Technische Universität Berlin
Fakultät I – Geistes- und Bildungswissenschaften
Institut für Sprache und Kommunikation
Fachgebiet Audiokommunikation

Tonhöhenverteilungen von Musikinstrumenten im klassischen Orchesterrepertoire

Masterarbeit

vorgelegt von:

Roman Alexander Quiring

[redacted]

Gutachter:

Prof. Dr. Stefan Weinzierl
Oliver Schwab-Felisch

Abgabedatum:

15. Juli 2016

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind kenntlich gemacht. Ich reiche die Arbeit erstmals als Prüfungsleistung ein.

Roman A. Quiring
Berlin, den 14. Juli 2016

Das Leben kann nur in der Schau nach rückwärts verstanden,
aber nur in der Schau nach vorwärts gelebt werden.

Sören Kierkegaard: „Die Tagebücher 1835 – 1855“

Danksagung

Mein großer Dank gebührt Herrn Prof. Dr. Stefan Weinzierl, der mich bei der Umsetzung dieser Masterarbeit in zahlreichen Sprechstunden mit seinen Ratschlägen unterstützt hat und mir darüber hinaus ermöglichte, erste im Rahmen der Arbeit entstandene Forschungsergebnisse auf der Jahrestagung für Akustik vorzustellen.
Weiterhin danke ich Herrn Oliver Schwab-Felisch herzlichst für die inspirierenden Gespräche und seine wertvollen Anregungen und Vorschläge, die in dieser Arbeit bedauerlicherweise nicht in Gänze umgesetzt werden konnten.
Nicht zuletzt danke ich Herrn Dr. Steffen Lepa für die Bereitstellung seiner Statistik-Expertise und sein selbstlose Hilfsbereitschaft bei der Entwicklung geeigneter Auswertungsmethoden.

Abstract

The examination of pitch distributions of musical instruments is not limited to the field of musicology, but it is also of interest regarding certain acoustic applications. The acoustic behaviour of musical instruments is strongly dependent on pitch with respect to properties like acoustic power, dynamic range, directivity and spectral composition. In this work a corpus study has been conducted, analysing the pitch distributions of 14 classical orchestral instruments in the symphonies of J. Haydn, W.A. Mozart, L.v. Beethoven and F. Schubert. The data basis consists of symphonic works of these composers in a MIDI representation, and a Java software application has been developed to extract pitch distributions. A comparison of the distributions of the different composers indicates individual preferences and historical changes in compositional and orchestrational practices. Further investigations demonstrate the versatile possibilities of application of the acquired data, amongst others for purposes of automatic classification of musical instruments or in the context of virtual acoustic environments.

Kurzfassung

Die Untersuchung von Tonhöhenverteilungen von Musikinstrumenten ist nicht nur vor musikwissenschaftlichem Hintergrund, sondern auch bezüglich akustischer Anwendungen von Interesse. So ist das akustische Verhalten von Musikinstrumenten im Hinblick auf Eigenschaften wie Schallleistung, Dynamikumfang, Richtcharakteristik oder spektrale Zusammensetzung stark von der gespielten Tonhöhe abhängig. In dieser Arbeit wurde eine Korpusstudie durchgeführt, in der die Tonhöhenverteilungen von 14 klassischen Orchesterinstrumenten in den Sinfonien der Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert analysiert wurden. Als Datengrundlage dienen sinfonische Werke in einer MIDI-Repräsentation, zur Berechnung der Tonhöhenverteilungen wurde eine Softwareanwendung in Java entwickelt. Ein Vergleich der Verteilungen der verschiedenen Komponisten weist auf individuelle und entwicklungshistorische Veränderungen in der Kompositions- und Instrumentationspraxis hin. Weitere Untersuchungen zeigen die vielseitigen Einsatzmöglichkeiten der Daten, die unter anderem zur automatischen Klassifikation von Musikinstrumenten oder im Kontext virtueller akustischer Umgebungen verwendet werden können.

Inhaltsverzeichnis

Inhaltsverzeichnis **I**

Abbildungsverzeichnis **III**

Tabellenverzeichnis **IV**

1 Einleitung **1**
1.1 Ziel und Aufbau dieser Arbeit 3

2 Vorbetrachtungen **5**
2.1 Korpusstudien 5
2.2 Musikrepräsentationen 6
2.2.1 MIDI 8

3 Methoden **12**
3.1 Korpuserstellung 12
3.1.1 Datengrundlage 13
3.1.2 Informationsextraktion 15
3.1.3 Datenstrukturierung und -reduktion 19
3.1.4 Fehlerschätzung 21
3.2 Vergleichende Untersuchungen 23
3.2.1 Grafische Auswertungen 24
3.2.2 Varianzanalyse 26
3.3 Klassifikation von Instrumenten 28
3.3.1 Maschinelles Lernen 28
3.3.2 Ausgewählte Klassifikationsverfahren 29
3.3.3 Vorgehensweise 30

4 Ergebnisse **32**
4.1 Korpus 32
4.1.1 Instrumente 34
4.1.2 Fehlerschätzung 34
4.1.3 Fazit 36

4.2 Vergleichende Untersuchungen 37
4.2.1 Tonhöhe 39
4.2.2 Tonumfang 53
4.2.3 Fazit 58
4.3 Klassifikation von Instrumenten 59
4.3.1 Fazit 64

5 Zusammenfassung und Ausblick 65

Literaturverzeichnis V

Anhang: Quellcode X

Abbildungsverzeichnis

3.1 Illustration einer Kerndichteschätzung mit Gauß-Kern und Bandbreite 6. Die Werte sind durch Kreuze markiert, die gestrichelten farbigen Linien entsprechen den einzelnen Kernen und die schwarze Linie der (unnormierten) Gesamtfunktion. 25

4.1 Summierte Tonhöhenverteilungen der Orchesterinstrumente in allen untersuchten Sinfonien für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert. Die Boxplots zeigen den Median, den Interquartilabstand und den gesamten Tonumfang der Instrumente. . . 38
4.2 Tonhöhenverteilungen der ersten Violine für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich. 39
4.3 Tonhöhenverteilungen der zweiten Violine für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich. . . 40
4.4 Tonhöhenverteilungen der Viola für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich. 40
4.5 Tonhöhenverteilungen des Violoncello für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich. 41
4.6 Tonhöhenverteilungen des Kontrabass für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich. 41
4.7 Residuen des statistischen Modells der mittleren Tonhöhe, aufgetragen über der Tonhöhe. 43
4.8 Histogramm der Residuen des statistischen Modells der mittleren Tonhöhe mit durch Kerndichteschätzung gewonnener Dichtefunktion. . . 44
4.9 Vergleich der tatsächlichen und angepassten Tonhöhenwerte. 44
4.10 Tatsächliche und angepasste mittlere Tonhöhe der Instrumente, aufgetragen über die gesamte Messreihe. 46
4.11 Tatsächliche und angepasste mittlere Tonhöhe der Instrumente, aufgetragen über die Werknummer. 46
4.12 Tatsächliche und angepasste mittlere Tonhöhe der Instrumente bei den Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert. . 48
4.13 Residuen des statistischen Modells des mittleren Tonumfangs. 54
4.14 Histogramm der Residuen des statistischen Modells des mittleren Tonumfangs mit durch Kerndichteschätzung gewonnener Dichtefunktion. 54
4.15 Tatsächlicher und angepasster mittlerer Tonumfang der Instrumente, aufgetragen über die gesamte Messreihe. 56

4.16 Tatsächlicher und angepasster mittlerer Tonumfang der Komponisten, aufgetragen über die Werknummer. 57
4.17 Tatsächlicher und angepasster mittlerer Tonumfang der Instrumente, aufgetragen über die Werknummer. 57
4.18 Korrelationsmatrix der zur Klassifikation verwendeten Features. 60
4.19 Genauigkeit und Cohens Kappa der angewandten Klassifikationsverfahren zum Konfidenzniveau 95%. 61
4.20 Genauigkeit des Random Forest-Verfahrens bei unterschiedlicher Anzahl ausgewählter Prädiktoren. 61
4.21 Gewicht der einzelnen Features bei der Verwendung des Random Forest-Verfahrens. 62
4.22 Wahrheitsmatrix der Ergebnisse des Random Forest-Verfahrens. 63

Tabellenverzeichnis

3.1 Tonumfang der untersuchten Orchesterinstrumente, angegeben in ISO-Tonhöhen und MIDI-Nummern. 22

4.1 Analysiertes Repertoire. 33
4.2 Anzahl der Datensätze für die einzelnen Instrumente und Komponisten. 33
4.3 Für die Fehlerschätzung verwendeten Stichproben. 35
4.4 Konfidenzintervalle der geschätzten Fehlerraten der einzelnen Autoren zum Konfidenzniveau 95%, berechnet nach dem Clopper-Pearson-Verfahren und anhand der hypergeometrischen Verteilung. 35
4.5 Modellanpassung für die mittlere Tonhöhe. 42
4.6 Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Haydn. 49
4.7 Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Mozart. 50
4.8 Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Beethoven. 51
4.9 Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Schubert. 52
4.10 Modellanpassung für den mittleren Tonumfang. 53
4.11 Parameterschätzer: Angepasster mittlerer Tonumfang im Vergleich zu Haydn. 55
4.12 Datengrundlage der Klassifikation. 59
4.13 Kenngrößen des Random Forest-Klassifikators. 63

Einleitung

Orchestermusik ist eine Form des künstlerisch-musikalischen Ausdrucks, die seit Jahrhunderten praktiziert und weiterentwickelt wird. Ihre Themen und Anwendungen sind vielseitig: Von den geistlich-religiösen Hintergründen eines Requiems erstreckt sich ihr Einsatzgebiet bis zur musikalischen Untermalung von Film- und Fernsehproduktionen. Im Zusammenspiel des Orchesters werden Kunst und Unterhaltung vereint, die Klänge der einzelnen Instrumente fügen sich zu einem Ganzen zusammen und werden zu einem vielfältigen Ausdrucksmittel, welches Orte zu beschreiben und Gefühle zu transportieren vermag[1]. Als solch vielfältiges Kulturgut erfreut sich Orchestermusik nicht nur großen gesellschaftlichen und wirtschaftlichen Interesses, sondern ist auch Gegenstand zahlreicher wissenschaftlicher Untersuchungen unterschiedlichster Forschungsgebiete. Diese reichen von der Kompositionslehre und musiktheoretischen Analysen des Tonmaterials über die Betrachtung seiner akustischen Merkmale und damit verbundener Konsequenzen für die Gestaltung von Aufführungsräumen bis zur Erforschung psychologischer Phänomene wie der Zusammenhänge von Musik und Emotionen [1] oder intermodaler Korrespondenzen zwischen bestimmten Orchesterwerken und den geschmackssinnlichen Eindrücken verschiedener Weinsorten [2].

Die zunehmende Digitalisierung macht auch vor der orchestralen Aufführungspraxis und ihrer Rezeption nicht halt: Musik wird mit Hilfe computergestützter Eingabeverfahren komponiert, Korrekturen erfolgen per Mausklick. Dirigierpartituren werden inzwischen nicht mehr umgeblättert sondern per Tablet navigiert, und auch die Werke selbst unterliegen mit zunehmender Verwendung elektroakustischer und elektronischer Verfahren der Klangerzeugung einem gewissen digitalen Wandel. Auf Seiten der Rezipienten ist dieser Wandel ebenfalls spürbar: Konzerterlebnisse in multimedialen oder gänzlich virtuellen Umgebungen sind keine Seltenheit, und

[1] Man denke beispielsweise an die „Szene am Bach“ oder „Frohe und dankbare Gefühle nach dem Sturm“ in Ludwig van Beethovens 6. Sinfonie (Pastorale).

auch der private Musikkonsum erfordert nicht mehr den Besitz eines physikalischen Tonträgers, sondern lediglich den Zugang zu digitalen Musikarchiven.

Der Musikforschung bietet das digitale Zeitalter neue Möglichkeiten der Datenerhebung und -auswertung. Das Internet ermöglicht nicht nur den Austausch, sondern auch das Zusammenführen von Daten in digitalen Sammlungen von Musikaufzeichnungen und Notenmaterial. Huron nimmt an, dass im nächsten Jahrzehnt nahezu alle je entstandenen kommerziellen Tonaufnahmen online verfügbar gemacht werden, und die Digitalisierung von 90% der nichtkommerziellen Tonarchive, beispielsweise aus dem Bereich des Rundfunks, in etwa 50 Jahren abgeschlossen sein könnte [3, p. 7]. Die statistische Untersuchung repräsentativer digitaler Datensammlungen nach bestimmten Kriterien – die Korpusanalyse – gewinnt daher im Kontext musikwissenschaftlicher Forschung zunehmend an Bedeutung. Die Verfügbarkeit großer Mengen an historischem Tonmaterial erlaubt beispielsweise die Untersuchung musikalisch-stilistischer Entwicklungen über verschiedene musikalische Epochen[2].

Mit Korpusstudien verknüpfte Betrachtungen sind allerdings keineswegs auf das Gebiet der historischen Musikwissenschaft beschränkt. Im Forschungsgebiet des *Music Information Retrieval* kommt der Entwicklung von Methoden zur Identifkation von Instrumenten [6], einzelner Töne [7] und Akkorden [8] sowie der Evaluation relevanter Methoden [9] eine große Bedeutung zu. Diesen Identifikationsmethoden liegen oftmals statistische Modelle und Distanzmaße zugrunde, welche ihren Ursprung ebenfalls in der Auswertung großer Datensammlungen haben. Weiterhin werden Korpora zur Klassifikation musikalischer Genres eingesetzt: So verwenden Tzanetakis, Ermolinskyi und Cook Tonhöhenhistogramme von Ausschnitten musikalischer Werke zur Unterscheidung fünf verschiedener Stilrichtungen [10]. Ähnlich dieses Vorgehens benutzen Dighe, Karnick und Raj derartige Histogramme für die Entwicklung von Methoden zur Beschreibung struktureller Eigenschaften und der Identifikation indischer Ragas [11], welche mit einer Identifikationsrate von über 94% eine beachtliche Genauigkeit aufweisen. Mardirossian und Chew greifen bei der Entwicklung allgemeiner musikalischer Ähnlichkeitsmaße ebenfalls auf die von Tzanetakis et al. vorgestellten Tonhöhenhistogramme zurück [12].

Die Untersuchung tonhöhenabhängiger musikalischer Merkmale ist darüber hinaus auch bei der akustischen Beschreibung von Musikinstrumenten von Interesse.

[2]So stellt White beim Vergleich der Korpora von 19 verschiedenen Komponisten fest, dass chronologisch nahe Repertoires eine hohe Korrelation hinsichtlich der Verwendung von Progressionen und damit eine kompositorische Ähnlichkeit aufweisen [4]. Albrecht und Huron untersuchen die historische Entwicklung von Dur- und Molltonarten anhand ausgewählter Werke von 259 Komponisten im Zeitraum vom 15. bis zum 18. Jahrhundert [5].

Wie Weinzierl feststellt, sind akustische Eigenschaften wie z.B. die Einschwingzeit von Orchesterinstrumenten bei der Tonproduktion stark von der gespielten Tonhöhe abhängig [13, p. 185], was sich auf die abgegebene Schallleistung auswirkt. Auch das Abstrahlverhalten ist in hohem Maße tonhöhenabhängig[3]. Die Dokumentation dieser tonhöhenabhängigen Eigenschaften von Musikinstrumenten ist auch im Kontext virtueller akustischer Umgebungen von Interesse, wo sie bei der Simulation einzelner Instrumente oder ganzer Orchester berücksichtigt werden können. Die Gewinnung entsprechender Kennwerte kann anhand korpusanalytischer Methoden erfolgen, bei denen aus einem musikalischen Repertoire repräsentative Mittelwerte für die verschiedenen Instrumente extrahiert werden.

1.1 Ziel und Aufbau dieser Arbeit

Diese Arbeit soll einen Beitrag zur Schaffung einer Datengrundlage im Hinblick auf die Tonhöhenverteilungen von Musikinstrumenten im klassischen Orchesterrepertoire leisten. Untersuchungsgegenstand sind Tonhöhenverteilungen von Orchesterinstrumenten in den sinfonischen Repertoires von vier Komponisten der Wiener Klassik. Es soll zunächst ein musikalisches Korpus erstellt werden, das eine repräsentative Auswahl der Sinfonien von Joseph Haydn (1732–1809), Wolfgang Amadeus Mozart (1756–1791), Ludwig van Beethoven (1770–1827) und Franz Schubert (1797–1828) enthält. Daraus werden anschließend die Tonhöhenverteilungen der einzelnen Instrumente extrahiert, welche unter Verwendung der von Tzanetakis et al. beschriebenen Methoden ebenfalls durch Tonhöhenhistogramme beschrieben werden sollen. In Anlehnung an die bereits erwähnten entwicklungshistorischen Betrachtungen vergangener Korpusstudien sollen die Tonhöhenverteilungen der Instrumente bei den verschiedenen Komponisten anschließend mit geeigneten statistischen Verfahren auf Unterschiede untersucht werden. Dabei sollen historisch-kompositorische Entwicklungen in der Verwendung der verschiedenen Orchesterinstrumente im Hinblick auf ihre Verwendungshäufigkeit wie auch ihre Tonhöhe und ihren Tonumfang aufgezeigt werden.

Zur Untersuchung der individuellen Präferenzen und kompositionshistorischen Entwicklungen der Instrumentationspraxis werden folgende Arbeitshypothesen angenommen:

[3] Bei Blechblasinstrumenten beispielsweise nimmt die Schallbündelung mit steigender Frequenz zu, da das Verhältnis von Strahlerfläche und Wellenlänge steigt. Bei Holzblasinstrumenten wird durch das Öffnen und Schließen der Grifflöcher bei der Tonproduktion zudem die Geometrie des Strahlers selbst verändert; ihre Richtcharakteristik ändert sich somit je nach produziertem Ton [13, p. 194].

1. Instrumente weisen bestimmte individuelle und entwicklungshistorische Tendenzen in der Verwendungshäufigkeit auf.

2. Es existieren bei den verschiedenen Komponisten signifikante Unterschiede hinsichtlich der mittleren Tonhöhe der Instrumente.

3. Der mittlere Tonumfang der Orchesterinstrumente weist bei den verschiedenen Komponisten ebenfalls signifikante Unterschiede auf.

Darüber hinaus soll die Eignung der Daten für Zwecke der automatischen Klassifikation von Orchesterinstrumenten anhand statistischer Parameter des von ihnen gespielten Tonmaterials geprüft werden. Dabei werden verschiedene Klassifikationsverfahren auf die Daten angewandt und ihre Ergebnisse miteinander verglichen.

Im ersten Teil dieser Arbeit werden zunächst die der Korpusanalyse zugrunde liegenden Vorbetrachtungen dargelegt. Hier wird auf die methodischen Prinzipien und allgemeinen Kriterien für der Auswahl geeigneter Ausgangsdaten sowie auf die als Korpusgrundlage in Frage kommenden Datenformate bzw. Musikrepräsentationen eingegangen. Im darauf folgenden Kapitel wird das methodische Vorgehen beschrieben: Es erläutert die verschiedenen Schritte der Korpuserstellung wie auch die der Auswertung des Datenmaterials zugrunde liegenden Verfahren, bevor im Folgekapitel die Ergebnisse der verschiedenen Untersuchungen hinsichtlich der oben genannten Fragestellungen dargestellt und zusammengefasst werden. Abschließend erfolgt eine Zusammenfassung der Forschungsergebnisse und eine Evaluation der ihnen zugrunde liegenden Methoden sowie die Betrachtung zukünftiger Einsatzmöglichkeiten der gewonnenen Daten und entwickelten Verfahren.

Vorbetrachtungen

In diesem Kapitel werden die bei der Durchführung von Korpusanalysen notwendigen Vorbetrachtungen dargelegt. Zunächst werden die methodischen Grundlagen von Korpusstudien und die Verfahren zur Auswahl einer geeigneten Datengrundlage erläutert. Anschließend werden verschiedene Musikrepräsentationen vorgestellt, die als Grundlage musikalischer Korpora in Betracht kommen.

2.1 Korpusstudien

Wie bereits in der Einleitung skizziert, werden Korpusstudien zur Überprüfung verschiedenster Hypothesen hinsichtlich struktureller inhaltlicher Merkmale des jeweils betrachteten Korpus eingesetzt. Je nach Forschungsgegenstand kann das Korpus daher bezüglich Umfang und Inhalt variieren. Dementsprechend existieren auch keine universellen Methoden zur Erstellung und Analyse musikalischer Korpora, was eine sorgfältige Auswahl sowohl der dem Korpus zugrunde liegenden Daten als auch der damit verbundenen Auswertungsmethoden erfordert. Folgt man Gjerdingen [14, p. 193], so beinhaltet ein allgemeiner Leitfaden für die Durchführung einer Korpusstudie folgende drei Punkte: „(1) Die Definition einer Menge von Elementen von denen angenommen wird, dass sie im Korpus auftreten, (2) die Berechnung angemessener Statistiken über der Zeitreihe dieser Elemente, und (3) die Ableitung entsprechender im Korpus vorherrschender Normen oder Regeln“.

Insbesondere der erste Punkt ist für die Validität der Ergebnisse entscheidend: Die Auswahl der zu untersuchenden Elemente sollte anhand zuvor definierter Kriterien erfolgen, die sich in erster Linie an der vorliegenden Fragestellung orientieren. Von großer Bedeutung ist dabei die Repräsentativität der Daten, die unter quantitativen und qualitativen Gesichtspunkten zu berücksichtigen sind. So muss ein Korpus eine hinreichende Größe aufweisen, um überhaupt allgemeine Schlüsse auf die Grundgesamtheit zulassen zu können. London [15, p. 78] schlägt für musikalische

Korpusstudien in Bezug auf die *common practice period*, d.h. die Perioden Barock, Klassik und Romantik, eine minimale Korpusgröße von 300 Werken vor. Dieser Wert kann als Orientierung angesehen werden, die jedoch nicht zwangsläufig auf andere Untersuchungsgegenstände übertragen werden muss. In der Praxis ist die Korpusgröße zudem oftmals durch die Menge an verfügbarem Datenmaterial begrenzt.
Weiterhin müssen die analysierten Elemente auch die inhaltlichen Aspekte der Grundgesamtheit widerspiegeln: Sollen beispielsweise Aussagen über das Gesamtwerk von W.A. Mozart getroffen werden, ist die alleinige Betrachtung seiner Klaviersonaten unzureichend. Das verwendete Korpus sollte stattdessen z.B. auch Sinfonien und Streichquartette umfassen, deren Anzahl ihrem prozentualen Anteil am Gesamtrepertoire des Komponisten entspricht. Zufallsstichproben sind die beste Möglichkeit zur Vermeidung systematischer Verzerrungen, die durch eine willkürliche Auswahl der Daten entstehen können, wie Kauermann und Küchenhoff feststellen [16, p. 9–11].
Huron spricht darüber hinaus mit der Akkumulierung von Messfehlern einen weiteren wichtigen Punkt hinsichtlich der Validität der Datengrundlage an [3, p. 7]: Während beispielsweise bei kodierten Tonhöhen eine Fehlerrate von 1% nur marginale Auswirkungen auf die Schätzung der Tonart hat, erhöht sie sich bei der Identifikation von Vierklängen bereits auf 4%, bei der Betrachtung von harmonischen Progressionen solcher Akkorde gar auf 8%. Schon geringfügige Fehler im verwendeten Ausgangsmaterial können somit große Auswirkungen auf die Ergebnisqualität der nachfolgenden statistischen Analysen haben.

Bei der Durchführung von Korpusstudien sind die Ausgangsdaten daher unbedingt einer kritischen Betrachtung zu unterziehen. Mögliche darin enthaltene Fehler müssen dokumentiert und in den Ergebnissen berücksichtigt werden.

2.2 Musikrepräsentationen

Bei der digitalen Darstellung musikalischer Inhalte ist zwischen zwei grundlegend verschiedenen Repräsentationsformen zu unterscheiden: Audiodaten und symbolische Repräsentationen von Musik.

Audiodaten, in der Regel das Ergebnis von Tonaufnahmen, beinhalten konkrete Informationen über den Klang, also die akustische Gestalt musikalischer Inhalte. Explizite semantische Informationen über das Material liegen hier allerdings nicht vor, so dass die Identifikation musikalischer Strukturen und Zusammenhänge nicht ohne Weiteres möglich ist. Stattdessen müssen diese Informationen durch komplexe computergestützte Verfahren aus dem Audiomaterial extrahiert werden.

In symbolischen Musikrepräsentationen hingegen wird das Material nicht durch seine auditive Manifestation beschrieben, sondern anhand seiner semantisch-strukturellen Gestalt. Musik wird hier als Folge symbolischer Einheiten aufgefasst, die erst durch deren Interpretation in hörbare Klänge überführt werden können[1]. Dieses gleichsam analoge Format ist für computergestützte automatisierte Analysemethoden jedoch nur bedingt verwendbar, da wie bei den Audiodaten auch hier zunächst die Inhalte interpretiert und in eine entsprechende digitale Form überführt werden müssen bevor sie verarbeitbar sind[2]. Insofern sind Repräsentationen zu bevorzugen, bei denen das Material bereits in einem maschinenlesbaren Format vorliegt. Im digitalen Notensatz verwendete Datenformate sind in diesem Kontext von besonderem Interesse, da die Inhalte hier bereits in digitaler Form vorliegen und nicht erst in eine solche transformiert werden müssen. Es existiert eine Vielzahl verschiedener Formate, die für verschiedene Anwendungszwecke entworfen wurden und daher unterschiedliche Stärken und Schwächen aufweisen. Im Folgenden soll daher ein kurzer Überblick über die gebräuchlichsten Musikrepräsentationen gegeben werden, die im digitalen Notensatz Verwendung finden.

Bei der 1993 von Chris Walshaw entwickelten *ABC*-Notation [18] handelt es sich um ein schlichtes, menschenlesbares textbasiertes Format, das zur Beschreibung monophoner Inhalte wie einstimmiger Melodien entworfen wurde und insbesondere zur Notation von Volksliedern eingesetzt wird. Mit später entwickelten Erweiterungen ist auch die Notation mehrstimmiger Inhalte möglich; diese erweiterten Standards werden von den gängigen Notensatzprogrammen allerdings nicht implementiert [19] und daher in der Praxis selten verwendet.

Das freie Notationsprogramm *LilyPond* [20] verwendet ein gleichnamiges textbasiertes Eingabeformat. Anders als die ABC-Notation wurde dieses Format für die Erstellung komplexer, polyphoner Werke entworfen: Es bietet dem Notensetzer daher bei der Kodierung des Materials deutlich mehr Ausdrucks- und Gestaltungsmöglichkeiten. Da das Format auf die Verarbeitung durch die LilyPond-Software zugeschnitten ist und daher sehr spezielle syntaktische und semantische Eigenheiten aufweist, ist seine Verwendung außerhalb der entsprechenden Anwendungen

[1] Nach dieser Definition stellt auch die traditionelle Notenschrift eine symbolische Repräsentation dar; die akustische Realisation der grafisch notierten Inhalte erfolgt erst durch den Interpreten.

[2] Es existieren verschiedene Verfahren für eine solche optische Musikerkennung (engl. *optical music recognition, OMR*), deren Ergebnisqualität je nach vorliegendem Material stark variieren kann: Schließlich muss unter Berücksichtigung unterschiedlicher Schriftarten eine zuverlässige Differenzierung von teils sehr ähnlichen Symbolen wie Halte- und Bindebögen erfolgen. Die damit verbundenen Aufgaben sind entsprechend komplex und stellen eine große Herausforderung dar, wie Bainbridge und Bell feststellen [17].

allerdings unüblich und die Verbreitung dementsprechend gering.

Anders als die bereits genannten symbolischen Musikrepräsentationen dient das seit 2004 kontinuierlich weiterentwickelte XML-basierte Format *MusicXML* [21] vor allem als Austauschformat zwischen verschiedenen Notensatzprogrammen: Ein mit einer bestimmten Software erstelltes Werk kann so mit einem gänzlich anderen Programm dargestellt und bearbeitet werden, sofern dieses den MusicXML-Spezifikationen folgt. Aufgrund dieser Unabhängigkeit von bestimmten Systemen oder Anwendungen ist MusicXML eine einfach zu handhabende und weit verbreitete Musikrepräsentation, deren einziger Nachteil in den großen inhaltlichen Redundanzen liegt, die dem zugrundeliegenden XML-Format geschuldet sind und sich in einem hohen Datenvolumen niederschlagen.

Neben der Übertragung in das MusicXML-Format bieten viele Notensatzprogramme Möglichkeiten zum Export der musikalischen Inhalte in die *MIDI*-Repräsentation, der im Rahmen dieser Arbeit besondere Bedeutung zukommen wird und die daher im folgenden Unterkapitel näher erläutert werden soll.

2.2.1 MIDI

Musical Instrument Digital Interface (MIDI) bezeichnet eine Reihe von Protokollen und Spezifikationen zum Austausch digitaler musikalischer Informationen [22]. 1983 erstmals veröffentlicht, lag der ursprünglicher Zweck dieser Repräsentation in der Kommunikation zwischen digitalen Synthesizern verschiedener Hersteller (Vgl. [23, p. 11]). Die Spezifikationen wurden jedoch im Laufe der Zeit erweitert und ihr Gebrauch auf andere Musikinstrumente und Anwendungen übertragen, so dass sie im Bereich der digitalen Klangsynthese zum Industriestandard wurden. Während in den ersten Spezifikationen lediglich Schnittstellen zum Austausch von Informationen definiert waren, wurde mit den *General MIDI Specifications* (GM) in den Folgejahren auch die konkrete Gestalt der eigentlichen Inhalte standardisiert. Die im Folgenden beschriebenen Parameter und Wertebereiche beziehen sich auf diesen Standard.

In der MIDI-Repräsentation werden sämtliche musikalischen Ereignisse als Stream sogenannter *Events* kodiert. Diese bestehen aus einem oder mehreren Bytes. Das erste Byte ist stets das sogenannte `STATUS`-Byte und gibt den Typ des Ereignisses an. Der eigentliche Inhalt wird – sofern derart atomare Informationen zur Beschreibung von Inhalten nicht ausreichen – in den darauf folgenden `DATA`-Bytes kodiert. Je nach Ereignistyp variieren Anzahl und Inhalte dieser Bytes. Im Folgenden werden relevante Events näher erläutert, wobei namentliche Bezüge stets das `STATUS`-Byte betreffen und inhaltliche Angaben den in den `DATA`-Bytes gespeicherten Informationen entsprechen.

Neben der direkten Erzeugung von MIDI-Daten im Rahmen von Live-Performances können diese auch aufgezeichnet oder automatisch generiert und als sogenannte *Standard MIDI Files* (SMF) mit der Dateiendung `.mid` gespeichert werden. Die von einem Instrument erzeugten Events werden dabei in einem *MIDI-Track* zusammengefasst. Dabei können drei verschiedene Dateiformate zum Einsatz kommen:

- SMF 0: Hier werden alle Events in einem einzigen Track gespeichert. Das Format eignet sich damit besonders für monophone Inhalte.
- SMF 1: Alle Stimmen werden in separaten, zeitlich synchronisierten Tracks gespeichert. Format 1 ist for allem bei der Betrachtung zusammenhängender mehrstimmiger Inhalte, wie der zu untersuchenden Orchesterwerke, von Interesse.
- SMF 2: Dieses Format wird selten verwendet. Es enthält mehrere zeitlich unabhängige Tracks vom Format 0 und kann z.B. der Speicherung verschiedener Rhythmusschemata eines Drumcomputers dienen.

Bei den einem Track zugeordneten Ereignissen wird zwischen *META-Events* und *MIDI-Events* unterschieden. Während erstere Gruppe ausschließlich zur Kodierung von Metainformationen und Textinhalten verwendet wird, entsprechen MIDI-Events den eigentlichen musikalischen Inhalten.

Innerhalb des MIDI-Tracks werden die MIDI-Events durch Angabe einer *channel number* auf einem von 16 Kanälen 'gesendet', was bei der Klangwiedergabe die Verwendung unterschiedlicher Syntheseparameter, bzw. *MIDI-Instrumente* erlaubt. Mit `PROGRAM`-Events wird den Kanälen jeweils eines von 128 MIDI-Instrumenten zugewiesen, deren Verwendung vielfältige klangliche Ausdrucksmöglichkeiten innerhalb einer Stimme ermöglicht: Mit Blick auf die Darstellung von Orchestermaterial erlauben sie beispielsweise die Differenzierung gedämpfter Klänge oder auch von Pizzicato-Sektionen der Streicher. Kanal 10 ist ausschließlich perkussiven Inhalten vorbehalten und wird daher auch als *drum channel* bezeichnet.

Töne werden im MIDI-Format durch `NOTE`-Events repräsentiert. Dabei wird unterschieden zwischen `NOTE ON`-Events, die das Erklingen eines Tons markieren, und `NOTE OFF`-Events, welche den Zeitpunkt seines Abklingens bestimmen. Beide Events enthalten neben der obligatorischen *channel number* auch Informationen über den tonalen Inhalt in Form der Parameter *note number* und *velocity*.

Die *note number* gibt die absolute Tonhöhe an. Sie wird durch einen von 128 diskreten Werten ausgedrückt, deren Abstände den Halbtonschritten der chroma-

tischen Tonleiter entsprechen. Der darstellbare Tonumfang umfasst in der MIDI-Repräsentation somit mehr als 10 Oktaven, mit Werten von 0 / C_{-1} bis 127 / G_9 und Grundfrequenzen von 8,176 Hz bis 12544 Hz. In vielen Anwendungsfällen ist eine Umrechnung der Nummern in gängigere Tonsymbole wie die im Folgenden verwendete ISO-Tonhöhe [24] wünschenswert. Die Tonbezeichnung Ton_n und die zugehörige Oktavlage $Oktave_n$ eines `NOTE`-Events mit der *note number* n werden wie folgt berechnet:

$$\begin{aligned} Ton_n &= n \bmod 12 \\ Oktave_n &= \left\lfloor \frac{n}{12} \right\rfloor - 1 \end{aligned} \tag{2.1}$$

Mit dem Parameter *velocity* werden Intensität und Dynamik eines Klanges ausgedrückt. Seine Bezeichnung leitet sich von der Anschlagsgeschwindigkeit bei der Verwendung von Tasteninstrumenten ab. Üblicherweise wird dieser Parameter als Lautstärkewert verstanden, die *velocity* kann bei der Klangsynthese je nach Anwendungsfall jedoch auch zur Beeinflussung anderer klanglicher Eigenschaften des produzierten Tons verwendet werden. Dabei ist zu unterscheiden zwischen der *velocity*-Angabe von `NOTE ON`-Events, die den Einschwingvorgang beschreibt, und der *release velocity* von `NOTE OFF`-Events, welche umgekehrt das Abklingen ausdrückt. Wie die Tonhöhen wird auch die *velocity* durch 128 diskrete Schritte beschrieben. In der GM-Spezifikation wird sie als Lautstärkewert interpretiert: Ein Wert von 0 bedeutet einen unhörbaren Ton, 128 hingegen ist der 'lauteste' Wert. Die Spezifikation führt weiterhin an, dass `NOTE ON`-Events mit einem *velocity*-Wert von 0 wie `NOTE OFF`-Events behandelt werden, was im Ergebnis eine Verringerung des Datenvolumens nach sich zieht. Mangels eines expliziten Standards zur Übersetzung konventioneller Dynamikangaben wie *piano* oder *forte* in *velocity*-Werte variiert ihre konkrete Implementierung bei den verschiedenen Softwareanwendungen.

Bei der Wiedergabe musikalischer Inhalte ist neben der Tonhöhe und den klanglichen Charakteristika auch die Angabe der Tondauer von Interesse. In der MIDI-Repräsentation wird die Zeit in diskreten Einheiten, sogenannten *Ticks* ausgedrückt. Jedes Event ist mit einem Zeitstempel versehen, der den exakten Zeitpunkt seines Auftretens markiert. Die Dauer tonaler Ereignisse ergibt sich somit aus der Differenz der Zeitstempel von `NOTE ON`- und `NOTE OFF`-Events. Es können dabei zwei unterschiedliche Zeitformate verwendet werden: Der von der Society of Motion Picture and Television Engineers eingeführte *SMPTE-Timecode*, der vor allem zur Synchronisation von Audio- und Videomaterial verwendet wird, und die relative Zeitangabe *Pulses*

Per Quarter Note (PPQN), welche die Dauer einer Viertelnote in Ticks ausdrückt und dadurch die zeitliche Auflösung der Daten bestimmt.

Mit dem Zeitmaß sind auch das Tempo und die metrische Struktur der musikalischen Sequenzen verbunden. Taktangaben werden in Notenschrift üblicherweise als Bruch dargestellt: Der Zähler gibt dabei die Anzahl an Zählzeiten pro Takt an, der Nenner legt den Notenwert der Zählzeiten fest. Dieser Konvention folgt auch die MIDI-Repräsentation, in der Taktangaben durch ein `TIME SIGNATURE`-Meta-Event ausgedrückt werden. Zähler und Nenner sind darin in jeweils einem Daten-Byte numerisch kodiert. Da bei MIDI-Daten im Format 1 alle Stimmen getrennt aufgeführt sind, würden wiederholte Angaben von Metrum und Tempo in den einzelnen Stimmen hier eine hohe inhaltliche Redundanz zur Folge haben, die es im Hinblick auf zeitkritische Anwendungsgebiete zu vermeiden gilt, in denen ein möglichst geringes Datenvolumen angestrebt wird. Deshalb werden diese Angaben beim SMF 1 ausschließlich in einem 'künstlichen' ersten Track definiert, der neben besagten Metainformationen keine weitere Inhalte hat. Bei der Wiedergabe wird nun während der Verarbeitung des ersten Tracks eine „Tempo-Karte“ (engl. *tempo map*) erstellt und zur Übertragung der Tempo- und Taktangaben auf die folgenden Tracks bzw. ihre musikalischen Inhalte verwendet.

Methoden

In diesem Kapitel werden die bei der Korpusanalyse verwendeten Methoden beschrieben. Bei der Erstellung des Korpus kommen die Programmiersprachen *Java* sowie die Software-Entwicklungsumgebung *MATLAB* zum Einsatz. Die verschiedenen Auswertungen des Materials erfolgen unter Verwendung der quelloffenen, frei verfügbaren und kostenlosen Programmiersprache *R*, die insbesondere für statistische Aufgaben und computergestützte Datenverarbeitung entwickelt wurde [25]. Für eine Übersicht dieser Programmiersprache und der ihr zugrunde liegenden Konzepte sei auf das Einführungswerk von Daniel Wollschläger verwiesen [26]. Der Quelltext aller eingesetzten Methoden ist dieser Arbeit beigefügt und über das Quellcodeverzeichnis einsehbar.

3.1 Korpuserstellung

In diesem Unterkapitel werden die Datengewinnung und die damit verbundenen methodischen Grundlagen beschrieben. Zunächst soll auf die Form und Herkunft der verwendeten Rohdaten und die damit verbundenen Überlegungen eingegangen werden. Sodann werden Kriterien hinsichtlich der Gültigkeit und Verwendbarkeit der Daten definiert, anhand derer daraufhin die Auswahl der Datengrundlage des zu erstellenden Korpus vorgenommen wird. Anschließend werden die bei der Informationsextraktion eingesetzten Verfahren beschrieben. Die digitale Natur der zu untersuchenden Inhalte erfordert softwarebasierte Analysemethoden; mit Blick auf die dieser Arbeit zugrunde liegenden Fragestellungen werden zunächst die Anforderungen an derartige Verfahren definiert und anschließend deren praktische Umsetzung in Form der für die Datenanalyse erstellten Softwareanwendung erläutert. Schließlich wird auf die Strukturierung und Speicherung der Datensätze eingegangen und die für die Schätzung von Messfehlern notwendigen Betrachtungen werden dargelegt.

3.1.1 Datengrundlage

Grundlage der Datengewinnung soll eine symbolische Musikrepräsentation sein, da diese – wie in Kapitel 2.2 dargelegt – im Gegensatz zu Tonaufnahmen sowohl die zuverlässige Identifikation einzelner Töne als auch der zugehörigen Instrumente erlaubt. Im Hinblick auf die Orchesterwerke der zu untersuchenden Komponisten gilt es zunächst, bestehende Datensammlungen mit entsprechendem Datenmaterial ausfindig zu machen und ihre Inhalte auf Eignung zu prüfen.

Während das MusicXML-Format sich aufgrund seines hohen Informationsgehaltes für inhaltliche Analysen grundsätzlich sehr gut eignet, gibt es in dieser Repräsentation keine größeren, frei zugänglichen Sammlungen von Orchesterwerken der Wiener Klassik. Es lassen sich in diversen Datenbanken zwar einzelne sinfonische Werke der zu betrachtenden Komponisten im MusicXML-Format finden, für die Erstellung eines umfassenden Korpus reichen diese jedoch nicht aus.

Das MIDI-Format hingegen war für lange Zeit der De-facto-Standard für den Notensatz und die digitale Klangsynthese. Dementsprechend viele musikalische Werke sind in diesem Format verfügbar und stehen in verschiedenen Datenbanken gebündelt zur Verfügung. Damit bietet diese Repräsentation eine optimale Grundlage für das Erstellen einer musikalischen Datensammlung. Für die Nutzung im Rahmen dieser Arbeit kommen grundsätzlich zwei Datenbanken in Frage: Neben der Online-Datenbank *Kunst der Fuge* [27], welche sich ausschließlich auf das Angebot von MIDI-Daten beschränkt, bietet die Online-Musikdatenbank *ClassicalArchives* [28] darüber hinaus auch klassische Werke in anderen digitalen Formaten an. Sie verfügt über ein umfangreicheres Angebot an Orchesterwerken im MIDI-Format; sinfonische Werke der zu betrachtenden Komponisten sind hier in großer Zahl und vielen Formaten vertreten. Sämtliche bei *Kunst der Fuge* vorhandenen Werke sind auch bei *ClassicalArchives* verfügbar, so dass diese Datenbank als Ausgangspunkt für die Datenerhebung gewählt wird.

Die bei *ClassicalArchives* in der MIDI-Repräsentation verfügbaren sinfonischen Werke der vier zu betrachtenden Komponisten werden von Privatpersonen kodiert und der Datenbank unentgeltlich zur Verfügung gestellt. In Ermangelung einheitlicher Richtlinien für die Erstellung von MIDI-Daten variieren die Arbeiten verschiedener Autoren daher hinsichtlich der Kodierung des musikalischen Materials. Bevor eine Analyse erfolgen kann, müssen die Daten daher einer strukturellen und inhaltlichen Überprüfung unterzogen werden um ihre Eignung für die Datenerhebung zu gewährleisten. Viele der sinfonischen Werke wurden darüber hinaus von unter-

schiedlichen Autoren kodiert und liegen daher in mehreren Versionen vor; es gilt also diese Ambiguitäten unter Berücksichtigung der individuellen Eigenheiten der Autoren aufzulösen bzw. die Datenmenge entsprechend zu reduzieren. Mit Blick auf die zu bearbeitenden Fragestellungen müssen die Datensätze daher folgende Anforderungen erfüllen:

- strukturelle Vollständigkeit: Alle Sätze einer Sinfonie müssen vorhanden sein.
- Trennbarkeit: Die Sätze müssen einzeln, d.h. in getrennten Dateien, und die verschiedenen Stimmen bzw. Instrumente separat kodiert sein (SMF 1).
- semantische Vollständigkeit: Alle Stimmen bzw. Instrumente müssen durch entsprechende Namensgebung identifizierbar sein.
- musikalische Vollständigkeit: Alle in der Partitur notierten Instrumente und ihr Tonmaterial müssen lückenlos kodiert sein.
- Singularität: Jedes Werk darf im finalen Korpus nur einmal vertreten sein.

In einem ersten Schritt werden also sämtliche Datensätze ausgeschlossen, bei denen nicht alle Sätze einer Sinfonie enthalten oder mehrere Sätze in einer MIDI-Datei zusammengefasst sind (strukturelle Vollständigkeit). Zur Überprüfung der Trennbarkeit des Materials einzelner Instrumentalstimmen wird das Werkzeug `mftext` des quelloffenen Programmpakets *abcMIDI* [29] verwendet, mit dem der Inhalt einer MIDI-Datei in Textform ausgegeben werden kann. Daten im SMF 0-Format können so identifiziert und verworfen werden[1] (Trennbarkeit). Unter Verwendung dieser Textrepräsentation wird in diesem Schritt zudem die inhaltliche Struktur aller Datensätze inspiziert und untersucht, ob alle Instrumentalstimmen mit entsprechenden Namen versehen sind; Datensätze ohne derartige Identifikationsmerkmale werden ausgeschlossen, da eine klare Differenzierung der Inhalte einzelner Instrumente dort nicht ohne weitere Untersuchungen und konkrete inhaltliche Vergleiche mit den Partituren möglich ist (semantische Vollständigkeit). Bei dieser ersten Inspektion der MIDI-Daten wurde festgestellt, dass diese keine `NOTE OFF`-Events beinhalten; das Ende eines Tons wird stattdessen durch ein zweites `NOTE ON`-Event mit einem `velocity`-Wert von 0 markiert.

Schließlich gilt es, die musikalische Vollständigkeit der Daten zu untersuchen: Dabei muss zunächst geprüft werden, ob in den MIDI-Datensätzen auch alle in der

[1] Format 2 findet bei den betrachteten Datensätzen keine Anwendung.

Partitur notierten Instrumentalstimmen repräsentiert sind. Zu diesem Zweck werden die Partituren aller verbleibenden in Frage kommenden sinfonischen Werke der vier Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert von einer weiteren Online-Datenbank, dem *International Music Score Library Project* [30], bezogen, welche Partituren und Stimmauszüge einer Vielzahl musikalischer Werke unterschiedlicher Komponisten zur Verfügung stellt.
Da ein manueller Abgleich der musikalischen Inhalte unter Verwendung der oben genannten Textrepräsentationen in Anbetracht der großen Datenmenge einen erheblichen Zeitaufwand darstellen würde, sollte dieser Schritt automatisiert werden. Es gilt daher, Methoden für die inhaltliche Analyse der MIDI-Daten zu entwickeln.

3.1.2 Informationsextraktion

Es sollen zunächst die grundlegenden Anforderungen an das Verfahren zur Datengewinnung beleuchtet und anschließend ihre Umsetzung beschrieben werden.

Mit Blick auf eine mögliche Wiederverwendung oder Erweiterung im Rahmen weiterer Studien stellen Zugänglichkeit und Modularität die Grundvoraussetzungen sämtlicher Komponenten der zu erstellenden Softwareanwendung dar. Die verwendeten Technologien sollen daher frei verfügbar, quelloffen und nach Möglichkeit unabhängig von bestimmten Softwareumgebungen oder Betriebssystemen anwendbar sein. Hinsichtlich des letzten Punktes gilt es allerdings auch die Kompatibilität mit den in diesem Forschungsbereich bzw. im Fachgebiet Audiokommunikation gängigen Entwicklungsumgebungen zu gewährleisten; insofern soll das zu entwickelnde Verfahren auch eine Verwendung innerhalb der Software-Entwicklungsumgebung *MATLAB* zulassen.

Die Wahl der verwendeten Softwaretechnologien erfolgt aufgrund einer Betrachtung der grundsätzlichen methodischen Anforderungen an die Applikation. Bei der zu erfolgenden Datenanalyse sind folgende Schritte notwendig:

- Sequenzielles Einlesen und Verarbeiten der MIDI-Daten bzw. einzelner Ausschnitte des darin kodierten Materials.

- Registrieren und Verknüpfen der für den Untersuchungsgegenstand wichtigen MIDI-Events: Töne, Tracks und Zeitangaben.

- Überführen der musikalischen Inhalte der betrachteten Bereiche in Tonhöhenverteilungen.

- Bereitstellung der Analyseergebnisse über eine anderen Anwendungen zugängliche Programmierschnittstelle.

Das sequenzielle, d.h. Byte-weise Auslesen der Daten und die Registrierung bzw. Verarbeitung der zu untersuchenden Inhalte sind grundsätzlich mit jeder Programmiersprache bzw. in jeder Umgebung möglich, so auch mit MATLAB. Hier stehen diverse Programmbibliotheken wie die *MIDI Toolbox* [31] zur Verfügung, die von diesem „Byte-Stream“ abstrahieren und die kodierten musikalischen Inhalte über Schnittstellen bereitstellen, bzw. bezüglich diverser musikalischer und psychoakustischer Parameter wie Tonart oder melodische Erwartung analysieren können. Die Verwendung solcher generischer Programmbibliotheken ist anzustreben, da der Gebrauch bestehender Funktionalitäten die Implementierung erheblich vereinfacht. Viele der Analysemethoden der MIDI Toolbox sind jedoch auf monophone Inhalte beschränkt, zudem werden die Track-Namen, welche für die Verknüpfung der Tonhöhenverteilungen mit den entsprechenden Instrumenten von zentraler Bedeutung sind, hier nicht bereitgestellt. Insofern ist die MIDI Toolbox für die relevanten Betrachtungen ungeeignet.

Neben der Analyse des gesamten Inhalts der MIDI-Daten – d.h. dem Einlesen ganzer Dateien, die im vorliegenden Fall den einzelnen sinfonischen Sätzen entsprechen – soll auch die Betrachtung bestimmter Teilbereiche bzw. Taktabschnitte ermöglicht werden[2]. Abseits der Taktangaben an entsprechender Stelle enthalten die MIDI-Daten jedoch keinerlei Informationen über die metrische und übergeordnete strukturelle Gestalt des Tonmaterials, für eine Zuordnung von Tönen zu bestimmten Takten oder Taktbereichen sind also weitere Schritte erforderlich.
Um das Tonmaterial in Takte einteilen zu können, gilt es zunächst die relativen Taktangaben in absolute Zeiteinheiten, also in eine Anzahl von MIDI-Ticks, zu überführen. Dabei ist neben der Taktangabe auch die zeitliche Auflösung der Daten in Form der *Pulses per Quarter Note* ($PPQN$) zu berücksichtigen. Zunächst wird die Anzahl der Ticks $Ticks_{Beat}$ berechnet, die eine Zählzeit ausmachen:

$$Ticks_{Beat} = \frac{PPQN}{\frac{M}{4}} \tag{3.1}$$

Dabei sei der Zähler der Taktangabe mit N und der Nenner mit M bezeichnet. Durch Multiplikation von $Ticks_{Beat}$ mit dem Zähler der Taktangabe kann so die Länge eines Taktes in Ticks ausgedrückt werden. Gleichung 3.2 zeigt die Formel zur

[2]Diese Anforderung ergibt sich aus den in Kapitel 3.1.4 beschriebenen Methoden zur Fehlerschätzung, die auf der inhaltlichen Analyse bestimmter Taktabschnitte fußen.

Berechnung der Taktlänge $Ticks_{Takt}$:

$$\begin{aligned} Ticks_{Takt} &= Ticks_{Beat} \cdot N \\ &= \frac{4 \cdot PPQN}{N \cdot M} \end{aligned} \tag{3.2}$$

Unter Kenntnis dieser Taktlänge und der exakten Zeitstempel der MIDI-Events können diese in ein Takt-Raster eingeordnet werden. Dabei wird vorausgesetzt, dass die MIDI-Daten entsprechend quantisiert sind, d.h. dass die kodierten Inhalte diesem Raster tatsächlich folgen. Insbesondere bei Auftakten und dem Auftreten gewisser expressiver Elemente wie Fermaten ist diese Bedingung jedoch nicht immer gegeben. Beim Vergleich der berechneten Taktsequenzen mit den korrespondierenden Musiknotationen ist weiterhin zu berücksichtigen, dass die lineare Struktur der MIDI-Inhalte keine Wiederholungen erlaubt: Wenn Wiederholungen auftreten, entspricht die Anzahl der Takte nicht mehr den Angaben der Partitur, womit auch die Taktnummern nicht mehr direkt vergleichbar sind.

Auch die letztgenannte Anforderung, die Berechnung und gezielte Analyse separater Taktabschnitte, ist in den verfügbaren Softwarelösungen zur Verarbeitung von MIDI-Daten nicht implementiert; MATLAB bietet keine nativen Programmierschnittstellen für diese Funktionalität. Es konnte darüber hinaus keine Programmbibliothek ermittelt werden, die eine direkte Umwandlung der tonalen Inhalte in Tonhöhenverteilungen ermöglicht, was die Entwicklung einer eigenständigen, auf diese Zwecke spezialisierten Anwendung nahelegt. Sie soll auf Grundlage der Programmiersprache Java entwickelt werden: Diese bietet mit der *Java Sound API* nicht nur fest eingebaute Schnittstellen zur Verarbeitung von MIDI-Daten, welche die Implementierung vereinfachen, in Java geschriebene Anwendungen sind darüber hinaus mit beliebigen Systemen kompatibel. Für die Ausführung ist allein eine (frei verfügbare) Java-Laufzeitumgebung (*Java Runtime Environment, JRE*) nötig. Da die MATLAB-Entwicklungsumgebung diese bereits per se beinhaltet, ist eine Integration in MATLAB problemlos möglich. Java-Anwendungen können so direkt eingebunden und verwendet werden[3].

Die im zweiten Anforderungspunkt angeführte Registrierung und Verknüpfung der relevanten MIDI-Events erfordert zunächst deren genaue Definition.
Im Kontext dieser Arbeit ist insbesondere die Erkennung von Tonhöhen und deren

[3]Es ist allerdings zu beachten, dass MATLAB derzeit standardmäßig nur die ältere Java-Version 7 unterstützt; die Verwendung des aktuellen Java 8 ist in MATLAB zwar grundsätzlich möglich, erfordert allerdings tiefgreifende Anpassungen durch den Benutzer, so dass auf dessen Verwendung zugunsten der Benutzerfreundlichkeit verzichtet und stattdessen Version 7 verwendet werden soll.

Verknüpfung mit den entsprechenden Instrumentalstimmen von Interesse. In der MIDI-Repräsentation wird dieser Zusammenhang bereits durch die MIDI-Tracks ausgedrückt, welche das musikalische Material der einzelnen Stimmen beinhalten: Durch eine getrennte Analyse der Tracks ergeben sich somit die Tonhöhenverteilungen der einzelnen Instrumentalstimmen.
Dabei werden die Töne bei der Analyse anhand der `NOTE ON`-Events registriert. Es gilt dabei den *velocity*-Wert der Töne zu berücksichtigen, da das Ende der Klangereignisse – wie im vorigen Abschnitt erläutert – nicht durch `NOTE OFF`-, sondern durch wiederholte `NOTE ON`-Events markiert wird. Gültig sind daher ausschließlich Töne mit einem positiven *velocity*-Wert: Ohne diese Einschränkung würden die Analyseergebnisse stark verzerrt. Auf die Untersuchung kanalspezifischer Informationen wie der eingestellten MIDI-Instrumente soll verzichtet werden. Allein das in MIDI-Kanal 10, dem *drum channel*, kodierte Material wird bei der Analyse des Tonmaterials kategorisch von der Betrachtung ausgeschlossen, da bei den vorzunehmenden Betrachtungen ausschließlich tonale Inhalte von Interesse sind.

Die Überführung der musikalischen Inhalte in Tonhöhenverteilungen wird durch die dynamische Erstellung von Histogrammen realisiert, welche den gesamten Tonumfang der MIDI-Repräsentation umfassen: Die Töne werden anhand ihrer *note number* einer der 128 Histogramm-Klassen zugeordnet, bzw. deren Werte entsprechend der Anzahl auftretender Töne inkrementiert. Das Tonmaterial der einzelnen Tracks wird auf diese Weise in Histogramme überführt. Tracks ohne musikalischen Inhalt, d.h. solche die ausschließlich dem Einbinden von Metainformationen und Textinhalten dienen, werden nach erfolgter Analyse automatisch verworfen. Einzig der erste Track spielt in diesem Zusammenhang eine Sonderrolle: Er beinhaltet zwar keine Toninformationen, enthält jedoch alle Taktangaben. Diese müssen vermerkt werden um sie im Fall der Analyse von Taktfenstern berücksichtigen zu können, da die Länge der Takte direkt von ihnen abhängig ist.

Der Quellcode der entwickelten Softwareanwendung wird im Anhang dieser Arbeit aufgeführt. Die Tonhöhenverteilungen und weitere Eigenschaften bzw. Analyseergebnisse der Tracks werden in der Applikation durch Objekte vom Typ `MidiTrackAnalysis` ausgedrückt, die wiederum in ein `MidiFileAnalysis`-Objekt eingebettet sind, welches die Schnittstelle des Analyseverfahrens darstellt und der Bereitstellung der Ergebnisse dient. Es beinhaltet weiterhin ein `MidiMeasureAnalysis`-Objekt, welches die Tonhöhenverteilungen der einzelnen Takte repräsentiert. Zur Analyse bestimmter Bereiche kann das interessierende Zeitfenster bei der Instanzierung der `MidiFileAnalysis`-Klasse durch Angabe von Zeitstempeln in Form von MIDI-Ticks

ausgedrückt werden; die Informationsextraktion wird dann (mit Ausnahme des ersten Tracks) auf diesen Bereich beschränkt. Zur Berücksichtigung von Auftakten kann dabei zusätzlich die Länge des ersten Taktes angegeben werden, die Länge der darauf folgenden Takte wird durch die Taktangaben im ersten Track bestimmt. Wenn derartige Angaben nicht gemacht werden, erfolgt eine Analyse der gesamten Datei unter automatischer Berechnung der Takte, wobei eine einheitliche Quantisierung ohne das Auftreten von Auftakten angenommen wird.

3.1.3 Datenstrukturierung und -reduktion

Durch Analyse der MIDI-Daten mit Hilfe der entwickelten und im vorigen Unterkapitel beschriebenen Softwareanwendung können die Tonhöhenverteilungen der einzelnen MIDI-Tracks extrahiert werden. Diese Analyseergebnisse müssen jedoch hinsichtlich der musikalischen Vollständigkeit überprüft werden: Es soll darin jedes in der Partitur notierte Instrument vertreten sein. Um diese Bedingung überprüfen zu können, wird für jeden Komponisten eine Tabelle angefertigt, in der sämtliche für die Analyse in Frage kommenden Werke aufgelistet und ihre Instrumentierung vermerkt ist. Sie wird für jeden Satz separat angegeben. Da die vorliegenden MIDI-Dateien jeweils einem Satz entsprechen, kann die Vollständigkeit auf Satzebene leicht überprüft werden, indem die Namen der MIDI-Tracks mit den Inhalten der Tabelle verglichen werden.

Da der manuelle Abgleich der Inhalte in Anbetracht der großen Datenmenge einen hohen Zeitaufwand benötigt, soll dieser Prozess nach Möglichkeit automatisiert werden. Die Übereinstimmung zwischen den Track-Namen und den in der Tabelle vermerkten Instrumenten kann mit Hilfe von *regulären Ausdrücken* (engl. *regular expression*, auch *Regex*) überprüft werden. Reguläre Ausdrücke dienen dem Musterabgleich von Zeichenketten und können nicht nur deren exakte Übereinstimmung, sondern auch Kombinationen gegebener Muster identifizieren (Vgl. Mitkovs Definition [32, p. 754]). Diese Eigenschaft ist vor dem Hintergrund der heterogenen Namensgebung der MIDI-Tracks zwingend erforderlich: Je nach Nationalität und persönlichen Präferenzen der Autoren kann die erste Violine beispielsweise als „Violin I“, „Violino 1“ oder „Violine 1.“ betitelt sein. Eine exakte Übereinstimmung mit der in der Referenztabelle verwendeten Zeichenkette „Violino I“ tritt somit in den wenigsten Fällen auf. Daher müssen also bestimmte Muster für die verschiedenen Instrumente definiert werden. Um die beschriebenen Variationen in der Namensgebung der ersten Violinstimme zu identifizieren, kann beispielsweise der reguläre Ausdruck `Violin([a-z]*)\s(I|1)[.]?$` eingesetzt werden: Dieser entspricht dem Textmuster

„Violin“, gefolgt von keinem, einem oder mehreren Buchstaben (`([a-z]*)`), einem Leerzeichen (`\s`) und entweder einer numerischen oder römischen Eins (`(I|1)`), auf die ein Punkt folgen kann (`[.]?`) bevor die Zeichenkette endet (`$`). Die Angabe des Endes der Zeichenkette ist notwendig, damit das Muster nicht auch auf eine Zeichenkette wie „Violino II“ zutrifft. Im Rahmen der entwickelten Softwareanwendung werden für alle Orchesterinstrumente reguläre Ausdrücke dieser Art erstellt und zum automatischen Musterabgleich verwendet. Solo-Teile einzelner Instrumente werden dabei als separate Stimmen aufgefasst.

Die Überprüfung soll mittels MATLAB erfolgen: Zum einen verfügt diese Programmierumgebung über effiziente Methoden zur Ausführung regulärer Ausdrücke, zum anderen soll dieser Schritt die Verwendbarkeit der Java-Anwendung in MATLAB illustrieren. Da in einem vollständig automatisierten Prozess trotz sorgfältig gewählter regulärer Ausdrücke Fehler nicht ausgeschlossen werden können, soll ein halbautomatischer Abgleich stattfinden: Dabei wird anhand der regulären Ausdrücke eine erste Zuordnung vorgenommen, die anschließend durch den Benutzer verifiziert bzw. korrigiert werden muss. In einem MATLAB-Skript werden also mit Hilfe der Java-Applikation alle MIDI-Dateien – die einzelnen Sätze der Sinfonien – nacheinander eingelesen und die Instrumentierung jedes Satzes wird betrachtet. Unter Verwendung der Liste regulärer Ausdrücke werden die Namen nun mit der in den Referenztabellen angegebenen Instrumentierung verglichen und die Übereinstimmungen dem Benutzer präsentiert, der die so gefundenen Wertpaare per Tastendruck bestätigt oder korrigiert. Nicht identifizierte Instrumente oder mehrfache Übereinstimmungen können durch entsprechende Eingaben manuell zugeordnet werden. Darüber hinaus können einzelne Tracks von der Aufnahme in das Korpus ausgeschlossen werden[4]. Nachdem das halbautomatische Klassifizierungsverfahren für alle Sätze einer Sinfonie abgeschlossen ist, werden die standardisierten Instrumentennamen nebst ihren zugehörigen Tonhöhenhistogrammen als `.mat`-Datei gespeichert. Darin werden auch die Namen ausgeschlossener und fehlender Instrumentalstimmen vermerkt, um im nächsten Schritt unvollständige Datensätze entfernen zu können.

In zwei Fällen ist in den Datensätzen L.v. Beethovens eine unvollständige Instrumentierung nachzuweisen, die lediglich auf eine unzureichende Kodierung des Datenmaterials zurückzuführen ist: Im vierten Satz der neunten Sinfonie sind die Klarinetten in B und A in einem Track zusammengefasst, und im ersten Satz der dritten

[4]Bei einigen von Haydns Werken ist beispielsweise eine Cembalostimme angegeben, welche die Violoncello- und Kontrabasssektionen doppelt und daher nicht berücksichtigt werden soll. Der Chor in Beethovens neunter Sinfonie ist ein weiteres Beispiel für Tonmaterial das von der Betrachtung ausgeschlossen werden muss.

Sinfonie sind die Hornstimmen in Es und F nur durch einen Track repräsentiert. Unter Verwendung des Text-Outputs von `mftext` und eines weiteren MATLAB-Skripts werden die jeweiligen Stimmen separiert und die entsprechenden Datensätze korrigiert. Diese und weitere Eigenheiten einzelner Datensätze – wie die oben genannten ausgeschlossenen Stimmen – werden schriftlich in einer Tabelle festgehalten.

Um eine Bearbeitung und Auswertung der Daten auch außerhalb von MATLAB zu ermöglichen, sollen diese in einem nächsten Schritt in eine einfachere Textform überführt werden. Dafür bietet sich das CSV-Format (engl. *comma separated values*) an: Tabellen in diesem Format sind nicht nur menschenlesbar sondern auch maschinell leicht zu verarbeiten. Mit einem zweiten MATLAB-Skript werden die zuvor gespeicherten `.mat`-Dateien eingelesen und in CSV-Tabellen überführt. Die zuvor festgehaltenen Kommentare des Verfassers zu einzelnen Datensätzen werden darin neben Angaben zu Quellen und Urheberschaft der zugrunde liegenden MIDI-Daten ebenfalls eingebunden. Auf diese Weise werden die Informationen über Herkunft und tonalen Inhalt jedes Satzes in eine Textrepräsentation überführt und so für die Auswertung bereitgestellt.

Im letzten Schritt gilt es, das in Kapitel 3.1.1 aufgestellte Singularitätskriterium zu erfüllen und noch vorhandene Duplikate – d.h. verschiedene Versionen eines Datensatzes von unterschiedlichen Autoren – zu eliminieren. In das Korpus sollen nur Datensätze aufgenommen werden, deren Autoren bereits vertreten sind, um eine möglichst große Konsistenz hinsichtlich der Kodierung zu erreichen. Damit wird das Korpus also auf zusammenhängende, d.h. vom selben Autor kodierte Datensatz-Sektionen reduziert. Zudem werden einige wenige Datensätze derjenigen Autoren entfernt, die lediglich durch ein einzelnes Werk vertreten sind.

3.1.4 Fehlerschätzung

Die Grundlage der Tonhöhenverteilungen bilden von unterschiedlichen Autoren kodierte Orchesterwerke im MIDI-Format, deren strukturelle Vollständigkeit mit den im vergangenen Unterkapitel beschriebenen Verfahren sichergestellt werden kann. Über die Richtigkeit der konkreten Inhalte können zu diesem Zeitpunkt allerdings noch keine Aussagen getroffen werden. Es bedarf also einer Überprüfung der Validität der Daten und einer Untersuchung, inwiefern sich die Daten einzelner Autoren in ihrer Qualität unterscheiden. In Anbetracht der enormen Menge an analysiertem Tonmaterial erscheint es sinnvoll, die Fehlerrate mit geeigneten Verfahren anhand einer Stichprobe zu schätzen. Für eine solche Schätzung bedarf es lediglich der Betrachtung einiger fehlerhafter Sektionen. Auf Grundlage der Anzahl der darin

ermittelten Fehler sowie der korrekt kodierten Töne kann anschließend die Fehlerrate geschätzt werden.

Es gilt also zunächst, fehlerhaft kodierte Sektionen in den Datensätzen zu identifizieren und das darin enthaltene Tonmaterial mit den Partituren abzugleichen. Zur Identifikation falscher Töne wird auf den Tonumfang der Orchesterinstrumente zurückgegriffen: Wenn Instrumente Töne außerhalb ihres physikalischen Tonumfangs spielen muss es sich dabei offensichtlich um fehlerhaft kodiertes Material handeln. Der Tonumfang der betrachteten Orchesterinstrumente wird daher in einer Tabelle festgehalten; bei transponierenden Instrumenten wie der Klarinette oder dem Horn werden dabei die entsprechenden Tonumfänge zusammengefasst (siehe Tabelle 3.1).

Tabelle 3.1: Tonumfang der untersuchten Orchesterinstrumente, angegeben in ISO-Tonhöhen und MIDI-Nummern.

Instrument	Tonumfang	
	ISO	MIDI
Violine	G_3–A_7	55–105
Viola	C_3–A_6	48–93
Violoncello	C_2–A_5	36–81
Kontrabass	H_0–G_4	23–67
Piccolo	H_4–C_8	71–108
Flöte	H_3–D_7	59–98
Oboe	B_3–G_6	58–91
Klarinette	B_1–B_6	34–94
Fagott	B_1–Es_5	34–75
Kontrafagott	A_0–B_3	21–58
Trompete	E_2–C_6	40–84
Horn	G_1–Fis_5	31–78
Posaune	E_2–F_5	40–77
Pauke	D_2–C_4	38–60

Unter Verwendung von R können die Extremwerte der Tonhöhenverteilungen nun in einem automatischen Prozess mit diesen Angaben verglichen und auf diese Weise ungültige Datensätze aufgespürt werden. Die so identifizierten Fehler stellen zwar lediglich eine Teilmenge aller im Datenmaterial enthaltenen Fehler dar, sind als Datengrundlage für die Fehlerschätzung jedoch ausreichend: Es müssen lediglich für jeden Autor Stücke identifiziert werden, die definitiv Fehler enthalten. Die Ergebnisse dieser Analyse, d.h. die fehlerhaft kodierten Instrumente und die Anzahl der Unter- und Überschreitungen des Tonumfangs werden in einer Tabelle gespeichert. Anhand

dieser Daten wird nun für jeden Autor der Datensatz ausgewählt, welcher die meisten Fehler enthält. Auf dessen Grundlage soll anschließend die Fehlerschätzung erfolgen.

Auch bei der Identifikation der fehlerhaften Bereiche erweist sich das Werkzeug `mftext` als hilfreich. In der damit erzeugten Textrepräsentation der MIDI-Dateien sind die im ersten Schritt gewonnenen Abweichungen vom gültigen Tonumfang leicht auffindbar und die entsprechenden Stellen können in der Partitur lokalisiert werden. Um bei der Fehlerschätzung nicht ausschließlich fehlerbehaftetes Material zu berücksichtigen, wird die Stichprobe über die fehlerhafte Sektion hinaus auf den gesamten sie beinhaltenden musikalischen Abschnitt erweitert: In der Partitur werden die nächstliegenden Doppelstriche identifiziert und die durch sie markierten Abschnitte für die Fehlerschätzung verwendet. Unter Anwendung der entwickelten Software-Applikation können nun die Tonhöhenverteilungen der Instrumente in den einzelnen Takten dieses 'Fensters' analysiert und mit der Partitur verglichen werden, wobei die Zahl der auftretenden Kodierungsfehler und die Gesamtzahl der betrachteten Töne ermittelt werden. Die Reihenfolge der Töne innerhalb des Taktes wird bei der Fehleranalyse nicht berücksichtigt, da sie keine Auswirkungen auf die Gestalt der Tonhöhenverteilungen hat.

Nun kann anhand der Stichproben die Fehlerquote der einzelnen Autoren geschätzt werden. Die damit verbundene Unsicherheit des Schätzers soll durch ein *Konfidenzintervall* ausgedrückt werden: Dieses Intervall gibt den Bereich an, in dem sich mit einer gegebenen Wahrscheinlichkeit der wahre Wert befindet, wobei üblicherweise eine Wahrscheinlichkeit von 95% angenommen wird. Das exakte Konfidenzintervall für die Fehlerquote kann anhand der *hypergeometrischen Verteilung* berechnet werden [16, p. 30–33]), die das hier erfolgte 'Ziehen ohne Zurücklegen' beschreibt. Diese Methode ist jedoch sehr rechenintensiv; Bei kleinen Stichproben, wie den vorliegenden, werden die Konfidenzintervalle daher üblicherweise nach dem *Clopper-Pearson*-Verfahren berechnet (Vgl. [16, p. 37]), welches die auf Basis der hypergeometrischen Verteilung gewonnenen Ergebnisse approximiert.

3.2 Vergleichende Untersuchungen

Das im Rahmen der Korpuserstellung gewonnene Datenmaterial soll nun statistisch ausgewertet werden. Für eine erste Übersicht über das vorhandene Datenmaterial sollen zunächst die Tonhöhenverteilungen aller Instrumente grafisch dargestellt und miteinander verglichen werden. Die für alle sinfonischen Sätze vorliegenden Histogramme der Instrumente werden dabei addiert, wodurch sich deren Gesamtver-

teilung ergibt. Um individuelle Unterschiede in der Verwendung der Instrumente aufzuzeigen, erfolgt diese Auswertung für alle Komponisten getrennt. Anschließend sollen die einzelnen Verteilungen der Instrumente mit geeigneten statistischen Methoden auf signifikante Unterschiede bei den vier Komponisten überprüft werden, wobei mit der Tonhöhe und dem Tonumfang sowohl ihr Lage- als auch das Streuungsmaß untersucht und verglichen werden soll.

Bei der Betrachtung dieser Parameter gilt es zunächst, geeignete Kennwerte für die 'mittlere' Tonhöhe und den 'mittleren' Tonumfang zu finden: Erstere ist durch den Median der Verteilung gegeben, der im Gegensatz zum Mittelwert deutlich robuster gegenüber Ausreißern[5] im unteren und oberen Tonbereich und daher zu bevorzugen ist. Da Ausreißer in der Tonhöhe sich ebenfalls direkt auf den Tonumfang auswirken, soll bei der Betrachtung des Tonumfangs daher nicht die gesamte Spannweite – d.h. die Differenz zwischen höchstem und tiefstem Ton – sondern der Interquartilsabstand dienen. Dieser ergibt sich aus der Differenz des 1. und 3. Quartils, jenen Schwellenwerten, unterhalb derer 25% bzw. 75% der Messwerte liegen: Der Interquartilsabstand markiert also den Bereich, in dem die mittleren 50% der Messwerte liegen und ist damit ein robustes Maß für den durchschnittlichen Tonumfang. Unter Kenntnis des Interquartilsabstandes lassen sich auch die Ausreißer formal definieren: Darunter werden üblicherweise diejenigen Werte verstanden, die außerhalb des $\pm 1{,}5$-fachen Interquartilsabstandes liegen [33, p. 49].

3.2.1 Grafische Auswertungen

Eine erste grafische Auswertung soll anhand des Vergleiches der Tonhöhenhistogramme erfolgen; R verfügt über entsprechende Methoden der grafischen Darstellung. Die im Histogramm dargestellten diskreten Häufigkeitswerte sollen darüber hinaus mittels Kerndichteschätzung in eine stetige Dichtefunktion überführt werden. Dabei wird jeder Histogramm-Klasse – d.h. in diesem Kontext jedem Ton – eine eingipflige Modellfunktion (Kern) zugewiesen, deren Maximalwert dem Wert der Klasse entspricht. Die Verteilung wird damit also nicht durch eine Anzahl diskreter Einzelwerte, sondern durch mathematische Funktionen ausgedrückt, durch deren Addition und Normierung auf eine Fläche von 1 sich eine stetige Dichtefunktion ergibt, wie in Abbildung 3.1 illustriert wird.

Die Güte dieser Schätzung hängt von der verwendeten Bandbreite, d.h. der Breite des Kerns ab: Eine zu geringe Bandbreite sorgt für eine multimodale, 'zackige'

[5] Unter Ausreißern sind dabei Werte zu verstehen, die im Vergleich zu den restlichen Werten besonders klein oder groß sind.

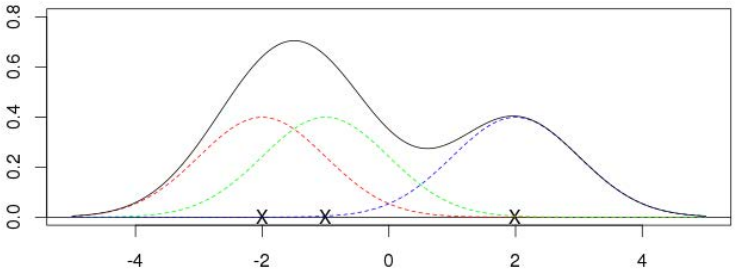

Abbildung 3.1: Illustration einer Kerndichteschätzung mit Gauß-Kern und Bandbreite 6. Die Werte sind durch Kreuze markiert, die gestrichelten farbigen Linien entsprechen den einzelnen Kernen und die schwarze Linie der (unnormierten) Gesamtfunktion.

Dichtefunktion mit vielen lokalen Maxima, während bei Verwendung eines zu hohen Bandbreitenwertes die Verteilung ungenau approximiert wird. Zur Bestimmung der optimalen Bandbreite existieren numerische Verfahren [34], die auch in R implementiert sind und zur Schätzung der Dichtefunktionen verwendet werden sollen. Die Kern-Funktion wird nach einer ersten Inspektion der Histogramme ausgewählt; es sei an dieser Stelle vorweggenommen, dass die Verteilungen annähernd normalverteilt sind, weshalb für die Dichteschätzung ein Gauß-Kern – d.h. eine Normalverteilung – verwendet werden kann.

Neben der Betrachtung der Tonhöhen-Histogramme sollen die wichtigsten Kennwerte der Verteilungen übersichtlich dargestellt und verglichen werden. Für diese Darstellung sollen sogenannte *Boxplots* oder *Box-Whisker-Plots* verwendet werden: Diese Diagramme stellen die genannten Lage- und Streuungsmaße übersichtlich dar und erlauben den direkten Vergleich der Kennzahlen verschiedener Verteilungen 'auf einen Blick'. Die namensgebende Box gibt die Position des 1. und 3. Quartils (und damit den Interquartilsabstand) sowie des Medians der Verteilung an. Durch zwei Antennen (engl. whiskers) an den Seiten der Box werden die außerhalb dieses Bereichs liegenden Messwerte dargestellt. Oftmals erstrecken diese sich maximal bis zum $\pm 1{,}5$-fachen Interquartilsabstand und etwaige darüber hinaus gehende Ausreißer werden separat gekennzeichnet [35, p. 52]. Die Antennen können aber auch die gesamte Spannweite der Messwerte anzeigen – wie auch in dieser Arbeit, in der sie den gesamten Tonumfang repräsentieren sollen.

3.2.2 Varianzanalyse

Während die grafischen Darstellungen der Tonhöhenverteilungen einen ersten Überblick über Form und inhaltliche Tendenzen des vorhandenen Datenmaterials liefern, erlauben sie keine quantitativen Befunde über die Unterschiede der einzelnen Verteilungen hinsichtlich Tonhöhe und Tonumfang. Diese müssen durch ein geeignetes Regressionsverfahren geschätzt werden, wobei die vorhandenen Daten – die Messungen an den verschiedenen Instrumenten, Werken und Komponisten – durch ein statistisches Modell beschrieben werden. Dieses erlaubt Rückschlüsse auf systematische Zusammenhänge und Unterschiede der Tonhöhenverteilungen.

Grundsätzlich werden bei der Regressionsanalyse Unterschiede in der Ausprägung einer abhängigen Variable (*Zielvariable*) auf eine oder mehrere unabhängige Variablen (*Faktoren* oder *Prädiktoren*) zurückgeführt. Bei der vorzunehmenden linearen Regression wird die Beziehung zwischen der Zielvariable und den sie bedingenden Faktoren durch eine lineare Gleichung dieser Form ausgedrückt:

$$y_t = \beta_0 + \beta_1 x_{t1} + \beta_2 x_{t2} + ... + \beta_n x_{tn} + u_t \tag{3.3}$$

Die Messungen der Zielvariable y und der Ausprägungen der n Faktoren $x_1 ... x_n$ liegen für verschiedene Zeitpunkte t vor. Es gilt die Regressionskoeffizienten $\beta_1 ... \beta_n$ so zu wählen, dass die Residuen u_t minimiert werden, d.h. dass die durch das Modell beschriebene Regressionsgerade die Verteilung der Daten möglichst optimal beschreibt. Die Struktur der Datensätze ist dabei maßgeblich für die Wahl des Modells verantwortlich: Im vorliegenden Fall gilt es also zu berücksichtigen, dass die ‚Messungen' der Tonhöhenverteilungen aller Instrumente nicht nur an den Orchesterwerken unterschiedlicher Komponisten vorgenommen, sondern bei jedem Werk auch alle Sätze separat analysiert werden. Die Modellierung dieser Mehrebenenstruktur erfordert ein *hierarchisch-lineares Modell* (auch *Mehrebenenmodell*). Damit kann das Risiko falscher Schlüsse bei der Interpretation von Zusammenhängen und Unterschieden minimiert werden, die eine Vernachlässigung dieser Hierarchie mit sich bringen würde, wie Eid et al. aufzeigen [36, p. 699–706].

Mehrebenenmodelle zeichnen sich unter anderem dadurch aus, dass sie sowohl feste als auch zufällige Effekte enthalten[6]; sie werden daher im Englischen auch

[6]Es wird zwischen Faktoren mit festem (*fixed effect*) und zufälligem Effekt (*random effect*) unterschieden: Ein fester Effekt liegt vor, wenn die möglichen Ausprägungen des Faktors bereits von vornherein feststehen und reproduzierbar sind. Sollten die Beobachtungen jedoch einer Zufallsstichprobe aus der Menge aller möglichen Ausprägungen entsprechen, handelt es sich um einen zufälligen Effekt (Vgl. [37, p. 2]).

als *mixed model* bezeichnet. Ziel des Verfahrens ist die Ermittlung der Faktoren, die in besonders starkem Zusammenhang zur Zielvariable stehen. Für die Umsetzung der Mehrebenenanalyse mit der Programmiersprache R steht unter anderem die Software-Bibliothek `lme4` bzw. deren Methode `lmer` bereit [37, 38].

Im vorliegenden Fall entspricht die Zielvariable der mittleren Tonhöhe bzw. dem mittleren Tonumfang. Beide Variablen sollen auf Zusammenhänge mit den Parametern „Komponist“, „Werknummer“, „Satznummer“ und „Instrument“ untersucht werden. Die Parameter „Komponist“ und „Instrument“ können als Faktoren mit festen Effekten angesehen werden, da es sich um kategoriale Variablen mit einer endlichen Anzahl an Ausprägungen handelt: Es wird schließlich bei genau vier Komponisten eine definierte Menge an Orchesterinstrumenten betrachtet. Auf die Werknummer trifft diese Bedingung nicht zu: Die Datensätze der verschiedenen Komponisten umfassen nicht nur unterschiedliche Werknummern, sondern auch eine unterschiedliche Anzahl davon. Die Werknummer könnte also als Variable mit zufälligem Effekt betrachtet werden, soll hier jedoch als metrische Variable mit festem Effekt in das Modell eingehen: So können systematische Zusammenhänge zwischen den Zielvariablen und der Entstehungszeit des Werkes berücksichtigt werden[7]. Die Satznummer soll schließlich als zufälliger Effekt modelliert werden, da die Anzahl der Sätze pro Werk variiert: Die Daten der verschiedenen Sätze sind in diesem Sinne als ‘Messwiederholung’ einer Sinfonie zu verstehen.

Zur Bestimmung der bedeutensten Faktoren wird ein *Likelihood-Ratio-Test* durchgeführt. Dieser gibt an, inwiefern sich das Modell durch das Hinzufügen eines neuen Terms verändert. Zunächst wird ein *Nullmodell* definiert, welches lediglich aus dem konstanten Term β_0 in Gleichung 3.3 besteht. Dieser wird auch *Intercept* genannt, es handelt sich beim Nullmodell daher um ein sogenanntes *Intercept-Only-Modell*, das zur Schätzung der Interklassenkorrelation verwendet werden kann [39, p. 14–15]. Diesem wird nun schrittweise einer der Faktoren hinzugefügt und untersucht, ob das Modell sich dadurch verbessert, d.h. ob es mehr Varianz der Daten zu erklären vermag. Neben den vier Haupteffekten „Komponist“, „Werknummer“, „Satznummer“ und „Instrument“ sind bei der Betrachtung von Tonhöhe und Tonumfang insbesondere die Interaktionen zwischen diesen Variablen von Interesse. Durch die Inklusion solcher zusätzlicher Parameter gewinnt das Modell oftmals an Güte, wird mit steigender Faktoranzahl allerdings auch schnell sehr komplex. Um eine Balance zwischen Anpassungsgüte und Modellkomplexität zu finden, wird ein *Informations-*

[7] Es ist schließlich möglich, dass späte Werke hinsichtlich des Tonumfangs oder der Tonöhe einzelner Instrumente grundsätzlich andere Tendenzen aufweisen als frühere.

kriterium verwendet. Die Komplexität eines Modells wird dabei anhand der Anzahl verwendeter Faktoren geschätzt und 'bestraft'. Als Abbruchbedingung des iterativen Prozesses der Modellfindung und zur Auswahl des besten Modells wird das *Bayessche Informationskriterium* (engl. *Bayesian Information Criterion*, *BIC*) herangezogen, eine Weiterentwicklung von *Akaikes Informationskriterium* (*AIC*), dessen Wert anders als bei letzterem von der Stichprobengröße unabhängig ist. Der BIC-Wert sollte bei der Aufnahme zusätzlicher Parameter in das Modell stets fallen; bei steigendem Wert liegt der Erklärungsgewinn hingegen nicht mehr im Verhältnis zur Modellkomplexität [39, p. 50–51] bzw. dem damit verbundenen interpretatorischen Aufwand.

Nach der Erstellung eines geeigneten Modells und der damit verbundenen Identifikation der relevanten Faktoren können die Schätzer der Haupt- und Interaktionseffekte auf Signifikanz[8] geprüft werden. Schließlich gilt es festzustellen wie viel Gesamtvarianz durch das Modell erklärt wird: Dieser Anteil ist durch das *Pseudo-Bestimmtheitsmaß* (*Pseudo-R^2*) gegeben, für dessen Berechnung verschiedene Verfahren existieren. Bereits Pseudo-R^2-Werte zwischen 0,2 und 0,4 weisen laut Hox et al. [39, p. 135] auf ein gutes Modell hin; dieser Wertebereich ist daher anzustreben.

3.3 Klassifikation von Instrumenten

Die Eignung der Datensätze zur automatischen Klassifikation von Instrumenten soll durch ein exploratives Verfahren überprüft werden. Zunächst sollen verschiedene Klassifikationsverfahren darauf angewandt und ihre Ergebnisse miteinander verglichen werden. Anschließend wird näher auf das Verfahren eingegangen, welches bei diesem Vergleich die höchste Genauigkeit aufweist. Im Folgenden werden die Grundlagen der automatischen Klassifikation erläutert und die zu betrachtenden Verfahren vorgestellt.

3.3.1 Maschinelles Lernen

Das übergreifende Themengebiet von automatischen Klassifikationsverfahren ist das maschinelle Lernen (engl. *machine learning*), welches in unterschiedlichen Forschungsgebieten Anwendung findet. Es geht dabei darum, aus vorhandenen Daten zu lernen, um anschließend Aussagen über neue Daten des selben Typs treffen zu können. Ein typisches Szenario wird von Friedman, Hastie und Tibshirani wie folgt beschrieben

[8] Signifikanz bedeutet hier, dass der Schätzer in der Grundgesamheit – d.h. der Menge aller Sinfonien der vier Komponisten – nicht Null ist, d.h. dass der in den Stichproben beobachtete Effekt wahrscheinlich auch in der Grundgesamtheit auftritt.

(Vgl. [40, p. 1–2]): Es existiert eine quantitative oder kategoriale Zielvariable, die anhand einer Menge von Merkmalen bzw. Prädiktoren (engl. *features*) vorhergesagt werden soll. In einer Trainingsphase wird eine bestimmte Menge an Messdaten observiert (das sogenannte *training set*) und anhand der darin auftretenden Zielvariablen und ihrer zugehörigen Features ein Prädiktionsmodell erstellt. Dieses ermöglicht – anschließend anhand der darin enthaltenen Features – die Klassifikation von neuen, d.h. dem Modell unbekannten Daten. Die Genauigkeit der Vorhersage wird durch verschiedene, modellspezifische Maße ausgedrückt.

Der hier beschriebene Prozess entspricht dem *überwachten Lernen* (engl. *supervised learning*), da die Zielvariable – bzw. im kategorialen Fall die Zielklasse – bekannt ist und während der Modellbildung berücksichtigt wird. Unüberwachte Verfahren hingegen betrachten nur die Features und werden zum Erlernen struktureller Eigenschaften der Daten eingesetzt. Bei den im Folgenden betrachteten Klassifikationsverfahren handelt es sich ausschließlich um Anwendungen des überwachten Lernens: Die kategoriale Zielvariable „Instrument“ soll anhand geeigneter Prädiktoren bestimmt werden. Es existiert eine Vielzahl an überwachten Lernmethoden, die unterschiedlichen methodischen Prinzipien folgen und für verschiedene Klassifikationsprobleme entworfen wurden. Je nach Einsatzszenario können sie hinsichtlich der Ergebnisqualität beträchtlich variieren; es gibt kein Verfahren, welches für jedes Problem bedingungslos geeignet ist[9]. Daher sind stets verschiedene Modelle bzw. Algorithmen in Betracht zu ziehen – und ggf. sogar zu kombinieren – um das optimale problemspezifische Klassifikationsverfahren zu entwickeln. Vor diesem Hintergrund gibt es dennoch einige Verfahren, die sich für bestimmte Problemtypen als zuverlässig erweisen. Caruana und Niculescu-Mizil werteten 10 verschiedene überwachte Lernmethoden aus, indem sie diese auf 11 binäre Klassifikationsprobleme anwandten und nach 8 verschiedenen Performanzkriterien bewerteten [42]. Die besten durchschnittlichen Resultate liefern demnach die Verfahren *Bagged Tree* und *Random Forest*, die an dieser Stelle kurz erläutert werden sollen.

3.3.2 Ausgewählte Klassifikationsverfahren

Die Algorithmen *Bagged Tree* und *Random Forest* basieren auf binären Entscheidungsbäumen, die während des Trainings erstellt werden. Es wird dabei das Feature ausgewählt, welches die Menge der Trainingsdaten am besten teilt; dafür existieren

[9] Man spricht in diesem Zusammenhang auch vom *No Free Lunch Theorem*, ein Begriff der von Wolpert und Macready eingeführt wurde [41] und auf die englische Redensart „There is no such thing as a free lunch“ zurückgeht, zu deutsch etwa „nichts ist umsonst“.

verschiedene Maße, welche die Homogenität der Zielvariable in den Teilmengen angeben. Das ausgewählte Feature wird in den Wurzelknoten eingetragen, anschließend wird dieser Prozess auf die beiden Teilmengen und die verbleibenden Features angewandt, so dass die Kindknoten den 'zweitwichtigsten' Prädiktoren entsprechen. Dieser Prozess wird so lange wiederholt, bis alle Features in den Baum aufgenommen wurden; seine Blätter entsprechen den verschiedenen Klassifikations-Labels. Bei der Klassifikation wird der Baum von der Wurzel aus durchlaufen, wobei an jedem Knoten ein Feature überprüft und der entsprechende Kindknoten ausgewählt wird bis ein Blatt erreicht wird.
Weiterhin verwenden beide Verfahren die von Breiman vorgestellte *Bagging*-Technik (von engl. *bootstrap aggregating*) [43], bei der die Prädiktionen verschiedener Klassifikationsmodelle vereint werden. Dabei werden aus der Menge der Trainingsdaten in einem iterativen Prozess zufällige Werte ausgewählt und anhand dieser wird jeweils ein Klassifikationsmodell erstellt. Bei der Klassifikation unbekannter Daten werden alle Modelle separat ausgewertet und ihre Ergebnisse gemittelt; das Endergebnis enspricht also der bei allen Modellen am häufigsten auftretenden Klassifikation. Durch die Betrachtung mehrerer Modelle kann die Varianz drastisch verringert werden (Vgl. [40, p. 283–285]) und die Genauigkeit gesteigert werden.
Das Bagged Tree-Modell entspricht somit einer Reihe unterschiedlicher Entscheidungsbäume, denen jeweils eine zufällige Auswahl der Trainingsdaten zugrunde liegt. Beim Random Forest wird zusätzlich bei jedem Entscheidungsbaum nur eine zufällige Teilmenge der Prädiktoren betrachtet (*feature bagging*); der Name des Verfahrens leitet sich von der Menge so entstehender zufälliger Bäume ab. Dieser Schritt verringert die Korrelation der verschiedenen Bäume: Wenn ein oder mehrere Features sich als starke Prädiktoren der Zielvariable erweisen, werden diese im Bagged Tree-Verfahren von vielen Bäumen ausgewählt. Durch die auf diese Weise ansteigende Korrelation der Bäume steigt auch die Fehlerrate, wie Breiman zeigt [44, p. 16–20].

3.3.3 Vorgehensweise

In R stehen mit dem `caret`-Paket [45] standardisierte Methoden zur Anwendung und Auswertung einer Vielzahl von Klassifikations- und Regressionsverfahren zur Verfügung. Unter Verwendung dieser Schnittstelle sollen 6 unterschiedliche Klassifikationsalgorithmen auf die Daten angewandt werden: Neben den beiden bereits vorgestellten Verfahren sollen die Algorithmen *KNN* (K-Nearest-Neighbours), *SVM* (Support Vector Machine), *LDA* (lineare Diskriminanzanalyse) und Naive Bayes verwendet und ihre Ergebnisse verglichen werden. Als Datengrundlage dienen sta-

tistische Kennwerte der Tonhöhenverteilungen. Neben dem bereits bei der Analyse von Tonhöhe und Tonumfang verwendeten Median und Interquartilsabstand werden acht weitere Features für die Prädiktion der Instrumente verwendet: Das Minimum und Maximum sowie der sich durch sie ergebende gesamte Tonumfang, der Wert des 1. und 3. Quartils, sowie der Mittelwert, die Varianz und die Standardabweichung der Tonhöhenverteilungen. Hinsichtlich der verwendeten Ausgangsdaten gilt es einige Einschränkungen vorzunehmen. So sollte zu jedem Instrument eine ähnliche Datenmenge vorliegen, um die Datenbasis nicht zugunsten bestimmter Zielklassen zu verzerren. Im Korpus selten vertretene Instrumente sind daher von der Betrachtungg auszuschließen. Zudem gilt es sicherzustellen dass die einzelnen Datensätze tatsächlich sinnvolle Messungen darstellen und genügend Tonmaterial beinhalten um die Features berechnen zu können[10]. Es wird daher festgelegt, dass die Tonhöhenverteilungen mindestens 10 Töne sowie mindestens zwei unterschiedliche Tonhöhenwerte enthalten müssen. Alle Datensätze müssen anhand dieser Verwendungskriterien überprüft und unzutreffende Elemente ausgeschlossen werden.

In einem weiteren Vorverarbeitungsschritt sollen die Daten vor der Modellbildung zentriert und skaliert werden. Dieser Schritt ist üblich, da einige der Algorithmen zumindest zentrierte Werte voraussetzen und bei skalierten Daten eine höhere Genauigkeit aufweisen [45, p. 5]. Schließlich müssen die Datensätze in Trainings- und Testdaten eingeteilt werden, was durch ein zufälliges Auswahlverfahren erfolgen soll. 70% der Daten sollen zum Training und 30% zur Auswertung verwendet werden, wobei das Gesamtverhältnis der verschiedenen Instrumente in beiden Gruppen beibehalten werden soll.

Die Genauigkeit der Klassifikationsverfahren wird schließlich durch N-fache Kreuzvalidierung geschätzt. Dabei werden die Algorithmen wiederholt anhand jeweils verschiedener Ausschnitte der Datengrundlage trainiert und unter Verwendung der verbleibenden Teile ausgewertet. Die so gewonnenen Ergebnisse entsprechen den Mittelwerten der N verschiedenen trainierten Modelle und sind damit aussagekräftiger als eine einfache Anwendung der Verfahren.

[10] Im vierten Satz von J. Haydns 41. Sinfonie ist der musikalische Beitrag der Posaune mit acht Tönen der selben Tonhöhe nicht nur sehr gering und daher wenig repräsentativ, es ergeben sich hier darüber hinaus auch keine sinnvollen Werte für die Streuungsmaße.

Ergebnisse

In diesem Kapitel sind die Resultate der im vorigen Kapitel vorgestellten Untersuchungs- und Auswertungsmethoden aufgeführt. Zunächst wird das im Korpus enthaltene musikalische Material vorgestellt und es werden quantitative wie auch qualitative Aspekte der Datengrundlage beleuchtet. Daraufhin werden die Ergebnisse der betrachteten Untersuchungsgegenstände ausgewertet. Schließlich werden die jeweiligen Befunde in einem kurzen Fazit zusammengefasst.

4.1 Korpus

Das erstellte Korpus umfasst 73 vollständige Sinfonien, die von vier verschiedenen Autoren kodiert wurden. Es stehen Datensätze von 14 Orchesterinstrumenten für alle Sinfonien und ihre einzelnen Sätze zur Verfügung. Dafür wurden insgesamt 286 Sätze und über 2,7 Millionen Einzeltöne analysiert. Tabelle 4.1 zeigt eine Übersicht der analysierten Werke.

Das sinfonische Repertoire L.v. Beethovens konnte vollständig analysiert werden, für die anderen drei Komponisten liegen lediglich Stichproben ihres Gesamtwerkes vor. Die analysierten Werke von J. Haydn und F. Schubert sind einigermaßen gleichmäßig über das gesamte sinfonische Repertoire der Komponisten verteilt. Der Datensatz von W.A. Mozart weist aufgrund einer 'Lücke' zwischen seiner 7. und der 27. Sinfonie allerdings eine Verzerrung auf: Die späten Sinfonien sind hier in der Überzahl und stehen nur zwei frühen Werken gegenüber. Das Korpus beinhaltet damit etwas weniger als ein Drittel des sinfonischen Gesamtrepertoires dieses Komponisten.

Tabelle 4.1: Analysiertes Repertoire.

Komponist	Anzahl Sinfonien	davon analysiert	Autor	Sinfonien
J. Haydn (1732–1809)	104	46	Belinda Campbell	1, 2, 6, 7, 8, 26, 28, 30, 31, 46, 47, 49, 60
			Steven Edwards	62-73, 82–102, 104
W.A. Mozart (1756–1791)	41	12	Daniel Viens	6, 7, 27, 30, 31, 35, 36, 38–41
L.v. Beethoven (1770–1827)	9	9	Daniel Viens	1–9
F. Schubert (1797–1828)	8	6	Belinda Campbell	1
			Jean-François Lucarelli	4–6, 8, 9

Tabelle 4.2: Anzahl der Datensätze für die einzelnen Instrumente und Komponisten.

Instrument	Haydn	Mozart	Beethoven	Schubert	$\sum$
1. Violine	182	45	37	22	286
2. Violine	182	45	37	22	286
Viola	182	45	37	22	286
Violoncello	182	45	37	22	286
Kontrabass	182	45	37	22	286
Piccolo	0	0	3	0	3
Flöte	120	24	37	22	203
Oboe	176	35	37	22	270
Klarinette	13	12	38	18	81
Fagott	174	26	37	22	259
Kontrafagott	0	0	2	0	2
Trompete	72	26	34	16	148
Horn	176	45	45	24	290
Posaune	0	0	5	6	11
$\sum$	1641	393	423	240	2697

4.1.1 Instrumente

Tabelle 4.2 zeigt die Anzahl der Datensätze, die für die betrachteten Orchesterinstrumente zur Verfügung stehen. Das analysierte Material umfasst insgesamt 2697 einzelne Tonhöhenverteilungen, von denen über die Hälfte aus dem Repertoire Haydns stammt. Mit 423 Datensätzen enthält das sinfonische Gesamtwerk Beethovens mehr einzelne Instrumentalstimmen als die 12 analysierten Werke von Mozart; dieser Unterschied ist auf den häufigen Einsatz der Klarinette bei Beethoven zurückzuführen. Bei diesem Instrument zeigt sich zwischen den Komponisten ein deutlicher Unterschied in der Verwendungshäufigkeit: Während die Klarinette bei Haydn und Mozart nur einen geringen Teil der Datensätze ausmacht, wird sie bei Beethoven und Schubert fast immer eingesetzt. Gleiches gilt für die anderen Holzbläser (mit Ausnahme des Kontrafagotts), die bei Beethoven und Schubert in jedem Werk vertreten sind. Das Fagott, welches bei Haydn zwar ebenfalls in fast allen Werken eingesetzt wird, verwendet Mozart nur in der knappen Hälfte der betrachteten Fälle. Auch die Flöte und die Oboe werden bei Haydn und Mozart seltener eingesetzt als bei Beethoven und Schubert. Das Horn tritt bei Haydn in fast allen untersuchten Sinfonien auf, bei den anderen Komponisten in jedem Werk. Hinsichtlich der Trompete zeigen sich gewisse kompositorische Präferenzen der Komponisten: Während sie von Haydn nur in gut einem Drittel seiner Werke eingesetzt wird, tritt sie bei Mozart wie die Flöte und das Fagott in etwa der Hälfte seiner Werke auf. Beethoven hingegen setzt sie fast immer ein, bei Schubert wiederum ist die Trompete nach der Posaune das am seltensten verwendete Instrument. Die Posaune wird nur von Beethoven und Schubert in seltenen Fällen eingesetzt, das Kontrafagott und die Piccoloflöte sind ausschließlich in Beethovens Repertoire vertreten.

4.1.2 Fehlerschätzung

Die für die Fehlerschätzung verwendeten Stichproben wurden für alle vier Autoren anhand des in Kapitel 3.1.4 beschriebenen Verfahrens gewonnen. In Tabelle 4.3 sind die betreffenden Werke bzw. Abschnitte aufgeführt. Durch manuellen Abgleich der entsprechenden MIDI-Auszüge mit den Partituren wurde das darin enthaltene Tonmaterial überprüft und alle fehlerhaft kodierten Töne identifiziert.

Auf dieser Grundlage konnte nun der gesamte Fehleranteil der Autoren geschätzt werden. Die Konfidenzintervalle wurden wie oben beschrieben nach der Clopper-Pearson-Methode sowie durch exakte Bestimmung anhand der hypergeometrischen Verteilung ermittelt. Tabelle 4.4 zeigt die Ergebnisse der Stichprobenanalyse; es sind

Tabelle 4.3: Für die Fehlerschätzung verwendeten Stichproben.

Autor	Komponist	Sinfonie	Satz	Takte
Campbell	Haydn	8	3	37–82
Edwards	Haydn	90	1	1–16
Lucarelli	Schubert	6	4	292–368
Viens	Beethoven	6	3	205–234

für beide Verfahren jeweils die untere Grenze g_{unten} und die obere Grenze g_{oben} des Konfidenzbereiches angegeben. Die Resultate beider Verfahren weisen eine hohe Ähnlichkeit auf. Die nach der Clopper-Pearson-Methode berechneten unteren Grenzwerte liegen, mit Ausnahme des vierten Autors, stets unterhalb der durch exakte Schätzung ermittelten Werte; analog dazu liegen die oberen Grenzwerte stets oberhalb ihrer exakten Pendants. Wie erwartet verringern sich diese Abweichungen mit zunehmender Größe der Grundgesamtheit, d.h. die Werte nähern sich zunehmend an. Da die aus beiden Schätzungen resultierenden Prozentwerte nahezu identisch sind – sie unterscheiden sich erst im höheren Nachkommabereich – werden sie in der Tabelle nicht getrennt aufgeführt.

Tabelle 4.4: Konfidenzintervalle der geschätzten Fehlerraten der einzelnen Autoren zum Konfidenzniveau 95%, berechnet nach dem Clopper-Pearson-Verfahren und anhand der hypergeometrischen Verteilung.

	Autor	Campbell	Edwards	Lucarelli	Viens
	Einzeltöne	327663	1001903	297250	931190
	Stichprobengröße	583	705	2908	470
	Fehler in Stichprobe	67	55	111	4
g_{unten}	Clopper-Pearson	29550	59422	9365	2164
	Hypergeom. Verteilung	29556	59427	9373	2164
	Prozentwert	9,02%	5,93%	3,15%	0,23%
g_{oben}	Clopper-Pearson	47066	100528	13610	20156
	Hypergeom. Verteilung	47058	100521	13600	20154
	Prozentwert	14,36%	10,03%	4,58%	2,16%

Hinsichtlich der Genauigkeit des kodierten Tonmaterials sind zwischen den einzelnen Autoren große Unterschiede feststellbar: Mit einem geschätzen maximalen Fehleranteil von 2,16% weist Daniel Viens von allen Autoren die höchste Genauigkeit auf. Die Datensätze der Komponisten W.A. Mozart und L.v. Beethoven gehen auf

seine Arbeiten zurück und können somit als weitestgehend fehlerfrei angesehen werden. Auch die von Jean-François Lucarelli kodierten Sinfonien von F. Schubert sind mit einem geschätzten Fehleranteil von 3,15%–4,58% relativ fehlerarm. Hier zeigen sich zudem die Auswirkungen eines großen Stichprobenumfangs: Mit einer Breite von 1,43% ist dieses Konfidenzintervall am schmalsten, die Schätzung somit am genauesten. Die Sinfonien J. Haydns, die von Belinda Campbell und Steven Edwards kodiert wurden, weisen mit zweistelligen prozentualen Obergrenzen eine deutlich höhere Fehlerrate auf als die Werke der anderen Komponisten (Vgl. Tabelle 4.2 im vorigen Abschnitt). Im Falle der von Campbell kodierten Daten ist gemäß der Schätzung sogar davon auszugehen, dass im Schnitt jeder zehnte Ton falsch notiert wurde. Die hohe Intervallbreite von ca. 5% deutet allerdings in beiden Fällen auf einen vergleichsweise geringen Stichprobenumfang hin. Mit Blick auf das nicht-zufällige Verfahren der Stichprobenziehung, bei dem gezielt fehlerhafte Stücke ausgewählt wurden, sind allerdings alle Fehlerschätzungen sehr pessimistisch und insofern als Grenzfall aufzufassen. Somit kann davon ausgegangen werden, dass die durchschnittliche Qualität der kodierten Daten höher ist, als durch die Ergebnisse der Fehlerschätzung suggeriert wird.

Bei der 'manuellen' Analyse der Stichproben wurden verschiedene Ursachen für die Inkonsistenzen ausgemacht. Die mit Abstand häufigste Fehlerquelle waren falsche Oktavangaben. In den meisten Fällen handelte es sich dabei um einzelne 'Ausrutscher' oder kurze Passagen, lediglich im Material von Steven Edwards war eine gesamte Hornstimme eine Oktave zu tief notiert. Alle 67 Fehler in der Stichprobe von Belinda Campbell gehen auf eine inkonsistente Instrumentierung zurück: In der hier untersuchten Trio-Sektion wurde dem Violoncello fälschlicherweise Notenmaterial der Bassstimme zugewiesen. Darüber hinaus wurden in der aufwändigen Fehleruntersuchung einzelne Fälle beobachtet, in denen in der Partitur nicht existente Töne kodiert waren. Dabei schien es sich in der Regel um methodische Fehler zu handeln, die gewissen Praktiken des Eingabeverfahrens geschuldet sein dürften: So wies in einem Fall ein in mehreren Stimmen pausierter Takt musikalisches Material eines vorangegangenen Taktes auf, dessen Inhalt scheinbar kopiert und fälschlicherweise für alle Stimmen übernommen worden war.

4.1.3 Fazit

Das analysierte Korpus umfasst einen Großteil der Werke der vier Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert und bildet damit eine repräsentative Auswahl an Orchesterwerken aus der Periode der Wiener Klassik.

Für den überwiegenden Teil der 14 Orchesterinstrumente stehen über 200 Datensätze zur Auswertung bereit. Die Häufigkeit des Auftretens der der einzelnen Instrumente variiert dabei nicht nur insgesamt, sondern auch bei den verschiedenen Komponisten. Deutliche Unterschiede zwischen den Komponisten zeigen sich in der Verwendungshäufigkeit der Klarinette wie auch der anderen Holzblasinstrumente (mit Ausnahme des Kontrafagotts). Auch bei der Verwendungshäufigkeit der Trompete konnten Unterschiede zwischen den Komponisten nachgewiesen werden.

Mit Blick auf die Validität des kodierten Tonmaterials zeigen das beschriebene Verfahren der Fehlerschätzung sowie der damit verbundene manuelle Abgleich der Stichproben mit den Partituren, dass Inkonsistenzen der Datengrundlage weitgehend auf methodische Fehler der Eingabepraxis sowie Fehler in der Kodierung der Instrumente zurückgehen. Bedingt durch die heterogene Urheberschaft des verwendeten Materials sind diese Unstimmigkeiten bei den analysierten Werken bzw. Komponisten unterschiedlich stark ausgeprägt.

4.2 Vergleichende Untersuchungen

Abbildung 4.1 zeigt eine Übersicht der analysierten Tonhöhenverteilungen, in der die summierten Verteilungen der einzelnen Instrumente für die verschiedenen Komponisten aufgeführt sind. Die hier dargestellten Boxplots zeigen den Interquartilsabstand, welcher den farbigen Boxen entspricht, die darin eingezeichnete vertikale Linie markiert den Median. Die Whisker geben den gesamten Tonumfang der Instrumente an.

Hinsichtlich ihrer Lage- und Streuungsmaße weisen die summierten Verteilungen der vier Komponisten bei diversen Instrumenten sichtbare Unterschiede auf, welche auf individuelle oder historisch bedingte Veränderungen beim Einsatz dieser Instrumente hindeuten und in den folgenden Unterkapiteln näher untersucht werden sollen. Der 'unmögliche' Tonumfang der zweiten Violine bei Haydn – mit einer Untergrenze bei F_3 spielt diese einen Ganzton tiefer als physikalisch möglich – ist durch den dritten Satz der 67. Sinfonie begründet, in dem die zweite Violinstimme zeitweise einen Ton tiefer gestimmt wird, wie der Abgleich mit der entsprechenden Partitur zeigt.

Neben dieser groben Übersicht wurden die Tonhöhenverteilungen einzelner Instrumente durch Histogramme visualisiert. In den Abbildungen 4.2, 4.3, 4.4, 4.5 und 4.6 sind exemplarisch die Tonhöhenverteilungen der Streichergruppe für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert dargestellt. Auch

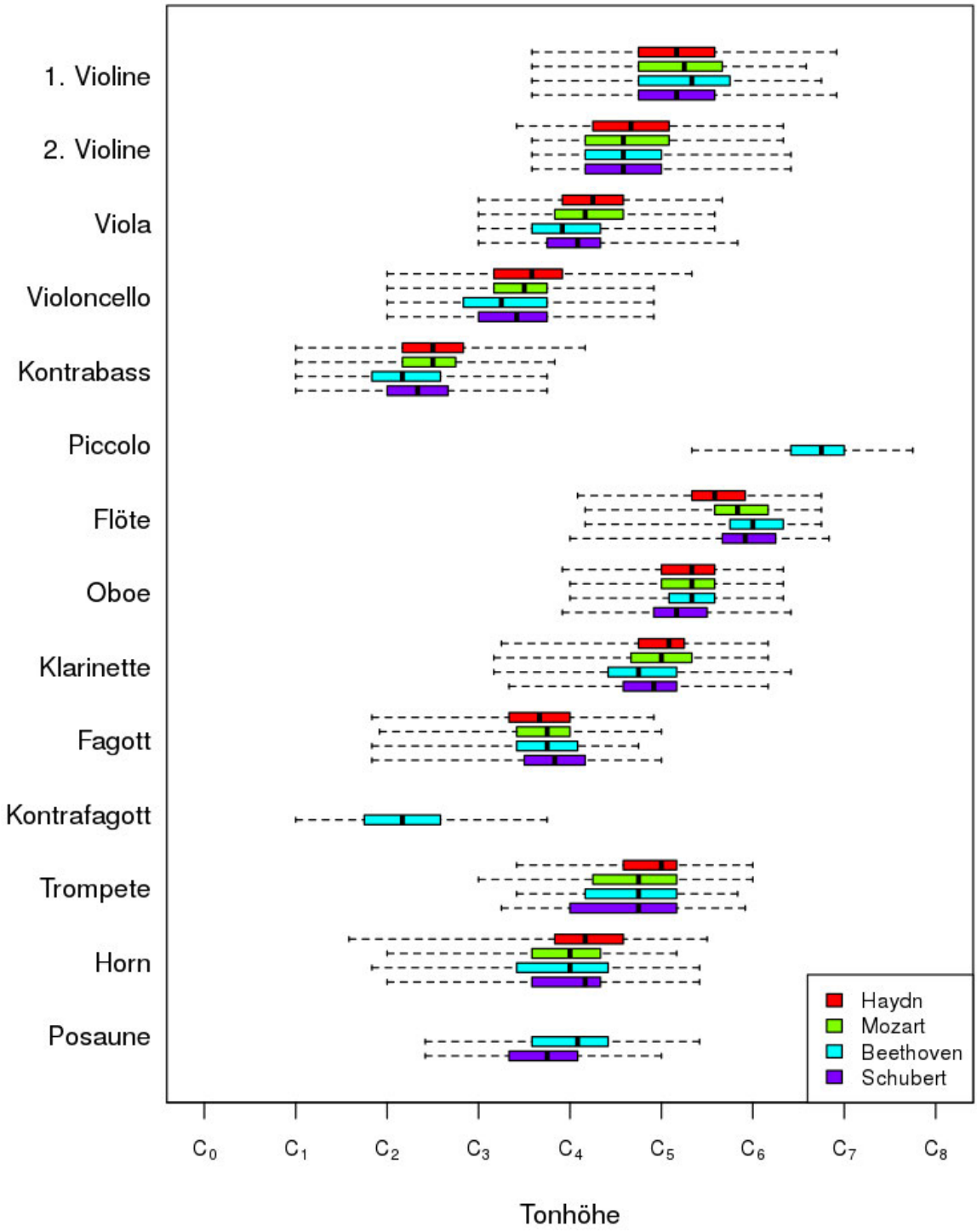

Abbildung 4.1: Summierte Tonhöhenverteilungen der Orchesterinstrumente in allen untersuchten Sinfonien für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert. Die Boxplots zeigen den Median, den Interquartilabstand und den gesamten Tonumfang der Instrumente.

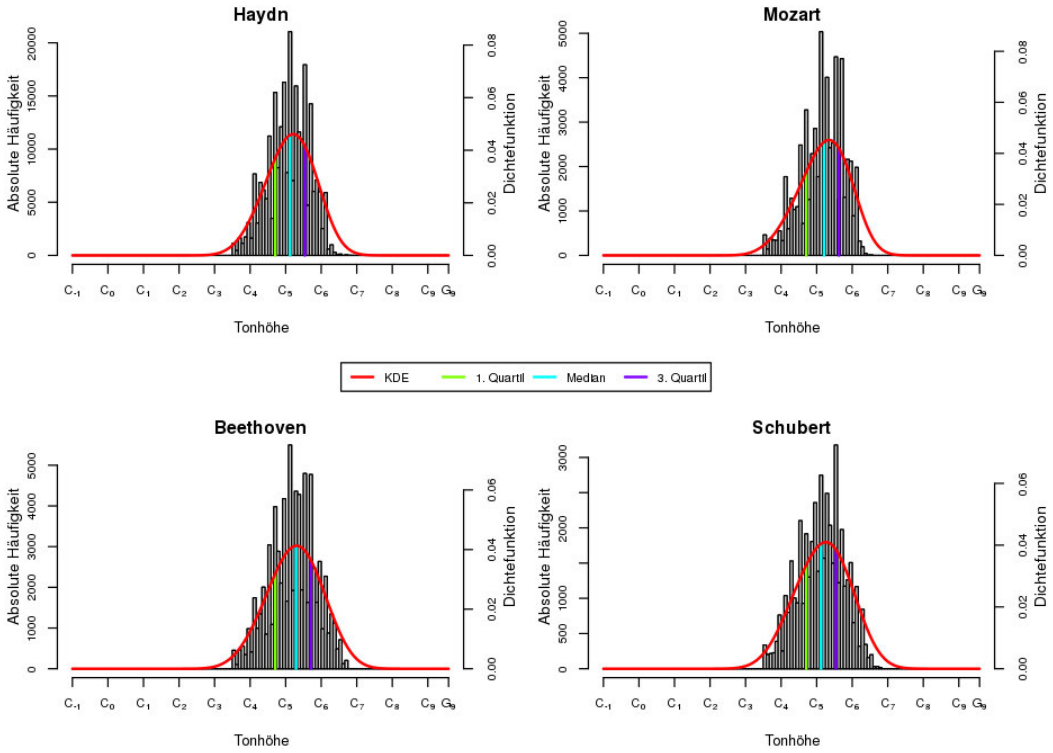

Abbildung 4.2: Tonhöhenverteilungen der ersten Violine für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich.

hier sind Median und Interquartilsabstand eingezeichnet, zusätzlich wird die durch Kerndichteschätzung approximierte Dichtefunktion (KDE) dargestellt. Die Abszissenachse entspricht dem gesamten Tonumfang der MIDI-Repräsentation.
Die Tonhöhenverteilungen sind mit einigen Abweichungen annähernd normalverteilt. Beim Vergleich der Histogramme der ersten und zweiten Violinstimme werden kompositorische Unterschiede hinsichtlich beider Stimmen deutlich: Die zweite Violine spielt nicht nur tiefer, darüber hinaus wird der tiefste spielbare Ton bei den Komponisten W.A. Mozart, L.v. Beethoven und F. Schubert deutlich häufiger eingesetzt als bei J. Haydn, die Verteilungen sind leicht rechtsschief. Gleiches gilt bei Beethoven für die Tonhöhenverteilungen von Viola und Violoncello.

4.2.1 Tonhöhe

Die Ergebnisse der Untersuchung der mittleren Tonhöhe werden im Folgenden dargelegt. Der unter Verwendung des Intercept-Only-Modells geschätzte Intraklassen-Korrelationskoeffizient des Faktors „Satz“ beträgt 0,0345, es werden also 3,45% der Gesamtvarianz durch die Gruppierung der Daten auf Satzebene erklärt.

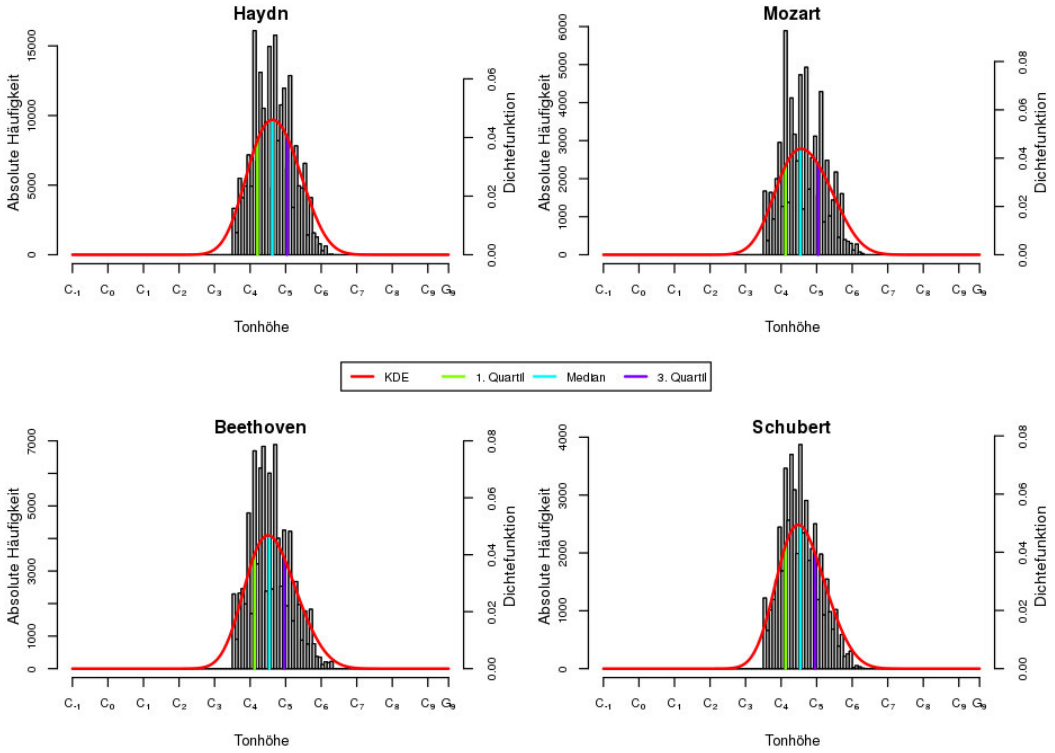

Abbildung 4.3: Tonhöhenverteilungen der zweiten Violine für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich.

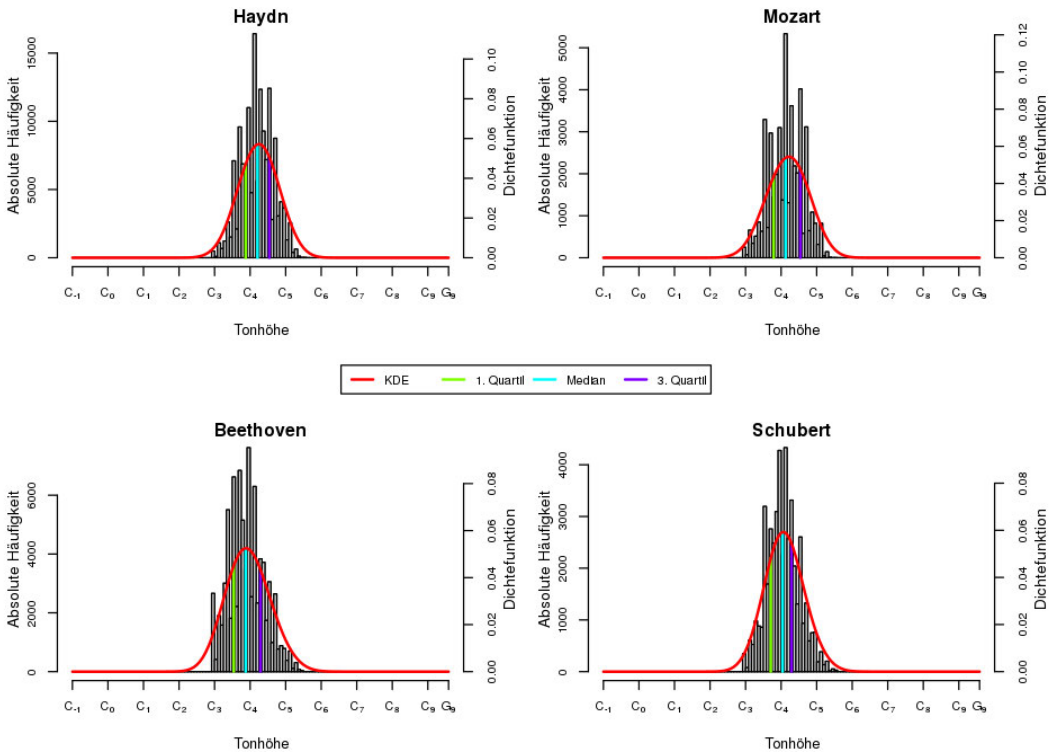

Abbildung 4.4: Tonhöhenverteilungen der Viola für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich.

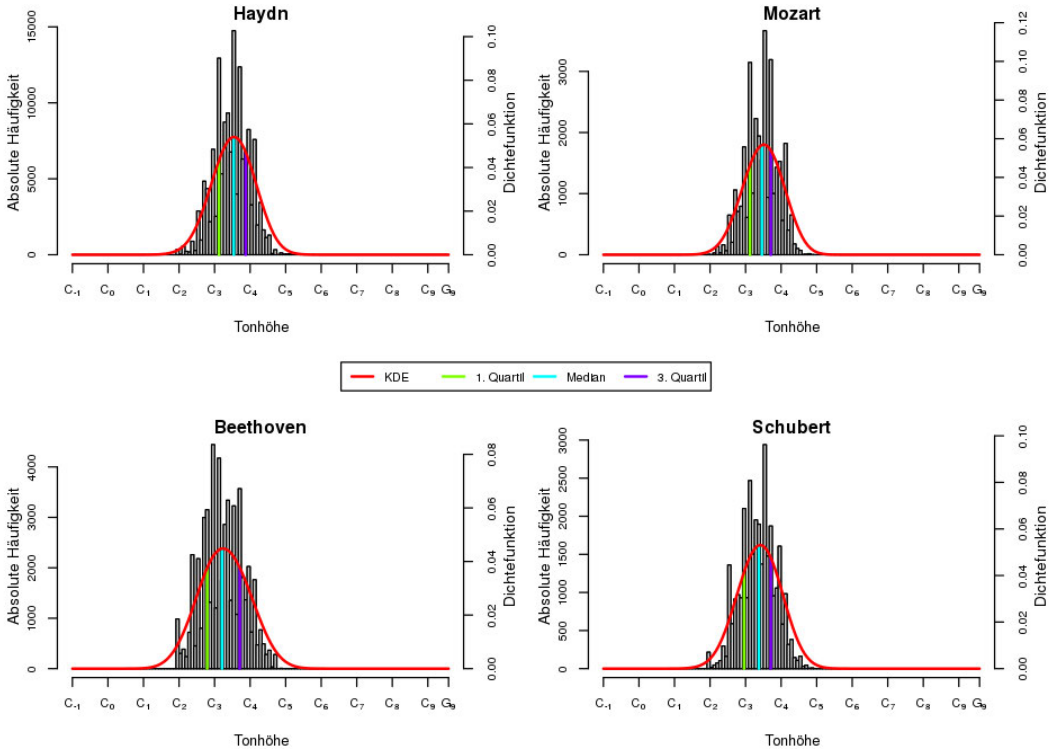

Abbildung 4.5: Tonhöhenverteilungen des Violoncello für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich.

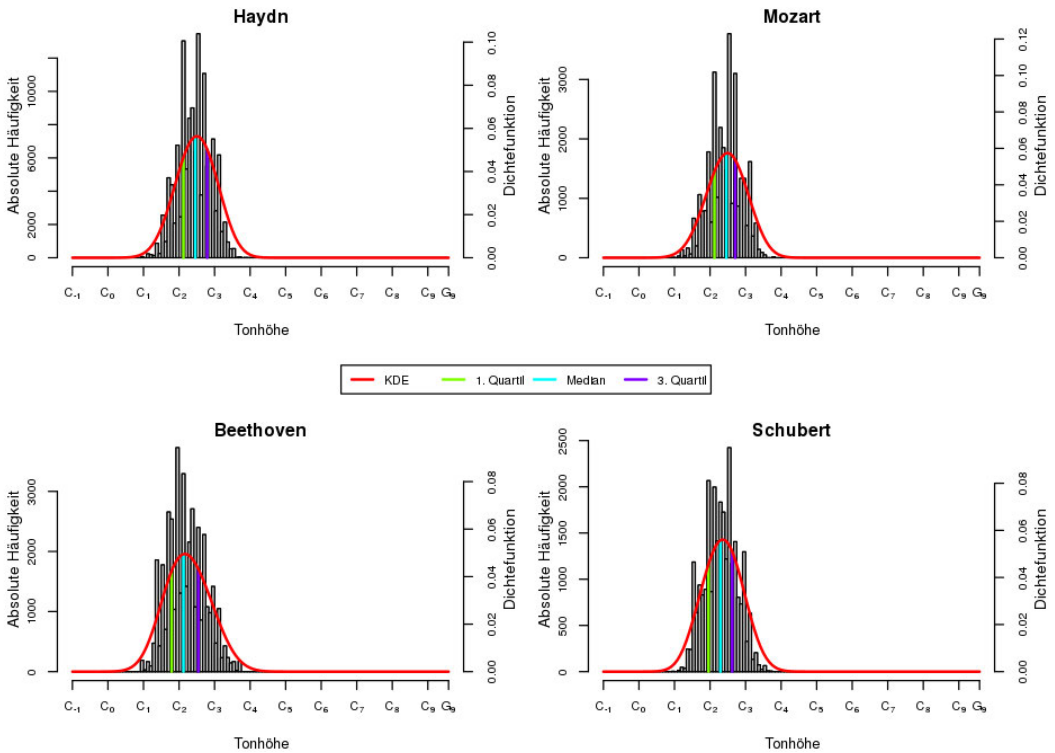

Abbildung 4.6: Tonhöhenverteilungen des Kontrabass für die Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert im Vergleich.

Tabelle 4.5 illustriert die verschiedenen Schritte der Anpassung des statistischen Modells der mittleren Tonhöhe. Darin ist erkennbar, dass die Instrumente einen hochsignifikanten Effekt bezüglich der mittleren Tonhöhe aufweisen. Ebenfalls signifikant ist die Interaktion zwischen Komponist und Instrument: Diese weist auf systematische Unterschiede hinsichtlich der Verwendung einzelner Instrumente bei den verschiedenen Komponisten hin. Weiterhin liegt eine Interaktion zwischen der mittleren Tonhöhe der Instrumente und der Werknummer vor – die mittlere Tonhöhe einzelner Instrumente variiert bei unterschiedlichen Werken also stark. Die abschließend betrachtete Dreifachinteraktion zwischen Komponist, Instrument und Werknummer ist ebenfalls signifikant, der steigende Wert des Bayesschen Informationskriteriums (BIC) weist allerdings auf ein Overfitting des Modells bei Verwendung dieses Faktors hin. Die dadurch erklärte Varianz steht somit nicht mehr im Verhältnis zur resultierenden Modellkomplexität, diese Dreifachinteraktion sollte also nicht ins finale Modell eingehen.

Tabelle 4.5: Modellanpassung für die mittlere Tonhöhe.

Variablenname	DF	-2LL	AIC	BIC	p (LR-Test)
Intercept		20661	20667	20685	
Komponist	3	20660	20672	20707	0,743
Instrument	10	12704	12736	12830	< 0,001***
Werknummer	1	12704	12738	12838	0,742
Komponist x Instrument	30	12265	12359	12636	< 0,001***
Komponist x Werknummer	3	12263	12363	12658	0,642
Instrument x Werknummer	10	12104	12218	12554	< 0,001***
Komponist x Instrument x Werknummer	30	12013	12193	12723	< 0,001***

Das optimale Modell setzt sich somit zusammen aus den drei Haupteffekten „Komponist“, „Instrument“ und „Werknummer“, sowie den Interaktionen „Komponist x Werknummer“ und „Instrument x Werknummer“. Wegen dieser Interaktionen müssen die Parameter „Komponist“ und „Werknummer“ in das Modell aufgenommen werden: Aufgrund der mangelnden Signifikanz der Haupteffekte könnten sie sonst vernachlässigt werden. Weiterhin gehen der Intercept und der Zufallseffekt des Satzes in das Modell ein. Die lineare Gleichung des Modells wird in R-Notation wie in Gleichung 4.1 ausgedrückt:

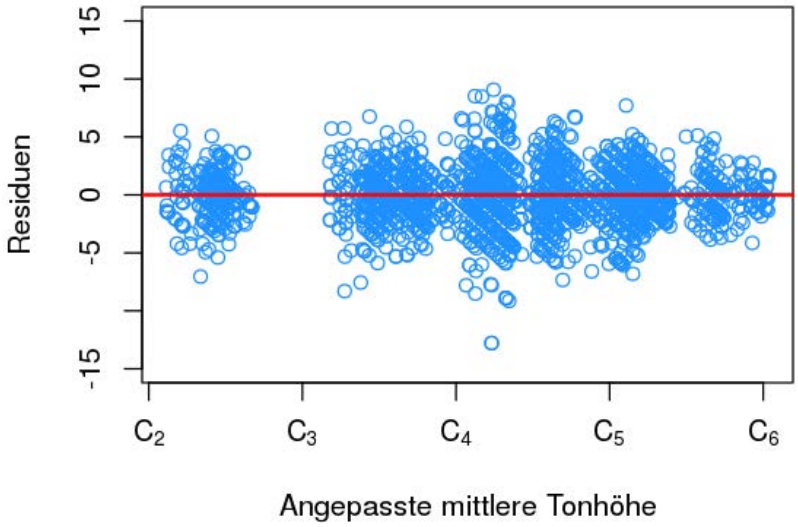

Abbildung 4.7: Residuen des statistischen Modells der mittleren Tonhöhe, aufgetragen über der Tonhöhe.

$$\begin{aligned} Pitch_{Median} \sim 1 + Komponist + Instrument + Werknummer + (1|Satz) \\ + Komponist : Instrument + Instrument : Werknummer \end{aligned} \tag{4.1}$$

Abbildung 4.7 zeigt die Residuen des Modells für alle betrachteten Tonhöhen. Ihre Verteilung besitzt einen Mittelwert von 0, ihre Standardabweichung beträgt 2,3 Halbtonschritte, die Varianz 5,3 Halbtonschritte. Bezüglich der Streuung der angepassten Werte sind keine Schwankungen bzw. systematische Tendenzen erkennbar, die Residuen weisen also eine konstante Varianz auf (*Homoskedastizität*). Diese Annahme wird durch Abbildung 4.8 erhärtet: Aus dem dort dargestellten Histogramm wird die Normalverteilung der Residuen ersichtlich. Auch Abbildung 4.9 bestätigt die Güte des Modells: Darin sind die angepassten gegen die tatsächlichen Tonhöhenkennwerte aufgetragen. Die Verteilung folgt der eingezeichneten Diagonale, die Schätzungen stimmen somit weitestgehend mit den Originalwerten überein.

Das nach McFadden berechnete Pseudo-Bestimmtheitsmaß hat einen Wert von 0,414 und zeigt, dass mit 41,4% ein großer Teil der Varianz in den mittleren Tonhöhen durch das Modell bzw. die darin enthaltenen Prädiktoren erklärt wird. Tatsächlich liegt der Wert sogar oberhalb des anzustrebenden Wertebereichs von 20%–40%[1], das Modell besitzt also eine hohe Güte.

[1]Vgl. Kapitel 3.2.2.

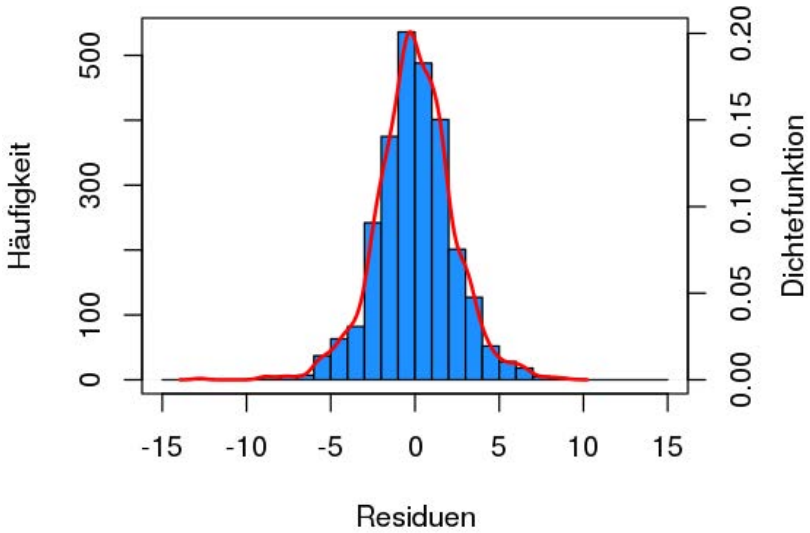

Abbildung 4.8: Histogramm der Residuen des statistischen Modells der mittleren Tonhöhe mit durch Kerndichteschätzung gewonnener Dichtefunktion.

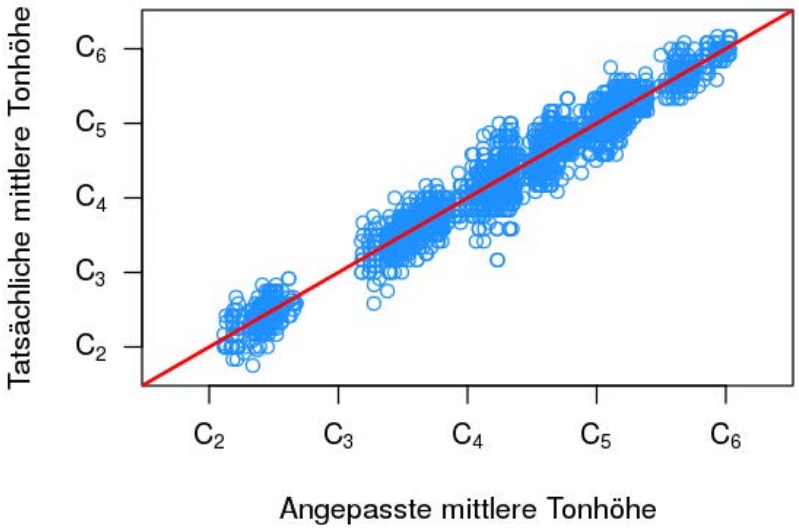

Abbildung 4.9: Vergleich der tatsächlichen und angepassten Tonhöhenwerte.

Die Betrachtung der einzelnen Modellparameter erfolgt auf Grundlage der Tabellen 4.6, 4.7, 4.8 und 4.9. Darin sind die geschätzte mittlere Tonhöhe der verschiedenen Faktoren bzw. deren Interaktionen sowie die zugehörigen Standardfehler, t-und p-Werte aufgeführt. Jede Tabelle zeigt einen anderen Komponisten als Default und gibt die Parameterwerte in Relation dazu an, wodurch die Parameterschätzungen aller vier Komponisten besser vergleichbar werden. Im Folgenden soll auf die aus den Parametertabellen ersichtlichen signifikanten Effekte eingegangen werden.

Das Tonmaterial von J. Haydn hat einen mittleren MIDI-Tonhöhenwert von 62,66, welcher zwischen den ISO-Tonhöhen D_4 und Dis_4 liegt. Die mittlere Tonhöhe des Materials von W.A. Mozart und L.v. Beethoven liegt etwa einen Halbtonschritt höher, die von F. Schubert liegt mit einem E_4 sogar einen Ganzton darüber. Die über alle Instrumente gemittelte Tonhöhe steigt also in der Periode der Wiener Klassik – in Bezug auf die hier analysierten Komponisten und Orchesterwerke – mit der Zeit leicht an.

Hinsichtlich der mittleren Tonhöhe der Komponisten Mozart, Beethoven und Schubert weisen alle Instrumente hochsignifikante Unterschiede auf. Lediglich in Hinblick auf Haydn weist die Tonhöhe der Viola keinerlei Signifikanz auf, auch die Klarinette zeigt nur einen knapp signifikanten Unterschied.

Die Werte des Parameterschätzers der Instrumente entsprechen den Differenzen zur mittleren Tonhöhe des Horns in Bezug auf den jeweiligen Komponisten und korrespondieren mit den in Abbildung 4.1 eingezeichneten Medianwerten der Tonhöhenverteilungen. Die Unterschiede in der mittleren Tonhöhe der einzelnen Instrumente werden durch Abbildung 4.10 veranschaulicht: Dort sind die tatsächlichen und die angepassten Modellwerte über alle Datensätze aufgetragen. Es ist gut erkennbar, dass die Kurve der geschätzten Tonhöhe die tatsächlichen Werte approximiert. Ihre Welligkeit ist als Resultat des Satz-Effektes zu deuten, der die geschätzte mittlere Tonhöhe im Laufe der Sinfonien leicht abfallen lässt.

Der Schätzer der Werknummer zeigt einen schwachen positiven Trend, demzufolge die mittlere Tonhöhe der Sinfonien mit jedem Werk um 2,7 Cent zunimmt. Bei Betrachtung der Interaktionen zwischen den Instrumenten und der Werknummer zeigt sich, dass diese Erscheinung vor allem auf die Klarinette zurückzuführen sein dürfte, deren mittlere Tonhöhe dem Modell zufolge mit jedem Werk um 26 Cent steigt. Diese Tendenz wird jedoch durch die signifikanten Unterschiede der Streicher und der Trompete ausgeglichen, welche bei zunehmendem Repertoire im Mittel jeweils ca. 4–5 Cent tiefer spielen. Abbildung 4.11 zeigt die tatsächliche und modellierte mittlere Tonhöhe in Bezug zur Werknummer, bei deren Betrachtung der Grund dieses Effektes

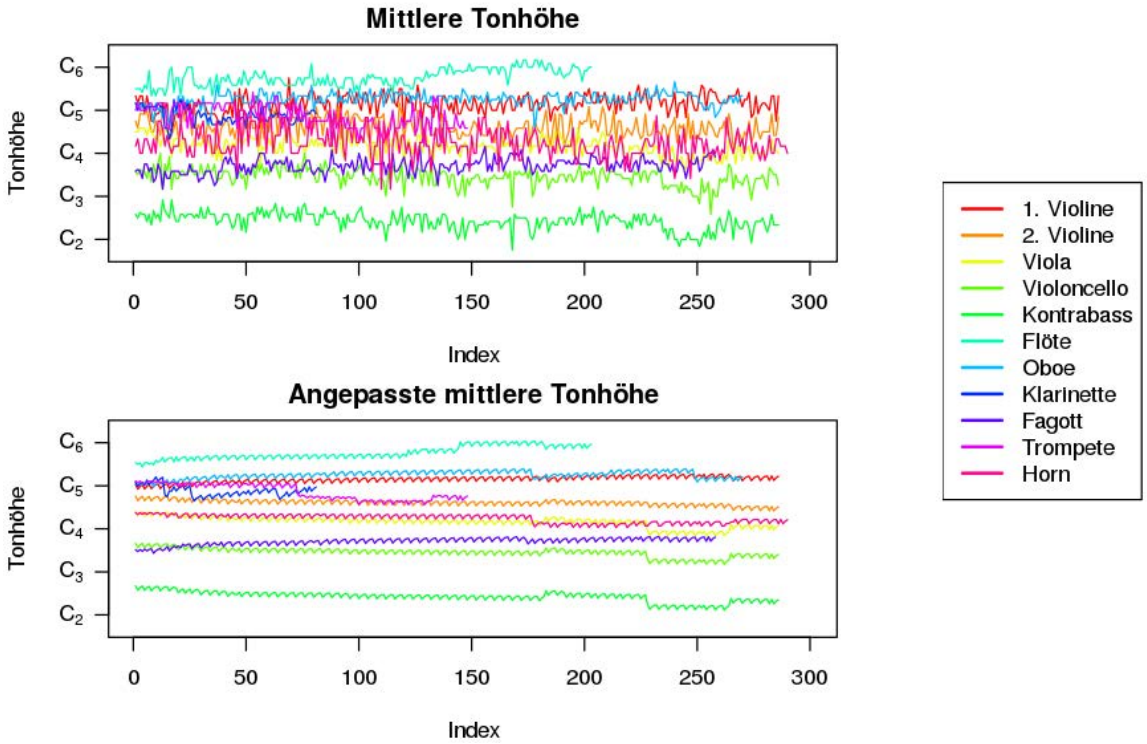

Abbildung 4.10: Tatsächliche und angepasste mittlere Tonhöhe der Instrumente, aufgetragen über die gesamte Messreihe.

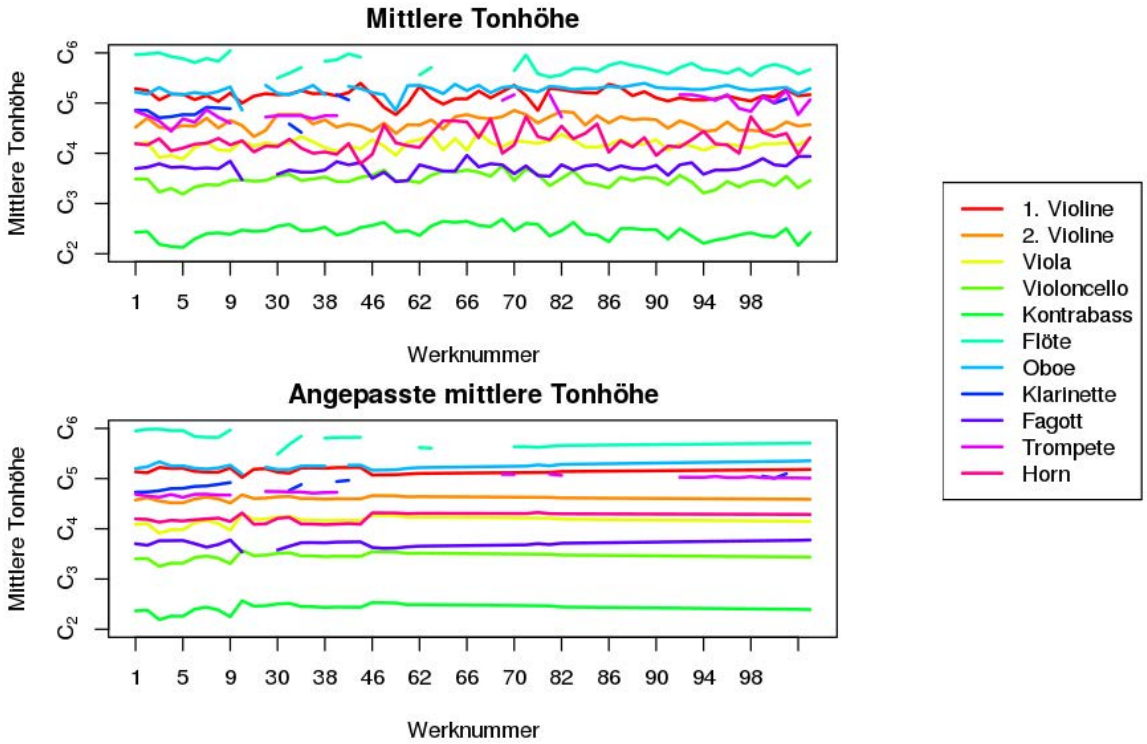

Abbildung 4.11: Tatsächliche und angepasste mittlere Tonhöhe der Instrumente, aufgetragen über die Werknummer.

ersichtlich wird: Der größte Teil der analysierten Werke geht auf den Komponisten J. Haydn zurück, dessen sinfonisches Repertoire deutlich umfangreicher ist als das der anderen Komponisten. Das Repertoire W.A. Mozarts liegt bis einschließlich der Werknummer 42 vor, die Werke von L.v. Beethoven und F. Schubert gehen lediglich in den einstelligen Bereich ein. Ab Werknummer 46 wird die mittlere Tonhöhe also nur auf Basis der Werke eines Komponisten geschätzt, was in einer Linearisierung des Schätzers resultiert. Die zuvor beobachteten Tendenzen von Klarinette und Streichern sind hier gut erkennbar.

Die Interaktionen zwischen Instrumenten und Komponisten geben Aufschluss über Unterschiede in der mittleren Tonhöhe der Instrumente zwischen einzelnen Komponisten, deren Beschreibung durch die grafische Darstellung in Abbildung 4.12 ergänzt wird. Die große Ähnlichkeit der darin abgebildeten angepassten Werte zu den tatsächlichen Mittelwerten bestätigt das hohe Bestimmtheitsmaß des Modells.

Während die Verteilungen der erste Violine keine signifikanten Unterschiede aufweisen, fällt die Tonhöhe der zweiten Violinstimme im Laufe des betrachteten Zeitraums. Im Vergleich zu J. Haydn spielt sie bei W.A. Mozart beinahe einen ganzen Ton tiefer, bei L.v. Beethoven beträgt die Differenz 3 Halbtonschritte und bei F. Schubert sogar zwei Ganztöne. Auch die mittlere Tonhöhe der anderen Streichinstrumente hat bei diesen drei Komponisten ausnahmslos einen niedrigeren Wert als bei Haydn, fällt jedoch nicht mit der Zeit: Beethoven weist bei diesen Instrumenten mit einer Differenz von etwa 6 Halbtonschritten zu Haydn stets die niedrigste mittlere Tonhöhe auf, gefolgt von Schubert, bei dem die mittleren Tonhöhen von Viola und Kontrabass 5 Halbtonschritte, die des Cellos gute 4 Halbtonschritte unter den Werten Haydns liegen. In den Werken W.A. Mozarts spielen diese Instrumente lediglich einen Ganzton tiefer als bei Haydn. Ein ähnliches Muster findet sich bei der Trompete: Auch hier fällt die mittlere Tonhöhe in den Werken von Haydn über Mozart und Beethoven sukzessive ab, bei Schubert steigt sie schließlich wieder um einen Halbtonschritt. Die mittlere Tonhöhe der Flöte hingegen steigt von Haydn über Mozart und Beethoven um jeweils etwa einen Ganzton an, bevor sie bei Schubert einen Halbtonschritt fällt. Die mittlere Tonhöhe der Oboe liegt bei Beethoven einen Ganzton über dem Kennwert der anderen Komponisten und unterscheidet sich signifikant von der Verwendung bei Schubert und Haydn. Zudem liegt ein hochsignifikanter Unterschied im Vergleich der Oboe zwischen Mozart und Schubert vor. Die mittlere Tonhöhe der Klarinette weist von allen betrachteten Instrumenten die größten Differenzen zwischen den Komponisten auf. Im geschätzten Mittel spielt sie bei Beethoven und Schubert 22 Halbtonschritte und damit beinahe zwei Oktaven höher als bei Haydn, und auch bei

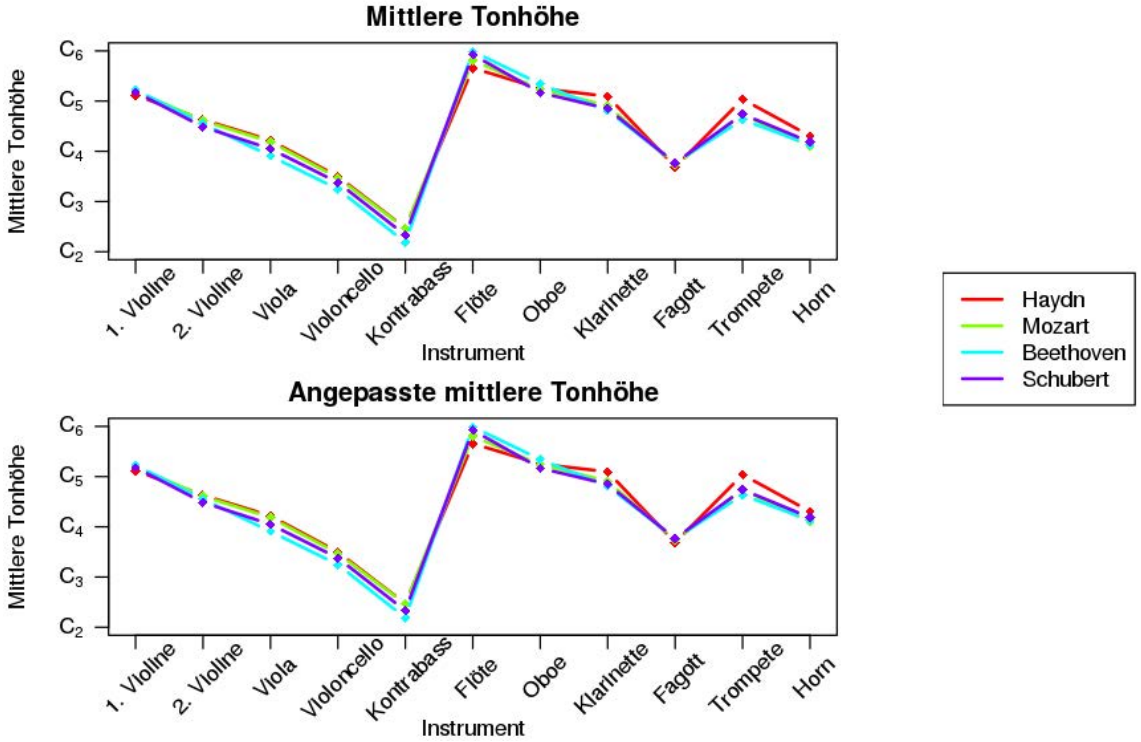

Abbildung 4.12: Tatsächliche und angepasste mittlere Tonhöhe der Instrumente bei den Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert.

Mozart liegt der Schätzwert eine gute Oktave höher als bei Haydn. Allerdings ist bei der Klarinette auch der Standardfehler sehr groß, ihre mittlere Tonhöhe scheint also starken Schwankungen zu unterliegen, die mit dem hohen Dynamikumfang dieses Instrumentes erklärt werden können.

Tabelle 4.6: Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Haydn.

Parameter	Schätzer	SE	t	p
Intercept	62,656	0,924	67,797	< 0,001***
Mozart[1]	1,065	0,584	1,823	0,068
Beethoven[1]	1,250	0,861	1,451	0,147
Schubert[1]	1,541	0,865	1,781	0,075
1. Violine[2]	9,079	0,985	9,218	< 0,001***
2. Violine[2]	5,893	0,985	5,984	< 0,001***
Viola[2]	1,667	0,985	1,693	0,091
Violoncello[2]	-7,208	0,985	-7,318	< 0,001***
Kontrabass[2]	-19,086	0,985	-19,378	< 0,001***
Flöte[2]	15,084	1,108	13,611	< 0,001***
Oboe[2]	9,468	1,000	9,470	< 0,001***
Klarinette[2]	-18,503	8,759	-2,113	< 0,05*
Fagott[2]	-9,057	1,007	-8,993	< 0,001***
Trompete[2]	11,322	1,786	6,341	< 0,001***
Werknummer[3]	0,027	0,009	2,958	< 0,01**
Mozart x 1. Violine	0,849	0,712	1,193	0,233
Beethoven x 1. Violine	1,522	1,001	1,521	0,128
Schubert x 1. Violine	0,712	1,043	0,683	0,495
Mozart x 2. Violine	-1,923	0,712	-2,699	< 0,01**
Beethoven x 2. Violine	-3,209	1,001	-3,206	< 0,01**
Schubert x 2. Violine	-4,264	1,043	-4,088	< 0,001***
Mozart x Viola	-2,446	0,712	-3,434	< 0,001***
Beethoven x Viola	-6,633	1,001	-6,628	< 0,001***
Schubert x Viola	-5,211	1,043	-4,996	< 0,001***
Mozart x Violoncello	-2,284	0,712	-3,207	< 0,01**
Beethoven x Violoncello	-5,695	1,001	-5,690	< 0,001***
Schubert x Violoncello	-4,400	1,043	-4,218	< 0,001***
Mozart x Kontrabass	-2,308	0,712	-3,240	< 0,01**
Beethoven x Kontrabass	-6,528	1,001	-6,523	< 0,001***
Schubert x Kontrabass	-5,080	1,043	-4,870	< 0,001***
Mozart x Flöte	1,940	0,826	2,350	< 0,05*
Beethoven x Flöte	4,679	1,110	4,214	< 0,001***
Schubert x Flöte	3,602	1,148	3,138	< 0,01**
Mozart x Oboe	0,362	0,739	0,490	0,624
Beethoven x Oboe	2,444	1,014	2,411	< 0,05*
Schubert x Oboe	0,057	1,055	0,054	0,957
Mozart x Klarinette	14,826	5,581	2,656	< 0,01**
Beethoven x Klarinette	22,749	8,325	2,733	< 0,01**
Schubert x Klarinette	22,904	8,315	2,754	< 0,01**
Mozart x Fagott	0,703	0,759	0,926	0,354
Beethoven x Fagott	2,110	1,020	2,069	< 0,05*
Schubert x Fagott	1,784	1,061	1,681	0,093
Mozart x Trompete	-5,933	1,241	-4,782	< 0,001***
Beethoven x Trompete	-7,704	1,730	-4,453	< 0,001***
Schubert x Trompete	-6,700	1,765	-3,796	< 0,001***
1. Violine x Werknummer	-0,004	0,010	-0,359	0,719
2. Violine x Werknummer	-0,042	0,010	-4,015	< 0,001***
Viola x Werknummer	-0,052	0,010	-5,004	< 0,001***
Violoncello x Werknummer	-0,049	0,010	-4,673	< 0,001***
Kontrabass x Werknummer	-0,055	0,010	-5,280	< 0,001***
Flöte x Werknummer	-0,001	0,012	-0,083	0,934
Oboe x Werknummer	0,012	0,011	1,169	0,243
Klarinette x Werknummer	0,260	0,086	3,007	< 0,01**
Fagott x Werknummer	0,008	0,011	0,782	0,434
Trompete x Werknummer	-0,045	0,019	-2,361	< 0,05*

[1] Dummykodiert (Default: Haydn)
[2] Effektkodiert (Default: Horn)
[3] metrische Variable

Tabelle 4.7: Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Mozart.

Parameter	Schätzer	SE	t	p
Intercept	63,721	0,415	238,814	<0,001***
Haydn[1]	-1,065	0,584	-1,781	0,075
Beethoven[1]	0,185	0,333	-1,377	0,169
Schubert[1]	0,476	0,346	-1,532	0,126
1. Violine[2]	9,929	0,498	20,674	<0,001***
2. Violine[2]	3,971	0,498	3,439	<0,001***
Viola[2]	-0,779	0,498	-7,483	<0,001***
Violoncello[2]	-9,492	0,498	-24,508	<0,001***
Kontrabass[2]	-21,394	0,498	-51,022	<0,001***
Flöte[2]	17,024	0,616	39,380	<0,001***
Oboe[2]	9,830	0,529	20,105	<0,001***
Klarinette[2]	-3,678	3,281	6,370	<0,001***
Fagott[2]	-8,353	0,585	-15,349	<0,001***
Trompete[2]	5,389	0,764	8,338	<0,001***
Werknummer[3]	0,027	0,009	2,958	<0,01**
Haydn x 1. Violine	-0,849	0,712	-0,683	0,495
Beethoven x 1. Violine	0,673	0,585	0,209	0,835
Schubert x 1. Violine	-0,137	0,657	1,365	0,172
Haydn x 2. Violine	1,923	0,712	4,088	<0,001***
Beethoven x 2. Violine	-1,286	0,585	3,566	<0,001***
Schubert x 2. Violine	-2,342	0,657	1,779	0,075
Haydn x Viola	2,446	0,712	4,996	<0,001***
Beethoven x Viola	-4,187	0,585	4,210	<0,001***
Schubert x Viola	-2,765	0,657	-2,397	<0,05*
Haydn x Violoncello	2,284	0,712	4,218	<0,001***
Beethoven x Violoncello	-3,410	0,585	3,221	<0,01**
Schubert x Violoncello	-2,115	0,657	-2,182	<0,05*
Haydn x Kontrabass	2,308	0,712	4,870	<0,001***
Beethoven x Kontrabass	-4,220	0,585	4,220	<0,001***
Schubert x Kontrabass	-2,772	0,657	-2,442	<0,05*
Haydn x Flöte	-1,940	0,826	-3,138	<0,01**
Beethoven x Flöte	2,739	0,681	-2,238	<0,05*
Schubert x Flöte	1,662	0,743	1,815	0,070
Haydn x Oboe	-0,362	0,739	-0,054	0,957
Beethoven x Oboe	2,082	0,611	0,449	0,654
Schubert x Oboe	-0,305	0,680	4,024	<0,001***
Haydn x Klarinette	-14,826	5,581	-2,754	<0,01**
Beethoven x Klarinette	7,923	2,870	-2,808	<0,01**
Schubert x Klarinette	8,079	2,877	-0,249	0,804
Haydn x Fagott	-0,703	0,759	-1,681	0,093
Beethoven x Fagott	1,407	0,657	-1,498	0,134
Schubert x Fagott	1,081	0,722	0,550	0,583
Haydn x Trompete	5,933	1,241	3,796	<0,001***
Beethoven x Trompete	-1,771	0,781	0,883	0,377
Schubert x Trompete	-0,767	0,869	-1,519	0,129
1. Violine x Werknummer	-0,004	0,010	-0,359	0,719
2. Violine x Werknummer	-0,042	0,010	-4,015	<0,001***
Viola x Werknummer	-0,052	0,010	-5,004	<0,001***
Violoncello x Werknummer	-0,049	0,010	-4,673	<0,001***
Kontrabass x Werknummer	-0,055	0,010	-5,280	<0,001***
Flöte x Werknummer	-0,001	0,012	-0,083	0,934
Oboe x Werknummer	0,012	0,011	1,169	0,243
Klarinette x Werknummer	0,260	0,086	3,007	<0,01**
Fagott x Werknummer	0,008	0,011	0,782	0,434
Trompete x Werknummer	-0,045	0,019	-2,361	<0,05*

[1] Dummykodiert (Default: Mozart)
[2] Effektkodiert (Default: Horn)
[3] metrische Variable

Tabelle 4.8: Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Beethoven.

Parameter	Schätzer	SE	t	p
Intercept	63,906	0,246	238,814	< 0,001***
Haydn[1]	-1,250	0,861	-1,781	0,075
Mozart[1]	-0,185	0,333	-1,377	0,169
Schubert[1]	0,291	0,190	-1,532	0,126
1. Violine[2]	10,601	0,366	20,674	< 0,001***
2. Violine[2]	2,685	0,366	3,439	< 0,001***
Viola[2]	-4,966	0,366	-7,483	< 0,001***
Violoncello[2]	-12,902	0,366	-24,508	< 0,001***
Kontrabass[2]	-25,614	0,366	-51,022	< 0,001***
Flöte[2]	19,763	0,367	39,380	< 0,001***
Oboe[2]	11,912	0,366	20,105	< 0,001***
Klarinette[2]	4,245	0,569	6,370	< 0,001***
Fagott[2]	-6,946	0,366	-15,349	< 0,001***
Trompete[2]	3,619	0,388	8,338	< 0,001***
Werknummer[3]	0,027	0,009	2,958	< 0,01**
Haydn x 1. Violine	-1,522	1,001	-0,683	0,495
Mozart x 1. Violine	-0,673	0,585	0,209	0,835
Schubert x 1. Violine	-0,810	0,593	1,365	0,172
Haydn x 2. Violine	3,209	1,001	4,088	< 0,001***
Mozart x 2. Violine	1,286	0,585	3,566	< 0,001***
Schubert x 2. Violine	-1,056	0,593	1,779	0,075
Haydn x Viola	6,633	1,001	4,996	< 0,001***
Mozart x Viola	4,187	0,585	4,210	< 0,001***
Schubert x Viola	1,422	0,593	-2,397	< 0,05*
Haydn x Violoncello	5,695	1,001	4,218	< 0,001***
Mozart x Violoncello	3,410	0,585	3,221	< 0,01**
Schubert x Violoncello	1,295	0,593	-2,182	< 0,05*
Haydn x Kontrabass	6,528	1,001	4,870	< 0,001***
Mozart x Kontrabass	4,220	0,585	4,220	< 0,001***
Schubert x Kontrabass	1,449	0,593	-2,442	< 0,05*
Haydn x Flöte	-4,679	1,110	-3,138	< 0,01**
Mozart x Flöte	-2,739	0,681	-2,238	< 0,05*
Schubert x Flöte	-1,077	0,593	1,815	0,070
Haydn x Oboe	-2,444	1,014	-0,054	0,957
Mozart x Oboe	-2,082	0,611	0,449	0,654
Schubert x Oboe	-2,387	0,593	4,024	< 0,001***
Haydn x Klarinette	-22,749	8,325	-2,754	< 0,01**
Mozart x Klarinette	-7,923	2,870	-2,808	< 0,01**
Schubert x Klarinette	0,156	0,627	-0,249	0,804
Haydn x Fagott	-2,110	1,020	-1,681	0,093
Mozart x Fagott	-1,407	0,657	-1,498	0,134
Schubert x Fagott	-0,326	0,593	0,550	0,583
Haydn x Trompete	7,704	1,730	3,796	< 0,001***
Mozart x Trompete	1,771	0,781	0,883	0,377
Schubert x Trompete	1,004	0,661	-1,519	0,129
1. Violine x Werknummer	-0,004	0,010	-0,359	0,719
2. Violine x Werknummer	-0,042	0,010	-4,015	< 0,001***
Viola x Werknummer	-0,052	0,010	-5,004	< 0,001***
Violoncello x Werknummer	-0,049	0,010	-4,673	< 0,001***
Kontrabass x Werknummer	-0,055	0,010	-5,280	< 0,001***
Flöte x Werknummer	-0,001	0,012	-0,083	0,934
Oboe x Werknummer	0,012	0,011	1,169	0,243
Klarinette x Werknummer	0,260	0,086	3,007	< 0,01**
Fagott x Werknummer	0,008	0,011	0,782	0,434
Trompete x Werknummer	-0,045	0,019	-2,361	< 0,05*

[1] Dummykodiert (Default: Beethoven)
[2] Effektkodiert (Default: Horn)
[3] metrische Variable

Tabelle 4.9: Parameterschätzer: Angepasste mittlere Tonhöhe im Vergleich zu Schubert.

Parameter	Schätzer	SE	t	p
Intercept	64,197	0,269	238,814	< 0,001***
Haydn[1]	-1,541	0,865	-1,781	0,075
Mozart[1]	-0,476	0,346	-1,377	0,169
Beethoven[1]	-0,291	0,190	-1,532	0,126
1. Violine[2]	9,792	0,474	20,674	< 0,001***
2. Violine[2]	1,629	0,474	3,439	< 0,001***
Viola[2]	-3,544	0,474	-7,483	< 0,001***
Violoncello[2]	-11,608	0,474	-24,508	< 0,001***
Kontrabass[2]	-24,165	0,474	-51,022	< 0,001***
Flöte[2]	18,686	0,475	39,380	< 0,001***
Oboe[2]	9,525	0,474	20,105	< 0,001***
Klarinette[2]	4,401	0,691	6,370	< 0,001***
Fagott[2]	-7,272	0,474	-15,349	< 0,001***
Trompete[2]	4,622	0,554	8,338	< 0,001***
Werknummer[3]	0,027	0,009	2,958	< 0,01**
Haydn x 1. Violine	-0,712	1,043	-0,683	0,495
Mozart x 1. Violine	0,137	0,657	0,209	0,835
Beethoven x 1. Violine	0,810	0,593	1,365	0,172
Haydn x 2. Violine	4,264	1,043	4,088	< 0,001***
Mozart x 2. Violine	2,342	0,657	3,566	< 0,001***
Beethoven x 2. Violine	1,056	0,593	1,779	0,075
Haydn x Viola	5,211	1,043	4,996	< 0,001***
Mozart x Viola	2,765	0,657	4,210	< 0,001***
Beethoven x Viola	-1,422	0,593	-2,397	< 0,05*
Haydn x Violoncello	4,400	1,043	4,218	< 0,001***
Mozart x Violoncello	2,115	0,657	3,221	< 0,01**
Beethoven x Violoncello	-1,295	0,593	-2,182	< 0,05*
Haydn x Kontrabass	5,080	1,043	4,870	< 0,001***
Mozart x Kontrabass	2,772	0,657	4,220	< 0,001***
Beethoven x Kontrabass	-1,449	0,593	-2,442	< 0,05*
Haydn x Flöte	-3,602	1,148	-3,138	< 0,01**
Mozart x Flöte	-1,662	0,743	-2,238	< 0,05*
Beethoven x Flöte	1,077	0,593	1,815	0,070
Haydn x Oboe	-0,057	1,055	-0,054	0,957
Mozart x Oboe	0,305	0,680	0,449	0,654
Beethoven x Oboe	2,387	0,593	4,024	< 0,001***
Haydn x Klarinette	-22,904	8,315	-2,754	< 0,01**
Mozart x Klarinette	-8,079	2,877	-2,808	< 0,01**
Beethoven x Klarinette	-0,156	0,627	-0,249	0,804
Haydn x Fagott	-1,784	1,061	-1,681	0,093
Mozart x Fagott	-1,081	0,722	-1,498	0,134
Beethoven x Fagott	0,326	0,593	0,550	0,583
Haydn x Trompete	6,700	1,765	3,796	< 0,001***
Mozart x Trompete	0,767	0,869	0,883	0,377
Beethoven x Trompete	-1,004	0,661	-1,519	0,129
1. Violine x Werknummer	-0,004	0,010	-0,359	0,719
2. Violine x Werknummer	-0,042	0,010	-4,015	< 0,001***
Viola x Werknummer	-0,052	0,010	-5,004	< 0,001***
Violoncello x Werknummer	-0,049	0,010	-4,673	< 0,001***
Kontrabass x Werknummer	-0,055	0,010	-5,280	< 0,001***
Flöte x Werknummer	-0,001	0,012	-0,083	0,934
Oboe x Werknummer	0,012	0,011	1,169	0,243
Klarinette x Werknummer	0,260	0,086	3,007	< 0,01**
Fagott x Werknummer	0,008	0,011	0,782	0,434
Trompete x Werknummer	-0,045	0,019	-2,361	< 0,05*

[1] Dummykodiert (Default: Schubert)
[2] Effektkodiert (Default: Horn)
[3] metrische Variable

4.2.2 Tonumfang

Analog zur statistischen Modellierung der Tonhöhe wird auch der Tonumfang durch ein Mehrebenenmodell beschrieben. Die Modellanpassung ist in Tabelle 4.10 angegeben. Im Ergebnis weisen nicht nur die Instrumente in den betrachteten Korpora hochsignifikante Unterschiede hinsichtlich ihres mittleren Tonumfangs auf. Letzterer variiert auch stark zwischen den Komponisten. Darüber hinaus gibt es signifikante Interaktionen sowohl zwischen der Werknummer und dem Tonumfang der Instrumente als auch zwischen Werknummer und Komponist. Des Weiteren sind die Interaktion zwischen Komponist und Instrument sowie die abschließend betrachtete Dreifachinteraktion zwischen Komponist, Instrument und Werknummer signifikant, aufgrund des steigenden BIC-Wertes werden diese Faktoren jedoch nicht in das Modell aufgenommen um ein Overfitting zu vermeiden.

Tabelle 4.10: Modellanpassung für den mittleren Tonumfang.

Variablenname	DF	-2LL	AIC	BIC	p (LR-Test)
Intercept		11681	11687	11705	
Komponist	3	11624	11636	11671	< 0,001***
Instrument	10	11107	11139	11233	< 0,001***
Werknummer	1	11106	11140	11240	0,396
Komponist x Werknummer	3	11075	11115	11233	< 0,001***
Instrument x Werknummer	10	10971	11031	11208	< 0,001***
Komponist x Instrument	30	10854	10974	11328	< 0,001***
Komponist x Instrument x Werknummer	30	10800	10980	11511	0,004**

Gleichung 4.2 zeigt das erstellte Modell in R-Notation: Es setzt sich neben dem Intercept aus den von der Untersuchung der mittleren Tonhöhe bereits bekannten Hauptfaktoren „Komponist“, „Instrument“ und „Werknummer“, dem zufälligen Effekt „Satz“ sowie den Interaktionen zwischen „Komponist“ und „Werknummer“ sowie „Instrument“ und „Werknummer“ zusammen.

$$\begin{aligned} Ambitus_{IQR} \sim 1 + Komponist + Instrument + Werknummer + (1|Satz) \\ + Komponist : Werknummer + Instrument : Werknummer \end{aligned} \tag{4.2}$$

Auch das Modell des Tonumfangs weist eine recht hohe Genauigkeit auf. Abbildung 4.13 zeigt einen Scatterplot der Residuen. Die darin eingezeichneten Werte sind abgesehen von drei Ausreißern gleichmäßg um die Abszissenachse verteilt. Die

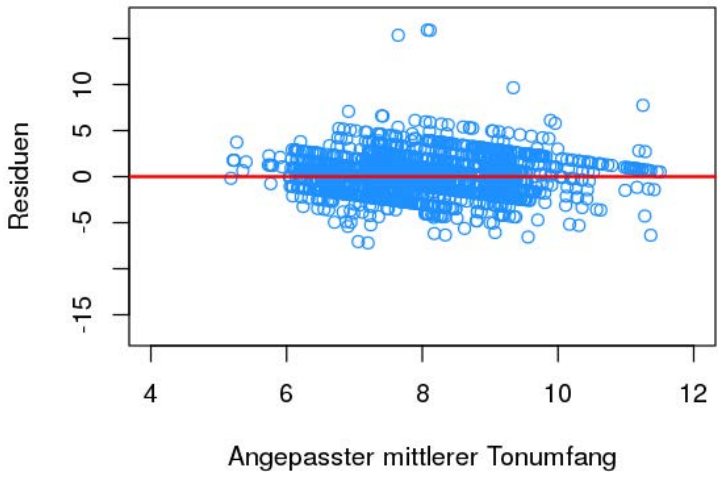

Abbildung 4.13: Residuen des statistischen Modells des mittleren Tonumfangs.

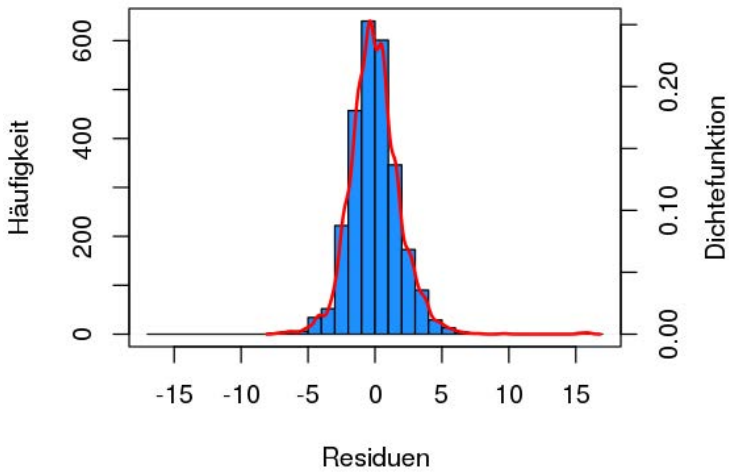

Abbildung 4.14: Histogramm der Residuen des statistischen Modells des mittleren Tonumfangs mit durch Kerndichteschätzung gewonnener Dichtefunktion.

Ausreißer deuten auf einzelne Messfehler bzw. fehlerhafte Datensätze hin. Das Histogramm der Residuen und die zugehörige Dichtefunktion sind in Abbildung 4.14 dargestellt; auch sie entsprechen einer Normalverteilung. Die Standardabweichung der Residuen beträgt 1,87 Halbtonschritte.

Wie bei der Betrachtung der Tonhöhen wurde auch hier das Pseudo-Bestimmtheitsmaß nach McFadden berechnet, es weist allerdings mit 0,06 einen deutlich niedrigeren Wert auf als beim Modell der mittleren Tonhöhe. Mit der Erklärung von 6% der Gesamtvarianz des Tonumfangs ist das hier verwendete Modell somit nicht optimal.

Anders als bei der Untersuchung der mittleren Tonhöhe müssen die Parameterschätzer nicht in Bezug auf alle Komponisten angegeben werden, da die Interaktionen zwischen Komponisten und Instrumenten nicht in das Modell des Tonhöhenumfangs eingehen und daher weniger Werte miteinander zu vergleichen sind. Tabelle 4.11 zeigt die Kennwerte der Parameterschätzer; der mittlere Tonumfang J. Haydns wurde hier als Default gewählt, um historisch-chronologische Entwicklungen leichter ablesen zu können, die anderen Parameter beziehen sich auf diesen Wert.

Tabelle 4.11: Parameterschätzer: Angepasster mittlerer Tonumfang im Vergleich zu Haydn.

Parameter	Schätzer	SE	t	p
Intercept	7.774	0.142	54.667	< 0,001***
Mozart	-1.158	0.313	-3.703	< 0,001***
Beethoven	0.444	0.243	1.825	0.068
Schubert	-0.419	0.303	-1.381	0.167
1. Violine	1.554	0.184	8.428	< 0,001***
2. Violine	0.491	0.184	2.666	< 0,01**
Viola	-0.600	0.184	-3.256	< 0,01**
Violoncello	0.183	0.184	0.993	0.321
Kontrabass	-0.357	0.184	-1.937	0.053
Flöte	-1.656	0.211	-7.830	< 0,001***
Oboe	-1.682	0.191	-8.786	< 0,001***
Klarinette	-1.001	0.244	-4.110	< 0,001***
Fagott	-0.331	0.200	-1.653	0.098
Trompete	2.764	0.236	11.730	< 0,001***
Werknummer	-0.002	0.002	-1.365	0.172
Mozart x Werknummer	0.042	0.009	4.813	< 0,001***
Beethoven x Werknummer	0.080	0.036	2.224	< 0,05*
Schubert x Werknummer	0.097	0.046	2.099	< 0,05*
1. Violine x Werknummer	0.000	0.003	0.086	0.931
2. Violine x Werknummer	0.011	0.003	3.698	< 0,001***
Viola x Werknummer	0.007	0.003	2.196	< 0,05*
Violoncello x Werknummer	-0.001	0.003	-0.249	0.803
Kontrabass x Werknummer	0.001	0.003	0.266	0.790
Flöte x Werknummer	0.009	0.003	2.773	< 0,01**
Oboe x Werknummer	0.008	0.003	2.757	< 0,01**
Klarinette x Werknummer	-0.003	0.005	-0.558	0.577
Fagott x Werknummer	0.002	0.003	0.573	0.566
Trompete x Werknummer	-0.034	0.004	-9.171	< 0,001***

[1] Dummykodiert (Default: Haydn)
[2] Effektkodiert (Default: Horn)
[3] metrische Variable

Hinsichtlich des durchschnittlichen Tonumfangs der Komponisten ist ein starker Unterschied zwischen Haydn und Mozart feststellbar, der in seiner 6. Sinfonie den tiefsten mittleren Tonumfang aller betrachteten Werke aufweist. Zudem unterscheidet sich der mittlere Tonumfang Schuberts von dem Beethovens und Mozarts, wobei hier jedoch nur eine niedrige Signifikanz nachweisbar ist. Die Werte der verschiedenen Komponisten lassen somit keine klaren entwicklungshistorischen Trends erkennen.

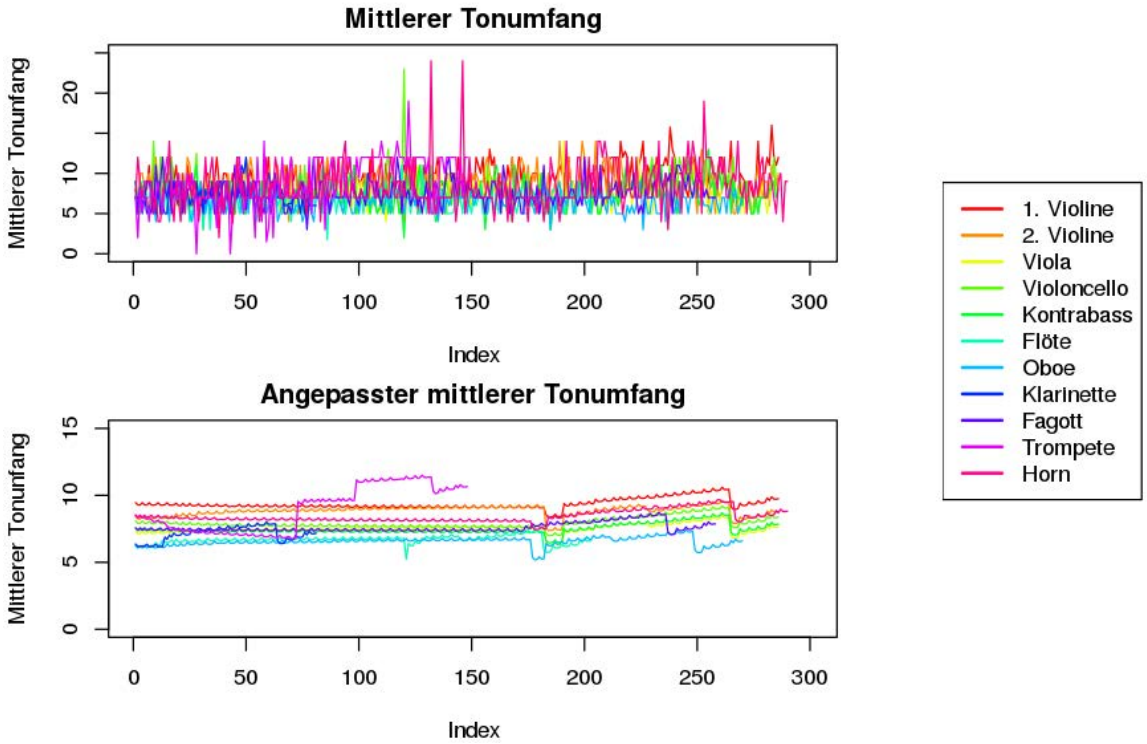

Abbildung 4.15: Tatsächlicher und angepasster mittlerer Tonumfang der Instrumente, aufgetragen über die gesamte Messreihe.

Der Tonumfang der verschiedenen Instrumente schwankt sehr stark und wird durch das Modell nicht gut approximiert, wie Abbildung 4.15 zeigt. Die untere Grafik zeigt den angepassten mittleren Tonumfang der einzelnen Instrumente, die sich darin kreuzenden Linien entsprechen den genannten Effekten. Der obere Teil zeigt die ursprünglichen Datenwerte; die bereits angesprochenen Ausreißer der Residuen sind hier deutlich als Peaks erkennbar. Dem Modell folgend ergeben sich signifikante Unterschiede bei den beiden Violinstimmen und der Viola, sowie der Flöte, Oboe, Klarinette und Trompete. Bei der Trompete ist ein kontinuierlicher Anstieg des Tonumfangs im Verlauf der Messreihe erkennbar.

Die Werknummer weist bei Mozart, Beethoven und Schubert einen niedrig signifikanten Einfluss auf den Tonumfang auf. Dieser lässt sich anhand von Abbildung 4.16 deuten:

Während der angepasste mittlere Tonumfang im Repertoire Haydns nahezu konstant bleibt, steigt er bei den anderen Komponisten im Mittel über ihre jeweilige Schaffenszeit leicht an. Insbesondere bei Beethoven ist dieser Effekt recht stark, die Kurve der Originalwerte zeigt lediglich bei seiner 8. Sinfonie einen Einbruch. Die Verteilung von Haydn zeigt allerdings, dass der Tonumfang in größeren Repertoires

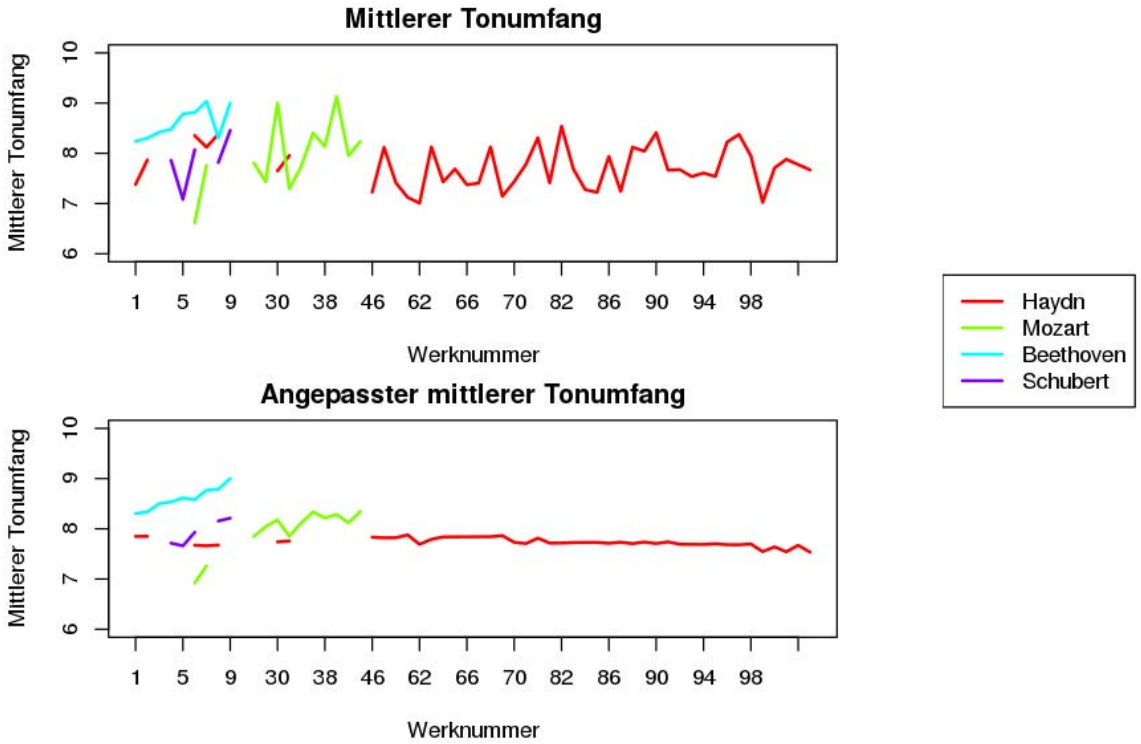

Abbildung 4.16: Tatsächlicher und angepasster mittlerer Tonumfang der Komponisten, aufgetragen über die Werknummer.

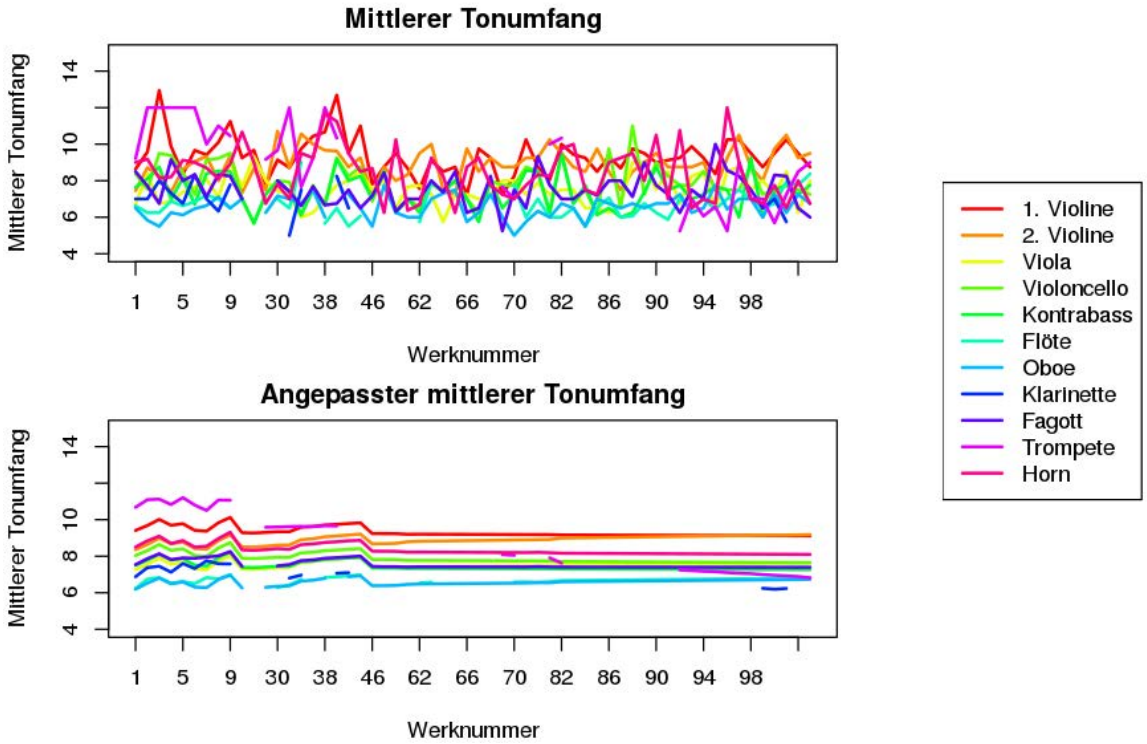

Abbildung 4.17: Tatsächlicher und angepasster mittlerer Tonumfang der Instrumente, aufgetragen über die Werknummer.

stark schwankt; vor diesem Hintergrund ist bei seriöser Vorgehensweise in der Interpretation der Aufwärtstrends von Mozart und Schubert Vorsicht geboten, da jeweils nur ein Teil ihres Repertoires analysiert wurde.

Schließlich werden die Interaktionen zwischen dem Tonumfang der Instrumente und der Werknummer betrachtet. Es zeigen sich hier nur wenige signifikante Unterschiede: Die zweite Violine und die Trompete weisen die stärksten Effekte auf, welche in Abbildung 4.17 ersichtlich sind. Darin ist erkennbar, dass die zweite Violine zwischen den Werknummern 46 und 82 kontinuierlich ansteigt; dieser Effekt geht somit allein auf das Repertoire Haydns zurück. Der mittlere Tonumfang der Trompete ist in den Werknummern 30–40 etwa einen Ganzton kleiner als in den einstelligen Werknummern und fällt daraufhin weiter ab. In den späten Werken Haydns nimmt er mit ca. 3 Ganztönen den geringsten Wert an, bei Beethoven und Schubert umfasst er hingegegen bis zu eine Oktave. Der Tonumfang von Flöte und Oboe umfasst 6–7 Halbtonschritte; seine Signifikanz lässt sich ebenso wie bei der zweiten Violine auf den leichten Anstieg in den Werken Haydns zurückführen, der im Gegensatz zu der Verringerung des Tonumfangs der restlichen Instrumente steht.

4.2.3 Fazit

Die Tonhöhenverteilungen von 11 Orchesterinstrumeten bei den vier Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert wurden zunächst grafisch ausgewertet und anschließend unter Verwendung eines linearen hierarchischen Modells statistisch untersucht. Mit einem Likelihood-Ratio-Test wurden relevante Variablen identifiziert und zur Modellbildung verwendet. Die Signifikanz der so geschätzten Parameter und ihre Interaktionen wurde tabellarisch und grafisch ausgewertet.

Im Bezug auf die mittlere Tonhöhe weisen einige Instrumente signifikante Unterschiede zwischen den einzelnen Komponisten auf: Mit Ausnahme der beiden Violinstimmen fällt die mittlere Tonhöhe der Streicher im Vergleich zu Haydn bei Mozart und Beethoven kontinuierlich um 1–2 Halbtonschritte ab und steigt bei Schubert wieder an, das Gleiche gilt für die Trompete. Die Flöte zeigt eine umgekehrte Tendenz: Sie steigt von Haydn über Mozart bis Beethoven um jeweils einen Ganzton, bevor sie bei Schubert einen Halbton fällt. Diese Tendenzen lassen sich als kompositionshistorische Entwicklungen interpretieren[2].

[2]Es gilt bei einer solchen Interpretation jedoch einschränkend die Ergebnisse von Kapitel 4.1.2 zu berücksichtigen sowie die Tatsache, dass – bedingt durch die Datengrundlage – nicht bei allen Komponisten das vollständige sinfonische Repertoire analysiert werden konnte.

Hinsichtlich des mittleren Tonumfangs zeigte sich, dass dieser bei allen Instrumenten starken Schwankungen unterliegt, die das Modell nicht zu erfassen vermag. Die Trompete hat von allen betrachteten Instrumenten den höchsten mittleren Tonumfang, die Oboe hingegen den niedrigsten. Darüber hinaus sind bei der Betrachtung des Tonumfangs keine ausgeprägten Tendenzen feststellbar.

4.3 Klassifikation von Instrumenten

Aufgrund der in 3.1.4 nachgewiesenen niedrigen Fehlerrate der Repertoires von W.A. Mozart und L.v. Beethoven wurden die Werke dieser Komponisten als Datengrundlage für die Klassifikation ausgewählt. In Tabelle 4.12 ist die Anzahl der verwendeten Datensätze für alle betrachteten Instrumente aufgeführt, nachdem das in Kapitel 3.3.3 beschriebene Ausschlusskriterium angewandt wurde. Dabei wurden die Piccoloflöte, das Kontrafagott und die Posaune von der Untersuchung ausgeschlossen. Wie bei den vorigen Betrachtungen werden die zwei Violinstimmen getrennt aufgeführt und bei der Klassifikation als separate Klassen behandelt.

Tabelle 4.12: Datengrundlage der Klassifikation.

Instrument	Datensätze		
	Gesamt	Training	Test
1. Violine	82	58	24
2. Violine	82	58	24
Viola	82	58	24
Violoncello	82	58	24
Kontrabass	82	58	24
Flöte	61	43	18
Oboe	72	51	21
Klarinette	50	35	15
Fagott	63	45	18
Trompete	60	42	18
Horn	88	62	26

Wie Abbildung 4.18 zeigt, sind einige Features stark korreliert. Dies trifft insbesondere auf die Lagemaße zu, doch auch die Streuungsmaße weisen Korrelationswerte von 20% und mehr auf. Zwischen den Streuungs- und Lagemaßen herrscht teilweise eine schwache negative Korrelation. Da nicht alle Klassifikationsverfahren gleichermaßen für Prädiktoren mit dieser Eigenschaft geeignet sind, gilt es daher in anderen

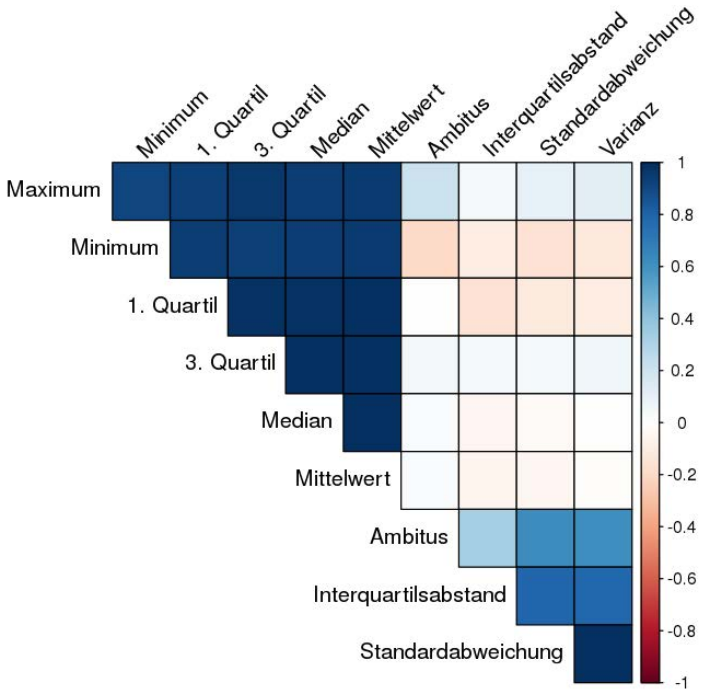

Abbildung 4.18: Korrelationsmatrix der zur Klassifikation verwendeten Features.

Anwendungsszenarien die Korrelation bei der Auswahl des Klassifikationsverfahrens zu berücksichtigen oder die Anzahl der Attribute entsprechend zu reduzieren.

Die Genauigkeit der Algorithmen wurde durch 10-fache Kreuzvalidierung geschätzt. In Abbildung 4.19 sind die geschätzten Genauigkeitsmaße der 6 untersuchten Klassifikationsverfahren in einem Boxplot aufgeführt. Alle Modelle weisen mit mittleren Werten von über 85% eine hohe Genauigkeit auf. Die Verfahren Random Forest und Bagged Tree zeigen mit Medianwerten von 89,25% und 87,57% die höchste Genauigkeit und bestätigen damit die Ergebnisse von Caruana und Niculescu-Mizil. Die schlechteste Performance zeigt das K-Nearest-Neighbours-Verfahren, dessen Genauigkeit auf bis zu 77,22% fallen kann. Die bayessche Klassifikation ist mit einem Interquartilsabstand von 5,6% der Genauigkeit hinsichtlich der Ergebnisqualität am dynamischsten, die Support Vector Machine hingegen zeigt die geringste Variabilität. Cohens Kappa-Werte ähneln bei allen Klassifikationsverfahren den jeweiligen Genauigkeitswerten sehr stark; die beobachtete Genauigkeit enspricht somit stets dem Erwartungswert.

Da das Random Forest-Verfahren in Bezug auf die verwendeten Datensätze und Prädiktoren die größte Genauigkeit aufweist, werden seine Ergebnisse im Folgenden näher betrachtet. Wie aus Abbildung 4.20 ersichtlich ist, erreicht dieser Algorithmus die höchste Genauigkeit bei der Betrachtung von 8 Prädiktoren. Der Zufallsstruktur

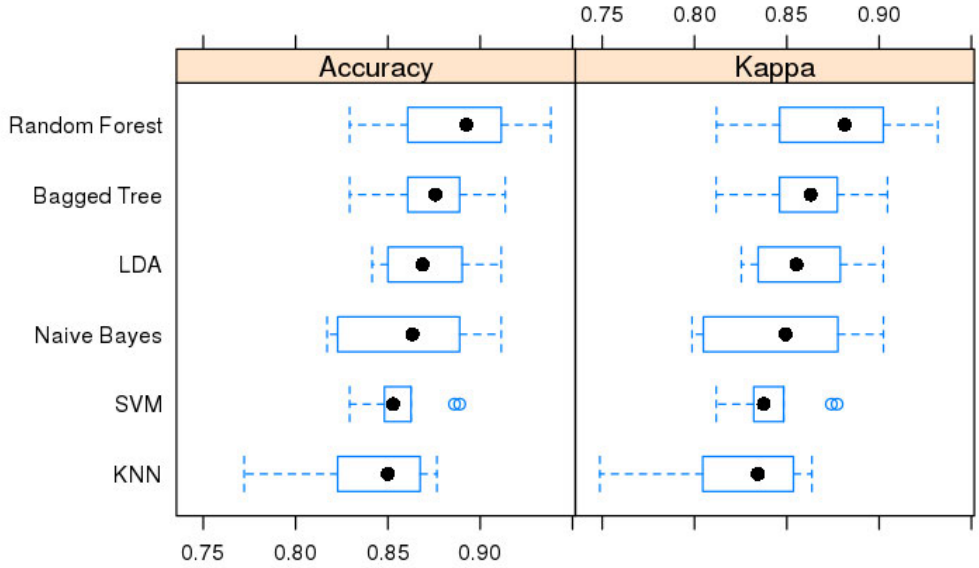

Abbildung 4.19: Genauigkeit und Cohens Kappa der angewandten Klassifikationsverfahren zum Konfidenzniveau 95%.

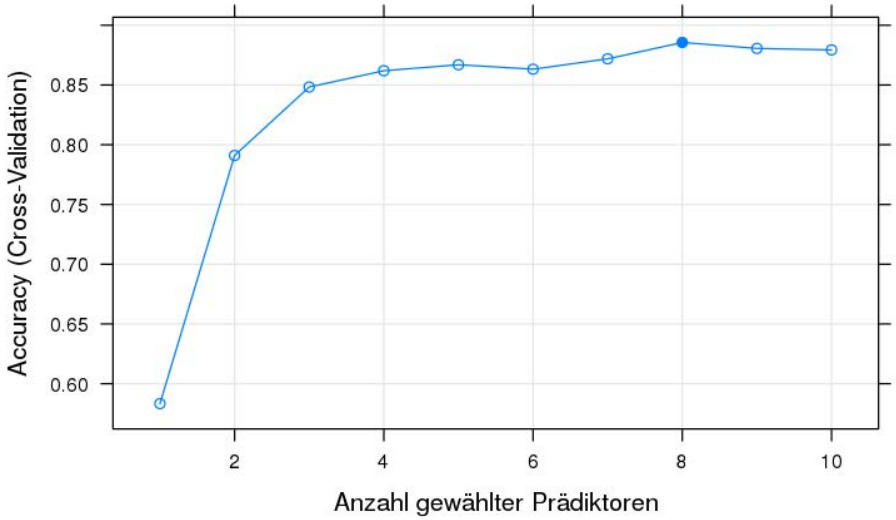

Abbildung 4.20: Genauigkeit des Random Forest-Verfahrens bei unterschiedlicher Anzahl ausgewählter Prädiktoren.

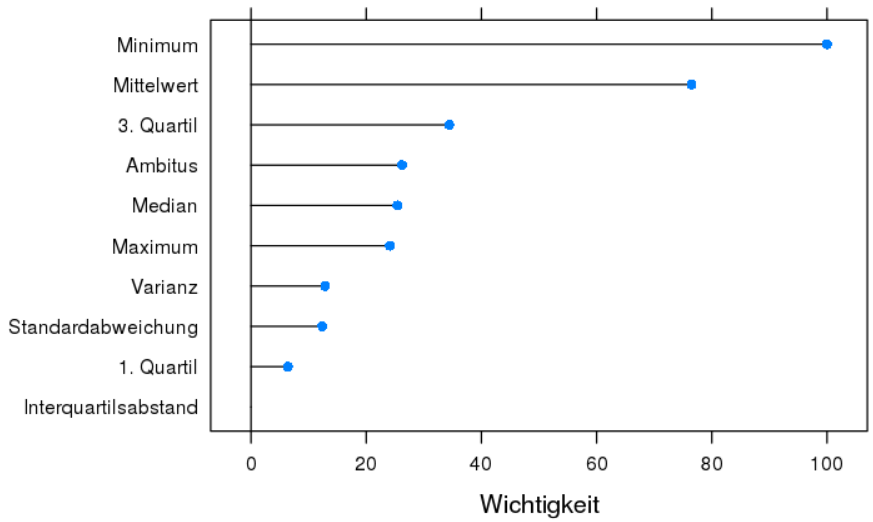

Abbildung 4.21: Gewicht der einzelnen Features bei der Verwendung des Random Forest-Verfahrens.

der Bäume – dem Feature Bagging – werden also 8 Merkmale zugrunde gelegt. Abbildung 4.21 zeigt die Wichtigkeit der einzelnen Prädiktoren. Hier ist aufgeführt, wie oft die jeweiligen Features zur Spaltung der Knoten herangezogen werden. Die im Random Forest-Verfahren als beste Prädiktoren identifizierten Attribute sind (in absteigender Reihenfolge): Minimum, Mittelwert, 3. Quartil, Ambitus, Median, Maximum, Varianz und Standardabweichung.

Abbildung 4.22 zeigt die Wahrheitsmarix der Klassifikation des Test-Sets: Anhand der darin erkennbaren Diagonale ist leicht zu erkennen, dass ein Großteil der Daten korrekt klassifiziert wird. Offenbar treten die meisten Klassifikationsfehler im Hinblick auf die Klarinette auf: Diese wurde jeweils zweimal als zweite Violine, Viola und Trompete sowie einmal als Oboe klassifiziert. Jeweils eine Trompete und erste Violinstimme wurden hingegen als Klarinette eingeordnet. Die Ursachen dieser hohen Fehlerrate sind vor allem dem hohen Dynamikumfang dieses Instruments geschuldet, dessen Tonhöhenverteilungen je nach Tonlage den Verteilungen anderer Instrumente stark ähneln können. Zudem ist die Klarinette mit 50 Datensätzen im für die Klassifikation verwendeten Datenmaterial das am wenigsten auftretende Instrument; durch Einbeziehung weiterer Datensätze könnte die Genauigkeit daher möglicherweise gesteigert werden. Auch die anderen transponierenden Instrumente sind für fehlerhafte Ergebnisse verantwortlich, die auf ihre verschiedenen Tonlagen zurückzuführen sind: Das Horn wurde jeweils dreimal als Viola und einmal als Trom-

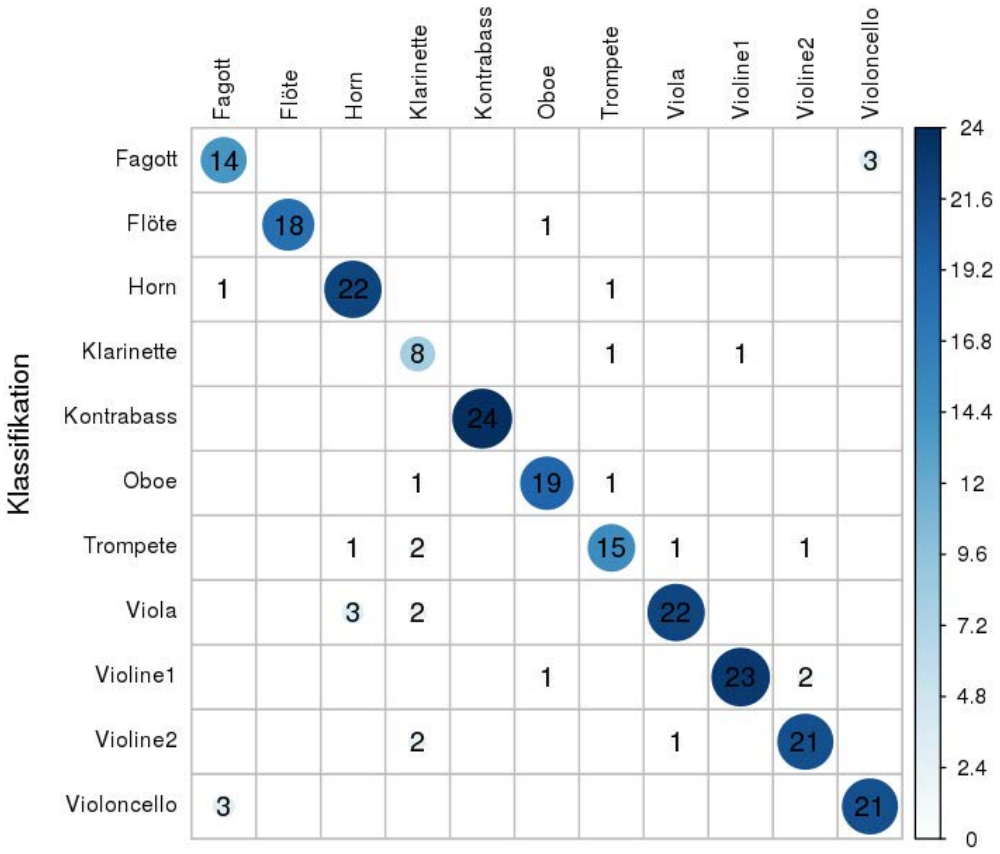

Abbildung 4.22: Wahrheitsmatrix der Ergebnisse des Random Forest-Verfahrens.

Tabelle 4.13: Kenngrößen des Random Forest-Klassifikators.

Instrument	Sensitivität	Spezifität	Relevanz	Segreganz	Genauigkeit
1. Violine	0,958	0,985	0,884	0,995	0,972
2. Violine	0,875	0,985	0,875	0,985	0,93
Viola	0,916	0,976	0,814	0,99	0,946
Violoncello	0,875	0,985	0,875	0,985	0,93
Kontrabass	1	1	1	1	1
Flöte	1	0,995	0,947	1	0,997
Oboe	0,904	0,990	0,904	0,990	0,947
Klarinette	0,533	0,99	0,8	0,969	0,762
Fagott	0,777	0,986	0,823	0,981	0,882
Trompete	0,833	0,977	0,75	0,986	0,905
Horn	0,846	0,99	0,916	0,981	0,918

pete klassifiziert, Fagott und Trompete wurden je einmal als Horn eingestuft. Auch die Klassifikation der Trompete ist mit einer hohen Fehlerrate verbunden, so wurde sie jeweils einmal als Horn, Klarinette und Oboe eingestuft. Zudem wurden vom Modell zwei Klarinetten sowie jeweils ein Horn, eine Viola und eine zweite Violine als Trompete klassifiziert. Schließlich gibt es Verwechslungen zwischen Violoncello und Fagott, die auf die Ähnlichkeit ihrer Tonlage zurückzuführen sein dürften. Die beiden Violinstimmen werden vom Modell bis auf zwei Fälle unterschieden, in denen die zweite Violine als erste Violine erkannt wird. Der Kontrabass wird als einziges Instrument in allen Fällen richtig erkannt.

In Tabelle 4.13 sind die Kenngrößen zur Beurteilung des Klassifikators aufgeführt. Die Sensitivität entspricht der Trefferquote, also dem Anteil der korrekt klassifizierten Instrumente an der Gesamtheit des jeweiligen Instruments. Die Spezifität gibt analog dazu den Anteil der korrekt als nicht der jeweiligen Klasse zugeordneten Elemente an ihrer Gesamtheit an. Die Relevanz- und Segreganzwerte werden ähnlich wie die Sensitivität und Spezifität berechnet, beziehen sich jedoch nicht auf die Grundgesamtheit sondern auf die jeweilige Anzahl positiv bzw. negativ erkannter Ergebnisse. Die Treffergenauigkeit gibt schließlich das Verhältnis korrekt klassifizierter Ergebnisse zur Anzahl aller Fälle an und entspricht den in Abbildung 4.19 dargestellten Werten.

4.3.1 Fazit

In diesem Kapitel wurden mehrere computergestützte Klassifikationsverfahren eingesetzt, um Instrumente anhand der statistischen Kennwerte ihrer Tonhöhenverteilungen zu bestimmen. Alle betrachteten Methoden zeigen hohe Genauigkeitswerte, das Random Forest-Verfahren weist die besten Ergebnisse auf.

Die zur Klassifikation verwendeten Prädiktoren sind teilweise stark korreliert und variieren in ihrer Aussagekraft hinsichtlich der zu bestimmenden Zielvariable. Der unteren Grenze des Tonumfangs kommt bei der Bestimmung der Instrumente die größte Bedeutung zu, andere Variablen wie der Interquartilsabstand der Tonhöhenverteilungen sind hingegen vernachlässigbar.

Grundsätzlich werden die meisten Instrumente vom Modell richtig zugeordnet. Transponierende Instrumente wie die Klarinette, das Horn und die Trompete weisen die höchste Fehlerrate auf; diese Beobachtung ist auf den hohen Dynamikumfang bzw. die verschiedenen Tonlagen dieser Instrumente zurückzuführen. Unterschiedliche Stimmen des selben Instruments, wie die erste und zweite Violinstimme, werden bei der Klassifikation zuverlässig unterschieden.

Zusammenfassung und Ausblick

In dieser Arbeit wurde auf Grundlage von Daten in der MIDI-Repräsentation ein musikalisches Korpus erstellt, welches die Tonhöhenverteilugnen eines großen Teils der sinfonischen Werke der Komponisten J. Haydn, W.A. Mozart, L.v. Beethoven und F. Schubert enthält. Es stellt somit eine repräsentative Auswahl der in der Periode der Wiener Klassik entstandenen Orchesterwerke dar. Diese Auswahl wurde zur Untersuchung individueller und entwicklungshistorischer Unterschiede in der Instrumentationspraxis der vier Komponisten herangezogen, sowie als Grundlage für den Vergleich und die Auswertung verschiedener computergestützer Verfahren zur automatischen Klassifikation von Instrumenten verwendet.

Die bei der Datengewinnung und Korpuserstellung eingesetzten Verfahren wurden begründet und erläutert. Es wurde eine Softwareanwendung zur Analyse der Tonhöhenverteilungen von MIDI-Daten entwickelt, deren generische Struktur eine Wiederverwendung oder Erweiterung im Rahmen weiterer Studien bzw. Anwendungen erlaubt. Aufgrund der ihr zugrunde liegenden plattformunabhängigen Java-Technologie ist ihr Einsatz in nahezu beliebigen Computersystemen und Entwicklungsumgebungen möglich. Ihre Anwendbarkeit in Kombination mit anderen Programmiersprachen wurde unter Verwendung der Programmierumgebung MATLAB aufgezeigt, in der die Applikation im Rahmen der Korpuserstellung zur Datenextraktion eingesetzt wurde.

Hinsichtlich des als Grundlage verwendeten Datenmaterials wurden einige Inkonsistenzen festgestellt, die auf fehlerhafte Eingaben des Notenmaterials durch die verschiedenen Autoren zurückgehen. Durch die Beschränkung auf das Material von vier Autoren und die Überprüfung des Tonmaterials auf gültige Wertebereiche sowie die Korrektur einzelner Datensätze konnten diese Unstimmigkeiten weitestgehend eliminiert werden. Das Auftreten fehlerhafter Einzeltöne, die auf fehlerhafte Oktavangaben und den interpretatorischen Spielraum bei der Kodierung des Notensatzes zurückgehen, konnte jedoch nicht gänzlich ausgeschlossen werden. In Anbetracht

der Größe des analysierten Korpus kann der Einfluss sporadischer Fehler auf die Kennwerte der Verteilungen bei der Auswertung allerdings vernachlässigt werden.

Das gewonnene und aufbereitete Datenmaterial wurde anschließend verwendet, um die Tonhöhenverteilungen von Orchesterinstrumenten bei den vier Komponisten vergleichend zu untersuchen. Dabei zeigten sich bei einigen Komponisten deutliche Unterschiede in der Verwendungshäufigkeit bestimmter Instrumente wie der Klarinette und der Trompete. Unter Verwendung eines hierarchisch-linearen Modells wurde eine Varianzanalyse durchgeführt und signifikante Unterschiede hinsichtlich der Tonhöhe und des Tonumfangs der Instrumente bei den verschiedenen Komponisten wurden aufgezeigt.
Mit Blick auf die mittlere Tonhöhe zeigten sich bei einigen Instrumenten gewisse kompositionshistorische Tendenzen: So spielen einige Streichinstrumente von Haydn über Mozart und Beethoven in immer tieferen Tonlagen, bei Schubert steigt die mittlere Tonhöhe schließlich wieder an. Bei der Flöte tritt dieses Phänomen in umgekehrter Form auf: Ihre mittlere Tonhöhe steigt im betrachteten Zeitraum sukzessive um etwa einen Ganzton an, bevor sie bei Schubert wieder einen Halbtonschritt fällt. Diese Entwicklungen werden stark durch die jeweils verwendeten Tonarten beeinflusst und können als das Ergebnis unterschiedlicher kompositorischer Präferenzen der verschiedenen Komponisten gedeutet werden.
Der Tonumfang aller Instrumente unterliegt in den einzelnen Werken sehr starken Schwankungen. Hinsichtlich des mittleren Tonumfangs der Instrumente konnten keine signifikanten Tendenzen ausgemacht werden, allein das Repertoire Beethovens zeigt einen geringen chronologischen Anstieg des mittleren Tonumfangs.

Die gesammelten Tonhöhenverteilungen wurden weiterhin zur Evaluation sechs verschiedener automatischer Klassifikationsverfahren eingesetzt. Bei der Klassifikation von Orchesterinstrumenten anhand statistischer Kennwerte ihrer Tonhöhenverteilungen zeigten alle betrachteten Verfahren eine mittlere Genauigkeit von über 85%. Das Bagged Tree-Verfahren sowie der darauf aufbauende Random-Forest-Algorithmus erwiesen sich als die Klassifikationsverfahren mit der geringsten Fehlerrate und bestätigen damit die Forschungsergebnisse von Caruana und Niculescu-Mizil [42].
Die Ergebnisse zeigen, dass eine zuverlässige Klassifikation von Instrumenten anhand statistischer Kennwerte ihrer Tonhöhenverteilungen möglich ist. Die Auswertung des Random-Forest-Verfahrens weist darauf hin, dass Lagemaßen wie dem Mittelwert der Tonhöhenverteilung und der unteren Grenze des Tonumfangs bei der Bestimmung der Instrumente eine entscheidende Bedeutung zukommt.
Einzig die korrekte Klassfikation transponierender Instrumente stellt eine gewisse

Herausforderung dar, da ihre Tonlage je nach Stimmung varriiert und die Kennwerte der Tonhöhenverteilung entsprechend beeinflusst: Es hat sich gezeigt, dass die Klarinette für die meisten Klassifikationsfehler verantwortlich ist, die zusätzlich durch den hohen Dynamikumfang dieses Instruments begünstigt werden. Die zuverlässige Unterscheidung von Instrumenten wie dem Fagott und dem Violoncello, deren Tonhöhenverteilungen ähnliche Lagemaße besitzen, erweist sich in einigen Fällen ebenfalls als problematisch.

Auch mit der Einschränkung bei der Klassifikation transponierender Instrumente sind die Ergebnisse der vorgestellten automatischen Klassifikationsverfahren robust und überzeugend. Insofern können die beschriebenen Klassifikationsmodelle der Ergänzung bestehender Methoden des Music Information Retrieval dienen, die auf einer spektralen Analyse des Tonmaterials beruhen: Durch die Einbeziehung der statistischen Kennwerte einer repräsentativen Menge von Tonhöhenverteilungen könnten ihre Ergebnisse verbessert werden. Zudem erlaubt das vorgestellte Verfahren die Unterscheidung unterschiedlicher Stimmen des gleichen Instruments, wie der ersten und zweiten Violine. Diese Unterscheidung wäre allein anhand der Klangeigenschaften nur schwer zu erzielen. Ebenso könnte das verwendete Verfahren von der Einbeziehung spektraler Eigenschaften profitieren: Das Fagott und das Violoncello könnten so z.B. durch zusätzliche Betrachtung der akustischen Merkmale zuverlässiger unterschieden werden, obwohl ihre Tonhöhenverteilungen starke Ähnlichkeiten aufweisen.

Die analysierten Tonhöhenverteilungen erlauben darüber hinaus die Bildung von gewichteten Mittelwerten für akustische Größen wie die Schallleistung, die Richtcharakteristik oder die spektrale Intensitätsverteilung von Orchesterinstrumenten. Derartige Betrachtungen sind im Bereich der musikalischen Akustik sowie vor dem Hintergrund virtueller akustischer Umgebungen von Interesse[1]. So besteht ein Mehrwert dieser Arbeit auch in der Verwendung der Datensätze für die gewichtete Mittelung von tonhöhenabhängigen Richtcharakteristiken bei der Simulation musikalischer Ensembles durch Grigoriev et al. [47]. Der Einfluss der Einzelton-Richtcharakteristiken der simulierten Orchesterinstrumente auf das akustische Gesamtbild wird dabei je nach Häufigkeit der Tonhöhen in einem 'typischen' Musikrepertoire unterschiedlich gewichtet. Als Grundlage dienen die neun Sinfonien von L.v. Beethoven, die als 'Kernbestand' klassischer Sinfoniekonzerte angesehen werden können und deren Datensätze für die Nachnutzung bereits elektronisch veröffentlicht wurden [48].

[1] Vgl. [46, p. 1489]

Neben der Betrachtung von Tonhöhe und Tonumfang können die analysierten Tonhöhenverteilungen zur Untersuchung vielfältiger musikwissenschaftlicher Fragen zu Praktiken der Instrumentation und Komposition dienen. So war in dieser Arbeit auch eine Untersuchung der Tonhöhenverteilungen bei verschiedenen Satztempi angedacht, deren Umsetzung und Auswertung den zeitlichen Rahmen jedoch gesprengt hätte. Daneben ermöglicht die entwickelte Softwareanwendung die Analyse von Teilbereichen eines Stücks, was beispielsweise der Untersuchung der Tonhöhenverteilungen verschiedener Formteile eines Satzes dienen kann.

Literaturverzeichnis

[1] Aramaki, Mitsuko; Mathieu Barthet; Richard Kronland-Martinet und Sølvi Ystad (Hrgs.) (2013): *From Sounds to Music and Emotions*. Heidelberg et al.: Springer.

[2] Spence, Charles und Qian Wang (2015): „Wine and music (I): on the crossmodal matching of wine and music." In: *Flavour*, 4(1) S. 1–14.

[3] Huron, David (2013): „On the Virtuous and the Vexatious in an Age of Big Data." In: *Music Perception: An Interdisciplinary Journal*, 31(1) S. 4–9.

[4] White, Christopher (2014): „Changing Styles, Changing Corpora, Changing Tonal Models." In: *Music Perception: An Interdisciplinary Journal*, 31(3) S. 244–253.

[5] Albrecht, Joshua und David Huron (2014): „A Statistical Approach to Tracing the Historical Development of Major and Minor Pitch Distributions, 1400-1750." In: *Music Perception: An Interdisciplinary Journal*, 31(3) S. 223–243.

[6] Cano, Estefanía; Gerald Schuller und Christian Dittmar (2014): „Pitch-informed solo and accompaniment separation towards its use in music education applications." In: *EURASIP Journal on Advances in Signal Processing*, 2014(23) S. 1–19.

[7] Nielsen, Andreas; Lars Hansen und Ulrik Kjems (2006): „Pitch based Sound Classification." In: *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP 2006, Tolouse, Frankreich, 14.–19. Mai 2006*. S. 788–791.

[8] Cheng, Heng-Tze; Yi-Hsuan Yang; Yu-Ching Lin; I-Bin Liao und Homer Chen (2008): „Automatic Chord Recognition for Music Classification and Retrieval." In: *International Conference on Multimedia and Expo, ICME 2008, Hannover, 23.–26. Juni 2008*. S. 1505–1508.

[9] Urbano, Julián; Markus Schedl und Xavier Serra (2013): „Evaluation in Music Information Retrieval." In: *Journal of Intelligent Information Systems*, 41(3) S. 345–369.

[10] Tzanetakis, George; Andrey Ermolinskyi und Perry Cook (2003): „Pitch Histograms in Audio and Symbolic Music Information Retrieval." In: *Journal of New Music Research*, 32(2) S. 143–152.

[11] Dighe, Pranay; Harish Karnick und Bhiksha Raj (2013): „Swara Histogram Based Structural Analysis and Identification of Indian Classical Ragas." In: *Proceedings of the 14th International Society for Music Information Retrieval Conference, ISMIR 2013, Curitiba, Brasilien, 4.–8. November 2013*. S. 35–40.

[12] Mardirossian, Arpi und Elaine Chew (2006): „Music Summarization Via Key Distributions: Analyses of Similarity Assessment Across Variations." In: *Proceedings of the 7th International Conference on Music Information Retrieval, ISMIR 2006, Victoria, Kanada, 8.–12. Oktober 2006*. S. 234–239.

[13] Weinzierl, Stefan (2014): „Zur Akustik musikalischer Schallquellen. Messverfahren – Modelle – Beschreibungsgrößen." In: Stefan Weinzierl (Hrg.) *Handbuch der Systematischen Musikwissenschaft 5*, Kap. 2. Laaber: Laaber Verlag, S. 179–198.

[14] Gjerdingen, Robert (2014): „"Historically Informed" Corpus Studies." In: *Music Perception: An Interdisciplinary Journal*, 31(3) S. 192–204.

[15] London, Justin (2013): „Building a Representative Corpus of Classical Music." In: *Music Perception: An Interdisciplinary Journal*, 31(1) S. 68–90.

[16] Kauermann, Göran und Helmut Küchenhoff (2011): *Stichproben. Methoden und praktische Umsetzung mit R*. Heidelberg et al.: Springer.

[17] Bainbridge, David und Tim Bell (2001): „The Challenge of Optical Music Recognition." In: *Computers and the Humanities*, 35(2) S. 95–121.

[18] Walshaw, Chris (2016): „The ABC Music Standard 2.1." `http://abcnotation.com/wiki/abc:standard:v2.1`. Online, zuletzt aufgerufen am 14. Juli 2016.

[19] Gonzato, Guido (2016): „The ABC Plus Project." `http://abcplus.sourceforge.net`. Online, zuletzt aufgerufen am 14. Juli 2016.

[20] Nienhuys, Han-Wen und Jan Nieuwenhuizen (2016): „LilyPond – Notationsreferenz." `http://lilypond.org/doc/v2.18/Documentation/notation/index.de.html`. Online, zuletzt aufgerufen am 14. Juli 2016.

[21] MakeMusic (2016): „MusicXML 3.0 Spezifikation." `http://www.musicxml.com/for-developers`. Online, zuletzt aufgerufen am 14. Juli 2016.

[22] Association, MIDI Manufacturers (2016): „The official MIDI Specifications." `https://www.midi.org/specifications`. Online, zuletzt aufgerufen am 14. Juli 2016.

[23] Rothstein, Joseph (1992): *MIDI: A Comprehensive Introduction*. Computer Music & Digital Audio Series. Madison: A-R Editions.

[24] ISO (1975): *Acoustics – Standard tuning frequency (Standard musical pitch*. ISO 16:1975, International Organization for Standardization, Genf.

[25] Ihaka, Ross und Robert Gentleman (1996): „R: A Language for Data Analysis and Graphics." In: *Journal of Computational and Graphical Statistics*, 5(3) S. 299–314.

[26] Wollschläger, Daniel (2012): *Grundlagen der Datenanalyse mit R*. Heidelberg et al.: Springer.

[27] Kunst der Fuge (2016): „kunstderfuge.com. The largest classical music resource in *.mid* files." `http://www.kunstderfuge.com`. Online, zuletzt aufgerufen am 14. Juli 2016.

[28] ClassicalArchives (2016): „ClassicalArchives. The largest classical music site in the world." `http://www.classicalarchives.com`. Online, zuletzt aufgerufen am 14. Juli 2016.

[29] Shlien, Seymour (2016): „abcMIDI." `http://abc.sourceforge.net/abcMIDI`. Online, zuletzt aufgerufen am 14. Juli 2016.

[30] IMSLP (2016): „International Music Score Library Project." `http://www.imslp.org`. Online, zuletzt aufgerufen am 14. Juli 2016.

[31] Eerola, Tuomas und Petri Toiviainen (2004): „MIR in MATLAB: The MIDI Toolbox." In: *Proceedings of the 5th International Society for Music Information Retrieval Conference, ISMIR 2004, barcelona, Spanien, 10.–14. Oktober 2004*. S. 22–27.

[32] Mitkov, Ruslan (2005): *The Oxford Handbook of Computational Linguistics*. Oxford: Oxford University Press.

[33] Kohn, Wolfgang (2005): *Statistik: Datenanalyse Und Wahrscheinlichkeitsrechnung*. Statistik und ihre Anwendungen. Heidelberg et al.: Springer.

[34] Botev, Zdravko; Joe Grotowski und Dirk Kroese (2010): „Kernel Density Estimation via Diffusion." In: *The Annals of Statistics*, 38(5) S. 2916–2957.

[35] Cleff, Thomas (2015): *Deskriptive Statistik und Explorative Datenanalyse*. Wiesbaden: Springer Gabler.

[36] Eid, Michael; Mario Gollwitzer und Manfred Schmitt (2010): *Statistik und Forschungsmethoden*. Weinheim: Beltz.

[37] Bates, Douglas (2010): „lme4: Mixed-effects modeling with R."

[38] Bates, Douglas; Martin Mächler; Ben Bolker und Steve Walker (2015): „Fitting Linear Mixed-Effects Models Using lme4." In: *Journal of Statistical Software*, 67(1) S. 1–48.

[39] Hox, Joop; Mirjam Moerbeek und Rens van de Schoot (2010): *Multilevel analysis: Techniques and Applications*. New York: Routledge.

[40] Friedman, Jerome; Trevor Hastie und Robert Tibshirani (2001): *The Elements of Statistical Learning*. Heidelberg et al.: Springer.

[41] Wolpert, David und William Macready (1997): „No Free Lunch Theorems for Optimization." In: *IEEE Transactions on Evolutionary Computation*, 1(1) S. 67–82.

[42] Caruana, Rich und Alexandru Niculescu-Mizil (2006): „An Empirical Comparison of Supervised Learning Algorithms." In: *Proceedings of the 23rd International Conference on Machine Learning (ICML), Pittsburgh, USA, 25.–28. Juni 2006*. S. 161–168.

[43] Breiman, Leo (1996): „Bagging Predictors." In: *Machine Learning*, 24(2) S. 123–140.

[44] Breiman, Leo (2001): „Random Forests." In: *Machine Learning*, 45(1) S. 5–32.

[45] Kuhn, Max (2008): „Building predictive Models in R Using the caret Package." In: *Journal of Statistical Software*, 28(5) S. 1–26.

[46] Quiring, Roman und Stefan Weinzierl (2016): „Tonhöhenverteilungen im klassischen Orchesterrepertoire." In: *Fortschritte der Akustik: Tagungsband der 42. Jahrestagung der Deutschen Gesellschaft für Akustik (DAGA 2016), Aachen, 14.–17. März 2016*. S. 1486–1489.

[47] Grigoriev, Dmitry; David Ackermann; Sönke Pelzer und Stefan Weinzierl (2016): „Ein psychologisches Messinstrument für die akustische Wahrnehmung von Räumen für Musik und Sprache: Stimulus-Erzeugung." In: *Fortschritte der Akustik: Tagungsband der 42. Jahrestagung der Deutschen Gesellschaft für Akustik (DAGA 2016), Aachen, 14.–17. März 2016*. S. 309–312.

[48] Quiring, Roman und Stefan Weinzierl (2016): „Pitch distributions for individual instruments in the symphonies No. 1-9 of L.v. Beethoven." `http://dx.doi.org/10.14279/depositonce-5040`.

Anhang: Quellcode

Java **i**
1 MidiAnalyzer.java i
2 MidiFileAnalysis.java iii
3 MidiTrackAnalysis.java iv
4 MidiMeasureAnalysis.java v
5 UnsupportedTimecodeException.java vii
6 IPitchHistogram.java vii
7 AbstractHistogram.java viii
8 ExtendableHistogram.java ix
9 CombinedHistogram.java ix
10 AbstractHistogramContainer.java x
11 MidiTimeSignature.java x
12 MidiKeySignature.java xi
13 MidiMeasure.java xi
14 MidiHelper.java xii

MATLAB **xiii**
15 convert_midi_to_mat.m xiii
16 parse_symphony_csv.m xiv
17 parse_symphony_data.m xiv
18 parse_movement_data.m xiv
19 parse_folder.m xv
20 parse_file.m xv
21 read_midi_file.m xv
22 parse_tracknames.m xvi
23 get_instrument_histograms.m xvii
24 fix_beethoven_histograms.m xvii
25 change_symphony_instrument.m xviii
26 get_histogram_from_mftext.m xviii
27 get_symphonies.m xix
28 get_valid_instruments.m xix
29 convert_mat_to_csv.m xix
30 symphony_to_csv.m xx
31 write_instrument_header.m xxi

R **xxi**
32 create_corpus_ambitus.R xxi
33 create_corpus_instruments.R xxii
34 get_histogram_csv_line.R xxiii
35 create_authors_note_stats.R xxiii
36 create_error_confidence_intervals.R xxiv
37 create_composer_boxplot.R xxv
38 create_composer_instrument_plots.R xxv
39 create_model_pitch_median.R xxvii
40 create_model_pitch_iqr.R xxx
41 get_lmm_data.R xxxii
42 get_composer_instrument_data.R xxxiii
43 compare_classification_models.R xxxiii
44 model_evaluation.R xxxv
45 get_features.R xxxvi
46 extract_features.R xxxvi
47 read_csv_histograms.R xxxvii
48 histogram_plot.R xxxvii
49 histogram_boxplot.R xxxviii
50 estimate_density.R xxxviii
51 get_iso_pitches.R xxxix
52 get_histogram_values.R xxxix
53 quad_plot.R xxxix

Quellcode-Datei 1: MidiAnalyzer.java

```java
package analysis;

import java io File;
import java io IOException;
import java util NavigableMap;

import javax sound midi InvalidMidiDataException;
import javax sound midi MetaMessage;
import javax sound midi MidiEvent;
import javax sound midi MidiMessage;
import javax sound midi MidiSystem;
import javax sound midi Sequence;
import javax sound midi ShortMessage;
import javax sound midi Track;

import midi MidiTimeSignature;
import exception UnsupportedTimecodeException;

/**
 * Main class for MIDI pitch analysis
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiAnalyzer {

    /**
     * Analyzes pitches of an entire MIDI file
     * @param file  The MIDI file to analyze
     * @return MidiFileAnalysis  The analysis results
     * @throws InvalidMidiDataException
     * @throws IOException
     * @throws UnsupportedTimecodeException
     */
    public static MidiFileAnalysis analyze(File file) throws InvalidMidiDataException, IOException,
        ↪UnsupportedTimecodeException {
        return analyze(file, 0, Long MAX_VALUE);
    }

    /**
     * Analyzes pitches of a section of a MIDI file
     * @param file  The MIDI file to analyze
     * @param start_tick  The start of the section as MIDI timestamp
     * @param end_tick  The end of the section as MIDI timestamp
     * @return MidiFileAnalysis  The analysis results
     * @throws InvalidMidiDataException
     * @throws IOException
     * @throws UnsupportedTimecodeException
     */
    public static MidiFileAnalysis analyze(File file, long start_tick, long end_tick) throws
        ↪InvalidMidiDataException, IOException, UnsupportedTimecodeException {
        return analyze(file, start_tick, end_tick, 0);
    }

    /**
     * Analyzes pitches of a section of a MIDI file where the length of the first measure is given
     * @param file  The MIDI file to analyze
     * @param start_tick  The start of the section as MIDI timestamp
     * @param end_tick  The end of the section as MIDI timestamp
     * @param anacrusis  The length of the first measure in MIDI ticks
     * @return MidiFileAnalysis  The analysis results
     * @throws InvalidMidiDataException
     * @throws IOException
     * @throws UnsupportedTimecodeException
     */
    public static MidiFileAnalysis analyze(File file, long start_tick, long end_tick, long anacrusis) throws
        ↪InvalidMidiDataException, IOException, UnsupportedTimecodeException {
        MidiFileAnalysis file_analysis = new MidiFileAnalysis(file getName());
        Sequence sequence = MidiSystem getSequence(file);

        if (sequence getDivisionType() != Sequence PPQ) {
            throw new UnsupportedTimecodeException("SMPTE");
        }

        int resolution = sequence getResolution();

        NavigableMap<Long, MidiTimeSignature> time_map = null;

        // loop tracks
        Track[] tracks = sequence getTracks();
        for (int i = 0; i < tracks length; i++) {
            Track track = tracks[i];
            MidiTrackAnalysis track_analysis;
            MidiMeasureAnalysis measure_analysis = null;

            // first track  extract the time signatures only (default to 4/4)
            // do not perform measure analyis on first track as the content is irrelevant
            if (i == 0) {
                track_analysis = new MidiTrackAnalysis(i, resolution);
            }

            // other tracks  apply time signatures of first track, perform measure analysis
```

```
            else {
                track_analysis = new MidiTrackAnalysis(i, resolution, time_map);

                // adjust size of first measure if anacrusis is set
                if (anacrusis > 0) {
                    measure_analysis = new MidiMeasureAnalysis(time_map, start_tick, anacrusis);
                }
                else {
                    measure_analysis = new MidiMeasureAnalysis(time_map, start_tick);
                }
            }

            // feed MIDI events to track analysis
            // this may be handled more efficiently by using a Java 8 Stream,
            // but we are limited to Java 7 for compatibility with MATLAB
            for (int j = 0; j < track.size(); j++) {
                processMidiEvent(track.get(j), track_analysis, measure_analysis, start_tick, end_tick);
            }

            // ignore tracks without musical material
            if (track_analysis.getHistogram().getNumNotes() > 0) {
                file_analysis.addTrackAnalysis(track_analysis);
                measure_analysis.finish(end_tick);
                file_analysis.addMeasureAnalysis(measure_analysis, i);
            }

            // remember time signature declarations in first track
            else if (i == 0) {
                time_map = track_analysis.getTimeSignatures();
            }
        }

        return file_analysis;
    }

    /**
     * Private helper to process MIDI events
     * Adds the pitches to a given MidiTrackAnalysis and MidiMeasureAnalysis
     * @param event  The MIDI event to process
     * @param track_analysis  The MidiTrackAnalysis object to add the events to
     * @param measure_analysis  The MidiMeasureAnalysis object to add the events to
     * @param start_tick  The start of the analyzed section as MIDI timestamp
     * @param end_tick  The end of the analyzed section as MIDI timestamp
     */
    private static void processMidiEvent(MidiEvent event, MidiTrackAnalysis track_analysis, MidiMeasureAnalysis
        ↪measure_analysis, long start_tick, long end_tick) {
        MidiMessage message = event.getMessage();
        long tick = event.getTick();

        if (message instanceof MetaMessage) {
            MetaMessage meta_message = (MetaMessage) message;
            byte[] data = meta_message.getData();

            switch (meta_message.getType()) {

                // sequence / track name
                case 3:
                    track_analysis.setName(new String(data));
                    break;

                // time signature
                case 0x58:
                    track_analysis.setTimeSignature(data[0] & 0xFF, 1 << (data[1] & 0xFF), tick);
                    break;

                default:
                    break;
            }
        }
        else if (message instanceof ShortMessage) {

            // only process events in desired tick range
            if (tick < start_tick || tick > end_tick) {
                return;
            }

            ShortMessage short_message = (ShortMessage) message;

            // ignore drum channel
            if (short_message.getChannel() == 9) {
                return;
            }

            if (short_message.getCommand() == ShortMessage.NOTE_ON) {

                // note onset  only accept velocity > 0
                if (short_message.getData2() > 0) {
                    track_analysis.addPitch(short_message.getData1(), tick);

                    if (measure_analysis != null) {
                        measure_analysis.addPitch(short_message.getData1(), tick);
                    }
```

```
                }
            }
        }
    }
}
```

Quellcode-Datei 2: MidiFileAnalysis.java

```
package analysis;

import java.util.Collection;
import java.util.Map;
import java.util.TreeMap;

/**
 * This class contains the pitch analysis results of a MIDI file
 * @author Roman Quiring <rquiring@gmx.de>
 */
public class MidiFileAnalysis {

    private String filename;
    private Map<Integer, MidiTrackAnalysis> track_analyses = new TreeMap<Integer, MidiTrackAnalysis>();
    private Map<Integer, MidiMeasureAnalysis> measure_analyses = new TreeMap<Integer, MidiMeasureAnalysis>();

    /**
     * Constructor
     * @param filename  The filename of the analyzed MIDI file
     */
    public MidiFileAnalysis(String filename) {
        this.filename = filename;
    }

    /**
     * Adds a MidiTrackAnalysis object to this analysis
     * @param analysis  The MidiTrackAnalysis object to be added
     */
    protected void addTrackAnalysis(MidiTrackAnalysis analysis) {
        track_analyses.put(analysis.getNumber(), analysis);
    }

    /**
     * Adds a MidiMeasureAnalysis object to this analysis
     * @param analysis  The MidiMeasureAnalysis object to be added
     * @param track_number  The track number of the MidiMeasureAnalysis object
     */
    protected void addMeasureAnalysis(MidiMeasureAnalysis analysis, int track_number) {
        measure_analyses.put(track_number, analysis);
    }

    /**
     * Private helper to translate track names to numbers
     * @param track_name  The name of the Track
     * @return Integer  The track number
     */
    private int getTrackNumber(String track_name) {
        for (MidiTrackAnalysis track_analysis : track_analyses.values()) {
            if (track_analysis.getName().equals(track_name)) {
                return track_analysis.getNumber();
            }
        }
        return -1;
    }

    /**
     * Returns the name of the analyzed MIDI file
     * @return String  The MIDI filename
     */
    public String getFilename() {
        return filename;
    }

    /**
     * Returns the results of all MIDI track analyses
     * @return Collection<MidiTrackAnalysis>  The MidiTrackAnalysis objects of the analyzed file
     */
    public Collection<MidiTrackAnalysis> getTrackAnalyses() {
        return track_analyses.values();
    }

    /**
     * Returns the results of a MIDI track analysis with a given number
     * @param track_number  The number of the MIDI track
     * @return MidiTrackAnalysis  The MidiTrackAnalysis object of the given MIDI track
     */
    public MidiTrackAnalysis getTrackAnalysis(int track_number) {
        return track_analyses.get(track_number);
    }

    /**
     * Returns the results of a MIDI track analysis with a given name
```

```
     * @param track_name  The name of the MIDI track
     * @return MidiTrackAnalysis  The MidiTrackAnalysis object of the given MIDI track
     */
    public MidiTrackAnalysis getTrackAnalysis(String track_name) {
        int track_number = getTrackNumber(track_name);
        return getTrackAnalysis(track_number);
    }

    /**
     * Returns the results of all MIDi measure analyses
     * @return Collection<MidiMeasureAnalysis>  The MidiMeasureAnalysis objects of the analyzed file
     */
    public Collection<MidiMeasureAnalysis> getMeasureAnalyses() {
        return measure_analyses values();
    }

    /**
     * Returns the results of a MIDI measure analysis with a given track number
     * @param track_number  The number of the MIDI track
     * @return MidiMeasureAnalysis  The MidiMeasureAnalysis object of the given MIDI track
     */
    public MidiMeasureAnalysis getMeasureAnalysis(int track_number) {
        return measure_analyses get(track_number);
    }

    /**
     * Returns the results of a MIDI measure analysis with a given track name
     * @param track_name  The name of the MIDI track
     * @return MidiMeasureAnalysis  The MidiMeasureAnalysis object of the given MIDI track
     */
    public MidiMeasureAnalysis getMeasureAnalysis(String track_name) {
        int track_number = getTrackNumber(track_name);
        return getMeasureAnalysis(track_number);
    }

}
```

Quellcode-Datei 3: MidiTrackAnalysis.java

```
package analysis;

import java util NavigableMap;
import java util TreeMap;

import midi MidiTimeSignature;
import histogram AbstractHistogramContainer;

/**
 * This class contains the pitch analysis of a single track of a MIDI file
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiTrackAnalysis extends AbstractHistogramContainer {

    private int number;
    private String name;
    private int resolution;
    private NavigableMap<Long, MidiTimeSignature> time_map;

    /**
     * Constructor
     * @param number  The MIDI track number
     * @param resolution  The MIDI resolution (PPQ)
     */
    public MidiTrackAnalysis(int number, int resolution) {
        this resolution = resolution;
        this number = number;
        time_map = new TreeMap<Long, MidiTimeSignature>();

        // assume default time signature (4/4), this may be overwritten
        time_map put(0L, new MidiTimeSignature(resolution));
    }

    /**
     * Constructor
     * @param number  The MIDI track number
     * @param resolution  The MIDI resolution (PPQ)
     * @param time_map  The time signature map for this track
     */
    public MidiTrackAnalysis(int number, int resolution, NavigableMap<Long, MidiTimeSignature> time_map) {
        this resolution = resolution;
        this number = number;
        this time_map = time_map;
    }

    /**
     * Sets the MIDI track name
     * @param name  The track name
     */
    protected void setName(String name) {
        this name = name;
```

```
    }

    /**
     * Adds a MIDI pitch to the track analysis
     * @param pitch  The MIDI note number
     * @param tick The timestamp of the MIDI note event
     */
    protected void addPitch(int pitch, long tick) {
        histogram addPitch(pitch);
    }

    /**
     * Sets a time signature at a given timestamp
     * @param numerator  The numerator of the time signature
     * @param denominator  The denominator of the time signature
     * @param tick  The timestamp of the time signature
     */
    protected void setTimeSignature(int numerator, int denominator, long tick) {
        time_map put(tick, new MidiTimeSignature(resolution, numerator, denominator));
    }

    /**
     * Returns the track number
     * @return Integer  The track number
     */
    public int getNumber() {
        return number;
    }

    /**
     * Returns the track name
     * @return String  The track name
     */
    public String getName() {
        return name;
    }

    /**
     * Returns a map of time signatures for this track
     * @return NavigableMap<Long, MidiTimeSignature>  The time signature map
     */
    public NavigableMap<Long, MidiTimeSignature> getTimeSignatures() {
        return time_map;
    }
}
```

Quellcode-Datei 4: MidiMeasureAnalysis.java

```
package analysis;
import histogram AbstractHistogramContainer;
import histogram CombinedHistogram;
import histogram IPitchHistogram;

import java util ArrayList;
import java util LinkedList;
import java util List;
import java util NavigableMap;
import java util TreeMap;

import midi MidiMeasure;
import midi MidiTimeSignature;

/**
 * This class contains the pitch analysis for all measures of a MIDI file
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiMeasureAnalysis extends AbstractHistogramContainer {

    private List<MidiMeasure> measures = new ArrayList<MidiMeasure>();
    private MidiMeasure current_measure;
    private NavigableMap<Long, MidiTimeSignature> time_map = new TreeMap<Long, MidiTimeSignature>();

    /**
     * Constructor
     * @param time_map  Map containing the MIDI time signatures for all MIDI tracks
     * @param tick_offset  Analysis offset in MIDI ticks
     * @param anacrusis  Length  of the first measure in MIDI ticks
     */
    public MidiMeasureAnalysis(NavigableMap<Long, MidiTimeSignature> time_map, long tick_offset, long anacrusis) {
        this time_map = time_map;
        current_measure = new MidiMeasure(getTimeSignatureForTick(tick_offset), tick_offset, tick_offset+anacrusis
            ↪-1);
    }

    /**
     * Constructor
     * @param time_map  Map containing the MIDI time signatures for all MIDI tracks
     * @param tick_offset  Analysis offset in MIDI ticks
     */
    public MidiMeasureAnalysis(NavigableMap<Long, MidiTimeSignature> time_map, long tick_offset) {
```

```
        this time_map = time_map;
        current_measure = new MidiMeasure(getTimeSignatureForTick(tick_offset), tick_offset);
    }

    /**
     * Constructor
     * @param time_map  Map containing the MIDI time signatures for all MIDI tracks
     */
    public MidiMeasureAnalysis(NavigableMap<Long, MidiTimeSignature> time_map) {
        this time_map = time_map;
        current_measure = new MidiMeasure(getTimeSignatureForTick(0), 0);
    }

    /**
     * Private helper for measure calculation
     * @param tick  The timestamp of the currently processed MIDI event
     */
    private void updateCurrentMeasure(long tick) {
        if (tick <= current_measure getEndTick()) {
            return;
        }

        MidiTimeSignature current_time_signature = current_measure getTimeSignature();
        int measure_length = current_time_signature getMeasureLength();
        long tick_dif = tick - current_measure getEndTick() - 1;
        long measure_dif = tick_dif / measure_length;

        for (int i = 0; i <= measure_dif; i++) {
            measures add(current_measure);
            MidiTimeSignature time_signature = time_map get(time_map floorKey(current_measure getEndTick() + 1));
            current_measure = new MidiMeasure(time_signature, current_measure getEndTick() + 1);
        }
    }

    /**
     * Private helper to get a time signature at a given timestamp
     * @param tick  The timestamp of the currently processed MIDI event
     * @return MidiTimeSignature  A MidiTimeSignature object
     */
    private MidiTimeSignature getTimeSignatureForTick(long tick) {
        return time_map get(time_map floorKey(tick));
    }

    /**
     * Adds a MIDI pitch to the measure analysis
     * @param pitch  The MIDI note number
     * @param tick The timestamp of the MIDI note event
     */
    protected void addPitch(int pitch, long tick) {
        updateCurrentMeasure(tick);

        if (tick >= current_measure getStartTick()) {
            current_measure addPitch(pitch);
            histogram addPitch(pitch);
        }
    }

    /**
     * Finishes the measure analysis
     * Fills in any empty measures for tracks that miss musical content at the end of track
     * @param end_tick  The timestamp of the end of the longest MIDI track
     */
    protected void finish(long end_tick) {
        measures add(current_measure);
        updateCurrentMeasure(end_tick);
    }

    /**
     * Blockwise processing utility method for measures
     * Implements a 'sliding window' with a length of multiple measures
     * @param block_size  The size of the measure window
     * @param hop_size  The number of measures to advance between blocks
     * @return List<IPitchHistogram>  List of merged pitch histograms
     */
    public List<IPitchHistogram> getMeasureBlocks(int block_size, int hop_size) {
        List<IPitchHistogram> blocks = new LinkedList<IPitchHistogram>();
        int n_measures = getNumberOfMeasures();

        //integer division + ceiling for positive numbers  (num + divisor - 1) / divisor
        int n_blocks = (n_measures - block_size + hop_size - 1) / hop_size + 1;

        for (int i = 0; i < n_blocks; i++) {
            int start_measure = i * hop_size;
            int end_measure = start_measure + block_size;

            //may have to shorten last block to stay in bounds
            if (end_measure > n_measures) {
                end_measure = n_measures;
            }

            List<IPitchHistogram> block = new LinkedList<IPitchHistogram>();
            for (MidiMeasure measure   getMeasures(start_measure, end_measure)) {
```

```
                block add(measure getHistogram());
            }

            blocks add(new CombinedHistogram(block));
        }

        return blocks;
    }

    /**
     * Returns the number of analyzed measures
     * @return Integer  The number of measures
     */
    public int getNumberOfMeasures() {
        return measures size();
    }

    /**
     * Returns a list of the analyzed MidiMeasure objects
     * @return List<MidiMeasure>  List of MidiMeasure objects
     */
    public List<MidiMeasure> getMeasures() {
        return measures;
    }

    /**
     * Returns a section of the analyzed MidiMeasure objects
     * @param from  Number of the first measure
     * @param to  Number of the last measure
     * @return List<MidiMeasure>  List of MidiMeasure objects
     */
    public List<MidiMeasure> getMeasures(int from, int to) {
        return measures subList(from, to);
    }

    /**
     * Returns a specific MidiMeasure object
     * @param number  Number of the measure
     * @return  MidiMeasure object
     */
    public MidiMeasure getMeasure(int number) {
        return measures get(number);
    }

}
```

Quellcode-Datei 5: UnsupportedTimecodeException.java

```
package exception;

/**
 * Excepption for unsupported MIDI timecodes
 * @author Roman Quiring <rquiring@gmx de>
 */
public class UnsupportedTimecodeException extends Exception {

    private static final long serialVersionUID = 1L;

    /**
     * Constructor
     * @param timecode  String containing the unsupported timecode, e g  "SMPTE"
     */
    public UnsupportedTimecodeException(String timecode) {
        super("Timecode is not supported   "+timecode);
    }

}
```

Quellcode-Datei 6: IPitchHistogram.java

```
package histogram;

/**
 * Interface for MIDI pitch histograms
 * @author Roman Quiring <rquiring@gmx de>
 */
public interface IPitchHistogram {

    /**
     * Returns the lowest pitch of the histogram
     * @return Integer  The lowest pitch
     */
    public int getPitchRangeStart();

    /**
     * Returns the highest pitch of the histogram
     * @return Integer  The highest pitch
```

```
     */
    public int getPitchRangeEnd();

    /**
     * Returns the number of notes in the histogram
     * @return Integer  The number of notes
     */
    public int getNumNotes();

    /**
     * Returns the number of pitches in the histogram
     * @return Integer  The number of pitches
     */
    public int getNumPitches();

    /**
     * Returns the pitch histogram
     * @return Integer[]  The histogram as an integer array
     */
    public int[] getPitches();

    /**
     * Returns the pitch with the most occurrences
     * @return Integer  The most frequent pitch
     */
    public int getMaxBin();

    /**
     * Returns the occurrence of the most frequent pitch
     * @return Integer  Occurrence of the most frequent pitch
     */
    public int getMaxValue();

    /**
     * Returns a string representation of the pitch histogram
     * @return String  Pitch histogram in string form
     */
    public String toString();

}
```

Quellcode-Datei 7: AbstractHistogram.java

```
package histogram;

import midi MidiHelper;

/**
 * Abstract implementation of the IPitchHistogram interface
 * @author Roman Quiring <rquiring@gmx de>
 */
public abstract class AbstractHistogram implements IPitchHistogram {

    protected static final int NUM_BINS = 128;
    protected int[] pitches = new int[NUM_BINS];
    protected int num_notes = 0;
    protected int num_pitches = 0;
    protected int pitch_range_start = NUM_BINS;
    protected int pitch_range_end = 0;
    protected int max_bin = 0;
    protected int max_value = 0;

    @Override
    public int getPitchRangeStart() {
        return pitch_range_start;
    }

    @Override
    public int getPitchRangeEnd() {
        return pitch_range_end;
    }

    @Override
    public int getNumNotes() {
        return num_notes;
    }

    @Override
    public int getNumPitches() {
        return num_pitches;
    }

    @Override
    public int[] getPitches() {
        return pitches;
    }

    @Override
    public int getMaxBin() {
        return max_bin;
```

```
    }

    @Override
    public int getMaxValue() {
        return max_value;
    }

    @Override
    public String toString() {

        StringBuilder string_builder = new StringBuilder();
        string_builder append("Histogram  "+num_notes+" notes in "+num_pitches+" pitches \n");
        for (int pitch = 0; pitch < NUM_BINS; pitch++) {
            if (pitches[pitch] > 0) {
                string_builder append("\t"+MidiHelper PitchToString(pitch)+"  "+pitches[pitch]+"x\n");
            }
        }

         return string_builder toString();
    }

}
```

Quellcode-Datei 8: ExtendableHistogram.java

```
package histogram;

/**
 * Extendable pitch histogram implementation
 * Allows pitches to be added to a histogram
 * @author Roman Quiring <rquiring@gmx de>
 */
public class ExtendableHistogram extends AbstractHistogram {

    /**
     * Adds a pitch to the pitch histogram
     * @param pitch  The MIDI note number
     */
    public void addPitch(int pitch) {
        if (pitches[pitch] == 0) {
            num_pitches++;
        }

        pitches[pitch]++;

        if(pitches[pitch] > max_value) {
            max_value = pitches[pitch];
            max_bin = pitch;
        }

        num_notes++;

        pitch_range_start = Math min(pitch_range_start, pitch);
        pitch_range_end = Math max(pitch_range_end, pitch);
    }

}
```

Quellcode-Datei 9: CombinedHistogram.java

```
package histogram;

import java util Collection;

/**
 * Representation of merged pitch histograms
 * @author Roman Quiring <rquiring@gmx de>
 */
public class CombinedHistogram extends AbstractHistogram {

    /**
     * Constructor
     * @param histograms  Collection of IPitchHistogram objects to merge
     */
    public CombinedHistogram(Collection<? extends IPitchHistogram> histograms) {
        for (IPitchHistogram histogram   histograms) {
            addHistogram(histogram);
        }
    }

    /**
     * Adds a histogram to the merged histogram
     * @param histogram  The IPitchHistogram object to merge
     */
    public void addHistogram(IPitchHistogram histogram) {
        num_notes += histogram getNumNotes();
        pitch_range_start = Math min(pitch_range_start, histogram getPitchRangeStart());
```

```
        pitch_range_end = Math max(pitch_range_end, histogram getPitchRangeEnd());

        if (histogram getMaxValue() > max_value) {
            max_value = histogram getMaxValue();
            max_bin = histogram getMaxBin();
        }

        int[] pitches = histogram getPitches();
        for (int i = 0; i < NUM_BINS; i++) {
            if (pitches[i] == 0) {
                continue;
            }

            if (this pitches[i] == 0) {
                num_pitches++;
            }

            this pitches[i] += pitches[i];
        }
    }

}
```

Quellcode-Datei 10: AbstractHistogramContainer.java

```
package histogram;

/**
 * Abstract implementation of a container class of multiple pitch histograms
 * @author Roman Quiring <rquiring@gmx de>
 */
public abstract class AbstractHistogramContainer {

    protected ExtendableHistogram histogram = new ExtendableHistogram();

    /**
     * Returns the pitch histogram
     * @return IPitchHistogram  The pitch histogram
     */
    public IPitchHistogram getHistogram() {
        return histogram;
    }

}
```

Quellcode-Datei 11: MidiTimeSignature.java

```
package midi;

/**
 * Representation of a MIDI time signature
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiTimeSignature {

    private int resolution;
    private int numerator = 4;
    private int denominator = 4;

    /**
     * Constructor
     * @param resolution  The MIDI resolution (PPQ)
     * @param numerator The numerator of the time signature
     * @param denominator The denominator of the time signature
     */
    public MidiTimeSignature(int resolution, int numerator, int denominator) {
        this resolution = resolution;
        this numerator = numerator;
        this denominator = denominator;
    }

    /**
     * Constructor
     * @param resolution  The MIDI resolution (PPQ)
     */
    public MidiTimeSignature(int resolution) {
        this resolution = resolution;
    }

    /**
     * Sets the numerator and denominator of this time signature
     * @param numerator  The new time signature numerator
     * @param denominator  The new time signature denominator
     */
    public void setTimeSignature(int numerator, int denominator) {
        this numerator = numerator;
        this denominator = denominator;
```

```
    }

    /**
     * Returns the MIDI resolution (PPQ)
     * @return Integer  The MIDI resolution
     */
    public int getResolution() {
        return resolution;
    }

    /**
     * Returns a string representation of this time signature
     * @return String  The time signature in string form
     */
    public String toString() {
        return numerator+"/"+denominator;
    }

    /**
     * Returns the length of a measure with this time signature
     * @return Integer  Number of MIDI ticks of a measure with this time signature
     */
    public int getMeasureLength() {
        return resolution * 4 / denominator * numerator;
    }

}
```

Quellcode-Datei 12: MidiKeySignature.java

```
package midi;

/**
 * Representation of a MIDI key signature
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiKeySignature {

    private int flats_sharps;
    private boolean minor;

    /**
     * Constructor
     */
    public MidiKeySignature() {
        flats_sharps = 0;
    }

    /**
     * Constructor
     * @param flats_sharps  Number of flats (negative) or sharps (positive)
     * @param minor  Boolean indicating whether this is a major or minor key
     */
    public MidiKeySignature(int flats_sharps, boolean minor) {
        this flats_sharps = flats_sharps;
        this minor = minor;
    }

    /**
     * Returns the flats/sharps value
     * @return Integer  The flats/sharps value
     */
    public int getFlatsSharps() {
        return flats_sharps;
    }

    /**
     * Returns whether the key is minor or major
     * @return Boolean  True = minor key, False = major key
     */
    public boolean isMinor() {
        return minor;
    }

}
```

Quellcode-Datei 13: MidiMeasure.java

```
package midi;

import histogram AbstractHistogramContainer;

/**
 * Representation of a measure
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiMeasure extends AbstractHistogramContainer {
```

```
    private MidiTimeSignature time_signature;
    private long start_tick;
    private long end_tick;

    /**
     * Constructor
     * @param time_signature  The MidiTimeSignature object for this measure
     * @param tick  The MIDI timestamp marking the start of the measure
     * @param end_tick  The MIDI timestamp marking the end of the measure
     */
    public MidiMeasure(MidiTimeSignature time_signature, long tick, long end_tick) {
        this time_signature = time_signature;
        start_tick = tick;
        this end_tick = end_tick;
    }

    /**
     * Constructor
     * @param time_signature  The MidiTimeSignature object for this measure
     * @param tick  The MIDI timestamp marking the start of the measure
     */
    public MidiMeasure(MidiTimeSignature time_signature, long tick) {
        this(time_signature, tick, tick + time_signature getMeasureLength() - 1);
    }

    /**
     * Adds a pitch to the pitch histogram of this measure
     * @param pitch  The MIDI note number
     */
    public void addPitch(int pitch) {
        histogram addPitch(pitch);
    }

    /**
     * Sets the time signature of this measure
     * @param time_signature  A MidiTimeSignature object
     */
    public void setTimeSignature(MidiTimeSignature time_signature) {
        this time_signature = time_signature;
        end_tick = start_tick + time_signature getMeasureLength() - 1;
    }

    /**
     * Returns the timestamp for the start of the measure
     * @return Long  The MIDI timestamp
     */
    public long getStartTick() {
        return start_tick;
    }

    /**
     * Returns the timestamp for the end of the measure
     * @return Long  The MIDI timestamp
     */
    public long getEndTick() {
        return end_tick;
    }

    /**
     * Returns the time signature of this measure
     * @return  The MidiTimeSignature object
     */
    public MidiTimeSignature getTimeSignature() {
        return time_signature;
    }

    /**
     * Returns a string representation of this measure
     * @return String  The measure in string form
     */
    public String toString() {
        return "Measure from Tick "+start_tick+" to "+end_tick+" \n"+histogram;
    }

}
```

Quellcode-Datei 14: MidiHelper.java

```
package midi;

/**
 * Static helper for translation of MIDI note numbers
 * @author Roman Quiring <rquiring@gmx de>
 */
public class MidiHelper {

    private static String[] pitches = {"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "H"};

    /**
```

```
     * Converts a MIDI note number to ISO pitch
     * @param pitch The MIDI note number
     * @return String  ISO pitch string
     */
    public static String PitchToString(int pitch) {
        return pitches[pitch % 12]+""+((int) Math floor(pitch/12)-1);
    }

}
```

Quellcode-Datei 15: convert_midi_to_mat.m

```
% This script analyzes the pitches in the MIDI files of a given composer
% The user will be prompted to verify the mapping of track names to instruments
% Instrument metadata must be provided via CSV file in <corpus_path>/Metadaten
% Results are stored as  mat files in <corpus_path>/MAT/Sinfonien
clearvars;
close all;
clc;
javaaddpath('MidiPitchAnalyzer jar');

composer = 'Ludwig van Beethoven';
corpus_path = '  /  /  /Korpus/';

composer_short = composer(find(composer == ' ', 1, 'last')+1 end);

data_file = [corpus_path, 'Metadaten/', composer_short, ' csv'];
midi_folder = [corpus_path, 'MIDI/', composer_short];
out_folder = [corpus_path, 'MAT/', composer_short];

instruments_regex = {
    'fl(a)?ut[oei]',
    'piccolo|ottavino',
    'obo[ei]',
    'amore',
    'clarinet',
    '(?<!contra)fagot|bassoon',
    'contrafagot',
    '[ch]orn([oi])?',
    '[ei]ngl[ei]s',
    'tromb[ea]|clarin[oi]|trumpet',
    'trombon[ei]',
    't[iy]mpani',
    'violin([a-z]*)\s(i|1)[ ]?$',
    'violin([a-z]*)\s(ii|2)[ ]?$',
    'viol[ae]',
    'cell[oie](?!\ssolo|\sconcertante)',
    '(contra)?bass[oei]?$',
    'violin[eo]?\s(solo|concertante)',
    'cello\s(solo|concertante)'
    'bass[o]?\s(solo|concertante)',
    '(fagot[a-z]*|bassoon)\s(solo|concertante)'
};

% get symphony data
symphony_data = parse_symphony_csv(data_file);

% get subfolders (= symphonies)
path_dir = dir(midi_folder);
subfolders = {path_dir([path_dir( ) isdir]) name}';
subfolders = subfolders(3 end);

for i = 1   length(subfolders)
    subfolder = subfolders{i};
    subfolder_path = fullfile(midi_folder, subfolder);

    % parse the folder name to get symphony number, creator and files
    [number, creator, midi_filenames] = parse_folder(subfolder_path, subfolder);

    fprintf('Symphony %s\n', number);

    % now that we know the symphony number we can access the metadata
    [key, opus, instrument_names, movements] = parse_symphony_data(symphony_data, number);

    % parse the files (= movements)
    parse_fn = @(f)(read_midi_file(subfolder_path, f, instrument_names, instruments_regex, movements));
    movements = cellfun(parse_fn, midi_filenames);

    % create data struct
    symphony = struct('composer', composer, 'number', number, 'opus', opus, 'midi_creator', creator);
    symphony movements = movements;

    % save struct
    composer_short = composer(find(composer == ' ', 1, 'last')+1 end);
    creator_short = creator(find(creator == ' ', 1, 'last')+1 end);
    out_filename = strcat(composer_short, '_', number, '_', creator_short, ' mat');
    save(fullfile(out_folder, out_filename), 'symphony');
end
```

Quellcode-Datei 16: parse_symphony_csv.m

```
% Parses the metadata csv file
% @file  The full path of the csv file as returned by fullfile()
% @return symphony_data  Cell array representation of the metadata
function [symphony_data] = parse_symphony_csv(file)
    fid = fopen(file, 'r');
    symphony_data = textscan(fid, repmat('%s', 1, 26), 'delimiter', ',', 'CollectOutput', true);
    fclose(fid);

    symphony_data = symphony_data{1};
end
```

Quellcode-Datei 17: parse_symphony_data.m

```
% Selects a given symphony from the metadata
% @symphony_data  The movement metadata table
% @number  The symphony number
% @return key  The key of the symphony
% @return opus  The opus string of the symphony
% @return instrument_names  The instruments appearing in the symphony
% @return movements  The movements of the symphony
function [key, opus, instrument_names, movements] = parse_symphony_data(symphony_data, number)

    % get instrument names and regex from header
    instrument_names = symphony_data(1, 6 end);

    % select symphony
    data = symphony_data(2 end,  );
    data = data(strcmp(data( , 1), number),  );

    opus = data(1, 2);
    key = data(1, 4);
    movements = data( , 3 end);
end
```

Quellcode-Datei 18: parse_movement_data.m

```
% Reads the metadata of a given movement
% @instrument_names  Cell array of valid instrument names
% @instrument_regex  Cell array of instrument regular expressions
% @movement_data  Movement metadata
% @number  The movement number
% @return instruments  The instruments appearing in the movement
% @return regex  The regular expressions matching the instruments
% @return key  The key of the movement
% @return description  The movement description
function [instruments, regex, key, description] = parse_movement_data(instrument_names, instrument_regex,
    ↪movement_data, number)
    multi_key_delimiter = ';';
    regex = instrument_regex;
    data = movement_data(strcmp(movement_data( , 1), number),  );
    key = data(2);
    description = data(3);

    % find undefined and keyless instruments
    instrument_data = data( , 4 end);
    find_fn = @(string)(@(c)(strcmp(string, c)));
    undefined = cellfun(find_fn('0'), instrument_data);

    instrument_data = instrument_data(~undefined);
    instruments = instrument_names(~undefined);
    regex = regex(~undefined);

    keyless = cellfun(find_fn('1'), instrument_data);

    % find instruments which have their key defined
    key_defined = ~keyless;
    key_defined_indices = find(key_defined);

    % get the keys of the instruments
    instrument_keys = instrument_data(key_defined);

    % split keys on delimiter to detect instruments in multiple keys
    key_split_fn = @(c)(strsplit(c, multi_key_delimiter));
    key_split = cellfun(key_split_fn, instrument_keys, 'UniformOutput', false);

    % add keys to instruments in a single key
    single = cellfun(@length, key_split) == 1;
    single_indices = key_defined_indices(single);
    single_keys = key_split(single);
    single_keys = [single_keys{ }];
    instruments(single_indices) = strcat(instruments(single_indices), {' in '}, single_keys);

    % add keys to instruments in more than one key
    multiple = ~single;
    multiple_indices = key_defined_indices(multiple);
```

```
    multiple_keys = key_split(multiple);

    offset = 0;
    for i = 1   length(multiple_keys)
        k = multiple_keys{i};
        n_keys = length(k);
        multiple_index_start = multiple_indices(i) + offset;
        instrument = instruments(multiple_index_start);

        instruments = [instruments(1 multiple_index_start-1),
            strcat(instrument, {' in '}, k),
            instruments(multiple_index_start+1 end)];

        regex = [regex(1 multiple_index_start-1),
            repmat(regex(multiple_index_start), 1, n_keys),
            regex(multiple_index_start+1 end)];

        offset = offset + n_keys - 1;
    end

    instruments = instruments';

end
```

Quellcode-Datei 19: parse_folder.m

```
% Searches a folder for MIDI files representing the movements of a symphony
% @folder  Full folder path as returned by fullfile()
% @folder_name  Name of the folder
% @return number  Symphony number
% @return creator  Creator of the MIDI files
% @return midi_filenames  Cell array of MIDI filenames
function [number, creator, midi_filenames] = parse_folder(folder, folder_name)
    folder_regex = 'Symphony No ([0-9]+) in ([A-Za-z-]+) *, ([A-Za-z]+ [0-9A-za-z ]+)  *-- ( *)';
    match = regexp(folder_name, folder_regex, 'tokens');

    if isempty(match) || length(match{1}) ~= 4
        error(['Unable to parse folder name  ', folder]);
    end

    number = match{1}{1};
    creator = match{1}{4};

    midi_files = dir(fullfile(folder, '* mid'));
    midi_filenames = {midi_files name};
end
```

Quellcode-Datei 20: parse_file.m

```
% Analyzes the pitches of a given MIDI file using the Java application
% @file  The MIDI file as returned by fullfile()
% @file_name  The filename of the MIDI file
% @return movement  The movement number
% @return description  The movement description
% @return file_analysis  The Java MidiFileAnalysis object
function [movement, description, file_analysis] = parse_file(file, file_name)
    file_regex = '([0-9;]+) ( *) mid';
    match = regexp(file_name, file_regex, 'tokens');

    if isempty(match) || length(match{1}) ~= 2
        error(['Unable to parse file name  ', file_name]);
    end

    movement = match{1}{1};
    description = match{1}{2};
    file_analysis = analysis MidiAnalyzer analyze(file);
end
```

Quellcode-Datei 21: read_midi_file.m

```
% Analyzes a MIDI file and matches the track names to given instrument names
% @folder  The folder the MIDI file resides in
% @filename  The name of the MIDI file
% @instrument_names  Cell array of valid instrument names
% @instruments_regex  Cell array of regular expressions to match <instrument_names>
% @movement_data  Cell array of movements
% @return movement  Struct containing the instruments and histograms as well as ignored and undefined instruments
function [movement] = read_midi_file(folder, filename, instrument_names, instruments_regex, movement_data)
    midi_file = fullfile(folder, filename);
    [movement_number, ~, file_analysis] = parse_file(midi_file, filename);

    fprintf('Movement %s\n', movement_number);

    [instruments, regex, key, description] = parse_movement_data(instrument_names, instruments_regex, movement_data,
        ↪ movement_number);
```

```
    [~, instrument_to_tracks, ignored_tracks, instruments, undefined_instruments] = parse_tracknames(file_analysis,
        ↪instruments, regex);
    [instrument_histograms] = get_instrument_histograms(file_analysis, instrument_to_tracks);

    movement = struct('number', movement_number, 'key', key, 'description', description);
    movement instruments = instruments;
    movement instrument_histograms = instrument_histograms;
    movement ignored_tracks = ignored_tracks;
    movement undefined_instruments = undefined_instruments;
end
```

Quellcode-Datei 22: parse_tracknames.m

```
% Maps the MIDI track names to given instrument names
% The user will be prompted to verify or adjust the mapping
% @file_analysis  Java MidiFileAnalysis object
% @possible_names  Possible instruments names
% @names_regex  Regular expression for matching of instrument names and tracks
% @return track_to_instrument  Cell array mapping track numbers to instrument names
% @return instrument_to_tracks  Cell array mapping instrument names to track numbers
% @return ignored_tracks  Numbers of ignored tracks
% @return instruments  Cell array of valid instrument names
% @return undefined_instruments  Cell array of undefined instrument names
function [track_to_instrument, instrument_to_tracks, ignored_tracks, instruments, undefined_instruments] =
    ↪parse_tracknames(file_analysis, possible_names, names_regex)
    instruments = possible_names;
    ignored_tracks = {};
    undefined_instruments = {};
    track_names = cell(file_analysis getTrackNames() toArray());
    track_numbers = file_analysis getValidTrackNumbers() toArray();
    n_tracks = length(track_names);

    track_to_instrument = cell(1, n_tracks);
    instrument_to_tracks = [possible_names( ), cell(length(possible_names), 1)];

    find_fn = @(string)(@(c)(strcmp(string, c)));

    for i = 1   n_tracks
        track_name = track_names{i};
        track_number = track_numbers(i);

        instrument = regexp(lower(track_name), names_regex, 'match');
        instrument = ~cellfun(@isempty, instrument);

        if ~any(instrument)
            fprintf('Unable to identify instrument name "%s"\n', track_name);
            instrument = select_instrument(track_number, track_name, possible_names);
        elseif sum(instrument) > 1
            fprintf('Instrument "%s" matches more than 1 category \n', track_name);
            instrument = select_instrument(track_number, track_name, possible_names(instrument));
        else
            instrument = possible_names{instrument};
            fprintf('"%s" => "%s"  ', track_name, instrument);
            confirm = input('Is that correct? [y]\n', 's');
            if ~strcmp(confirm, 'y')
                instrument = select_instrument(track_number, track_name, possible_names);
            end
        end

        if instrument > -1
            instrument_index = cellfun(find_fn(instrument), instrument_to_tracks( , 1));

            current_tracks = instrument_to_tracks(instrument_index, 2);
            instrument_to_tracks(instrument_index, 2) = {[current_tracks{1}, track_number]};

            track_to_instrument(i) = {instrument};
        else
            ignored_tracks = [ignored_tracks, strjoin({[num2str(track_number), ' '], track_name}, ' ')];
        end
    end

    empty_instruments = cellfun(@isempty, instrument_to_tracks( , 2));

    if any(empty_instruments)
        empty_instrument_names = instrument_to_tracks(empty_instruments);
        message = ['Some instruments are not defined '; empty_instrument_names];
        warning('%s\n', message{ });
        undefined_instruments = [undefined_instruments, empty_instrument_names];

        instrument_to_tracks = instrument_to_tracks(~empty_instruments,  );
        instruments = instruments(~empty_instruments,  );
    end

    function instrument = select_instrument(track_number, track_name, possible_names)
        n_names = length(possible_names);
        name_indices = num2cell(1 n_names)';
        name_choice = [name_indices, possible_names];

        fprintf('Track %i  "%s"\n', track_number, track_name);
        disp(name_choice);
```

```
        name_index = input('Select an instrument category ');

        if name_index == -1
            disp('Ignoring this instrument ');
            instrument = -1;
        elseif isempty(name_index) || name_index > n_names || name_index < 1
            error('Invalid instrument category!')
        else
            instrument = possible_names{name_index};
        end
    end

end
```

Quellcode-Datei 23: get_instrument_histograms.m

```
% Extracts the pitch histograms from a MidiFileAnalysis object
% @file_analysis  Java MidiFileAnalysis object
% @instrument_to_tracks  Cell array mapping instrument names to track numbers
% @return instrument_histograms  Cell array of instrument pitch histograms
function [instrument_histograms] = get_instrument_histograms(file_analysis, instrument_to_tracks)
    tracks = instrument_to_tracks( , 2);

    get_fn = @(c)(file_analysis getTrackAnalysis(c));
    track_fn = @(c)(arrayfun(get_fn, c, 'UniformOutput', false));
    track_analyses = cellfun(track_fn, tracks, 'UniformOutput', false);

    histogram_fn = @(a)(to_histogram(a{1}));
    instrument_histograms = arrayfun(histogram_fn, track_analyses);

    function histogram = to_histogram(track_analyses)

        pitches_fn = @(ta)(ta getHistogram() getPitches());
        pitches = cellfun(pitches_fn, track_analyses, 'UniformOutput', false);

        % merge multiple histograms, convert to double
        pitches = sum(cell2mat(pitches), 2);

        n_notes = sum(pitches);
        n_pitches = nnz(pitches);
        range_start = find(pitches, 1) - 1;
        range_end = find(pitches, 1, 'last') - 1;

        histogram = struct('n_notes', n_notes, 'n_pitches', n_pitches,
            'range_start', range_start, 'range_end', range_end,
            'pitches', pitches);

    end

end
```

Quellcode-Datei 24: fix_beethoven_histograms.m

```
% This script fixes the pitch distributions of two wrongly coded instruments in Beethoven's symphonies
% The new distributions are based on the output from mftext, stored as  txt files in subdirectory 'mftext'
clearvars;
clc;

% define composer
composer = 'Beethoven';

% define corpus location
corpus_path = '  /  /  /Korpus';

% define file locations
txt_path = fullfile('mftext');
mat_path = fullfile(corpus_path, 'MAT', composer);
out_path = fullfile(corpus_path, 'MAT', composer);

% define file name suffix for output files
out_suffix = '_Fixed';

% define incorrect instruments
incorrect_instruments = {
    struct('symphony', 9, 'movement', 4, 'name', 'Clarinetto in B/A',
        'real_names', {{'Clarinetto in B', 'Clarinetto in A'}}),
    struct('symphony', 3, 'movement', 1, 'name', 'Corno in Es',
        'real_names', {{'Corno in Es', 'Corno in F'}})};

% iterate incorrect instruments
for i = 1   length(incorrect_instruments)
    incorrect = incorrect_instruments{i};

    % get symphony file
    mat_files = dir(fullfile(mat_path, '* mat'));
    regex = [composer, '_', num2str(incorrect symphony)];
    match = regexp({mat_files name}, regex, 'match');
```

```
    index = find(~cellfun(@isempty, match));
    symphony_file = fullfile(mat_path, mat_files(index) name);

    % get text files
    txt_name = strcat(composer, '_', num2str(incorrect symphony),
        '_', num2str(incorrect movement), '_',
        strrep(incorrect real_names, ' ', '_'), ' txt');

    % compute corrected data
    symphony = change_symphony_instrument(symphony_file, incorrect movement,
        incorrect name, incorrect real_names, fullfile(txt_path, txt_name));

    % save the corrected data
    out_name = strrep(mat_files(index) name, ' mat', [out_suffix, ' mat']);
    save(fullfile(out_path, out_name), 'symphony');

end
```

Quellcode-Datei 25: change_symphony_instrument.m

```
% Removes data of a given instrument in a symphony mat file and inserts new instruments in its place
% Pitch histograms of new instruments are calculated based on given mftext output files
% @symphony_file  The full filename of the  mat file
% @movement_no  The number of the movement to change
% @incorrect_instrument_name  Name of instrument to be removed
% @new_instrument_names  Names of instruments to be inserted
% @new_instrument_txt  Cell array of mftext output files containing the pitches of the new instruments
% @return new_symphony  The new symphony struct
function new_symphony = change_symphony_instrument(symphony_file, movement_no,
    incorrect_instrument_name, new_instrument_names, new_instrument_txt)

    % load data of original  mat-file
    new_symphony = load(symphony_file);
    new_symphony = new_symphony symphony;
    movement = new_symphony movements(movement_no);
    N_instruments = length(movement instruments);

    % find the index of the incorrect instrument to get indices of new instruments
    N_new_instruments = length(new_instrument_names);
    start_index = find(strcmp(movement instruments, incorrect_instrument_name));
    end_index = start_index + N_new_instruments - 1;
    indices = start_index   end_index;

    % remove incorrect instrument name, insert new names in its place
    new_instruments = movement instruments(1 start_index-1);
    new_instruments(indices) = new_instrument_names;
    new_instruments(end_index+1 N_instruments+N_new_instruments-1) = movement instruments(end_index N_instruments);

    % create new histogram array with gaps for new instruments
    new_histograms = movement instrument_histograms(1 start_index-1);
    new_histograms(end_index+1 N_instruments+N_new_instruments-1) = movement instrument_histograms(end_index
        ↪N_instruments);

    % iterate new instruments
    for i = 1   N_new_instruments
        index = indices(i);

        % get the histogram and insert it into the histograms array
        histogram = get_histogram_from_mftext(new_instrument_txt{i});
        new_histograms(index) n_notes = sum(histogram);
        new_histograms(index) n_pitches = nnz(histogram);
        new_histograms(index) range_start = find(histogram, 1) - 1;
        new_histograms(index) range_end = find(histogram, 1, 'last') - 1;
        new_histograms(index) pitches = histogram;
    end

    % remove undefined instruments if they were defined by new instruments
    fn = @(x) find(strcmp(x, movement undefined_instruments));
    i = cell2mat(cellfun(fn, new_instrument_names, 'UniformOutput', false));
    movement undefined_instruments(i) = [];

    if isempty(movement undefined_instruments)
        movement undefined_instruments = {};
    end

    % save modified instrument data to symphony struct
    movement instruments = new_instruments;
    movement instrument_histograms = new_histograms;
    new_symphony movements(movement_no) = movement;
end
```

Quellcode-Datei 26: get_histogram_from_mftext.m

```
% Scans a mftext output file and creates a histogram from its pitch information
% @file  Output file from mftext, possibly reduced to a specific instrument
function histogram = get_histogram_from_mftxt(file)
    histogram = zeros(128, 1);
```

```
    % regular expression  capture pitch in a group, only match volume > 0
    regex = 'Note on *pitch=([0-9]+) vol=[1-9]\d*';

    % read file line by line
    fid = fopen(file);
    tline = fgetl(fid);
    while ischar(tline)

        % try to match line with regular expression
        [~, pitch] = regexp(tline, regex, 'match', 'tokens');

        % found a match  add pitch to histogram
        if ~isempty(pitch)
            pitch = str2double(pitch{1});
            histogram(pitch+1) = histogram(pitch+1) + 1;
        end

        tline = fgetl(fid);
    end

    fclose(fid);
end
```

Quellcode-Datei 27: get_symphonies.m

```
% Returns the MATLAB symphony files in a given folder
% @mat_path  Path to the  mat files
% @return symphonies  Cell array of symphony structs
function symphonies = get_symphonies(mat_path)

    % select  mat-files
    mat_files = dir(fullfile(mat_path, '* mat'));
    mat_files = fullfile(mat_path, {mat_files name});
    N = length(mat_files);

    % initialise return array
    symphonies = cell(1, N);

    % fill array with symphony objects
    for i = 1   N
        data = load(mat_files{i});
        symphonies{i} = data symphony;
    end

end
```

Quellcode-Datei 28: get_valid_instruments.m

```
% Returns the valid instruments and their pitch histograms from a movement struct
% @movement  The movement struct
% @return instruments  The names of the valid instruments
% @return histograms  The pitch histograms of the valid instruments
function [instruments, histograms] = get_valid_instruments(movement)

    % access instrument data of this movement
    defined_instruments = movement instruments;
    undefined_instruments = movement undefined_instruments;

    % get indices of valid entries
    valid_indices = ~ismember(defined_instruments, undefined_instruments);

    % return valid instruments and their histograms
    instruments = defined_instruments(valid_indices);
    histograms = movement instrument_histograms(valid_indices);
end
```

Quellcode-Datei 29: convert_mat_to_csv.m

```
% This script converts pitch distribution data stored in  mat files to CSV files
% CSV files are created for every movement of every composer
clearvars;
clc;

% define composers and data URLs
composers = {'Beethoven', 'Haydn', 'Mozart', 'Schubert'};
urls = {'http //www classicalarchives com/midi/composer/2156 html',
        'http //www classicalarchives com/midi/composer/2679 html',
        'http //www classicalarchives com/midi/composer/3052 html',
        'http //www classicalarchives com/midi/composer/3308 html'};

% define CSV delimiter
delimiter = ',';

% define corpus folder location
```

```
corpus_path = '  /  /  /Korpus/';

% define output folder name to be added in corpus folder
output_folder = 'CSV';

% define comments location
comments_file = fullfile(corpus_path, 'Metadaten', 'Kommentare csv');

% read comments
fid = fopen(comments_file, 'r');
comments = textscan(fid, repmat('%q', 1, 4), 'delimiter', delimiter);
comments = [comments{ }]; % unwrap nested cell arrays
comments = comments(2 end,  ); % skip header
fclose(fid);

% iterate composers
for c = 1   length(composers)

    % select composer name and URL
    composer = composers{c};
    url = urls{c};

    % get the  mat-files for this composer
    mat_path = fullfile(corpus_path, 'MAT', composer);
    symphonies = get_symphonies(mat_path);

    % find comments for this composer
    composer_comments_indices = ismember(comments, composer);
    composer_comments = comments(composer_comments_indices, 2 end);

    % select output location, create folder if necessary
    output_path = fullfile(corpus_path, output_folder, composer);

    if ~exist(output_path, 'dir')
        mkdir(output_path);
    end

    % apply conversion function to all symphonies  this creates a CSV file
    % for every movement of every symphony
    fn = @(s)(symphony_to_csv(s, output_path, delimiter, url, composer_comments));
    cellfun(fn, symphonies);
end
```

Quellcode-Datei 30: symphony_to_csv.m

```
% Writes CSV files containing the pitch histograms for all movements of a symphony
% @symphony  The symphony struct
% @output_folder  The output directory for the CSV file
% @delimiter  The CSV delimiter
% @url  Origin URL of the MIDI files
% @comments  Cell array of comments regarding MIDI file analysis
function symphony_to_csv(symphony, output_folder, delimiter, url, comments)

    % define header
    header_captions = {'Composer', 'Opus', 'Symphony', 'Movement',
                       'MIDI Author', 'URL', 'Comments'};
    header_format = ['%s', delimiter, '%s\n'];

    % define format of instrument data
    instruments_format = ['%s', delimiter,
                          repmat(['%d', delimiter], 1, 131), '%d\n'];

    % get comments for this symphony
    comments_indices = ismember(comments( , 1), symphony number);
    comments_symphony = comments(comments_indices, 2 end);

    % iterate movements
    for m = 1   length(symphony movements)

        % select movement data
        movement_number = num2str(m);
        movement = symphony movements(m);
        composer = symphony composer;
        composer_short = composer(find(composer == ' ', 1, 'last') + 1 end);

        % get comment for this movement
        comments_indices = ismember(comments_symphony( , 1), movement_number);
        if isempty(comments_indices) || sum(comments_indices) == 0
            comment = '';
        else
            comment = ['"', comments_symphony{comments_indices, 2}, '"'];
        end

        % generate output filename
        filename = [composer_short, '_', symphony number, '_', movement_number, ' csv'];

        % generate header
        header_values = {symphony composer, symphony opus, symphony number,
                         movement_number, symphony midi_creator, url, comment};
        header = [header_captions; header_values];
```

```
        % open output file
        csv_file = fullfile(output_folder, filename);
        fid = fopen(csv_file, 'w');

        % print headers to output file
        fprintf(fid, header_format, header{ , });
        write_instrument_header(fid, delimiter);

        % get instrument data of this movement
        [instruments, histograms] = get_valid_instruments(movement);

        % iterate instrument data
        for i = 1   length(instruments)
            instrument = instruments(i);
            histogram = histograms(i);

            % select instrument data values
            data = [instrument, {histogram n_notes, histogram n_pitches,
                    histogram range_start, histogram range_end},
                    num2cell(histogram pitches)'];

            % print instrument data to output file
            fprintf(fid, instruments_format, data{ , });
        end

        % close output file
        fclose(fid);
    end
end
```

Quellcode-Datei 31: write_instrument_header.m

```
% Appends a header to a CSV file that stores instrument pitch information
% @fid  The file ID of the CSV file
% @csv_delimiter  The CSV delimiter
function write_instrument_header(fid, csv_delimiter)

    % calculate MIDI pitch names
    midi_bins = 0 127;
    notes = {'C' 'C#' 'D' 'D#' 'E' 'F' 'F#' 'G' 'G#' 'A' 'A#' 'B'};
    pitch_names = strrep(strcat(notes(mod(midi_bins, 12)+1)', '/',
                  num2str(floor(midi_bins/12)'))', ' ', '');

    % define MIDI header
    header_midi = ['MIDI Value', midi_bins, 'Pitch', pitch_names];
    header_midi_format = [repmat(csv_delimiter, 1, 4), '%s', csv_delimiter,
                          repmat(['%d', csv_delimiter], 1, 127), '%d\n',
                          repmat(csv_delimiter, 1, 4), '%s', csv_delimiter,
                          repmat(['%s', csv_delimiter], 1, 127), '%s\n'];

    % define instrument header
    header_instruments = {'Instrument', 'Notes', 'Pitches', 'Range Start', 'Range End'};
    header_instruments_format = [repmat(['%s', csv_delimiter], 1, 4), '%s\n'];

    % write header to output file
    fprintf(fid, header_midi_format, header_midi{ , });
    fprintf(fid, header_instruments_format, header_instruments{ , });
end
```

Quellcode-Datei 32: create_corpus_ambitus.R

```
# This script matches the ambitus of the datasets of all <composers> against the range defined in <ambitus_file>
# <ambitus_file> is a CSV file with 5 columns, matching the original instrument name in the corpus to its ambitus
# The first column is the instrument name, followed by written and numeric MIDI pitch minimum and maximum values
# Pitch distributions for the datasets of all <composers> are read from the corpus in <corpus_folder>
# The distributions are corrected by clipping them to the valid pitch range and the corrected CSV files stored in
# <out_folder>/<composer>  The original file headers are maintained
# Any distributions outside the valid pitch range are logged in <error_log_file> as defined in <error_header>

# clear environment, set working directory, import helpers
rm(list=ls())
setwd("<path>/R/1-Datengewinnung/")
source("get_histogram_csv_line R")

# input  ambitus information for all instruments
ambitus_file = "Instrumente_Tonumfänge csv"
ambitus_data = read csv(ambitus_file, skip=1, header=FALSE, stringsAsFactors=FALSE)

# input  corpus data
corpus_folder = "  /  /  /Korpus/CSV"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
csv_header_length = 10

# output  output folder and error log file
out_folder = "  /Korpus - Ambitus korrigiert"
error_log_file = file path(out_folder, "Instrumente_Tonumfänge_Fehler csv")
```

```
error_header = rbind(c("Autor", "Komponist", "Sinfonie", "Satz", "Instrument", "< Ambitus", "> Ambitus"))
write table(error_header, file=error_log_file, sep=",", row names=FALSE, col names=FALSE)

# iterate composers
for (composer in composers) {
  data_folder = file path(corpus_folder, composer)
  data_files = list files(data_folder)

  # create directory for this composer if it doesn't already exist
  composer_folder = file path(out_folder, composer)
  dir create(composer_folder, showWarnings=FALSE)

  # iterate files of this composer
  for (file in data_files) {
    in_file = file(file path(data_folder, file))

    # create out file if it doesn't already exist
    out_file = file path(composer_folder, file)
    file create(out_file, showWarnings=FALSE)

    # copy header
    header = readLines(in_file, csv_header_length)
    writeLines(header, out_file)

    # iterate data
    data = read table(in_file, skip=csv_header_length, header=FALSE, sep=",", stringsAsFactors=FALSE)
    for (i in 1 nrow(data)) {
      instrument = data[i, 1]

      # read ambitus of this instrument
      ambitus_index = ambitus_data[1] == sub("  *", "", instrument)
      ambitus_min = ambitus_data[ambitus_index, 4]
      ambitus_max = ambitus_data[ambitus_index, 5]

      # wrong pitch range  need to correct histogram
      if (data[i, 4] < ambitus_min || data[i, 5] > ambitus_max) {
        histogram = data[i, 6 133]

        # get movement metadata, count notes below and above ambitus
        author = sub("^ *,", "", header[5])
        composer = sub("^ *,", "", header[1])
        symphony = sub("^ *,", "", header[3])
        movement = sub("^ *,", "", header[4])
        n_below = sum(histogram[1 ambitus_min])
        n_above = sum(histogram[(ambitus_max+2) 128])

        # write line to error log
        log_line = rbind(c(author, composer, symphony, movement, instrument, n_below, n_above))
        write table(log_line, file=error_log_file, sep=",", row names=FALSE, col names=FALSE, append=TRUE)

        # clip histogram
        histogram_clipped = rep(0, 128)
        histogram_clipped[(ambitus_min+1) (ambitus_max+1)] = histogram[(ambitus_min+1) (ambitus_max+1)]
        out_line = get_histogram_csv_line(instrument, histogram_clipped)
      }
      else { # no need to change the histogram
        out_line = data[i, ]
      }

      # write line to output file
      write table(out_line, file=out_file, sep=",", row names=FALSE, col names=FALSE, append=TRUE)
    }
  }
}
```

Quellcode-Datei 33: create_corpus_instruments.R

```
# This script merges the pitch histograms of all instruments defined in <instruments_file> for every composer
# <instruments_file> is a CSV file with two columns, matching the original instrument name in the corpus
# to a custom label for the output (e g  the German translation)
# Pitch distributions for the instruments of all <composers> are read from the corpus in <corpus_folder>
# Results are stored as CSV <out_folder>/<out_filename>, according to the columns in <out_header>
# Keys of transposed instruments  are removed, i e  their pitch distributions are merged
# Solo parts are merged with their respective instruments
# Instrument frequencies are counted for every composer and stored in separate CSV files

# clear environment, set working directory, import helpers
rm(list=ls())
setwd("<path>/R/1 - Datengewinnung/")
source("get_histogram_csv_line R")

# input  instrument list and translation to German
instruments_file = "Instrumente_Übersetzung csv"
instruments_data = read csv(instruments_file, skip=1, header=FALSE, stringsAsFactors=FALSE)
N_instruments = nrow(instruments_data)

# input  corpus data
corpus_folder = "  /  /  /Korpus - Ambitus korrigiert"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
csv_header_length = 10
```

```
# output  output folder and file header
out_folder = "  /Korpus - Instrumente"
out_header = rbind(c("Instrument", "Notes", "Pitches", "Range Start", "Range End", seq(0, 127)))

# iterate composers
for (composer in composers) {
  data_files = list files(file path(corpus_folder, composer), full names=TRUE)

  # create named list of emtpy histograms for all instruments  <instrument label> -> <histogram>
  instruments_map = vector("list", N_instruments)
  names(instruments_map) = instruments_data[[1]]
  instruments_map = lapply(instruments_map, function(x) rep(0, 128))

  # create list for the instrument frequencies of this composer
  instrument_count = as list(rep(0, N_instruments))

  # iterate files of this composer
  for (file in data_files) {
    data = read table(file, skip=csv_header_length, header=FALSE, sep=",", stringsAsFactors=FALSE)

    # iterate instrument data in this file
    for (i in 1 nrow(data)) {
      line = data[i, ]
      instrument_label = data[i, 1]
      histogram = data[i, 6 133]

      # remove keys of transposed instruments, map solo parts to their respective instruments
      instrument = sub(" in  *", "", instrument_label)

      # if instrument is defined in <instruments_file> and not a solo part  count it
      if (!is null(instruments_map[[instrument]])) {
        instrument_index = which(instruments_data[[1]]==instrument)
        instrument_count[instrument_index] = instrument_count[[instrument_index]] + 1
      }

      instrument = sub(" solo", "", instrument)
      if (instrument == "Violino") {
        instrument = "Violino I"
      }

      # if instrument is defined in <instruments_file>  add histogram to map
      if (!is null(instruments_map[[instrument]])) {
        instruments_map[[instrument]] = unlist(instruments_map[[instrument]] + histogram)
      }
    }
  }

  # write merged histograms to csv file
  out_filename = paste0(composer, "_Instrumente csv")
  out_file = file path(out_folder, out_filename)
  write table(out_header, file=out_file, sep=",", row names=FALSE, col names=FALSE)
  for (i in 1 N_instruments) {
    out_line = get_histogram_csv_line(instruments_data[i, 2], instruments_map[[i]])
    write table(out_line, file=out_file, sep=",", row names=FALSE, col names=FALSE, append=TRUE)
  }

  # write instrument frequencies to csv file
  out_filename = paste0(composer, "_Instrumente_Häufigkeiten csv")
  out_file = file path(out_folder, out_filename)
  write table(rbind(c("Instrument", "Häufigkeit")), file=out_file, sep=",", row names=FALSE, col names=FALSE)
  for (i in 1 N_instruments) {
    write table(rbind(c(instruments_data[[2]][i], instrument_count[[i]])), file=out_file, sep=",", row names=FALSE,
        ↪col names=FALSE, append=TRUE)
  }
}
```

Quellcode-Datei 34: get_histogram_csv_line.R

```
# Converts a pitch histogram to an array for use in CSV conversion
# The first column is a <label> for the histogram, e g  the instrument name,
# Other columns are number of notes, number of pitches, the pitch range and the histogram itself
# <label>  string  Label for the histogram
# <histogram>  num [1x128]  The pitch histogram
# returns; string array [1x133]
get_histogram_csv_line = function(label, histogram) {
  histogram = unlist(histogram)
  nonzero_midi_bins = which(histogram != 0) - 1
  n_notes = sum(histogram)
  n_pitches = length(nonzero_midi_bins)
  range_start = min(suppressWarnings(min(nonzero_midi_bins)), 0) # convert -Infinity to 0
  range_end = max(suppressWarnings(max(nonzero_midi_bins)), 0) # convert +Infinity to 0

  return(rbind(c(label, n_notes, n_pitches, range_start, range_end, histogram)))
}
```

Quellcode-Datei 35: create_authors_note_stats.R

```
# This script counts the coded notes of all MIDI authors and writes the sum to the CSV file <out_file>
# Pitch distributions for the datasets of all <composers> are read from the corpus in <corpus_folder>
# The total number of notes for each author is computed and written to <out_file> as defined in <out_header>

# clear environment, set working directory, import helpers
rm(list=ls())
setwd("<path>/R/2 - Auswertung/Fehlerschätzung/")

# input  corpus data
corpus_folder = "  /  /  /  /Korpus/CSV"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
csv_header_length = 10

# output  author stats file
out_file = "Autoren_Gesamtzahl_Töne csv"
out_header = rbind(c("Autor", "Anzahl Töne"))
write table(out_header, file=out_file, sep=",", row names=FALSE, col names=FALSE)

# use a list to map authors to number of notes
authors_map = list()

# iterate composers
for (composer in composers) {
  data_files = list files(file path(corpus_folder, composer), full names=TRUE)

  # iterate files for this composer
  for (file in data_files) {

    # get file contents, extract author
    data = read table(file, skip=4, header=FALSE, sep=",", stringsAsFactors=FALSE, fill=TRUE)
    author = data[1, 2]

    # put author in list
    if (is null(authors_map[[author]])) {
      authors_map[[author]] = 0
    }

    # add notes in this file to author's list entry
    authors_map[[author]] = authors_map[[author]] + sum(as numeric(data[7 nrow(data), 2]))
  }
}

# write results to output file
authors = names(authors_map)
for (author in authors) {
  out_line = rbind(c(author, authors_map[[author]]))
  write table(out_line, file=out_file, sep=",", row names=FALSE, col names=FALSE, append=TRUE)
}
```

Quellcode-Datei 36: create_error_confidence_intervals.R

```
# This script computes confidence intervals of the error rate of the MIDI authors and writes them to <out_file>
# Data is taken from <in_file>, a CSV file with 4 columns  The author names, total number of notes, number of notes
# in analyzed sample, and number of errors in analyzed sample
# Intervals are computed using both the Clopper-Pearson method and the hypergeometric distribution for a given
# <confidence_level> and rounded to a given number of <decimal_places>
# Results are stored as CSV in <out_file> according to <out_header>

# clear environment, set working directory
rm(list=ls())
setwd("<path>/R/2 - Auswertung/Fehlerschätzung/")

# input file
in_file = "Autoren_Fehler csv"
data = read csv(in_file, skip=1, header=FALSE, stringsAsFactors=FALSE)

# output  results file
out_file = "Autoren_Fehlerschätzung csv"
out_header = c("Autor", "Noten gesamt", "Noten Stichprobe", "Fehler Stichprobe", "g_u Clopper-Pearson",
               "g_o Clopper-Pearson", "g_u Hypergeom ", "g_o Hypergeom ", "g_u %", "g_o %")
write table(rbind(out_header), file=out_file, quote=FALSE, sep=",", row names=FALSE, col names=FALSE, append=FALSE)

# set confidence level
confidence_level = 0 95

# set number of decimal places of percentages
decimal_places = 2

# iterate authors
for (i in 1 length(data)) {
  author = data[i, 1]
  N = data[i, 2] # number of notes of this author
  n = data[i, 3] # number of notes in sample
  k = data[i, 4] # number of errors in sample

  # compute confidence interval using Clopper-Pearson method
  binom_test = binom test(k, n, conf level=confidence_level)
  lower_limit  = binom_test$conf int[[1]]
  upper_limit  = binom_test$conf int[[2]]
```

```
  pearson = c(ceiling(lower_limit*N), floor(upper_limit*N))

  # compute confidence interval using hypergeometric distribution
  # See Küchenhoff (2011)  "Stichproben  Methoden und praktische Umsetzung mit R"  Springer
  # The following algorithm is part of <http //www statistik lmu de/~helmut/VPL/hypergeo r>
  lower_limit = k
  while (phyper(k-1, lower_limit, N-lower_limit, n) > (confidence_level+1)/2) {
    lower_limit = lower_limit+1
  }
  lower_limit = max(k, lower_limit-1)

  upper_limit = N-(n-k)
  while (phyper(k, upper_limit, N-upper_limit, n) < (1-confidence_level)/2) {
    upper_limit = upper_limit-1
  }
  upper_limit = min(N-(n-k), upper_limit+1)

  percent = round(c(lower_limit/N, upper_limit/N)*100, decimal_places)
  hypergeo = c(lower_limit, upper_limit)

  # write confidence intervals to csv
  out_data = c(author, N, n, k, pearson[1], pearson[2], hypergeo[1], hypergeo[2], percent[1], percent[2])
  write table(rbind(out_data), file=out_file, quote=FALSE, sep=",", row names=FALSE, col names=FALSE, append=TRUE)
}
```

Quellcode-Datei 37: create_composer_boxplot.R

```
# This script creates a boxplot showing the merged instrument pitch distributions of all composers
# Pitch distributions for the instruments of all <composers> are read from the corpus in <corpus_folder>
# The boxplot is stored as a PNG image in <out_file>

# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
source("read_csv_histograms R")
source("get_histogram_values R")
source("histogram_boxplot R")

# input  corpus data
corpus_folder = "  /  /  /Korpus - Instrumente"
csv_header_length = 1

# input  composer data
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
N_composers = length(composers)
colors = palette(rainbow(N_composers))
N_instruments = 14

# define x axis ticks and limits
x_ticks = seq(0, 127, 12) # mark all C pitches
x_limits = c(11,109) # limit pitch range to C0-C8

# output  image file
out_file = "Vergleich Komponisten/Boxplots/Komponisten_Instrumente_Boxplot png"

# iterate composers and put their histograms in a list
boxplot_data = vector("list", N_composers*N_instruments)
for (c in seq(1, N_composers)) {
  filename = paste0(composers[c], "_Instrumente csv")
  data = read_csv_histograms(file path(corpus_folder, filename), csv_header_length)

  # put histograms in list, sorted by instrument first, composer second
  for (i in seq(0, N_instruments-1)) {
    boxplot_data[[i*N_composers+c]] = get_histogram_values(data$histograms[i+1, ])
  }
}

# create boxplot and add a legend
png(out_file,
    width = 16,
    height = 20,
    units = "cm",
    res = 120,
    pointsize = 12
)

# set margins and font size
par(mar=c(4, 5 2, 1, 1),
    cex axis = 0 8,
    cex lab = 1
)
histogram_boxplot(boxplot_data, data$labels, colors, 0, 2 2, 1 2, x_ticks, x_limits)
legend("bottomright", composers, fill=colors, cex=par("cex axis"))
dev off()
```

Quellcode-Datei 38: create_composer_instrument_plots.R

```r
# This script plots QQ-plots, ECDF and histograms for the merged instrument pitch distributions of all composers
# Pitch distributions for the instruments of all <composers> are read from the corpus in <corpus_folder>
# Instrument data of the 4 composers is shown in a 2x2 grid to compare the distributions of each instrument
# Results are stored as PNG images in <out_folder_qq> and <out_folder_ecdf>

# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
source("Vergleich Komponisten/get_composer_instrument_data R")
source("quad_plot R")
source("get_histogram_values R")
source("get_iso_pitches R")
source("histogram_plot R")

# input  corpus and composer data
corpus_folder = "  /  /  /Korpus - Instrumente"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")

# output  image folders
out_folder_qq = "Vergleich Komponisten/QQ-Plots"
out_folder_ecdf = "Vergleich Komponisten/ECDF"
out_folder_histogram = "Vergleich Komponisten/Histogramme"
dir create(out_folder_qq, showWarnings=FALSE)
dir create(out_folder_ecdf, showWarnings=FALSE)
dir create(out_folder_histogram, showWarnings=FALSE)

# output  plot properties

qq_x_limits = c(-4 5, 4 5)
ecdf_x_limits = c(24, 108)
ecdf_x_ticks = seq(0, 127, 12)
values_color = "darkgrey"

# get merged instrument distributions for all composers
composer_data = get_composer_instrument_data(composers, corpus_folder)

# create plots for each instrument
for (i in seq(1, length(composer_data$labels))) {
  instrument = composer_data$labels[i]

  # create QQ-plot
  out_filename_qq = paste0(instrument, "_QQ-Plot png")
  png(file path(out_folder_qq, out_filename_qq),
      width = 20,
      height = 14,
      units = "cm",
      res = 120,
      pointsize = 12
  )
  par(mar = c(4, 4, 2, 1))
  quad_plot(composer_data, i, function(histogram, composer, instrument) {
    values = get_histogram_values(histogram)
    qqnorm(values, xlab="Erwartete Quantile", ylab="Beobachtete Quantile", main=composer, col=values_color, xlim=
        ↪qq_x_limits)
    qqline(values, col="red", lw=2 5)
  })
  dev off()

  # create ECDF plot
  out_filename_ecdf = paste0(instrument, "_ECDF png")
  png(file path(out_folder_ecdf, out_filename_ecdf),
      width = 20,
      height = 14,
      units = "cm",
      res = 120,
      pointsize = 12
  )
  par(mar = c(4, 4, 2, 1))
  quad_plot(composer_data, i, function(histogram, composer, instrument) {
    values = get_histogram_values(histogram)
    plot(ecdf(values), col=values_color, main=composer, xlim=c(24, 108), xaxt ="n", xlab="Tonhöhe", ylab="
        ↪Verteilungsfunktion")
    axis(1, at=ecdf_x_ticks, labels=get_iso_pitches(ecdf_x_ticks), las=1)
  })
  dev off()

  # create histogram plot
  out_filename_histogram = paste0(instrument, "_Histogramm png")
  png(file path(out_folder_histogram, out_filename_histogram),
      width = 20,
      height = 14,
      units = "cm",
      res = 120,
      pointsize = 12
  )

  quad_plot(composer_data, i, function(histogram, composer, instrument) {
    par(cex lab = 1,
        cex axis = 0 8,
        mar = c(4,3 5,1,3 5)
    )
```

```
    histogram_plot(histogram, composer, FALSE)
  }, TRUE)

  # plot legend in window center
  par(xpd=TRUE,
      mar=c(0,0,0,0)
  )
  plot new()
  graph_names = c("KDE", "1  Quartil", "Median", "3  Quartil")
  legend("center", graph_names, col=palette(rainbow(4)), lw=2, cex=par("cex axis"), horiz=TRUE)
  dev off()
}
```

Quellcode-Datei 39: create_model_pitch_median.R

```
# This script creates a linear mixed model of the pitch interquartile range for instruments of all <composers>
# Instruments are defined in CSV file <instruments_csv_file>
# Model parameters are determined based on a likelihood-ratio-test
# Various graphical evaluations are stored in directory <plot_folder>

# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
library(lme4)
library(lmerTest)
source("Vergleich Komponisten/get_lmm_data R")
source("get_iso_pitches R")

# input  corpus and composer data
corpus_folder = "  /  /  /Korpus - Ambitus korrigiert"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
N_composers = length(composers)

# input  read instrument list
instruments_csv_file = "Vergleich Komponisten/Instrumente_Vergleich csv"
instruments_data = read csv(instruments_csv_file, stringsAsFactors=FALSE)
instruments = list(labels=instruments_data[[1]], names=instruments_data[[2]])
N_instruments = length(instruments$labels)

# define folder, colors and line width for plots
plot_folder = "Vergleich Komponisten/Model-Fit"
color_1 = "dodgerblue"
color_2 = "red"
linewidth = 2

# get the data for building the models
stats = get_lmm_data(composers, corpus_folder, instruments$labels)

# create the multilevel models
model_null = lmer(Median~1+(1|Satz), stats, REML=FALSE)
model_1 = lmer(Median~1+Komponist+(1|Satz), stats, REML=FALSE)
model_2 = lmer(Median~1+Komponist+Instrument+(1|Satz), stats, REML=FALSE, contrasts=list(Instrument=contr sum))
model_3 = lmer(Median~1+Komponist+Instrument+Sinfonie+(1|Satz), stats, REML=FALSE, contrasts=list(Instrument=contr
      ↪sum))
model_4 = lmer(Median~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Instrument, stats, REML=FALSE, contrasts=
      ↪list(Instrument=contr sum))
model_5 = lmer(Median~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Instrument+Komponist Sinfonie, stats, REML=
      ↪FALSE, contrasts=list(Instrument=contr sum))
model_6 = lmer(Median~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Instrument+Instrument Sinfonie, stats, REML
      ↪=FALSE, contrasts=list(Instrument=contr sum))
model_7 = lmer(Median~1+Komponist*Instrument*Sinfonie+(1|Satz)+Komponist Instrument+Instrument Sinfonie, stats, REML
      ↪=FALSE, contrasts=list(Instrument=contr sum))

# summary of intercept-only model for ICC calculation
summary(model_null)

# perform likelihood ratio test, print results
lrt = anova(model_null, model_1, model_2, model_3, model_4, model_5, model_6, model_7)
print(lrt)

# calculate pseudo R^2
pseudo_r2 = 1-(lrt$logLik[7]/lrt$logLik[1])

# use final model (model 6), re-level composer factor to change intercept to all 4 composers
formula = Median~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Instrument+Instrument Sinfonie
stats_releveled = stats
stats_releveled$Komponist = relevel(stats$Komponist, 2)
summary(model_6) # intercept = Haydn
model_6_mozart = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_6_mozart) # intercept = Mozart
stats_releveled$Komponist = relevel(stats$Komponist, 3)
model_6_beethoven = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_6_beethoven) # intercept = Beethoven
stats_releveled$Komponist = relevel(stats$Komponist, 4)
model_6_schubert = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_6_schubert) # intercept = Schubert

# get residuals and fitted values
residuals = residuals(model_6)
fitted = fitted(model_6)
```

```r

# plot residuals histogram and estimated density function
png(file path(plot_folder, "Residuals_Histogram_Pitch png"),
    width = 12,
    height = 8,
    units = "cm",
    res = 120,
    pointsize = 12
)
par(mar=c(4, 4, 1, 4))
residuals_limits = c(-15, 15)
hist(residuals, breaks=seq(residuals_limits[1], residuals_limits[2]), col=color_1, xlab="Residuen", ylab="Häufigkeit", main="")
par(new=TRUE)
density = density(residuals)
plot(density, col=color_2, xlim=residuals_limits, lwd=linewidth, xaxt="n", yaxt="n", xlab="", ylab="", main="")
axis(4, ylim=c(0, max(density$y)))
mtext(4, text="Dichtefunktion", line=2 5, cex=par("cex lab"))
dev off()

# plot residuals vs pitches
png(file path(plot_folder, "Residuals_Pitch png"),
    width = 12,
    height = 8,
    units = "cm",
    res = 120,
    pointsize = 12
)
x_limits = c(32, 88)
y_limits = c(-15, 15)
ticks = seq(36, 84, 12)
par(mar=c(4, 4, 1, 1))
plot(fitted, residuals, xlab='Angepasste mittlere Tonhöhe', ylab='Residuen', col="dodgerblue", ylim=y_limits, xaxt="n")
abline(h=0, col=color_2, lw=linewidth)
axis(1, at=ticks, las=1, labels=get_iso_pitches(ticks))
dev off()

# plot fitted values vs original values
png(file path(plot_folder, "Fitted_Values_Pitch png"),
    width = 12,
    height = 8,
    units = "cm",
    res = 120,
    pointsize = 12
)
par(mar=c(4, 4, 1, 1))
plot(fitted, stats$Median, xlab="Angepasste mittlere Tonhöhe", ylab="Tatsächliche mittlere Tonhöhe", col=color_1, xlim=x_limits, ylim=x_limits, xaxt="n", yaxt="n")
segments(0, 0, 127, 127, col=color_2, lwd=linewidth)
axis(1, at=ticks, las=1, labels=get_iso_pitches(ticks))
axis(2, at=ticks, las=1, labels=get_iso_pitches(ticks))
dev off()

# plot main effect  Instrument
cols = rainbow(N_instruments)
png(file path(plot_folder, "Instrument_Original_Fitted_Pitch png"),
    width = 20,
    height = 12,
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
plot(1, type="n", xlim=c(1, 300), ylim=x_limits, yaxt="n", xlab="Index", ylab="Tonhöhe", main="Mittlere Tonhöhe")
for (i in 1 N_instruments) {
  values = stats$Median[stats$Instrument==i]
  lines(values, col=cols[i])
}
axis(2, at=ticks, las=1, labels=get_iso_pitches(ticks))
plot(1, type="n", xlim=c(1, 300), ylim=x_limits, yaxt="n", xlab="Index", ylab="Tonhöhe", main="Angepasste mittlere Tonhöhe")
for (i in 1 N_instruments) {
  values = fitted[stats$Instrument==i]
  lines(values, col=cols[i])
}
axis(2, at=ticks, las=1, labels=get_iso_pitches(ticks))
plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=instruments$names, col=cols, lw=linewidth)
dev off()

# interaction  composer-instrument
x_limits = c(1, N_instruments)
y_limits = c(36, 84)
x_ticks = 1 11
y_ticks= seq(36, 84, 12)
x_label_offset = 6

png(file path(plot_folder, "Instrument_Composer_Original_Fitted_Pitch png"),
    width = 20,
    height = 12,
```

```
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
plot(1, type="n", xlab="Instrument", ylab="Mittlere Tonhöhe", xlim=x_limits, ylim=y_limits, xaxt="n", yaxt="n", main
    ↪="Mittlere Tonhöhe")
graph_colors = rainbow(N_composers)
N = nrow(stats)
for (c in seq(1, N_composers)) {
  composer_data = stats[stats$Komponist==c, ]
  row = c()
  for (i in seq(1, N_instruments)) {
    instrument_data = composer_data[composer_data$Instrument==i,]
    row[i] = mean(instrument_data$Median)
  }
  lines(1 11, row, col=graph_colors[c], pch=18, type="b", lw=linewidth)
}
text(x_ticks, par("usr")[3]-x_label_offset, labels=instruments$names, srt=45, pos=1, xpd=TRUE, cex=par("cex axis"))
axis(1, at=x_ticks, las=1, labels=rep("", N_instruments))
axis(2, at=y_ticks, labels=get_iso_pitches(y_ticks), las=2)

plot(1, type="n", xlab="Instrument", ylab="Mittlere Tonhöhe", xlim=x_limits, ylim=y_limits, xaxt="n", yaxt="n", main
    ↪="Angepasste mittlere Tonhöhe")
for (c in seq(1, N_composers)) {
  composer_data = stats[stats$Komponist==c, ]
  row = c()
  selection = fitted[stats$Komponist==c]
  for (i in seq(1, N_instruments)) {
    instrument_data = selection[composer_data$Instrument==i]
    row[i] = mean(instrument_data)
  }
  lines(1 11, row, col=graph_colors[c], pch=18, type="b", lw=linewidth)
}

text(x_ticks, par("usr")[3]-x_label_offset, labels=instruments$names, srt=45, pos=1, xpd=TRUE, cex=par("cex axis"))
axis(1, at=x_ticks, las=1, labels=rep("", N_instruments))
axis(2, at=y_ticks, labels=get_iso_pitches(y_ticks), las=2)

plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=composers, col=graph_colors, lw=linewidth, cex=par("cex axis"))
dev off()

# interaction  instrument-symphony
symphony_numbers = sort(unique(stats$Sinfonie))
N_symphonies = length(symphony_numbers)
colors = rainbow(N_instruments)
x_limits = c(1, N_symphonies)
y_limits = c(36, 84)
x_ticks = seq(1, N_symphonies, 4)
y_ticks = seq(36, 84, 12)

png(file path(plot_folder, "Instrument_Symphony_Original_Fitted_Pitch png"),
    width = 20,
    height = 12,
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
plot(1, type="n", xlab="Werknummer", ylab="Mittlere Tonhöhe", xlim=x_limits, ylim=y_limits, xaxt="n", yaxt="n", main
    ↪="Mittlere Tonhöhe")
for (i in 1 N_instruments) {
  instrument_data = stats[stats$Instrument==i,]
  row = c()

  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = instrument_data[instrument_data$Sinfonie==symphony_number, ]
    row[s] = mean(symphony_data$Median)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])
axis(2, at=y_ticks, labels=get_iso_pitches(y_ticks), las=3)

plot(1, type="n", xlab="Werknummer", ylab="Mittlere Tonhöhe", xlim=x_limits, ylim=y_limits, xaxt="n", yaxt="n",main=
    ↪"Angepasste mittlere Tonhöhe")
for (i in 1 N_instruments) {
  instrument_data = stats[stats$Instrument==i,]
  selection = fitted[stats$Instrument==i]
  row = c()
  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = selection[instrument_data$Sinfonie==symphony_number]
    row[s] = mean(symphony_data)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
```

```
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])
axis(2, at=y_ticks, labels=get_iso_pitches(y_ticks), las=3)

plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=instruments$names, col=colors, lw=linewidth)
dev off()
```

Quellcode-Datei 40: create_model_pitch_iqr.R

```
# This script creates a linear mixed model of the pitch median for instruments of all <composers>
# Instruments are defined in CSV file <instruments_csv_file>
# Model parameters are determined based on a likelihood-ratio-test
# Various graphical evaluations are stored in directory <plot_folder>

# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
library(lme4)
library(lmerTest)
source("Vergleich Komponisten/get_lmm_data R")
source("get_iso_pitches R")

# input  corpus and composer data
corpus_folder = "  /  /  /Korpus - Ambitus korrigiert"
composers = c("Haydn", "Mozart", "Beethoven", "Schubert")
N_composers = length(composers)

# input  read instrument list
instruments_csv_file = "Vergleich Komponisten/Instrumente_Vergleich csv"
instruments_data = read csv(instruments_csv_file, stringsAsFactors=FALSE)
instruments = list(labels=instruments_data[[1]], names=instruments_data[[2]])
N_instruments = length(instruments$labels)

# define folder, colors and line width for plots
plot_folder = "Vergleich Komponisten/Model-Fit"
color_1 = "dodgerblue"
color_2 = "red"
linewidth = 2

# get the data for building the models
stats = get_lmm_data(composers, corpus_folder, instruments$labels)

# create the multilevel models
model_null = lmer(IQR~1+(1|Satz), stats, REML=FALSE)
model_1 = lmer(IQR~1+Komponist+(1|Satz), stats, REML=FALSE)
model_2 = lmer(IQR~1+Komponist+Instrument+(1|Satz), stats, REML=FALSE, contrasts=list(Instrument=contr sum))
model_3 = lmer(IQR~1+Komponist+Instrument+Sinfonie+(1|Satz), stats, REML=FALSE, contrasts=list(Instrument=contr sum)
    ↪)
model_4 = lmer(IQR~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Sinfonie, stats, REML=FALSE, contrasts=list(
    ↪Instrument=contr sum))
model_5 = lmer(IQR~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Sinfonie+Instrument Sinfonie, stats, REML=
    ↪FALSE, contrasts=list(Instrument=contr sum))
model_6 = lmer(IQR~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Sinfonie+Instrument Sinfonie+Komponist
    ↪Instrument, stats, REML=FALSE, contrasts=list(Instrument=contr sum))
model_7 = lmer(IQR~1+Komponist*Instrument*Sinfonie+(1|Satz)+Komponist Sinfonie+Instrument Sinfonie+Komponist
    ↪Instrument, stats, REML=FALSE, contrasts=list(Instrument=contr sum))

# summary of intercept-only model for ICC calculation
summary(model_null)

# perform likelihood ratio test, print results
lrt = anova(model_null, model_1, model_2, model_3, model_4, model_5, model_6, model_7)
print(lrt)

# calculate pseudo R^2
pseudo_r2 = 1-(lrt$logLik[6]/lrt$logLik[1])

# use final model (model 5), re-level composer factor to change intercept to all 4 composers
formula = IQR~1+Komponist+Instrument+Sinfonie+(1|Satz)+Komponist Sinfonie+Instrument Sinfonie
stats_releveled = stats
stats_releveled$Komponist = relevel(stats$Komponist, 2)
summary(model_5) # intercept = Haydn
model_5_mozart = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_5_mozart) # intercept = Mozart
stats_releveled$Komponist = relevel(stats$Komponist, 3)
model_5_beethoven = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_5_beethoven) # intercept = Beethoven
stats_releveled$Komponist = relevel(stats$Komponist, 4)
model_5_schubert = lmer(formula, stats_releveled, REML=FALSE, contrasts=list(Instrument=contr sum))
summary(model_5_schubert) # intercept = Schubert

# get residuals and fitted values
residuals = residuals(model_5)
fitted = fitted(model_5)

# plot residuals histogram and estimated density function
png(file path(plot_folder, "Residuals_Histogram_Ambitus png"),
    width = 12,
    height = 8,
```

```
    units = "cm",
    res = 120,
    pointsize = 12
)
par(mar=c(4, 4, 1, 4))
residuals_limits = c(-17, 17)
hist(residuals, breaks=seq(residuals_limits[1], residuals_limits[2]), col=color_1, xlab="Residuen", ylab="Häufigkeit", main="")
par(new=TRUE)
density = density(residuals)
plot(density, col=color_2, xlim=residuals_limits, lwd=linewidth, xaxt="n", yaxt="n", xlab="", ylab="", main="")
axis(4, ylim=c(0, max(density$y)))
mtext(4, text="Dichtefunktion", line=2 5, cex=par("cex lab"))
dev off()

# plot residuals vs pitches
png(file path(plot_folder, "Residuals_Ambitus png"),
    width = 12,
    height = 8,
    units = "cm",
    res = 120,
    pointsize = 12
)
x_limits = c(4, 12)
par(mar=c(4, 4, 1, 1))
plot(fitted, residuals, xlab='Angepasster mittlerer Tonumfang', ylab='Residuen', col="dodgerblue", xlim=x_limits, ylim=residuals_limits)
abline(h=0, col=color_2, lw=linewidth)
dev off()

# plot fitted values vs original values
png(file path(plot_folder, "Fitted_Values_Ambitus png"),
    width = 12,
    height = 8,
    units = "cm",
    res = 120,
    pointsize = 12
)
x_limits = c(0, 25)
plot(fitted, stats$IQR, xlab="Angepasster mittlerer Tonumfang", ylab="Tatsächlicher mittlerer Tonumfang", col=color_1, xlim=x_limits, ylim=x_limits)
segments(0, 0, 127, 127, col=color_2, lwd=linewidth)
dev off()

# plot main effect  Instrument
cols = rainbow(N_instruments)
y_limits = c(0, 25)
png(file path(plot_folder, "Instrument_Original_Fitted_Ambitus png"),
    width = 20,
    height = 12,
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
plot(1, type="n", xlim=c(1, 300), ylim=y_limits, xlab="Index", ylab="Mittlerer Tonunfang", main="Mittlerer Tonumfang")
for (i in 1 N_instruments) {
  values = stats$IQR[stats$Instrument==i]
  lines(values, col=cols[i])
}
y_limits = c(0, 15)
plot(1, type="n", xlim=c(1, 300), ylim=y_limits, xlab="Index", ylab="Mittlerer Tonunfang", main="Angepasster mittlerer Tonumfang")
for (i in 1 N_instruments) {
  values = fitted[stats$Instrument==i]
  lines(values, col=cols[i])
}
plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=instruments$names, col=cols, lw=linewidth)
dev off()

# interaction  instrument-symphony
symphony_numbers = sort(unique(stats$Sinfonie))
N_symphonies = length(symphony_numbers)
colors = rainbow(N_instruments)
x_limits = c(1, N_symphonies)
y_limits = c(4, 15)
x_ticks = seq(1, N_symphonies, 4)
y_ticks = seq(36, 84, 12)

png(file path(plot_folder, "Instrument_Symphony_Original_Fitted_Ambitus png"),
    width = 20,
    height = 12,
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
plot(1, type="n", xlab="Werknummer", ylab="Mittlerer Tonumfang", xaxt="n", xlim=x_limits, ylim=y_limits, main="
```

```
Mittlerer Tonumfang")
for (i in 1 N_instruments) {
  instrument_data = stats[stats$Instrument==i,]
  row = c()

  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = instrument_data[instrument_data$Sinfonie==symphony_number, ]
    row[s] = mean(symphony_data$IQR)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])

plot(1, type="n", xlab="Werknummer", ylab="Mittlerer Tonumfang", xlim=x_limits, ylim=y_limits, xaxt="n", main="Angepasster mittlerer Tonumfang")
for (i in 1 N_instruments) {
  instrument_data = stats[stats$Instrument==i,]
  selection = fitted[stats$Instrument==i]
  row = c()
  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = selection[instrument_data$Sinfonie==symphony_number]
    row[s] = mean(symphony_data)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])

plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=instruments$names, col=colors, lw=linewidth)
dev off()

# interaction  composer-symphony
png(file path(plot_folder, "Composer_Symphony_Original_Fitted_Ambitus.png"),
    width = 20,
    height = 12,
    units = "cm",
    res = 120,
    pointsize = 12
)
layout(mat=matrix(c(1, 2, 3, 3), ncol=2), widths=c(2, 1))
par(mar=c(4, 4, 2, 0))
colors = rainbow(N_composers)
y_limits = c(6, 10)
plot(1, type="n", xlab="Werknummer", ylab="Mittlerer Tonumfang", xaxt="n", xlim=x_limits, ylim=y_limits, main="Mittlerer Tonumfang")
for (i in 1 N_composers) {
  composer_data = stats[stats$Komponist==i,]
  row = c()

  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = composer_data[composer_data$Sinfonie==symphony_number, ]
    row[s] = mean(symphony_data$IQR)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])

plot(1, type="n", xlab="Werknummer", ylab="Mittlerer Tonumfang", xlim=x_limits, ylim=y_limits, xaxt="n", main="Angepasster mittlerer Tonumfang")
for (i in 1 N_composers) {
  composer_data = stats[stats$Komponist==i,]
  selection = fitted[stats$Komponist==i]
  row = c()
  for (s in 1 N_symphonies) {
    symphony_number = symphony_numbers[s]
    symphony_data = selection[composer_data$Sinfonie==symphony_number]
    row[s] = mean(symphony_data)
  }

  lines(1 N_symphonies, row, col=colors[i], lw=linewidth)
}
axis(1, at=x_ticks, labels=symphony_numbers[x_ticks])

plot(1, type="n", axes=FALSE, xlab="", ylab="")
legend("left", legend=composers, col=colors, lw=linewidth)
dev off()
```

Quellcode-Datei 41: get_lmm_data.R

```
# get_lmm_data(composers, corpus_folder, instrument_labels)
# Returns a data frame containing the pitch median and IQR of every instrument for all <composers>
# @composers  String vector [Nx1]  Names of the composers, subfolders of <corpus folder>
# @corpus_folder  Root location of the CSV files of merged instrument distributions
# @instrument_labels  String vector [Nx1]  Names of instruments to be analyzed
```

```
library(stringr) # regular expressions
source("read_csv_histograms R")
source("get_histogram_values R")
get_lmm_data = function(composers, corpus_folder, instrument_labels) {
  csv_header_length = 10
  stats_names = c("Komponist", "Sinfonie", "Satz", "Instrument", "Median", "IQR")
  stats = as data frame(setNames(replicate(6, numeric(0)), stats_names))

  # iterate composers
  for (c in 1 length(composers)) {
    composer = composers[c]
    data_files = list files(file path(corpus_folder, composer), full names=TRUE)

    # iterate files of this composer
    for (file in data_files) {
      data = read_csv_histograms(file, csv_header_length)

      # use regular expression to extract symphony and movement number from filename
      regex = str_match(file, "([0-9]+)_([0-9]+) csv$")
      symphony = as numeric(regex[2])
      movement = as numeric(regex[3])

      # iterate instruments in this movement
      for (i in seq(1, length(data$labels))) {
        instrument_label = data$labels[i]

        # remove keys of transposed instruments
        instrument_label = sub(" in  *", "", instrument_label)

        # check if instrument should be analyzed
        if (any(instrument_labels == instrument_label)) {
          instrument_index = which(instrument_labels == instrument_label)
          values = get_histogram_values(data$histograms[i, ])

          # ignore Fagotto with only one pitch
          if (length(unique(values)) > 1) {
            quartiles = quantile(values, probs=c(0 25, 0 5, 0 75), na rm=FALSE, names=FALSE)
            line = c(c, symphony, movement, instrument_index, quartiles[2], quartiles[3]-quartiles[1])
            stats = rbind(stats, line)
          }
        }
      }
    }
  }

  # re-apply stat names because they got changed by rbind()
  stats = setNames(stats, stats_names)

  # convert composer, movement and instrument to factors
  stats[[1]] = factor(stats[[1]]) # composer
  stats[[3]] = factor(stats[[3]]) # movement
  stats[[4]] = factor(stats[[4]]) # instrument

  return(stats)
}
```

Quellcode-Datei 42: get_composer_instrument_data.R

```
# function get_composer_instrument_data(composers, corpus_folder)
# Returns a list of merged instrument pitch distributions for all <composers>
# The list has the named properties <data> containing the data and <labels> containing instrument names
# The <data> property is itself a list with entries for each composer  These entries have the properties
# <histograms>, with the pitch distribution data sorted as in <labels>, and <composer> for the composer name
# @composers  String vector [Nx1]  Names of the composers, subfolders of <corpus folder>
# @corpus_folder  Root location of the CSV files of merged instrument distributions
source("read_csv_histograms R")
get_composer_instrument_data = function(composers, corpus_folder) {
  N_composers = length(composers)
  composer_data = vector("list", N_composers)

  for (c in seq(1, N_composers)) {
    filename = paste0(composers[c], "_Instrumente csv")
    data = read_csv_histograms(file path(corpus_folder, filename), 1)
    composer_data[[c]]$histograms = data$histograms
    composer_data[[c]]$composer = composers[c]
  }

  return(list(data=composer_data, labels=data$labels))
}
```

Quellcode-Datei 43: compare_classification_models.R

```
# This script compares the instrument classification performance of 6 different classification algorithms
# Training data is taken from <corpus_folder> for given <composers>
# The instruments defined in CSV file <instruments_file> are used as classes
# Evaluation plots are stored in <out_folder>,

```

```
# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
source("Klassifikation/get_features R")
library(corrplot)
library(caret)
library(MASS)
library(kernlab)
library(klaR)
library(ipred)
library(randomForest)

# set seed for random numbers
seed = 123

# input  instrument class list
instruments_file = "Klassifikation/Instrumente_Klassen csv"
instruments_data = read csv(instruments_file, skip=1, header=FALSE, stringsAsFactors=FALSE)
instrument_classes = unique(instruments_data[[2]])

# input  corpus and composer data
corpus_folder = "  /  /  /Korpus - Ambitus korrigiert"
composers = c("Mozart", "Beethoven")

# output  image folder
out_folder = "Klassifikation/Plots"

# get feature data
features = get_features(corpus_folder, composers, instruments_data)

# plot feature correlation
png(file path(out_folder, "Feature_Correlation png"),
    width = 16,
    height = 16,
    units = "cm",
    res = 120,
    pointsize = 12
)
correlation_matrix = cor(features$values[, ])
corrplot(correlation_matrix, type = "upper", method = "color", order="hclust", diag=FALSE, outline=TRUE,
         tl pos="td", tl cex=1, tl col='black', tl srt=45)
dev off()

# train the models
control = trainControl(method="cv", number=10, repeats=10, allowParallel=TRUE)
metric = "Accuracy"
pre_process = c("center", "scale")

# Linear Discriminant Analysis
set seed(seed)
fit lda = train(features$values, features$labels, method="lda", metric=metric, pre_process, trControl=control)
# SVM Radial
set seed(seed)
fit svmRadial = train(features$values, features$labels, method="svmRadial", metric=metric, preProc=pre_process,
    ↪trControl=control)
# KNN
set seed(seed)
fit knn = train(features$values, features$labels, method="knn", metric=metric, pre_process, trControl=control)
# Naive Bayes
set seed(seed)
fit nb = train(features$values, features$labels, method="nb", metric=metric, trControl=control, preProcess=
    ↪pre_process)
# Bagged CART
set seed(seed)
fit treebag = train(features$values, features$labels, method="treebag", metric=metric, trControl=control, preProcess
    ↪=pre_process)
# Random Forest
set seed(seed)
fit rf = train(features$values, features$labels, method="rf", metric=metric, trControl=control, preProcess=
    ↪pre_process)

# print results
results = resamples(list(fit lda, fit knn, fit nb, fit treebag, fit rf, fit svmRadial),
                    modelNames=c("LDA", "KNN", "Naive Bayes", "Bagged Tree", "Random Forest", "SVM"))
summary(results)

# create boxplot
png(file path(out_folder, "Modelle_Boxplot_Accuracy_Kappa png"),
    width = 16,
    height = 10,
    units = "cm",
    res = 120,
    pointsize = 12
)
bwplot(results)
dev off()

# create dotplot
png(file path(out_folder, "Modelle_Dotplot_Accuracy_Kappa png"),
    width = 16,
    height = 10,
    units = "cm",
```

```
    res = 120,
    pointsize = 12
)
dotplot(results)
dev off()
```

Quellcode-Datei 44: model_evaluation.R

```
# This script evaluates the Random Forest classification algorithm
# Training data is taken from <corpus_folder> for given <composers>
# The instruments defined in CSV file <instruments_file> are used as classes
# Evaluation plots are stored in <out_folder>,

# clear environment, set working directory, import helper functions
rm(list=ls())
setwd("<path>/R/2 - Auswertung")
source("Klassifikation/get_features R")
library(corrplot)
library(caret)
library(randomForest)

# set seed for random numbers
seed = 123

# input  instrument class list
instruments_file = "Klassifikation/Instrumente_Klassen csv"
instruments_data = read csv(instruments_file, skip=1, header=FALSE, stringsAsFactors=FALSE)

# input  corpus and composer data
corpus_folder = "  /  /  /Korpus - Ambitus korrigiert"
composers = c("Mozart", "Beethoven")

# output  image folder
out_folder = "Klassifikation/Plots"

# get feature data
features = get_features(corpus_folder, composers, instruments_data)

# divide data into test and train set
set seed(seed)
partition = createDataPartition(y=features$labels, p=0 7, list=FALSE)
features_train = features$values[partition, ]
labels_train = features$labels[partition]
features_test = features$values[-partition, ]
labels_test = features$labels[-partition]

# train classifier
set seed(seed)
control = trainControl(method="cv", number=10, allowParallel=TRUE, classProbs=TRUE)
metric = "Accuracy"
pre_process = c("center", "scale")
model = train(features_train, labels_train, method="rf", metric=metric, trControl=control, preProcess=pre_process)

# predict test set and create confusion matrix
predictions = predict(model, newdata=features_test)
confusion = confusionMatrix(predictions, labels_test)

# plot confusion matrix
print(confusion)
png(file path(out_folder, "Random_Forest_Confusion_Matrix png"),
    width = 16,
    height = 16,
    units = "cm",
    res = 120,
    pointsize = 12
)
corrplot(confusion$table, p mat=confusion$table, method="circle", is corr=FALSE, tl col="black", insig="p-value", tl
    ↪ cex=0 8, cl cex=0 8, mar=c(0, 3, 0, 0))
text(1, 8, "Klassifikation", srt=90, xpd=TRUE, pos=2, offset=6, cex=par("cex lab"))
mtext("Instrument", side=3, line=2, las=1, cex=par("cex lab"))
dev off()

# plot variable importance
png(file path(out_folder, "Random_Forest_Variable_Importance png"),
    width = 16,
    height = 10,
    units = "cm",
    res = 120,
    pointsize = 12
)
plot(varImp(model, type=2), xlab="Wichtigkeit")
dev off()


# define the control using a random forest selection function
rfe_control = rfeControl(functions=rfFuncs, method="cv", number=3)

# run the RFE algorithm
results = rfe(features$values, features$labels, sizes=c(1 10), rfeControl=rfe_control)
```

```
# summarize the results
print(results)

# plot the results
png(file path(out_folder, "Random_Forest_Selected_Features png"),
    width = 16,
    height = 10,
    units = "cm",
    res = 120,
    pointsize = 12
)
plot(results, type=c("g", "o"), metric=metric, xlab="Anzahl gewählter Prädiktoren")

dev off()

# list the chosen features
print(predictors(results))
```

Quellcode-Datei 45: get_features.R

```
# get_features(corpus_folder, composers, instruments_data)
# Returns a data frame containing the features of the pitch distributions of all instruments
# defined in <instruments_data> for all <composers>
# @corpus_folder  String  The path to the corpus folder
# @composers  String array [1xN]  Contains composer names
# @instruments_data  Data frame [2xN]  Instrument names and labels
source("read_csv_histograms R")
source("Klassifikation/extract_features R")
get_features = function(corpus_folder, composers, instruments_data) {
  csv_header_length = 10
  min_pitches = 2
  min_notes = 10
  N_features = length(extract_features("names"))
  feature_matrix = as data frame(matrix(nrow=0, ncol=N_features))
  labels = list()

  # iterate composers
  for (composer in composers) {
    data_files = list files(file path(corpus_folder, composer), full names=TRUE)

    # iterate files of this composer
    for (file in data_files) {
      data = read_csv_histograms(file, csv_header_length)

      # iterate instruments in this movement
      for (i in seq(1, length(data$labels))) {
        instrument_label = data$labels[i]

        # remove keys of transposed instruments
        instrument_label = sub(" in  *", "", instrument_label)

        # get class of this instrument
        instrument_class = instruments_data[[2]][instruments_data[[1]] == instrument_label]

        # add features of this pitch distribution if this is a valid instrument class
        # and it contains at least the minimum amount of pitches and notes
        if (length(instrument_class) > 0 &&
            sum(data$histograms[i, ]) >= min_notes &&
            sum(data$histograms[i, ] > 0) >= min_pitches) {

          features = extract_features(data$histograms[i, ])
          feature_matrix = rbind(feature_matrix, features$features)
          labels = rbind(labels, instrument_class)
        }
      }
    }
  }

  # name feature columns, put features and labels in a list and save it
  names(feature_matrix) = features$names
  features = list(values=feature_matrix, labels=factor(unlist(labels)))

  return(features)
}
```

Quellcode-Datei 46: extract_features.R

```
# function extract_features(histogram)
# Returns the features of a pitch histogram for classification
# @histogram  Numeric vector [128x1]  The MIDI pitch histogram
source("get_histogram_values R")
extract_features = function(histogram) {
  names = c("Median", "Mittelwert", "Standardabweichung", "Varianz", "Minimum", "Maximum",
            "Ambitus", "1  Quartil", "3  Quartil", "Interquartilsabstand")
  if (is character(histogram) && histogram == "names") {
    return(names)
  }
```

```
  else {
    values = get_histogram_values(histogram)
    quartiles = quantile(values, probs = seq(0, 1, 0 25), na rm=FALSE, names=FALSE)
    range_start = min(values)
    range_end = max(values)
    features = c(median(values), mean(values), sd(values), var(values), range_start, range_end,
                 range_end-range_start, quartiles[2], quartiles[4], quartiles[4]-quartiles[2])
    return(list(names=names, features=features))
  }
}
```

Quellcode-Datei 47: read_csv_histograms.R

```
# read_csv_histograms(csv_file, header_length?)
# Reads MIDI histograms from a CSV file and returns their data
# @csv_file  String  Absolute path to CSV file
# @header_length  Number  Header size, i e  how many lines are skipped before reading data  Defaults to 0
read_csv_histograms = function(csv_file, header_length) {
  if (missing(header_length)) {
    header_length = 0;
  }

  data = read csv(csv_file, skip=header_length, skipNul = TRUE, header = FALSE, stringsAsFactors=FALSE)

  return(list("labels" = data[, 1], "histograms" = data[, 6 133]))
}
```

Quellcode-Datei 48: histogram_plot.R

```
# histogram_plot(histogram, title?, plot_legend?)
# Creates a plot of a pitch histogram  Plots estimated density function over absolute pitch frequencies
# @histogram  Numeric vector [1x128]  Contains the absolute pitch frequencies
# @title  String  Optional title for the plot  Defaults to empty string
# @plot_legend  Boolean  If TRUE a legend will be shown in the topleft corner of the plot  Defaults to TRUE
source("get_iso_pitches R")
source("get_histogram_values R")
source("estimate_density R")
histogram_plot = function(histogram, title, plot_legend) {
  if (missing(title)) {
    title = ""
  }

  if (missing(plot_legend)) {
    plot_legend = TRUE
  }

  bins = c(0 127)
  x_ticks = seq(0, 127, 12) # mark all C pitches
  x_ticks[length(x_ticks)+1] = 127 # add last pitch (G10)
  x_limits = c(0, 127)
  x_label = "Tonhöhe"
  y_label = "Absolute Häufigkeit"
  y_label_2 = "Dichtefunktion"
  graph_names = c("KDE", "1  Quartil", "Median", "3  Quartil")
  graph_colors = palette(rainbow(length(graph_names)))
  graph_linewidth = 2
  bar_color = "lightgrey"

  # perform KDE
  values = get_histogram_values(histogram)
  density = estimate_density(values)
  density_values = rep(density$x, histogram)

  # compute quartiles, interpolate values to center on histogram bins
  quartiles = quantile(density_values, probs = seq(0, 1, 0 25), na rm=FALSE, names=FALSE)[2 4]
  quartiles_x = quartiles-0 5
  quartiles_y = (density$y[quartiles]+density$y[quartiles+1])/2

  # plot histogram, label x axis and first y axis (absolute frequencies), set x axis limits
  hist(values, breaks=bins, probability = FALSE, col=bar_color, xaxt="n", xlim=x_limits, plot=TRUE, main="", xlab=""
       ↪, ylab="")

  # plot again with relative frequencies to scale viewport, maintain x axis limits
  par(new=T)
  hist(values, breaks=bins, probability = TRUE, col=bar_color, xaxt="n", xlim=x_limits, yaxt="n", plot=TRUE, main=
       ↪title, xlab=x_label, ylab="")

  # plot quartiles and KDE
  segments(quartiles_x, c(0,0,0), quartiles_x, quartiles_y, col=graph_colors[2 4], lw=graph_linewidth, xaxt="n",
       ↪yaxt="n")
  lines(density, col=graph_colors[1], lw=graph_linewidth)

  # add second y axis and label it
  axis(4, ylim=c(0, max(density$y)))
  mtext(4, text=y_label_2, line=2 1, cex=par("cex lab")-0 3)
  mtext(2, text=y_label, line=2, cex=par("cex lab")-0 3)

```

```
  # set x axis
  axis(1, at=x_ticks, labels=get_iso_pitches(x_ticks), las=1)

  # add a legend if desired
  if (plot_legend == TRUE) {
    legend("topleft", graph_names, col=graph_colors, lw=graph_linewidth, cex=par("cex axis"))
  }
}
```

Quellcode-Datei 49: histogram_boxplot.R

```
# histogram_boxplot(data, labels, colors, box_range?, group_step?, box_step?, x_ticks?, x_limits?)
# Creates a boxplot of given pitch distributions with given
# Distributions are grouped by the number of elements in <colors>
# @data  List of numeric vectors [NxM]  List of MIDI pitch data
# @labels  String vector [Nx1]  Labels for the distributions (e g  instrument names)  Used to label y axis
# @colors  String vector [0x1]  Determines the box colors and box grouping
# @box_range?  Numeric value  Determines whisker range, see boxplot()  Defaults to 0 (entire range)
# @group_step?  Numeric value  Space between box groups  Defaults to 2
# @box_step?  Numeric value  Space between boxes within a group  Defaults to 1
# @x_ticks?  Numeric vector [Px1]  Ticks on x axis  Values are MIDI note numbers  Marks all Cs on default
# @x_limits?  Numeric vector [2x1]  X axis limits as minimum and maximum MIDI note number  Defaults to entire range
source("get_iso_pitches R")
histogram_boxplot = function(data, labels, colors, box_range, group_step, box_step, x_ticks, x_limits) {
  if (missing(box_range)) {
    box_range = 0
  }

  if (missing(group_step)) {
    group_step = 2
  }

  if (missing(box_step)) {
    box_step = 1
  }

  if (missing(x_ticks)) {
    x_ticks = seq(0, 127, 12)
  }

  if (missing(x_limits)) {
    x_limits = c(0,127)
  }

  # calculate y positions of boxes and labels
  N_groups = length(labels)
  N_boxes_per_group = length(colors)
  box_pos = rep(c((N_groups-1) 0), each=N_boxes_per_group)*(group_step+N_boxes_per_group)+c(N_boxes_per_group 1)*box_step
  y_label_pos = c((N_groups-1) 0)*(group_step+N_boxes_per_group)+N_boxes_per_group*box_step/2+0 5

  # create boxplot
  boxplot(data,
          at = box_pos,
          xaxt ="n",
          yaxt = "n",
          xlab = "Tonhöhe",
          ylim = x_limits,
          range = box_range,
          col = colors,
          horizontal = TRUE
  )

  # label axes
  axis(1, at=x_ticks, labels=get_iso_pitches(x_ticks), las=1)
  mtext(labels, side=2, line=0 5, at=y_label_pos, las=1, cex=par("cex lab"))
}
```

Quellcode-Datei 50: estimate_density.R

```
# estimate_density(values)
# Returns the estimated density function and quantiles of a given array of numeric values
# @values  Numeric vector [1xN]
library(provenance)
estimate_density = function(values) {

  # perform KDE with guassian kernel, try to use Botev's optimal bandwith estimation
  bandwidth = max(botev(values), 2)
  density = density(values, bw=bandwidth, n=128, from=0, to=127)

  # scale density values so their sum adds up to exactly 1
  scaled_values = density$y+((1-sum(density$y))/128)

  # variables and helper function for quartile calculation
  sum = 0;
  density_index = 1
  get_quartile = function(quartile) {
```

```
    while (sum < quartile) {
      sum = sum+scaled_values[density_index]
      density_index = density_index+1
    }
    return(density_index-1) # return MIDI bin
  }

  # calculate quantiles and add them to density object
  density$quartiles[1] = get_quartile(0 25)
  density$quartiles[2] = get_quartile(0 5)
  density$quartiles[3] = get_quartile(0 75)

  return(density)
}
```

Quellcode-Datei 51: get_iso_pitches.R

```
# get_iso_pitches(midi_numbers?)
# Returns the ISO pitch names of MIDI note numbers as an expression (subscripted octave) for use in plots
# @midi_numbers  Numeric vector [1xN]  Contains zero-indexed MIDI note numbers from range [0   127]
#                If not set, the function will return the names of all 128 MIDI pitches
get_iso_pitches = function(midi_numbers) {
  bins = c(0 127)
  notes = c('C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'H')
  note_names = notes[bins %% 12+1]
  octaves = floor(bins/12)-1

  if (missing(midi_numbers)) {
    midi_numbers = bins
  }

  octave_subscript = function(i) {
    as expression(substitute(note[octave], list(note=as name(note_names[i+1]), octave=octaves[i+1])))
  }

  return(sapply(midi_numbers, octave_subscript))
}
```

Quellcode-Datei 52: get_histogram_values.R

```
# get_histogram_values(histogram)
# Converts a MIDI pitch histogram to a data vector of the original MIDI pitch values
# @histogram  Numeric vector [1x128]  Contains the absolute pitch frequencies
get_histogram_values = function(histogram) {
  return(rep(c(0 127), histogram))
}
```

Quellcode-Datei 53: quad_plot.R

```
# quad_plot(composer_data, instrument_index, plot_function, center_legend?)
# Plots instrument pitch distributions for each composer in a 2x2 grid using a given <plot_function>
# @composer_data  List of composer instrument data as returned by get_composer_instrument_data()
# @instrument_index  Number  Index of the instrument to plot
# @plot_function  Callback function for actual plot of type f(histogram, composer, instrument) where <histogram>
# is the MIDI pitch histogram, <composer> the name of the composer and <instrument> the instrument name
# <plot_function> is not called when data values are empty, a blank plot will be drawn instead
# @center_legend  Boolean  If TRUE, a third row will be added to the center where a legend can be placed
quad_plot = function(composer_data, instrument_index, plot_function, center_legend) {
  if (missing(center_legend)) {
    center_legend = FALSE
  }

  # use a standard 2x2 grid for the plots
  if (center_legend == FALSE) {
    par(mfrow = c(2, 2))
  }
  else { # use a 2x3 grid, leave the middle row empty for the legend
    layout(matrix(c(1,2,5,5,3,4), byrow=T, ncol=2), heights=c(4,1,4))
  }

  data = composer_data$data
  for (c in seq(1, length(data))) {
    histogram = data[[c]]$histograms[instrument_index,]

    if (sum(histogram) > 0) { # pass data to callback function
      plot_function(histogram, data[[c]]$composer, composer_data$labels[instrument_index])
    }
    else { # no data for this composer  blank plot
      plot(0, type='n', axes=FALSE, ann=FALSE)
    }
  }
}
```