maximalen Lautstärkegradienten.
Input	
notes	Eine Notenmatrix
d	Zeitintervall zur Glättung der Lautstärkekurve
p	Grenzwert für die Unterscheidung von kleinen und großen Gradienten
Output	
nSeq	Anzahl der Sequenzen plötzlicher Lautstärkezunahme
maximum	Maximaler Gradient
Benennungsschema	sOIV_Stimmgruppe_p
Quellcode im Anhang	A.5

melody(notes, p)	Diese Funktion berechnet verschiedene Themenattribute.
Input	
notes	Eine Notenmatrix, die ein Thema enthält
p	Grenzwert für kurze Notenwerte
Output	
var	Varianz des Themas
ratioMax	Verhältnis des Auftauchens der höchsten Note zur Dauer des Stückes
ratioMin	Verhältnis des Auftauchens der tiefsten Note zur Dauer des Stückes
ratioLength	Verhältnis der kurzen Noten ($<$ p) zur Anzahl aller Noten
Benennungsschema	mel_v_p mel_max_p mel_min_p mel_l_p
Quellcode im Anhang	A.6

ratioOfScalicNotes(notes)	Diese Funktion berechnet das Verhältnis der tonleitereigenen Töne zur Anzahl aller Töne.
Input	
notes	Eine Notenmatrix
Output	
ratio	Verhältnis der tonleitereigenen Töne zur Anzahl aller Töne
Benennungsschema	rOSN
Quellcode im Anhang	A.7

kSequences(notes, k)	Diese Funktion berechnet die Häufigkeit aller Sequenzen von k musikalischen Intervallen.
Input	
notes	Eine Notenmatrix
k	Zahl der musikalischen Intervalle pro Sequenz
Output	
freq	Eine Map, die als Schlüssel alle auftauchenden Sequenzen enthält, und als Wert ihre Häufigkeiten
Benennungsschema	z.B. 4|0|2 für k=3
Quellcode im Anhang	A.8

attributesFromIntervalKeyPoints(notes, p)	Diese Funktion berechnet das Verhältnis tonleitereigener Schlüsselstellen zur Anzahl aller Schlüsselstellen. Grundlage sind dabei Schlüsselstellen, an denen große Intervallsprünge vorliegen.
Input	
notes	Eine Notenmatrix
p	Grenzwert für Schlüsselstellenintervalle. Intervalle > p gelten als Schlüsselstellen.
Output	
ratio	Verhältnis tonleitereigener Schlüsselstellen zur Anzahl aller Schlüsselstellen. Tonleitereigen meint hier, dass beide Töne der Schlüsselstelle tonleitereigen sind.
Benennungsschema	aFIKP_p
Quellcode im Anhang	A.10

attributesFromVelocityKeyPoints(keyPoints, notes)	Diese Funktion berechnet das Verhältnis tonleitereigener Schlüsselstellen zur Anzahl aller Schlüsselstellen und gibt die drei häufigsten Intervalle, die an Schlüsselstellen auftauchen, an. Grundlage sind dabei Schlüsselstellen, die stark betont sind.
Input	
keyPoints	Ein Vektor der Zeitpunkte von Schlüsselstellen enthält. Generiert mit der Funktion *keyPointsFromVelocity* in Code A.21.
notes	Eine Notenmatrix
Output	
ratio	Verhältnis tonleitereigener Schlüsselstellen zur Anzahl aller Schlüsselstellen
intervals	Die drei häufigsten musikalischen Intervalle (0-11), die an Schlüsselstellen auftauchen (-1, falls nicht vorhanden).
Benennungsschema	aFVKP_r aFVKP_i_1st aFVKP_i_2nd aFVKP_i_3rd
Quellcode im Anhang	A.9

4.3. Selektion

Die Feature Selection ist im vorliegenden Fall ein besonders wichtiger Schritt. Wie in Kapitel 4.2 beschrieben, liegt die Zahl der Attribute nach der Berechnung der initialen Featurevektoren bei 7332. Einen Großteil davon machen die 7189 Attribute aus der Analyse der *k-Folgen* aus, alle restlichen Attribute ergeben durch Parametrisierung eine Anzahl von 143. Bei einer Datensatzgröße von nur 209 Musikstücken ist die Zahl der Attribute damit viel zu groß. Tan *et al.* sagen über den *Fluch der Dimensionalität*:

> „The curse of dimensionality refers to the phenomenon that many types of data analysis become significantly harder as the dimensionality of the data increases. Specifically, as dimensionality increases, the data becomes

increasingly sparse in the space that it occupies."

([21], 51)

Die Zahl der Attribute muss also deutlich verringert werden.

Ein erster Ansatzpunkt dafür ist die große Menge der k-Folgen-Attribute. Im Zuge dieser Analyse wird, wie in Kapitel 3.2.6 beschrieben, ein Attribut für jede verschiedene Intervallfolge aufgestellt, die in einem Stück erklingt. Daraus resultiert eine große Zahl von Attributvektoren, die nur sehr dünn besetzt sind, da manche Intervallfolgen in nur einem oder einigen wenigen Stücken auftauchen. Die Aussagekraft solcher Attribute, deren Wert für fast alle Samples gleich Null ist, dürfte vernachlässigbar sein. Filtert man alle k-Folgen-Attribute heraus, die für mehr als 100 Samples eine Null liefern, reduziert sich die Dimensionalität um 6978 Attribute auf 354. Ausgehend von dieser ersten Reduktion, kann nun weiter Feature Selection betrieben werden.

Fast alle Attributideen wurden, wie in Kapitel 4.2 beschrieben, durch Paramterisierung in vielen verschiedenen Varianten berechnet. Da alle Varianten aber immer die gleiche Grundidee repräsentieren, sind sie mit hoher Wahrscheinlichkeit korreliert. Eine Menge korrelierter Attribute beeinflusst die Genauigkeit eines Klassifizierers aber negativ, da die Dimensionalität hoch ist, die Menge der Attribute aber nicht oder nur unwesentlich mehr Informationen enthält als ein einzelnes aus ihr. Deshalb gilt es, aus den parametrisierten Attributen einen Repräsentanten auszuwählen, der der Klassifizierung am besten dient. Abb. 4.1 zeigt dies anschaulich.

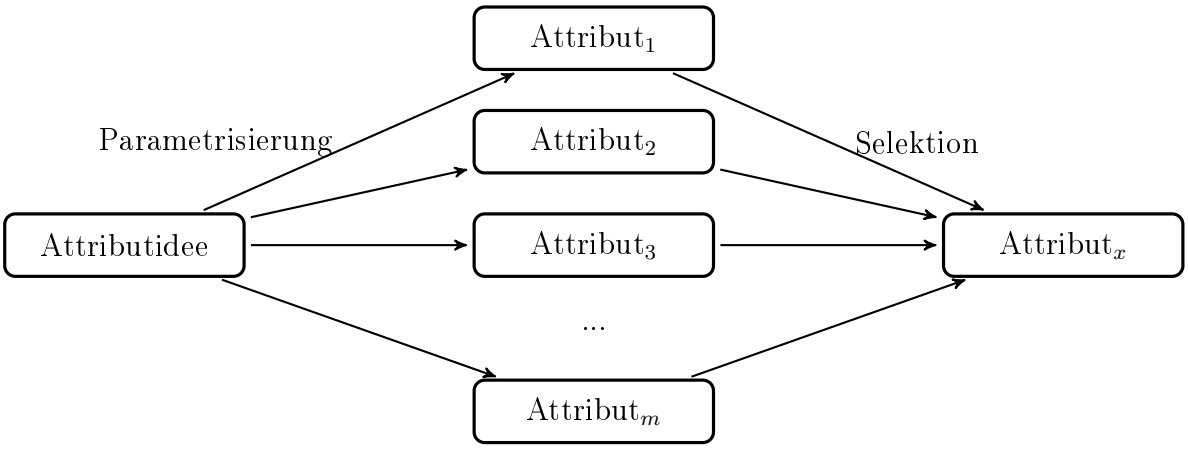

Abbildung 4.1.: Parametrisierung & Selektion

Als Algorithmus zur Feature Selection soll der in Kapitel 2.4.1.1 ausführlich beschriebene korrelationsbasierte Selektionsalgorithmus dienen. Dieser berechnet für jedes At-

tribut die Korrelation mit dem Label. Ein hoher Korrelationswert bedeutet eine gute *Erklärungskraft* beider Variablen über die jeweils andere (vgl. [12], 70 f.). Eine Wrapper-Methode kommt in diesem Fall nicht in Betracht, da sie Teilmengen evaluiert. Da die Anzahl der Teilmengen einer so großen Attributmenge allerdings unbeherrschbar wäre, ist hier eine Filter-Methode die bessere Wahl. Da ein Filter aber immer nur generelle Charakteristiken der Daten bewertet und keine Korrelation der Attribute untereinander, muss noch sichergestellt werden, dass aus dem Ranking der besten Attribute eine unkorrelierte Teilmenge generiert wird. Dies soll dadurch erreicht werden, dass aus jeder der Attributgruppen - Distanz, Autokorrelation, Rhythmik, Dynamik, Thema, Schlüsselstellen und k-Folgen - nur das bestbewertete Attribut Eingang in die Klassifikation findet. Eine Selektion des Attributs Tonales Zentrum ist nicht nötig, da dort keine Parametrisierung erfolgte. Die Annahme ist, dass innerhalb einer Gruppe eine hohe Korrelation der Attribute besteht, da sie durch Parametrisierung derselben Idee entstanden. Es genügt also, nur jeweils einen Repräsentanten zu wählen. So fließen am Schluss der Selektion 8 Attribute - eines pro Attributgruppe - in die Klassifikation ein.

Distanz		Autokorrelation	
0.15226	D_s2_h_8	0.214	AC_m_16
0.14191	D_s2_h_16	0.1845	AC_m_8
0.12929	D_s2_h_4	0.179	AC_h_32

Rhythmik		Dynamik		Tonales Zentrum	
0.1175	rOA_1_r_50	0.09621	sOIV_m_90	rOSN	0.089491
0.1175	rOA_1_r_60	0.08534	sOIV_n_50	-	-
0.1175	rOA_1_r_65	0.08485	sOIV_n_40	-	-

Thema		Schlüsselstellen		k-Folgen	
0.2404	mel_l_8	0.1217	aFVKP_i_1st	0.300727	0|0|0|1
0.0668	mel_max_4	0.0899	aFIKP_6	0.29053	0|1
0.0668	mel_max_16	0.0772	aFVKP_i_2nd	0.288318	0|0|1

Tabelle 4.1.: Bestbewertete Features pro Kategorie nach Korrelation mit dem Label

Die Implementierung der korrelationsbasierten Selektion im WEKA Data Mining Tool [16] liefert für die acht Attributgruppen die Korrelationswerte in Tabelle 4.1. Dargestellt sind jeweils die drei bestbewerteten Features. Bei der Auswahl des Datensatzes für die Selektion besteht ein Problem, das aus der geringen Datensatzgröße resultiert. Um ein zuverlässiges Modell zu trainieren und zu evaluieren, muss der gesamte Datensatz von 209 Stücken zur Verfügung stehen. Allerdings sollte der Datensatz für die Attributse-

lektion strikt von den übrigen Datensätzen getrennt sein, die Resultate in Tabelle 4.1 basieren jedoch schon auf zwei Drittel des gesamten Datensatzes. Ein Experiment zeigt, dass diese zwei Drittel bedenkenlos in der Modellbildung und Evaluation weiterverwendet werden können. Dazu wird die Feature Selection auf einem Drittel des Datensatzes wiederholt. Die nach Tabelle 4.1 besten Features werden dort ebenfalls gut bewertet. Auf den übrigen zwei Dritteln des Datensatzes findet dann die Modellbildung und Evaluation, wie in den folgenden beiden Kapiteln beschrieben, statt. Der F-Score (siehe Kapitel 5) ist auf diese Weise nur um 0.01 geringer als mit der Verwendung des gesamten Datensatzes. Dies zeigt, dass für die Modellbildung und Evaluation die Daten, die in diese Attributselektion einflossen, wiederverwendet werden können.

Bewertung der Selektionsergebnisse

Die Selektionsergebnisse lassen einige interessante Schlüsse zu. Zunächst fällt auf, dass die bestbewerteten Attribute allesamt thematische Aspekte der Stücke vermessen. Das Thema-Feature selbst zeigt eine hohe Korrelation mit dem Label, das Autokorrelationsattribut des Themas ebenso. Das bestbewertete Feature stammt aus der k-Folgen Analyse, in der Tonfolgen aus harmonischen Strukturen und dem Thema extrahiert werden. Die deutlich bessere Bewertung thematischer Features ist dabei aus musikalischer Perspektive nicht verwunderlich. Intuitiv würde man den *Ohrwurmcharakter* eines Stückes eher dem Thema (der Melodie) als harmonischen oder rhythmischen Aspekten zuschreiben. Hört man ein beliebtes Stück im Radio, pfeift man die Melodie oft mit. Kaum jemand würde allerdings auf die Idee kommen, den Harmonieverlauf zu pfeifen oder den Rhythmus zu klopfen. Jourdain bestätigt dieses Verständnis der Melodie als zentralen Aspekt eines Stückes.

> „Es ist schon merkwürdig, daß man für eine Melodie ein Urheberrecht anmelden, daß man also ein bestimmtes Schallmuster als Eigentum besitzen kann. Dagegen würde man Gelächter ernten, wollte man seine Rechte an einem Rhythmus oder an einer Harmoniefolge geltend machen, denn beides gilt als zu allgemein, um Urheberschutz zu ermöglichen."
>
> ([2], 88)

Dies legt nahe, dass der Melodie eine Sonderstellung unter den musikalischen Aspekten eines Stückes zukommt. Die Selektionsergebnisse erscheinen in diesem Zusammenhang plausibel.

Die allgemein bestbewerteten Features entstammen mit einer Korrelation von ca. 0.3 der k-Folgen Analyse. Den drei in Abb. 4.1 gezeigten Folgenmustern (0|0|0|1, 0|1 und 0|0|1) ist dabei gemeinsam, dass sie jeweils mit einem Intervallsprung von einer kleinen Sekunde (ein Halbton) enden. Die vorangehenden Töne bewegen sich alle auf gleicher Höhe, die Sprünge sind mit 0 Halbtönen angegeben. Über den musikalischen Grund für die hohe Korrelation kann nur spekuliert werden. Einem Intervall von einem Halbton wohnt eine Spannung inne, die aus ihrer Dissonanz resultiert. Nahe beieinanderliegende Töne erscheinen dissonant, weil die Rezeptoren für ihre Frequenzen im menschlichen Ohr dicht beeinander liegen (vgl. [2], 136). Diese Spannung macht sich ebenfalls der in Kapitel 2.1 beschriebene Leittoneffekt zu nutze. Der Grund für die hohe Korrelation der Attribute 0|0|0|1 und 0|0|1 kann ebenfalls in dieser Spannung liegen. Gerade nach einer langen Folge von gleichen Tönen (dargestellt durch die Nullen) kann der Intervallsprung der kleinen Sekunde eine noch stärkere musikalische Wirkung und Spannung erzeugen, als er das normalerweise durch seine Dissonanz schon tut. Einige erfolgreiche Stücke im Datensatz besitzen für beide Attribute einen hohen Attributwert, während die nicht erfolgreichen überwiegend niedrigere Häufigkeiten besitzen. Abb. 4.2 zeigt die Häufigkeiten beider k-Folgen im Datensatz. Blau dargestellt sind die Stücke der *kein-Hit-Klasse*, rot die der *Hit-Klasse*.

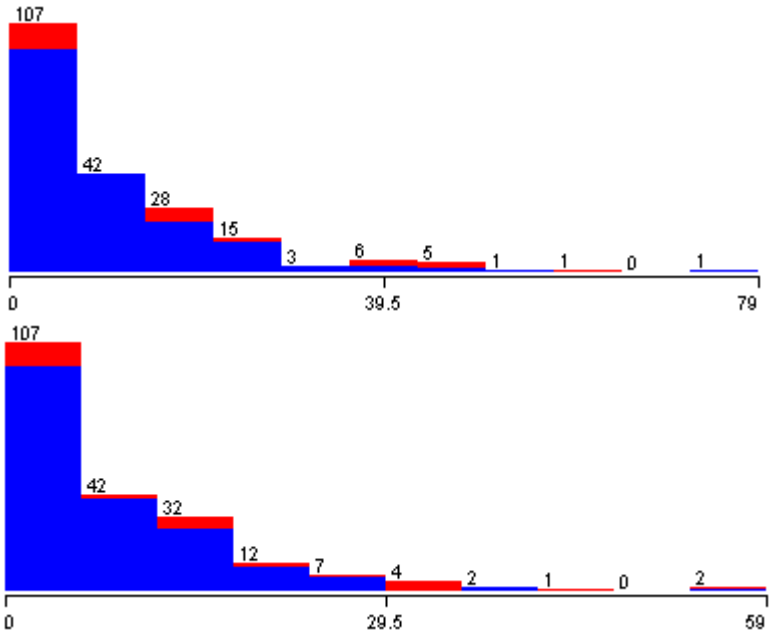

Abbildung 4.2.: Häufigkeitsverteilung für die k-Folgen Attribute 0|0|1 (oben) und 0|0|0|1 (unten) im Datensatz

4.4. Transformation & Klassifikation

Trotz Feature Selection kann man bei einem kleinen Datensatz von lediglich 209 Samples und 8 Attributen immer noch von einer hohen Dimensionalität sprechen. Es muss also ein Klassifikationsalgorithmus Anwendung finden, der diesen Umstand berücksichtigt. Ein solcher ist nach Nisbet *et al.* der Naive Bayes Algorithmus, dessen Funktionsweise ausführlich in Kapitel 2.4.1.2 beschrieben ist.

> „In fact, the Naïve Bayesian Classifier technique is particularly suited when the number of variables (the *dimensionality* of the inputs) is high."
>
> ([12], 51)

Um zusätzlich der geringen Größe des Datensatzes Rechnung zu tragen, bietet es sich an, die Vielfalt der Attributwerte durch Diskretisierung zu reduzieren. In Kapitel 2.4.1.3 ist beschrieben, wie die Methode der überwachten Diskretisierung arbeitet, die hier zum Einsatz kommt.

Ein gutes Mittel, auf kleinen Datensätzen zu evaluieren, bietet die Cross-Validierung. Bei der Cross-Validierung wird der Datensatz mehrfach in Test- und Trainingsdatensatz geteilt, wobei in jedem Durchgang der Testdatensatz aus andere Samples besteht. Die Validierung wird so oft wiederholt, bis jedes Sample einmal getestet wurde (vgl. [12], 295 f.). In der Regel teilt man die Datensätze in 10% Test- und 90% Trainigsdatensatz und wiederholt die Validerung 10 Mal. Dies hat den Vorteil, auf dem kompletten Datensatz testen zu können, was die Beurteilung der Genauigkeit des Algorithmus' erheblich erleichtert, insbesondere wenn nur ein sehr kleiner Datensatz zur Verfügung steht.

Alle aufgeführten Methoden werden im Knowledge Flow von WEKA 3.7.10 [16] ausgeführt. Abb. 4.3 zeigt den Aufbau des Prozesses. Der *ArffLoader* lädt die Daten im WEKA-eigenen ARFF-Format, der *ClassAssigner* ordnet dem Datensatz das richtige Label zu. Anschließend generiert der *CrossValidationFoldMaker* in 10 Durchgängen jeweils einen Trainingsdatensatz, der 90% und einen Testdatensatz, der 10% der Samples enthält. Der *FilteredClassifier* führt in jedem der 10 Durchgänge eine überwachte Diskretisierung mit den Trainingsdaten durch und diskretisiert die Testdaten nach den so gewonnenen Regeln. Anschließend trainiert er einen Naive Bayes Klassifikator auf den Trainingsdaten. Der *ClassifierPerformanceEvaluator* wertet den Vorhersagealgorithmus anhand der Testdaten aus und kummuliert die Ergebnisse über die 10 Durchgänge. Schließlich gibt er eine Confusion Matrix und weitere Evaluationsmetriken aus, die im nächsten Kapitel ausführlich behandelt werden.

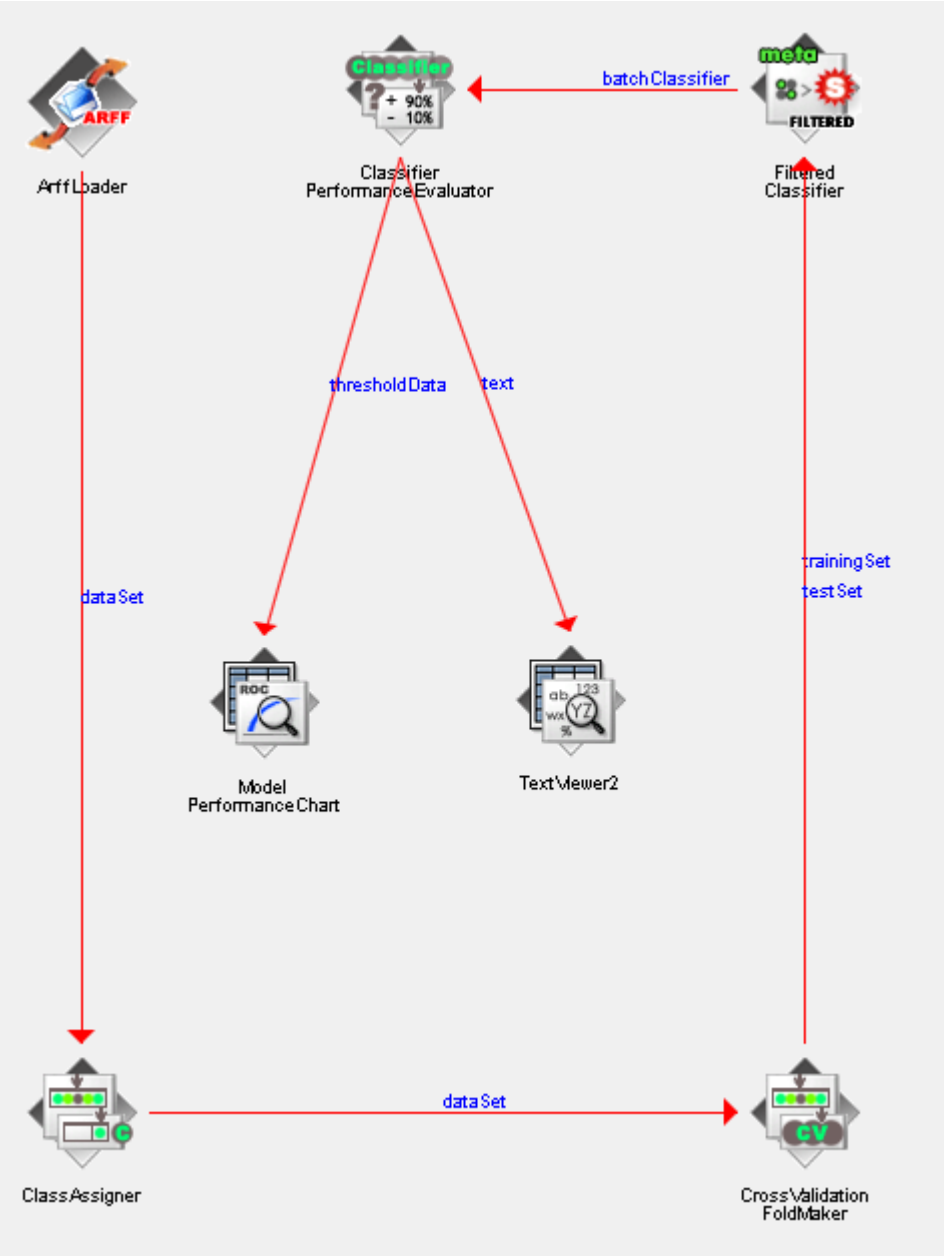

Abbildung 4.3.: Knowledge Flow

5. Evaluation

5.1. Bewertung der Vorhersage

Ausgangsbasis der Bewertung eines binären Vorhersagealgorithmus' ist die Confusion Matrix (vgl. [12], 293 f.). In ihren Spalten werden die vorhergesagten Labels aufgetragen, in ihren Zeilen die tatsächlichen Labels. Dabei werden die Klassen im binären Fall üblicherweise mit *Positive* (P) und *Negative* (N) bezeichnet. Entsprechend finden sich in den einzelnen Einträgen der Matrix die

- *True Positives* (TP), die Zahl der korrekt als positiv klassifizierten Samples,
- *False Positives* (FP), die Zahl der inkorrekt als positiv klassifizierten Samples,
- *True Negatives* (TN), die Zahl der korrekt als negativ klassifizierten Samples und
- *False Negatives* (FN), die Zahl der inkorrekt als negativ klassifizierten Samples.

Tab. 5.1 zeigt die Struktur der Confusion Matrix. Um ein gutes Vergleichsmaß für die Ge-

		vorhergesagtes Label	
		N	P
tatsächliches Label	N	TN	FP
	P	FN	TP

Tabelle 5.1.: Struktur der Confusion Matrix

nauigkeit des Algorithmus' zu erhalten, soll im Folgenden ein zufälliger Klassifikator konstruiert werden. Dieser stellt die untere Genauigkeitsgrenze dar, die ein nicht-zufälliger Klassifikator einhalten sollte. Zunächst ist dafür die Wahrscheinlichkeitsverteilung der Labels im Datensatz von Interesse, die aus Tab. 5.2 hervorgeht. Daraus ergibt sich die in Tab. 5.3 gezeigte Wahrscheinlichkeitsverteilung für die Confusion Matrix einer zufälligen Klassifikation. Diese Wahrscheinlichkeiten, übertragen auf den vorliegenden Datensatz von 209 Stücken, führen zur Confusion Matrix in Tab. 5.4. Der Naive Bayes Klassifika-

Hit (P)	kein Hit (N)
$12,4\%$	$87,6\%$

Tabelle 5.2.: Wahrscheinlichkeitsverteilung der Labels im Testdatensatz

		vorhergesagtes Label	
		kein Hit (N)	Hit (P)
tatsächliches Label	kein Hit (N)	$0.876 * 0.876 = 0.767$	$0.876 * 0.124 = 0.109$
	Hit (P)	$0.124 * 0.876 = 0.109$	$0.124 * 0.124 = 0.015$

Tabelle 5.3.: Wahrscheinlichkeitsverteilung für Confusion Matrix einer zufälligen Klassifikation

tor, der in den vorangehenden Kapiteln trainiert wurde, erreicht die Werte in Tab. 5.5. Nun existieren verschiedene Metriken, die sich aus der Confusion Matrix ableiten lassen. Dies sind

- $Recall = \frac{TP}{TP+FN}$, der Anteil der korrekt klassifizierten Positives an allen Positives,
- $Precision = \frac{TP}{TP+FP}$, der Anteil der korrekt klassifizierten Positives an allen als Positive klassifizierten,
- $Accuracy = \frac{TP+TN}{P+N}$, der Anteil der korrekten Vorhersagen,
- $Fallout = \frac{FP}{N}$, die Wahrscheinlichkeit, dass ein Negative falsch klassifiziert wird und
- $F - Score = 2 * \frac{Precision*Recall}{Precision+Recall}$.

Recall, Precision und F-Score liegen für die *Hit-Klasse* deutlich niedriger als für die *kein-Hit-Klasse*. Dies zeigen die Tabellen 5.6 und 5.7. Der Grund dafür ist die sehr ungleiche Verteilung von positiven und negativen Labels im Datensatz. Der eingesetzte Naive Bayes Algorithmus nutzt die Information über die Verteilung, wie in Kapitel 2.4.1.2 gezeigt, für die Vorhersage. Das führt zu einem guten F-Score in der negativen Klasse. Die Stärke des Algorithmus' liegt darüberhinaus in einer hohen Precision in der

		vorhergesagtes Label	
		kein Hit (N)	Hit (P)
tatsächliches Label	kein Hit (N)	160	23
	Hit (P)	23	3

Tabelle 5.4.: Confusion Matrix einer zufälligen Klassifikation

		vorhergesagtes Label	
		kein Hit (N)	Hit (P)
tatsächliches Label	kein Hit (N)	180	3
	Hit (P)	20	6

Tabelle 5.5.: Confusion Matrix des Naive Bayes Klassifikators

Hit-Klasse, positiv klassifizierte Stücke sind also sehr wahrscheinlich tatsächlich Hits. Defizite weist der Algorithmus hingegen auf, wenn es darum geht, alle Hits zu erkennen. Dies lässt sich am geringen Recall der *Hit-Klasse* ablesen. Es ist also anzunehmen, dass viele tatsächliche Hits falsch klassifiziert werden. Die in Tab. 5.8 aufgeführte Genauigkeit

	Naive Bayes	zufälliger Klassifikator
Recall	0,231	0,115
Precision	0,667	0,115
F-Score	0,343	0,115

Tabelle 5.6.: F-Score der Hit-Klasse

	Naive Bayes	zufälliger Klassifikator
Recall	0,984	0,874
Precision	0,9	0,874
F-Score	0,94	0,874

Tabelle 5.7.: F-Score der kein-Hit-Klasse

des Algorithmus' liegt aufgrund der guten Leistung in der *kein-Hit-Klasse* bei einem Wert von nahezu 90%.

Ein weiteres Analyseinstrument ist die Receiver Operating Characteristic (ROC). Dabei wird das zu analysierende Modell in einem Diagramm dargestellt, auf dessen X-Achse der Fallout, und auf dessen Y-Achse der Recall aufgetragen sind. Das bestmögliche Modell liegt bei (0,1), das schlechtestmögliche bei (1,0). Zufällige Modelle finden sich auf der Winkelhalbierenden wieder. Abb. 5.1 zeigt die ROC des vorliegenden Klassifikators. Der geringe Fallout spricht für das Modell. Allerdings ist der Recall, wie bereits gezeigt, sehr gering. Insgesamt erscheint das Modell aber besser als ein zufälliger Klassifikator.

5.2. Vergleich mit alternativen Algorithmen

Um die allgemeine Gültigkeit des Datenmodells sicherzustellen, soll im Folgenden die Vorhersage mit alternativen Algorithmen betrachtet werden. So kann überprüft werden,

	Naive Bayes	zufälliger Klassifikator
Accuracy	0,8899	0,7799

Tabelle 5.8.: Accuracy

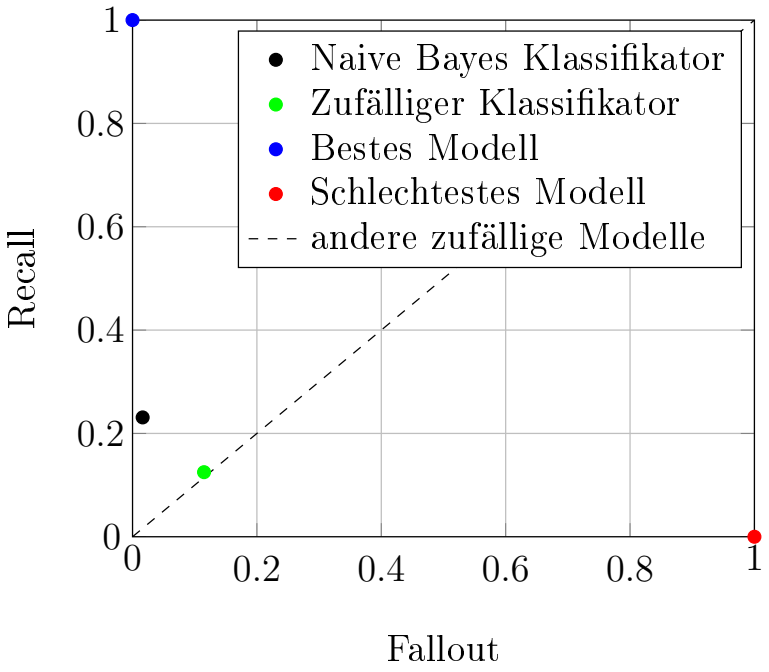

Abbildung 5.1.: ROC

ob die Vorhersageleistung des Naive Bayes Algorithmus zufällig ist, oder ob andere Algorithmen für den gleichen Datensatz ähnlich vorhersagen. Zur Anwendung kommen dabei die WEKA-3.7.10-Implementierungen folgender Algorithmen.

- ZeroR
- RandomTree
- J48
- DecisionStump
- MultilayerPerceptron

ZeroR ist ein trivialer, regelbasierter Algorithmus, der jedem Sample den häufigsten im Datensatz vorkommenden Klassenwert zuweist. Da die Häufigkeit der *kein-Hit-Klasse*

überwiegt, werden alle Samples negativ klassifiziert. RandomTree, J48 und DecisionStump sind Entscheidungsbaum-Algorithmen. Sie wählen das Attribut aus, anhand dessen Attributwerten die Menge der Samples im Hinblick auf die Klassifizierung am besten in Teilmengen zerlegt werden kann. Dieses Vorgehen wird für die resultierenden Teilmengen rekursiv wiederholt. Der durch den J48- und den RandomTree-Algorithmus entstehende Baum muss dabei im Gegensatz zu dem des DecisionStumps nicht binär sein. Der DecisionStump ist ein einfacher Baum-Algorithmus, dessen Baum lediglich eine Eben tief ist und somit nur ein Attribut zur Vorhersage heranzieht. Der MultilayerPerceptron-Algorithmus nimmt die Klassifikation anhand einer Hyperebene vor, die beide Klassen voneinander trennt und iterativ optimiert wird.

	Recall	Precision	F-Score	Fallout
Naive Bayes	0,231	0,667	0,343	0,016
ZeroR	0	0	0	0
RandomTree	0,308	0,333	0,320	0,087
J48	0,192	1	0,323	0
DecisionStump	0,154	0,571	0,242	0,016
MultilayerPerceptron	0,269	0,5	0,350	0,038

Tabelle 5.9.: Vergleich alternativer Vorhersagealgorithmen

Tabelle 5.9 zeigt das Ergebnis des Vergleichs. Vom ZeroR abgesehen, resultieren die Vorhersagen mit allen Algorithmen in ähnlichen F-Scores. Der MultilayerPerceptron übertrifft sogar noch den Naive Bayes, er erkennt einen Hit mehr korrekt. Dies deutet darauf hin, dass die hohe Precision und der geringe Recall nicht in der Wahl des Naive Bayes begründet liegen. Dieser nutzt die Häufigkeitsverteilung der Klassenwerte für die Vorhersage, begünstigt somit die Zahl der Negatives und sagt seltener Positives voraus. Somit wäre der geringe Recall und die hohe Precision erklärbar. Dass aber auch fast alle anderen Algorithmen zu einem ähnlichen Ergebnis kommen, gibt Anlass zur Vermutung, dass der Grund an anderer Stelle zu suchen ist. Eventuell liegt die geringe Zahl an Positives an der musikalisch sehr tiefgehenden Analyse. Ein *flacheres* Verfahren, dass oberflächlichere Attribute verwendet, könnte deutlich öfter positiv klassifizieren und somit den Recall erhöhen, nähme aber wahrscheinlich auch eine geringere Precision in Kauf. Die Kombination des vorliegenden Verfahrens mit einem flacheren wäre daher eine interessante Erweiterung, um Recall und Precision zu optimieren. Zur Einordnung aller alternativen Algorithmen zeigt Abb. 5.2 ihre Position in der Receiver Operating Characteristic.

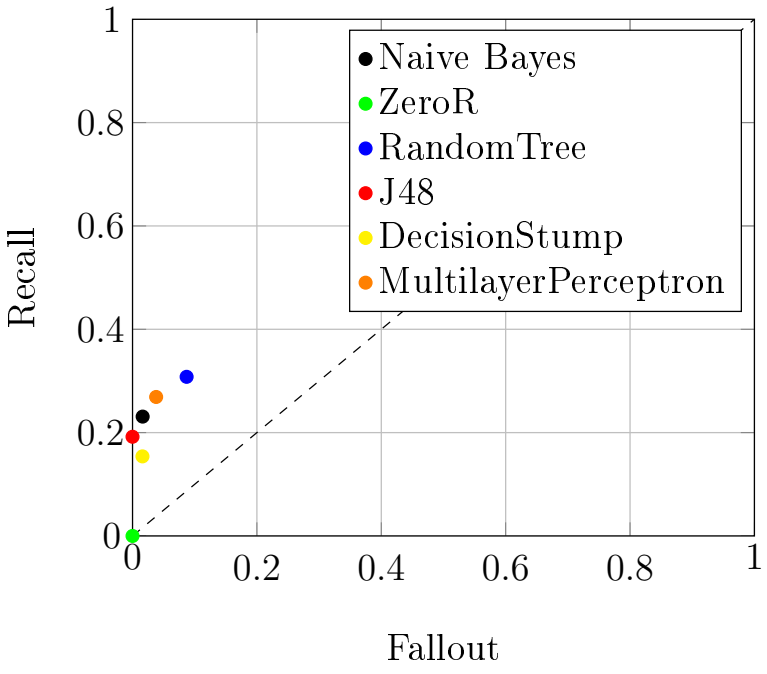

Abbildung 5.2.: ROC-Vergleich alternativer Algorithmen

5.3. Validität des Modells

Um die Validität des Modells zu beurteilen, werden im Folgenden zwei Baum-Algorithmen herangezogen. In den resultierenden Entscheidungsbäumen kann man ablesen, welche Attribute für die Vorhersage relevant sind. Diese stehen im Baum weit oben. Zurückgegeriffen wird dabei auf die Bäume des J48- (Abb. 5.3) und des DecisionStump-Algorithmus' (Abb. 5.4).

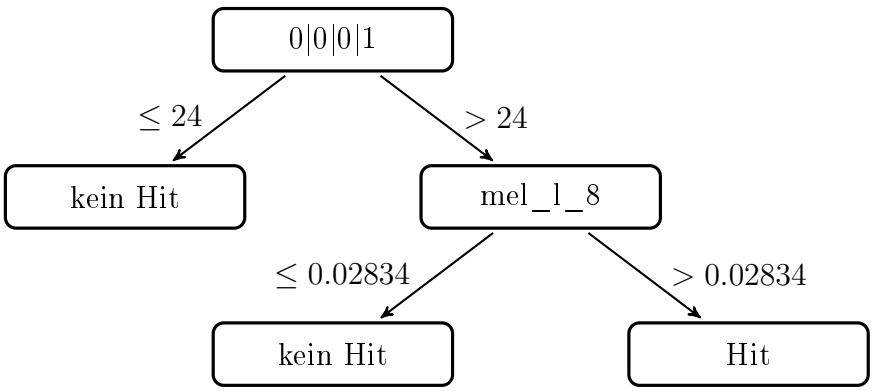

Abbildung 5.3.: J48 Entscheidungsbaum

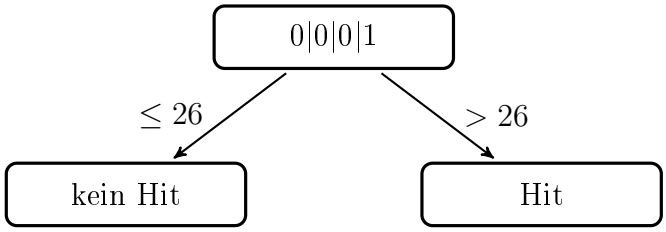

Abbildung 5.4.: DecisionStump Entscheidungsbaum

Beide Bäume bewerten das k-Folgen Attribut 0|0|0|1, das auch in der Attributselektion den höchsten Korrelationswert zeigte, am besten. Stücke mit 24 oder weniger dieser Intervallmuster werden negativ klassifiziert, Stücke mit mehr als 24 positiv. In Kapitel 4.3 wurde bereits die Sinnhaftigkeit dieses Attributs diskutiert. Da die kleine Sekunde (ein Halbton) ein musikalisch sehr interessantes und wichtiges Intervall darstellt, ist die hohe Gewichtung dieses Attributes durchaus plausibel. Ebenso wie die Tatsache, dass ein häufigeres Vorkommen des Intervallmusters zu einer positiven Klassifikation führt. Das Stück wirkt dann *spannungsgeladener*.

Das zweite Attribut mel_1_8, das im J48-Baum vorkommt, ist ebenfalls ein melodiebezogenes Attribut. Es misst das Verhältnis der Noten, die kürzer als 1/8-Note sind, zu allen Noten in der Melodiestimme. Da wenige kurze Noten zu einer negativen Klassifikation führen, kann man schließen, dass *energiegeladene*, schnelle Stücke bevorzugt als Hits empfunden werden. Ebenfalls in Kapitel 4.3 wurde die Relevanz einer Melodie für das Hitpotential eines Stückes betrachtet. Da es gängigen Ansichten in der Musik entspricht, dass die Melodie der wichtigste Aspekt eines Stückes ist, deutet das Vorkommen dieses Attributes im Entscheidungsbaum auf die Gültigkeit des Modells hin.

Gegen die Validität des Modells spricht, dass lediglich zwei Attribute Eingang in den Entscheidungsbaum finden. Ein Musiker würde argumentieren, dass die Komplexität einer Komposition niemals nur durch diese beiden Attribute ausgedrückt werden kann und dass viele weitere Aspekte Einfluss auf den Erfolg eines Stückes nehmen. Insofern bleibt fraglich, ob das vorliegende Modell auch für andere Datensätze zuverlässig vorhersagen kann.

5.4. Fazit

Insgesamt lässt sich feststellen, dass das Modell Erfolg deutlich genauer vorhersagt als der zufällige Klassifikator, was die Minimalanforderung an einen Vorhersagealgorithmus darstellt. Es hat allerdings große Schwächen: Viele Hits werden nicht erkannt. Ande-

rerseits kann man sich mit einiger Sicherheit darauf verlassen, dass ein prognostizierter Hit tatsächlich ein Hit wird. Für eine praktische Anwendung, in der man sich von der Vorhersage einen Hinweis erhofft, ob sich die Veröffentlichung und Vermarktung eines Songs finanziell lohnen würde, ist der Algorithmus unter diesen Umständen nur begrenzt zu gebrauchen. Man verfolgte eine sehr vorsichtige Strategie, verließe man sich einzig auf die Vorhersage. Viele potenzielle Hits würden nie veröffentlicht werden.

Aus musikalischer Sicht ist das Ergebnis kaum verwunderlich. Das eingangs erwähnt Zitat von Li *et al.* hat das Ergebnis bereits vorweggenommen.

> „[...] the goal ist to understand better the relation between intrinsic characteristics of songs [...] and their popularity, regardless of the complex and poorly understood mechanisms of human appreciation and social pressure at work."
>
> ([1], 311)

Der Algorithmus hat erwartungsgemäß Schwierigkeiten damit, die unerforschten Zusammenhänge zwischen musikalischen Merkmalen eines Songs und dessen Popularität zu simulieren. Seine Vorhersagen sind zwar besser als die eines zum Vergleich herangezogenen zufälligen Algorithmus'. Die Abdeckung in der *Hit-Klasse* ist jedoch gering. Ein Großteil der Hits wird nicht erkannt, somit ist auch der Publikumsgeschmack nur zu einem kleinen Teil erfasst. Der musikalische Schaffensprozess ist und bleibt kreativ geheimnisvoll und kann nicht in letzter Konsequenz und Ausprägung nachvollzogen werden.

Jourdain sagt über die Analyse von Melodien am Computer:

> „[...] es macht auch keinen Sinn, einen Supercomputer mit rasender Geschwindigkeit Melodien erzeugen zu lassen, denn wer hätte eine Billion Jahre Zeit, sich die Ergebnisse anzuhören und zu beurteilen?"
>
> ([2], 89)

Einen praktischen Ansatz, Melodien ebenfalls automatisch *anzuhören* und zu beurteilen, hat diese Arbeit geliefert.

A. Anhang

A.1. Datensatz

Die MIDI-Dateien folgender Songs sind sämtlich aus [20] entnommen, die zugehörigen Chartpositionen aus [17].

Titel	Interpret	beste Chart-position [17]
Cryin	Aerosmith	17
Don't Bring Me Down	The Animals	6
Don't Let Me Be Misunderstood	The Animals	3
House Of The Rising Sun	The Animals	1
Sky Pilot	The Animals	40
We Gotta Get Out Of This Place	The Animals	2
Ain't Seen Nothin Yet	Bachmann Turner Overdrive	2
Can't Get Enough	Bad Company	15
Feel Like Makin' Love	Bad Company	20
Barbara Ann	Beach Boys	3
California Girls	Beach Boys	26
Do It Again	Beach Boys	1
Hey Jude	The Beatles	1
I Want To Hold Your Hand	The Beatles	1
Love Me Do	The Beatles	17
Please Please Me	The Beatles	2
Strawberry Fields Forever	The Beatles	2
Deadweight	Beck	23
Devil's Haircut	Beck	22

Loser	Beck	15
Where It's At	Beck	35
Paranoid	Black Sabbath	4
Maria	Blondie	1
The Tide Is High	Blondie	1
Don't Fear The Reaper	Blue Oyster Cult	16
Charmless Man	Blur	5
Country House	Blur	1
Girls & Boys	Blur	5
Song 2	Blur	2
Tender	Blur	2
Lay Lady Lay	Bob Dylan	5
Like A Rolling Stone	Bob Dylan	4
Positively 4th Street	Bob Dylan	8
Rainy Day Women 12 & 35	Bob Dylan	7
Subterranean Homesick Blues	Bob Dylan	9
Livin' On A Prayer	Bon Jovi	4
Back To You	Bryan Adams	18
Can't Stop That Thing We Started	Bryan Adams	12
Christmas Time	Bryan Adams	55
Cloud No 9	Bryan Adams	6
Have You Ever Really Loved A Woman	Bryan Adams	4
I Finally Found Someone	Bryan Adams	10
It's Only Love	Bryan Adams	29
Let's Make A Night To Remember	Bryan Adams	10
On A Day Like Today	Bryan Adams	13
Please Forgive Me	Bryan Adams	2
Run To You	Bryan Adams	11
Somebody	Bryan Adams	35

Straight From The Heart	Bryan Adams	51
Summer Of 69	Bryan Adams	42
The Only Thing That Looks Good On You	Bryan Adams	6
This Time	Bryan Adams	41
Thought I'd Dies And Gone To Heaven	Bryan Adams	8
Peggy Sue	Buddy Holly	6
No Particular Place	Chuck Berry	3
School Days	Chuck Berry	24
Sweet Little Sixteen	Chuck Berry	16
Viva La Vida	Coldplay	1
Badge	Cream	18
Strange Brew	Cream	17
Sunshine Of Your Love	Cream	46
White Room	Cream	28
Bad Moon Rising	Creedence Clearwater Revival	1
Down On The Corner	Creedence Clearwater Revival	31
Marrakesh Express	Crosby, Stills, Nash & Young	17
Be Quiet And Drive Far Away	Deftones	50
Break On Through	The Doors	64
Hello I Love You	The Doors	15
Light My Fire	The Doors	49
Riders On The Storm	The Doors	22
A View To A Kill	Duran Duran	2
Girls On Film	Duran Duran	5
Hungry Like A Wolf	Duran Duran	5
New Moon On Monday	Duran Duran	9
Ordinary Worls	Duran Duran	6
Rio	Duran Duran	9
Save A Prayer	Duran Duran	2
The Reflex	Duran Duran	1

Union Of The Snake	Duran Duran	3
Wild Boys	Duran Duran	2
Heartache Tonight	Eagles	40
Hotel California	Eagles	8
Love Will Keep Us Alive	Eagles	52
Lyin' Eyes	Eagles	23
New Kid In Town	Eagles	20
One Of These Nights	Eagles	23
Please Come Home For Christmas	Eagles	30
Take It to The Limit	Eagles	12
The Long Run	Eagles	66
Carrie	Europe	22
The Final Countdown	Europe	1
Blue Morning	Foreigner	45
Cold As Ice	Foreigner	24
Feels Like The First Time	Foreigner	39
Hot Blooded	Foreigner	42
I Want To Know What Love Is	Foreigner	1
That Was Yesterday	Foreigner	28
Urgent	Foreigner	54
Iris	Goo Goo Dolls	50
Slide	Goo Goo Dolls	43
Don't Cry	Guns N' Roses	8
Knockin' On Heavens Door	Guns N' Roses	2
Live And Let Die	Guns N' Roses	5
Nightrain	Guns N' Roses	17
Pardon Me	Incubus	61
Cowboy	Kid Rock	36
All Day And All Of The Night	The Kinks	2
Lola	The Kinks	2
So Tired Of Waiting For You	The Kinks	1

Sunny Afternoon	The Kinks	1
Waterloo Sunset	The Kinks	2
You Really Got Me	The Kinks	1
Stairway To Heaven	Led Zeppelin	37
Are You Gonna Go My Way	Lenny Kravitz	4
Fly Away	Lenny Kravitz	1
Heaven Help	Lenny Kravitz	20
I Belong To You	Lenny Kravitz	75
Free Bird	Lynyrd Skynyrd	21
Sweet Home Alabama	Lynyrd Skynyrd	31
Bat Out Of Hell	Meat Loaf	15
I Would Do Anything For Love	Meat Loaf	1
Two Out Of Three Ain't Bad	Meat Loaf	32
Down Under	Men At Work	1
Who Can It Be Now	Men At Work	45
Heart Of Gold	Neil Young	10
Jeremy	Pearl Jam	15
Last Kiss	Pearl Jam	42
Another Brick In The Wall	Pink Floyd	1
Can't Stand Losing You	The Police	42
De Do Do Do De Da Da Da	The Police	5
Don't Stand So Close To Me	The Police	1
Back On The Chain Gang	The Pretenders	17
Don't Get Me Wrong	The Pretenders	10
I'll Stand By you	The Pretenders	10
Kid	The Pretenders	33
Another One Bites The Dust	Queen	7
Bicycle Race	Queen	11
Bohemian Rhapsody	Queen	1
Crazy Little Thing Called Love	Queen	2
Killer Queen	Queen	2
Play The Game	Queen	14

Radio Ga Ga	Queen	2
To Much Love Will Kill You	Queen	15
We Are The Champions	Queen	2
You're My Best Friend	Queen	7
Californication	Red Hot Chili Peppers	16
My Friends	Red Hot Chili Peppers	29
Otherside	Red Hot Chili Peppers	33
Scar Tissue	Red Hot Chili Peppers	15
Under The Bridge	Red Hot Chili Peppers	26
Everybody Hurts	R.E.M.	7
Great Beyond	R.E.M.	3
Losing My Religion	R.E.M.	19
Man On The Moon	R.E.M.	18
Shiny Happy People	R.E.M.	6
Angie	The Rolling Stones	5
Brown Sugar	The Rolling Stones	2
Get Off My Cloud	The Rolling Stones	1
Honky Tonk Woman	The Rolling Stones	1
Jumping Jack Flash	The Rolling Stones	1
Let's Spend The Night Together	The Rolling Stones	3
Miss You	The Rolling Stones	3
Mixed Emotions	The Rolling Stones	36
Paint It Black	The Rolling Stones	1
Ruby Tuesday	The Rolling Stones	59
Satisfaction	The Rolling Stones	1
Start Me Up	The Rolling Stones	7
Time Is On My Side	The Rolling Stones	62
Waiting On A Friend	The Rolling Stones	50
Send Me An Angel	Scorpions	27
Wind Of Change	Scorpions	53
Two Princes	Spin Doctors	3
Born To Be Wild	Steppenwolf	30

All This Time	Sting	22
Fields Of Gold	Sting	16
Fortress Around Your Heart	Sting	49
If I Ever Lose My Faith In You	Sting	14
If You Love Somebody Set Them Free	Sting	26
Moon Over Bourbon Street	Sting	44
The Soul Cages	Sting	57
Breakfast In America	Supertramp	9
Dreamer	Supertramp	13
Give A Little Bit	Supertramp	29
Goodbye Stranger	Supertramp	57
It's Raining Again	Supertramp	26
Logical Song	Supertramp	7
Once In A Lifetime	Talking Heads	14
Wild Wild Life	Talking Heads	43
Hold Me Thrill Me Kiss Me	U2	2
I Still Haven't Found What I'm Looking For	U2	6
One	U2	7
Pride	U2	3
Sweetest Thing	U2	3
When Love Comes To Town	U2	6
Where The Streets Have No Name	U2	4
With Or Without You	U2	4
Dreams	Van Halen	62
Jump	Van Halen	7
Panama	Van Halen	61
Runnin' With The Devil	Van Halen	52
Baba O'Riley	The Who	55
I Can See For Miles	The Who	10

My Generation	The Who	2
Pinball Wizard	The Who	4
Squeeze Box	The Who	10
Substitute	The Who	5
Summertime Blues	The Who	38
Won't Get Fooled Again	The Who	9
Gimme All Your Lovin	ZZ Top	10

A.2. Quellcode

A.2.1. Berechnung der Featurevektoren

Code A.1: Sequentieller Aufruf aller Attributfunktionen auf allen Stücken

```
1 function [ M attrNames ] = getFeatureVectors( file )
2 %getFeatureVectors returns matrix M of samples in rows
3 %and features in columns
4 %
5 %Input:
6 %   file: path to index file structured as follows
7 %
8 %         every line is a song:
9 %
10 %         path to txt file; entrance chart position;
11 %         peak chart position;
12 %         midi channels containing melody tracks (sep. by,);
13 %         midi channels containing harmony tracks (sep. by,);
14 %         midi channels containing rhythm tracks (sep. by,)
15 %
16 %         use mf2txt.exe to generate txt files
17 %
18 %         example: Paint_it_Black.txt;5;1;12,5;4,8,3;10
19 %Output:
20 %   M: matrix of samples in rows and features in columns
21 %   attrNames: cell array containing column of M as key
22 %              and name of attribute as value
```

```
23
24 % time capturing
25 begin = tic;
26 fprintf('Starting ...\n');
27
28 [name entrance peak mel harm rhyth] = textread(file, '%s %d %d
     %s %s %s','delimiter',';');
29
30 mel = cellfun(@(x) strsplit(x,','), mel, 'UniformOutput', false
     );
31 harm = cellfun(@(x) strsplit(x,','), harm, 'UniformOutput',
     false);
32 rhyth = cellfun(@(x) strsplit(x,','), rhyth, 'UniformOutput',
     false);
33
34 mel = cellfun(@(x) cellfun(@(y) str2num(y),x), mel, '
     UniformOutput', false);
35 harm = cellfun(@(x) cellfun(@(y) str2num(y),x), harm, '
     UniformOutput', false);
36 rhyth = cellfun(@(x) cellfun(@(y) str2num(y),x), rhyth, '
     UniformOutput', false);
37
38 % matrix with samples in rows and features in columns
39 M = zeros(length(name),1);
40 M(:,1) = 1:length(name);
41 attrNames = cell(0,0);
42 attrNames{1} = 'id';
43
44 % cell array of maps to store k-sequences temporarily
45 % during iteration over samples
46 freq = cell(length(name),0);
47 for i = 1 : length(name)
48     freq{i} = containers.Map('KeyType','char','ValueType','
         int32');
49 end
```

```
for i = 1 : length(name)
    songBegin = tic;
    nmat = mftxt2nmat(name{i});
    m = drop(nmat, [harm{i} rhyth{i}]);
    h = drop(nmat, [mel{i} rhyth{i}]);
    hm = drop(nmat, [rhyth{i}]);
    r = drop(nmat, [harm{i} mel{i}]);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % AC & D
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    [avg_m, s2_m] = computeD(m, [1 2 4 8 16 32], 4);
    [avg_h, s2_h] = computeD(h, [1 2 4 8 16 32], 4);
    M(i,2:25) = [avg_m avg_h s2_m s2_h];
    M(i,26:31) = computeAC(m, [1 2 4 8 16 32], 4);
    M(i,32:37) = computeAC(h, [1 2 4 8 16 32], 4);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % ratio of accentuation
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    p = 50 : 5 : 70;
    [r1_r r2_r] = arrayfun(@(x) ratioOfAccentuation(r, x), p);
    [r1_hm r2_hm] = arrayfun(@(x) ratioOfAccentuation(hm, x), p
        );
    M(i,38:47) = [r1_r r1_hm];
    M(i,48:57) = [r2_r r2_hm];

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % sequences of increasing velocity
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    p = 40 : 5 : 200;
    [n max] = arrayfun(@(x) sequencesOfIncreasingVelocity(nmat,
        0.5, x), p);
    M(i,58:90) = n;
```

```
    M(i,91:123) = max;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % melody
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    p = [1/16 1/8 1/4];
    [v rMax rMin rL] = arrayfun(@(x) melody(m, x), p);
    M(i,124:135) = [v rMax rMin rL];


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % ratio of scalic notes
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    M(i,136) = ratioOfScalicNotes( nmat );

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % key points
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    rKP = keyPointsFromVelocity( r, 4, 0.5 );
    hmKP = keyPointsFromVelocity( hm, 4, 0.5 );
    keyP = unique([rKP hmKP]);
    [ratio int] = attributesFromVelocityKeyPoints(keyP, hm);
    M(i,137) = ratio;
    M(i,138:140) = int;

    p = [ 4 5 6 7 ];
    ratio = arrayfun(@(x) attributesFromIntervalKeyPoints(m, x)
        , p);
    M(i,141:144) = ratio;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % k-sequences
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    f = kSequences(h, 2);
    freq{i} = joinMaps(freq{i}, f);
```

```
117     f = kSequences(h, 3);
118     freq{i} = joinMaps(freq{i}, f);
119     f = kSequences(h, 4);
120     freq{i} = joinMaps(freq{i}, f);
121     f = kSequences(m, 2);
122     freq{i} = joinMaps(freq{i}, f);
123     f = kSequences(m, 3);
124     freq{i} = joinMaps(freq{i}, f);
125     f = kSequences(m, 4);
126     freq{i} = joinMaps(freq{i}, f);
127
128     fprintf('Song %d done. Took %f seconds.\n', i, toc(
            songBegin));
129 end
130
131 % map k-seq names to columns of M
132 freqToM = containers.Map('KeyType','char','ValueType','int32');
133 start = 145;
134 for i = 1 : length(freq)
135     allKeys = keys(freq{i});
136     for j = 1 : length(allKeys)
137         k = allKeys{j};
138         if (~isKey(freqToM, k))
139             freqToM(k) = start;
140             start = start + 1;
141         end
142     end
143 end
144
145 % transfer particular freq{i}'s to M
146 allKeys = keys(freqToM);
147 for i = 1 : length(allKeys)
148     k = allKeys{i};
149     for j = 1 : length(name)
150         if (isKey(freq{j}, k))
```

```
                M(j, freqToM(k)) = freq{j}(k);
            else
                M(j, freqToM(k)) = 0;
            end
        end
end

% label
M(:,end+1) = entrance;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% attribute names
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
attrNames(2:7) = [{'D_avg_m_1'},{'D_avg_m_2'},{'D_avg_m_4'},{'
    D_avg_m_8'},{'D_avg_m_16'},{'D_avg_m_32'}];
attrNames(8:13) = [{'D_avg_h_1'},{'D_avg_h_2'},{'D_avg_h_4'},{'
    D_avg_h_8'},{'D_avg_h_16'},{'D_avg_h_32'}];
attrNames(14:19) = [{'D_s2_m_1'},{'D_s2_m_2'},{'D_s2_m_4'},{'
    D_s2_m_8'},{'D_s2_m_16'},{'D_s2_m_32'}];
attrNames(20:25) = [{'D_s2_h_1'},{'D_s2_h_2'},{'D_s2_h_4'},{'
    D_s2_h_8'},{'D_s2_h_16'},{'D_s2_h_32'}];
attrNames(26:31) = [{'AC_m_1'},{'AC_m_2'},{'AC_m_4'},{'AC_m_8'
    },{'AC_m_16'},{'AC_m_32'}];
attrNames(32:37) = [{'AC_h_1'},{'AC_h_2'},{'AC_h_4'},{'AC_h_8'
    },{'AC_h_16'},{'AC_h_32'}];
attrNames(38:42) = [{'rOA_1_r_50'},{'rOA_1_r_55'},{'rOA_1_r_60'
    },{'rOA_1_r_65'},{'rOA_1_r_70'}];
attrNames(43:47) = [{'rOA_1_hm_50'},{'rOA_1_hm_55'},{'
    rOA_1_hm_60'},{'rOA_1_hm_65'},{'rOA_1_hm_70'}];
attrNames(48:52) = [{'rOA_2_r_50'},{'rOA_2_r_55'},{'rOA_2_r_60'
    },{'rOA_2_r_65'},{'rOA_2_r_70'}];
attrNames(53:57) = [{'rOA_2_hm_50'},{'rOA_2_hm_55'},{'
    rOA_2_hm_60'},{'rOA_2_hm_65'},{'rOA_2_hm_70'}];

for j = 40 : 5 : 200
```

```
176     attrNames{end+1} = sprintf('sOIV_n_%d',j);
177 end
178 for j = 40 : 5 : 200
179     attrNames{end+1} = sprintf('sOIV_m_%d',j);
180 end
181
182 attrNames(124:126) = [{'mel_v_16'},{'mel_v_8'},{'mel_v_4'}];
183 attrNames(127:129) = [{'mel_max_16'},{'mel_max_8'},{'mel_max_4'
    }];
184 attrNames(130:132) = [{'mel_min_16'},{'mel_min_8'},{'mel_min_4'
    }];
185 attrNames(133:135) = [{'mel_l_16'},{'mel_l_8'},{'mel_l_4'}];
186 attrNames{136} = 'rOSN';
187 attrNames{137} = 'aFVKP_r';
188 attrNames(138:140) = [{'aFVKP_i_1st'},{'aFVKP_i_2nd'},{'
    aFVKP_i_3rd'}];
189 attrNames(141:144) = [{'aFIKP_4'},{'aFIKP_5'},{'aFIKP_6'},{'
    aFIKP_7'}];
190
191 allKeys = keys(freqToM);
192 for i = 1 : length(allKeys)
193     k = allKeys{i};
194     attrNames{freqToM(k)} = k;
195 end
196
197 attrNames{end+1} = 'label';
198
199 % time capture
200 fprintf('Done. Took %f seconds.\n', toc(begin));
201 end
202
203 function nmat = drop( nmat, channels)
204     for i = 1 : length(channels)
205         nmat = dropmidich(nmat, channels(i));
206     end
```

```
207 end
208
209 function m = joinMaps(m, f)
210     % join map f into m
211     allKeys = keys(f);
212     n = length(allKeys);
213
214     for i = 1 : n
215         k = allKeys{i};
216         if (isKey(m, k))
217             m(k) = m(k) + f(k);
218         else
219             m(k) = f(k);
220         end
221     end
222
223 end
```

A.2.2. Funktionen zur Attributberechnung

Code A.2: Distanz

```
1  function [ avg s2 ] = computeD( notes, step, timeSigNum )
2  %computeD returns distance index of nmat notes
3  %    Input:
4  %            notes: notematrix
5  %            step: vector of numbers of bars to shift
6  %            timeSigNum: numerator of time signature
7  %            (= beats per bar)
8  %    Output:
9  %            avg: average difference of signal and
10 %            signal shifted by step bars
11 %            s2: variance
12 %
13 avg = zeros(length(step),1);
14 s2 = zeros(length(step),1);
15
```

```
16 dur = notes(end, 7);
17 beat = notes(end, 2);
18 beatsPerSec = beat/dur;
19 SecPerBeat = dur/beat;
20 trackLength = max(notes(:,6) + notes(:,7));
21 nBeats = beatsPerSec * trackLength;
22
23 nBars = nBeats / timeSigNum;
24 f = @(x,y) abs(x-y);
25 m = SecPerBeat/4;
26
27 % segment notes in m intervals
28 c = cache(notes, m);
29
30 % distance matrix
31 mat = computeMatrix(c, f);
32
33 nIntervals = size(mat, 1);
34 b = ceil(nIntervals/nBars);
35
36 % function handle for accesing mat at column j
37 % and corresponding row for computing distance
38 p = @(j,s) mat(mod(j + b*s - 1, nIntervals) + 1, j);
39
40 % compute avg and s2 for every given step
41 for count = 1 : length(step)
42     coefficients = arrayfun(@(j) p(j, step(count)), 1 :
          nIntervals);
43     avg(count) = sum(coefficients/nIntervals);
44     s2(count) = sum((coefficients - avg(count)).^2)/(nIntervals
          - 1);
45 end
46
47 avg = avg';
48 s2 = s2';
```

```
49
50 end
```

Code A.3: Autokorrelation

```
1 function [ ac ] = computeAC( notes, step, timeSigNum)
2 %computeD returns distance index of nmat notes
3 %    Input:
4 %             notes: notematrix
5 %             step: vector of numbers of bars to shift
6 %             timeSigNum: numerator of time signature
7 %             (= beats per bar)
8 %    Output:
9 %             ac: autocorrelation
10 %
11 ac = zeros(length(step),1);
12 cov = zeros(length(step),1);
13
14 dur = notes(end, 7);
15 beat = notes(end, 2);
16 beatsPerSec = beat/dur;
17 SecPerBeat = dur/beat;
18 trackLength = max(notes(:,6) + notes(:,7));
19 nBeats = beatsPerSec * trackLength;
20
21 nBars = nBeats / timeSigNum;
22 m = SecPerBeat/2;
23
24 % segment notes in m intervals
25 c = cache(notes, m);
26
27 % convert vector of notes into vector of intervals
28 g = @(x,y) abs(x-y);
29 c = arrayfun(@(x) similarity(c(x),c(x+1),g), 1:length(c)-1);
30
31 % autocovariance matrix
```

```
32 avg = sum(c)/length(c);
33 f = @(x,y) (x - avg)*(y - avg);
34 mat = computeMatrix(c, f);
35
36 var = sum((c - avg).^2)/(length(c)-1);
37
38 nIntervals = size(mat, 1);
39 b = ceil(nIntervals/nBars);
40
41 % function handle for accesing mat at column j
42 % and corresponding row for computing autocorrelation
43 p = @(j,s) mat(mod(j + b*s - 1, nIntervals) + 1, j);
44
45 % compute an autocorrelation for every given step
46 for count = 1 : length(step)
47     coefficients = arrayfun(@(j) p(j, step(count)), 1 :
          nIntervals);
48     cov(count) = sum(coefficients)/(nIntervals - 1);
49     ac(count) = cov(count) / var;
50 end
51
52 ac = ac';
53
54 end
```

Code A.4: Betonungsverhältnis

```
1 function [ r1 r2 ] = ratioOfAccentuation( notes, p )
2 %ratioOfAccentuation returns r1: ratio of beat velocity
3 %to max beat velocity (127*numBeats) and r2: ratio
4 %of beats with velocity less than p to all beats
5 %   Input:
6 %           notes: nmat of notes
7 %           p: threshold for unaccented beats
8 %   Output:
9 %           r1: ratio of beat velocity to max beat
```

```
10 %             velocity (127*numBeats)
11 %             r2: ratio of beats with velocity less than p
12 %             to #beats
13 %
14 if (size(notes,1) == 0)
15     r1 = 0;
16     r2 = 1;
17     return;
18 end
19
20 dur = notes(end, 7);
21 beat = notes(end, 2);
22 beatsPerSec = beat/dur;
23 trackLength = max(notes(:,6) + notes(:,7));
24 nBeats = ceil(beatsPerSec * trackLength);
25
26 r1max = nBeats * 127;
27
28 idx = find(notes(:,1) == floor(notes(:,1)));
29 beats = notes(idx,:);
30 velocities(:,1) = unique(beats(:,1));
31 % find rows of beats with index x,
32 % pass them to maxVel to find maximum velocity.
33 % for x, vector of unique beat times is ran through.
34 velocities(:,2) = arrayfun(@(x) maxVel(beats(x == beats(:,1),:)
       ), velocities(:,1));
35
36 r1 = sum(velocities(:,2))/r1max;
37 r2 = (length(find(velocities(:,2) < p)) + (nBeats - length(
       velocities)))/nBeats;
38
39 end
40
41 function m = maxVel(notes)
42 [~,idx] = max(notes(:,5));
```

```
43 m = notes(idx,5);
44 end
```

Code A.5: Sequenzen plötzlicher Lautstärkezunahme

```
1  function [ nSeq maximum ] = sequencesOfIncreasingVelocity(
       notes, d, p )
2  %sequencesOfIncreasingVelocity returns number of
3  %sequences of increasing velocity and the maximum
4  %velocity gradient over all sequences
5  %    Input:
6  %            notes: nmat
7  %            d: interval for smoothing velocity curve
8  %            p: threshold to classify gradients
9  %            in small and large
10 %    Output:
11 %            nSeq: number of sequences
12 %            maximum: maximum gradient
13 %
14 trackLength = max(notes(:,6) + notes(:,7));
15 gradients = zeros(floor(trackLength/d),1);
16
17 % compute vector of gradients
18 for i = 0 : d : trackLength-d
19     gradients(i/d+1) = (grad(seekActiveNotes(i, notes, 'time'),
           seekActiveNotes(i+d, notes, 'time'), i, i+d));
20 end
21
22 tempSequence = [];
23 sequences = {};
24 state = 0;
25 % find sequences with gradients<p followed by gradients>p
26 for i = 1 : length(gradients)
27     if (gradients(i) <= p && (state == 0 || state == 1))
28         tempSequence(end+1) = gradients(i);
29         state = 1;
```

```
30      elseif (gradients(i) > p && (state == 1 || state == 2))
31          tempSequence(end+1) = gradients(i);
32          state = 2;
33      elseif (gradients(i) <= p && state == 2)
34          sequences{end+1} = tempSequence;
35          tempSequence = [];
36          state = 0;
37      end
38  end
39
40  % find max gradient
41  maximum = cellfun(@max, sequences);
42  maximum = max(maximum);
43  if (isempty(maximum))
44      maximum = 0;
45  end
46  % count gradients
47  nSeq = length(sequences);
48  end
49
50  function g = grad( notesI, notesJ, i, j )
51      g = velocity(notesJ, 'max') - velocity(notesI, 'max') / j -
            i;
52  end
```

Code A.6: Themenanalyse

```
1  function [ var ratioMax ratioMin ratioLength ] = melody( notes,
       p )
2  %melody examines a melody-notematrix in terms
3  %of contour and structure
4  %    Input:
5  %            notes: nmat containing a melody (no harmony track!)
6  %            p:     critical value for distinguishing
7  %                   between short and long notes
8  %    Output:
```

```
9 %             var:          variance of the melody
10 %            ratioMax:     ratio of #appearances of highest
11 %                          note to duration of nmat
12 %            ratioMin:     ratio of #appearances of lowest
13 %                          note to duration of nmat
14 %            ratioLength:  ratio of short notes (<=p) to #notes
15 duration = notes(end,6) + notes(end,7);
16 notesPitch = notes(:,4);
17 notesDurBeat = notes(:,2);
18 maximum = max(notesPitch);
19 minimum = min(notesPitch);
20 n = length(notesPitch);
21
22 % variance
23 avg = mean(notesPitch);
24 var = sum((notesPitch - avg).^2)/(n-1);
25
26 % count maximum's and minimum's appearances
27 ratioMax = length(find(notesPitch == maximum))/duration;
28 ratioMin = length(find(notesPitch == minimum))/duration;
29
30 % count "short" (<p) notes
31 ratioLength = length(find(notesDurBeat <= p))/n;
32 end
```

Code A.7: Harmonieverhältnis

```
1 function [ ratio ] = ratioOfScalicNotes( notes )
2 %ratioOfScalicNotes returns ratio of scalic notes to #notes
3 %   Input:
4 %           notes:  nmat
5 %   Output:
6 %           ratio:  ratio of scalic notes to #notes
7
8 % get key
9 [gt g] = mapToPitchFirst( kkkey( notes ) );
```

```
10 notes = notes(:,4);
11 n = length(notes);
12
13 % get scale from key
14 sc = scale(g, gt);
15
16 % count scalic notes
17 ratio = arrayfun(@(x) test(notes(x), sc), 1:n);
18 ratio = sum(ratio);
19 ratio = ratio / n;
20
21 end
```

Code A.8: k-Folgen

```
1 function [ freq ] = kSequences( notes, k )
2 %kSequences returns a map that indicates the frequency
3 %of all sequences of k intervals in nmat notes
4 %   Input:
5 %           notes:  nmat
6 %           k:      length of sequences
7 %   Output:
8 %           freq:   map that indicates the
9 %                   frequency of k-sequences
10
11 freq = containers.Map('KeyType','char','ValueType','int32');
12
13 notes = sequentialize(notes, 'first');
14 intervals = zeros(length(notes)-1);
15
16 % convert notevector into vector of intervals
17 for i = 1 : length(notes)-1
18     intervals(i) = abs(notes(i)-notes(i+1));
19 end
20
21 % get all k-sequences and update their freuquency in map freq
```

```
22 for i = 1 : length(intervals)-(k-1)
23     sequence = intervals(i:i+k-1);
24     seqName = toString(sequence);
25     if (isKey(freq, seqName))
26         freq(seqName) = freq(seqName) + 1;
27     else
28         freq(seqName) = 1;
29     end
30 end
31
32 end
33
34 % converts sequence into a key name
35 function name = toString( pitches )
36     name = '';
37
38     for i = 1 : length(pitches)
39         name = [name int2str(pitches(i))];
40         if ~(i == length(pitches))
41             name = [name '|'];
42         end
43     end
44 end
```

Code A.9: Schlüsselstellenattribute aus Betonung

```
1 function [ ratio intervals ] = attributesFromVelocityKeyPoints(
      keyPoints, notes )
2 %attributesFromKeyPoints computes ratio of scalic key points
3 %and most common intervals at key points
4 %   Input:
5 %           keyPoints:  vector of keyPoints
6 %           notes:      nmat
7 %   Output:
8 %           ratio:      ratio of scalic key points
9 %           intervals:  three (or less) most common
```

```
10 %                              intervals at key points
11 %                              (if less, filled with -1)
12
13 % get key
14 [gt g] = mapToPitchFirst( kkkey( notes ) );
15
16 %eliminate keyPoints with no notes playing (may result from
       drum solo parts)
17 for i = 1 : length(keyPoints)
18     if (isempty(seekActiveNotes(keyPoints(i), notes, 'time')))
19         keyPoints(i) = -1;
20     end
21 end
22 keyPoints(keyPoints == -1) = [];
23
24 n = length(keyPoints);
25 sc = scale(g, gt);
26
27 % handle for testing wether a key point is scalic
28 isScalic = @(x) test(mapToPitchFirst(kkkey(seekActiveNotes(x,
       notes, 'time'))), sc);
29
30 ratio = sum(arrayfun(isScalic, keyPoints))/n;
31
32 % computes set of differences between key point pitch and it's
33 % forerunner's/follower's pitch
34
35 forerunners = arrayfun(@(x) diff(x, notes, 'down'), keyPoints);
36 followers = arrayfun(@(x) diff(x, notes, 'up'), keyPoints);
37 forerunners(forerunners == -1) = [];
38 followers(followers == -1) = [];
39
40 % map to lowest octave
41 forerunners = mod(forerunners, 12);
42 followers = mod(followers, 12);
```

```
43
44
45 frequencies = zeros(12,2);
46 % first column determines interval
47 frequencies(:,1) = 0:11;
48
49 % count frequencies
50 for j = 0 : 11
51     frequencies(j+1,2) = length(find(forerunners == j));
52     frequencies(j+1,2) = frequencies(j+1,2) + length(find(
           followers == j));
53 end
54
55 % sort and get three most common intervals
56 [~, idx] = sort(frequencies(:,end),'descend');
57 frequencies = frequencies(idx,:);
58
59 % we dont want intervals, that dont occur
60 % delete them
61 nonZeroValues = find(frequencies(:,end) ~= 0);
62 frequencies = frequencies(nonZeroValues, :);
63
64 % are there three ore more intervals?
65 if (size(frequencies,1) < 3)
66     % return less than three if there are less
67     % fill with -1
68     numF = size(frequencies,1);
69     intervals = frequencies(1:numF, 1)';
70     intervals(numF+1:3) = -1;
71 else
72     % three else
73     intervals = frequencies(1:3, 1)';
74 end
75
76 end
```

```
77
78 function d = diff(x, notes, mode)
79
80     % handle for computing pitch at point in time x
81     pitch = @(x) mapToPitchOrig(kkkey(seekActiveNotes(x, notes,
           'time')), seekActiveNotes(x, notes, 'time'));
82
83     neighb = neighbour(x, notes, mode);
84     % test wether neighbour has no notes playing
85     if (isempty(seekActiveNotes(neighb, notes, 'time')))
86         d = -1;
87     else
88         d = abs(pitch(x) - pitch(neighb));
89     end
90 end
91
92 function nb = neighbour( x, notes, mode)
93     off = notes(end,1) + notes(end,2);
94     nb = 0;
95
96     % search for forerunner or follower?
97     if (strcmp(mode, 'up'))
98         i = x+(1/32);
99     elseif (strcmp(mode, 'down'))
100        i = x-(1/32);
101    end
102
103    % search for first i such that notes playing at i
104    % and playing at x are not the same
105    while (i >= 0) && (i <= off)
106        I = seekActiveNotes(i, notes, 'beat');
107        X = seekActiveNotes(x, notes, 'beat');
108
109        % compare the number of notes
110        if ( size(I) == size(X) )
```

```
111             % if they're the same, compare the notematrices
112             if ( I ~= X )
113                 % neighbour found
114                 nb = i;
115                 return;
116             end
117         else
118             % neighbour found
119             nb = i;
120             return;
121         end
122
123         % neighbour not found
124         if (strcmp(mode, 'up'))
125             i = i+(1/32);
126         elseif (strcmp(mode, 'down'))
127             i = i-(1/32);
128         end
129
130     end
131 end
```

Code A.10: Schlüsselstellenattribute aus Intervallen

```
1 function [ ratio ] = attributesFromIntervalKeyPoints( notes, p
    )
2 %attributesFromIntervalKeyPoints identifies key points
3 %as intervals greater than p and returns
4 %ratio of scalic key points to #key points
5 %   Input:
6 %           notes:      nmat
7 %           p:          critical value for determining
8 %                       key point intervals
9 %   Output:
10 %          ratio:      ratio of scalic key points
11 %                      to #key points
```

```
12
13 % get key
14 [gt g] = mapToPitchFirst( kkkey( notes ) );
15 sc = scale(g, gt);
16
17 % sequentialize
18 notes = sequentialize(notes, 'orig');
19 n = length(notes);
20
21 % handles to find out wether an interval is large/scalic
22 large = @(x) (abs(notes(x) - notes(x+1)) >= p);
23 scalic = @(x) (large(x) && test(notes(x), sc) && test(notes(x
       +1), sc));
24
25 numLarge = sum(arrayfun(large, 1:length(notes)-1));
26 numScalic = sum(arrayfun(scalic, 1:length(notes)-1));
27
28 ratio = numScalic/numLarge;
29 end
```

A.2.3. Hilfsfunktionen

Code A.11: Vorverarbeitung für Matrixberechnung

```
1 function [ c ] = cache( notes, m )
2 %cache returns vector of segmentation
3 %of notes in nmat into m intervals
4 %    Input:
5 %        notes: nmat
6 %        m: segment size
7 %    Output:
8 %        c: segmented vector
9 %
10
11 ons = notes(:,6);
12 offs = ons + notes(:,7);
13 n = max(offs);
```

```
14 t = 0 : m : n;
15 t = t(1:end-1);
16
17 c = arrayfun(@(x) toPitch(x, notes), t);
18
19 end
20
21 function pitch = toPitch( t, notes )
22 notes = seekActiveNotes(t, notes, 'time');
23
24 if (~isempty(notes))
25     % return key of chord in order to represent set of notes
26     pitch = mapToPitchOrig(kkkey(notes), notes);
27 else
28     pitch = 0;
29 end
30
31 end
```

Code A.12: Matrixberechnung

```
1 function [ mat ] = computeMatrix( cache, f )
2 %computeMatrix returns matrix that
3 %contains f(cache(i), cache(j)) at mat(i,j)
4 %   Input:
5 %           cache: vector of notes
6 %           f: elementary similarity function for notes
7 %           (is being passed to similarity(g,h,f))
8 %   Output:
9 %           mat: resulting matrix
10
11 n = length(cache);
12 mat = zeros(n,n);
13
14 for i = 1 : n
15     for j = 1 : n
```

```
16            mat(i,j) = similarity(cache(i), cache(j), f);
17        end
18    end
19
20    end
```

Code A.13: Umwandeln eines Tonartindex in Pitch (tiefste Oktave)

```
1  function [ pitch, majOrMin ] = mapToPitchFirst( key )
2  %mapToPitchFirst returns corresponding tone
3  %for key in first octave [0 .. 12]
4  %and 'major' or 'minor'
5  %    Input:
6  %            key:          key of notes
7  %    Output:
8  %            pitch:        pitch of tone in notes that
9  %                          correspondes to key
10 %            majOrMin:     major or minor?
11 %
12
13 majOrMin = 'major';
14
15 if (key > 12)
16     key = key - 12;
17     majOrMin = 'minor';
18 end
19 key = key - 1;
20
21 pitch = key;
22
23 end
```

Code A.14: Umwandeln eines Tonartindex in Pitch (ursprüngliche Oktave)

```
1  function [ pitch ] = mapToPitchOrig( key, notes )
2  %mapToPitchOrig returns corresponding tone for key
3  %in originating octave
4  %    Input:
```

```
5 %             key: key of notes
6 %             notes: set of notes (e.g. a chord)
7 %    Output:
8 %             pitch: pitch of tone in notes that
9 %             correspondes to key
10
11 notes = notes(:,4);
12 pitch = -1;
13
14 if (key > 12)
15     key = key - 12;
16 end
17 key = key - 1;
18
19 %%%%%%%%%%%%%%%%%%%%%%%
20 %Key is in notes
21
22 %transpose all notes to lowest octave
23 notesTrans = mod(notes, 12);
24 %find position of note that corresponds to key
25 idx = find(notesTrans == key);
26 if (~isempty(idx))
27     pitch = notes(idx);
28     %in case two or more notes match
29     pitch = max(pitch);
30     return;
31 end
32
33 %%%%%%%%%%%%%%%%%%%%%%%
34 %Key is not in notes
35
36 %max 11 octaves in midi data
37 keyVec = [0 12 24 36 48 60 72 84 96 108 120];
38 %key tone in all 11 octaves
39 keyVec = keyVec + key;
```

```
40
41 %select tone from keyVec with minimal distance to all tones in
       notes
42 distances = arrayfun(@(x) distance(x, notes), keyVec);
43 [~, idx] = min(distances);
44 if (~isempty(idx))
45     pitch = keyVec(idx);
46     return;
47 end
48
49 end
50
51 function dist = distance( x, vec )
52 dist = abs(vec - x);
53 dist = sum(dist);
54 dist = dist/length(vec);
55 end
```

Code A.15: Berechnung von Tonleitern aus Grundton

```
1 function sc = scale( g, gt )
2 %scale returns major or minor scale for key gt
3 if (strcmp(g,'major'))
4     sc = [gt gt+2 gt+4 gt+5 gt+7 gt+9 gt+11 gt+12];
5     sc = mod(sc, 12);
6 elseif (strcmp(g,'minor'))
7     sc = [gt gt+2 gt+3 gt+5 gt+7 gt+8 gt+10 gt+12];
8     sc = mod(sc ,12);
9 end
10 end
```

Code A.16: Finden von aktiven Noten

```
1 function [ activeNotes ] = seekActiveNotes( t, notes, mode )
2 %seekActiveNotes returns set of notes
3 %currently playing at time t
4 %   Input:
5 %           t: point in time
```

```
6 %             notes: set of notes
7 %             mode: is t in beats
8 %             or in time ('beat'/'time') ?
9 %     Output:
10 %            activeNotes: notes playing at t
11
12 activeNotes = zeros(0,7);
13 i = 1;
14
15 if (strcmp(mode,'beat'))
16     column = 1;
17 elseif (strcmp(mode,'time'))
18     column = 6;
19 else
20     %default
21     println('no mode, choose time as default');
22     column = 6;
23 end
24
25 for i = 1 : size(notes, 1)
26
27     if (notes(i,column) <= t) && (t < notes(i,column) + notes(i
       ,column+1))
28         activeNotes(end+1,:) = notes(i,:);
29     end
30     if (notes(i,column) > t)
31         break;
32     end
33 end
34
35 end
```

Code A.17: Sequentialisierung einer Notenmatrix

```
1 function [ sequence ] = sequentialize( notes, mode )
2 %sequentialize turn an nmat notes into a vector
```

```
3 %of sequentialized intervals
4 %    Input:
5 %              notes:        nmat
6 %              mode:         'first' to map all chord keys
7 %                            (as result of kkkey)
8 %                            to lowest octave,
9 %                            'orig' to map them to
10 %                           their originating octave
11 %   Output:
12 %             sequence:     interval vector
13
14 % get vector of start and ent times of all notes
15 times = [notes(:,6)' (notes(:,6)+notes(:,7))'];
16 times = sort(times);
17 times = unique(times);
18
19 % handle for mapping active notes at x to key in first octave
20 if (strcmp(mode, 'first'))
21     map = @(x) mapToPitchFirst(kkkey(seekActiveNotes(x, notes,
           'time')));
22 elseif (strcmp(mode, 'orig'))
23     map = @(x) mapToPitchOrig(kkkey(seekActiveNotes(x, notes, '
           time')), seekActiveNotes(x, notes, 'time'));
24 end
25
26 % handle for testing if no note is playing at x (return NaN
       then)
27 test = @(x) iff(~isempty(seekActiveNotes(x, notes, 'time')),
       map, x);
28
29 % sequentialize and delete all NaNs
30 sequence = arrayfun(@(x) test(x), times);
31 sequence = sequence(~isnan(sequence));
32 end
33
```

```
34 function res = iff(cond, handle, arg)
35     if cond
36         res = handle(arg);
37     else
38         res = [ NaN ];
39     end
40 end
```

Code A.18: Ähnlichkeitsberechnung von Notenmengen

```
1 function [ sim ] = similarity( g, h, f )
2 %similarity computes similarity of two notes or chords
3 %    Input:
4 %              g: note or chord (row(s) of nmat)
5 %              h: note or chord (row(s) of nmat)
6 %              f: elementary similarity function for two notes
7 %    Output:
8 %              sim: similarity measure
9
10 sim = 0;
11
12 if (size(g,1) ~= 1)
13     g = mapToPitchOrig(kkkey(g), g);
14 end
15 if (size(h,1) ~= 1)
16     h = mapToPitchOrig(kkkey(h), h);
17 end
18
19 sim = f(g,h);
20
21 end
```

Code A.19: Tonleiterzugehörigkeit von Noten testen

```
1 function bool = test(note, scale)
2 %test tests wether a note is part of a scale
3 if (~isempty(find(scale == mod(note, 12))))
4     bool = 1;
```

```
5 else
6     bool = 0;
7 end
8 end
```

Code A.20: Lautstärke einer Notenmenge feststellen

```
1 function [ vel ] = velocity( notes, mode )
2 %velocity returns max/sum of velocity of nmat notes
3 %   Input:
4 %           notes:  nmat
5 %           mode:   'max' or 'sum'
6 %           (maximum or summed up velocity)
7 %   Output:
8 %           vel:    max/summed up velocity
9
10 if (strcmp(mode, 'max'))
11     vel = unique(max(notes(:,5)));
12 elseif (strcmp(mode, 'sum'))
13     vel = sum(notes(:,5));
14 end
15
16 if (isempty(vel))
17     vel = 0;
18 end
19
20 end
```

Code A.21: Finden von Schlüsselstellen

```
1 function [ keyPoints ] = keyPointsFromVelocity( notes, r, t )
2 %keyPointsFromVelocity returns set of
3 %local velocity maxima in notes
4 %   Input:
5 %           notes:      nmat
6 %           t:          resolution for segmentation of time
7 %           r:          interval in which local maximum
8 %                       is to be found
```

```
9  %    Output:
10 %              keyPoints:  set of local maxima (=key points)
11
12 keyPoints = [];
13
14 if (size(notes,1) == 0)
15     return;
16 end
17
18 % end of track
19 off = notes(end, 6) + notes(end, 7);
20
21 % find local maxima
22 for i = 0 : t : off
23     isMax = true;
24     for j = outOfRange(i-r, off) : t : outOfRange(i+r, off)
25         if (velocity(seekActiveNotes(j, notes, 'time'), 'sum')
              >= velocity(seekActiveNotes(i, notes, 'time'), 'sum'
              )) && (i ~= j)
26             isMax = false;
27         end
28     end
29     if (isMax)
30         keyPoints(end+1) = i;
31     end
32 end
33
34 end
35
36 function num = outOfRange(x, n)
37     if (x < 0)
38         num = 0;
39     elseif (x > n)
40         num = n;
41     else
```

```
42              num = x;
43          end
44      end
```

Literaturverzeichnis

[1] Li, Tao, Mitsunori Ogihara und George Tzanetakis: *Music Data Mining*. CRC Press / Taylor & Francis Group, Boca Raton, USA, 1. Auflage, 2012.

[2] Jourdain, Robert: *Das wohltemperierte Gehirn*. Spektrum Akademischer Verlag, Berlin und Heidelberg, Deutschland, 1. Auflage, 2001.

[3] Steiger, Barbara: *Der Gewinner der Eurovision Song Contest: Fragt doch Big Data!* `http://blogs.technet.com/b/microsoft_presse/archive/2013/05/17/der-gewinner-der-eurovision-song-contest-fragt-doch-big-data.aspx`, abgerufen am 18.11.2013.

[4] Schmitt, Peter Philipp: *Dänin gewinnt den Eurovision Song Contest*. `http://www.faz.net/aktuell/gesellschaft/eurovision-song-contest/cascada-enttaeuscht-daenin-gewinnt-den-eurovision-song-contest-12188147.html`, abgerufen am 18.11.2013.

[5] Drux, Rudolf: *Martin Opitz und sein poetisches Regelsystem*. Bouvier Verlag, Bonn, Deutschland, 1976.

[6] Schmieder, Jürgen: *Algorithmus und Pinsel*. Süddeutsche Zeitung, 254:16, 2013.

[7] Sagalnik, M.J., P.S. Dodds und D.J. Watts: *Experimental study of inequality and unpredictability in an artifical cultural market*. Science, 311(5762):854–856, 2006.

[8] MIDI Manufacturers Association: *An Introduction to MIDI*, 2009. `http://www.midi.org/aboutmidi/intromidi.pdf`, abgerufen am 18.11.2013.

[9] Eerola, Tuomas und Petri Toiviainen: *MIDI Toolbox: MATLAB Tools for Music Research*. University of Jyväskylä, Jyväskylä, Finland, 2004. `www.jyu.fi/musica/miditoolbox/`.

[10] Krumhansl, Carol L. und Jack A. Taylor: *Cognitive Foundations of Musical Pitch*. Psychomusicology: Music, Mind & Brain, 11(1):50–63, 1990.

[11] Krumhansl, Carol L. und Edward J. Kessler: *Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys.* Psychological Review, 89(4):334–368, 1982.

[12] Nisbet, Robert, John Elder und Gary Miner: *Handbook of Statistical Analysis & Data Mining Applications.* Academic Press, Amsterdam, Niederlande, 2009.

[13] Alpaydin, Ethem: *Maschinelles Lernen.* Oldenbourg Wissenschaftsverlag, München, Deutschland, 1. Auflage, 2008.

[14] Hastie, Trevor, Robert Tibshirani und Jerome Friedmann: *The Elements of Statistical Learning Data Mining, Inference and Prediction.* Springer Science+Business Media, New York, USA, 2001.

[15] Kreiß, Jens Peter und Georg Neuhaus: *Einführung in die Zeitreihenanalyse.* Springer-Verlag, Berlin und Heidelberg, Deutschland, 1. Auflage, 2006.

[16] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann und Ian H. Witten: *The WEKA Data Mining Software: An Update.* SIGKDD Explorations, 11:Issue 1, 2009.

[17] polyhex.com: *UK Singles - 1952-2013.* `http://www.polyhex.com/music/chartruns/chartruns.php`, abgerufen am 05.08.2013.

[18] Altenmüller, Eckart, Oliver Grewe, Frederik Nagel und Reinhard Kopiez: *Der Gänsehaut-Faktor.* Gehirn&Geist, 6(1):58 – 63, 2007.

[19] Sony Corporation: *12 Tone Analysis Technology.* `http://www.sony.net/SonyInfo/technology/technology/theme/12tonealalysis_01.html`, abgerufen am 13.07.2013.

[20] mididb.com: *Rock MIDI.* `http://mididb.com/rock/`, abgerufen am 13.07.2013.

[21] Tan, Pang Ning, Michael Steinbach und Vipin Kumar: *Introduction to Data mining.* Pearson Education, Boston, USA, 2006.